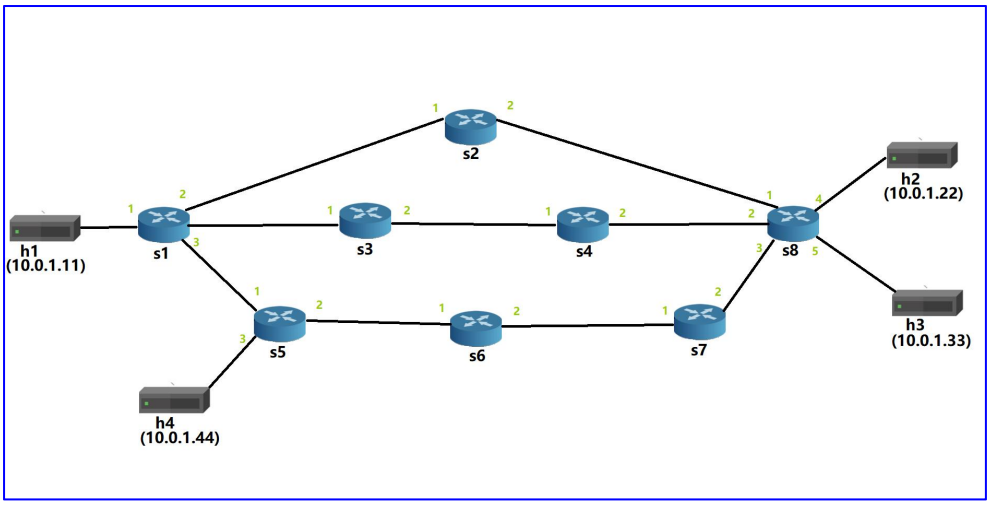
Report

Network QoS management using SDN: a delay focused study

SU Runbo

29/06/2020



UNIVERSITÉ DE LORRAINE

Loria
Laboratoire lorrain de recherche en informatique et ses applications

Abstract

This internship report presents our investigation on using SDN for dynamically controlling network QoS, with a focus on delay performance. The QoS management using SDN has been largely exploited during the recent years, but there are still challenging issues that the current researches do not invest enough, especially when explicit time and delay are concerned. Moreover, so many different types of SDN controller platforms exist, but they provide very different functions for managing QoS. So how to choose the good controller platform is also an important issue since it conditions the success or unsuccess of the QoS management objectives. In this work, we investigate the possibility of using POX and P4 for delay performance control. For this purpose, we experiment on Mininet a typical multi-hop and multi-path switched network composed of three paths (high, middle and low) between sender hosts and receiver hosts. We developed a dynamic delay aware automatic re-routing algorithm using SDN controller that better balances the total traffic between high, middle and low paths. Through this example, we compare POX and P4 in terms of both the QoS perspectives and ease of use. The results of delay aware dynamic rerouting mechanism show great potentials of SDN in delay QoS management. The comparisons between POX and P4 revealed some inaccuracy in queue length indication and the lack of explicit time management components in P4.

Keywords: SDN (Software Defined Networking), QoS (Quality of service), delay management, Mininet, P4, POX.

Résumé

Ce rapport de stage présente notre étude sur l'utilisation de SDN pour la gestion dynamique de la QoS, avec un focus sur le délai. Utiliser SDN pour gérer la QoS a été largement investi ces dernières années par la communauté de chercheurs, mais il reste encore des défis quant à la gestion explicite de performances temporelles comme délais de bout en bout. En plus il existe de très nombreuses plateformes de contrôleur SDN qui proposent des fonctionnalités très

différentes pour la gestion de la QoS, le bon choix d'une telle plateforme consiste aussi un défi, car il conditionne le succès ou non dans les objectifs de la gestion de la QoS. Dans ce travail, nous investiguons la possibilité d'utiliser P4 et POX pour la gestion explicite de délais. Pour cela, nous avons expérimenté les deux approches sur Mininet un réseau commuté typique multi-saut et multi-chemin composé de trois chemins (high, middle, low) entre les émetteurs et récepteurs. Nous avons développé an algorithme de re-routage dynamique qui change de chemin en fonction des délais constatés et des seuils de délais à garantir. Au travers de cet exemple, nous comparons POX et P4 en termes de facilités de gérer la QoS et d'implémentation. Les résultats obtenus montrent les grands potentiels de SDN pour la gestion de la QoS et en particulier les délais. La comparaison entre POX et P4 a aussi révélé le manque du mécanisme lié au temps réel et des imprécisions sur les indicateurs de la taille de files d'attente dans la version actuelle de P4.

Mots clés : SDN (Software Defined Networking), QoS (Quality of service), delay management, Mininet, P4, POX.

1.Introduction

With the development of the Internet and the emergence of new networking applications, different operations and treatments are required for better transporting application flows that share the same network but under different QoS (Quality of Service) constraints. Addressing these requirements needs accordingly network QoS mechanisms. Having predictable quality for a network service so that the needs of network operators of the service are met, is the aim of QoS [ITU-T08]. However, traditional networking architectures have not been established enough to solve these QoS issues [Tomovic15]. In fact, the traditional networks are actually complex and difficult to manage because of their control and data transport functions running inside the network specific devices, network operators have to configure each networking devices individually to express the desired network QoS policies, this means the networking configurations are still static, each modification needs amount of reconfiguration effort. For this reason, more and more current technological advancements are re-examining the traditional network architectures from a QoS perspective, a new networking architecture, which enables dynamic networking configuration, needs to be proposed to satisfy their requirements. Software Defined Networking (SDN) is a new emerging and promising architecture in recent years, it decouples the control and data planes in order to simplify the management of the network. SDN brings two essential advantages [ONF14-1]: First, SDN has a tendency to simplify network management by means of its programmability, so various operations can be conveniently automated; Second, the centralized control logic in SDN allows for a more efficient resource utilization in comparison with traditional network architecture, due to its global overview of the network resources.

Although SDN provides these advantages, the QoS management for network operators remain a challenging task [Murat17]. This is because SDN does not fully provide QoS policies and management mechanisms to define constraints, especially on an end-to-end network (e.g., lack of time-aware scheduling).

In this context, the research question for this internship is formulated as follows:

How can QoS management using SDN be applied? In particular, how can QoS

management be implemented using different SDN platforms? To answer the questions stated above, the research in this internship aims to leverage the benefits brought by SDN to improve QoS management for network services. As the most QoS demanding applications are real-time ones, in this respect, this internship work will mainly focus on investigating the different approaches of delay management using SDN.

The rest of the report is organized as follows: Chapter 2 gives an overview of SDN, QoS, and their relation by listing the current solutions and remaining issues. Furthermore, the contexts of POX controller and P4 language are introduced. In Chapter 3, we analyse the main problem to solve and introduce the scenario to consider. Chapter 4 presents different approaches that we have implemented. Chapter 5 presents and discusses the results. Chapter 6 wraps the report up with concluding remarks.

2.Context and the state of the art

2.1. Context of SDN and OpenFlow

Compared with the traditional networks, SDN enables network network operators to treat flows in a finer-granular way employing controllers. As mentioned in the previous chapter, SDN introduces a new architecture which decouples the control and data planes, so that the control plan logic can be centralized as SDN controller and the data plane becomes simple forwarding devices [Bruno14]. Such network architecture, as shown in Fig. 2.1, generally includes three planes: data plane, control plane and application plane [ONF14-2].

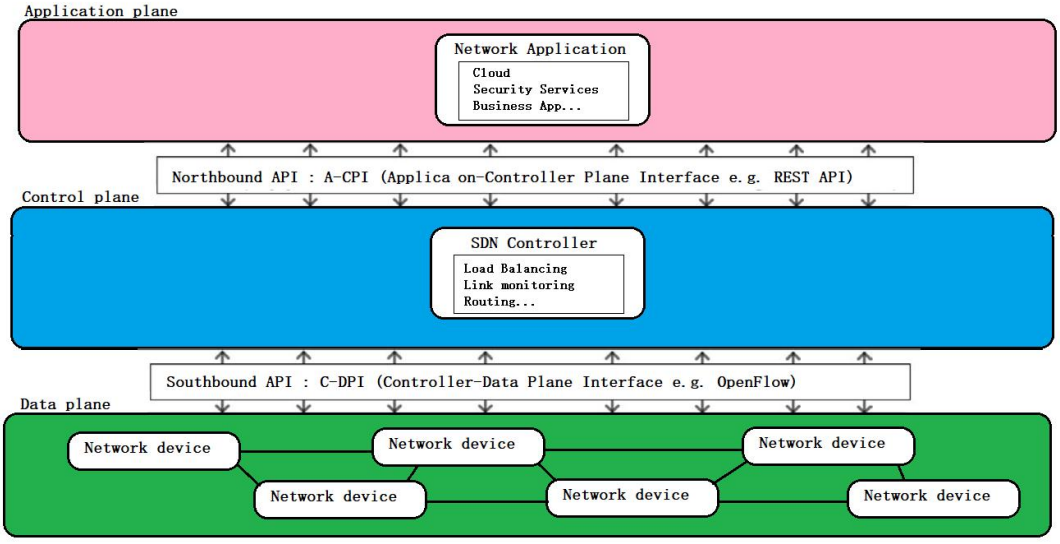


Fig. 2.1: SDN architecture

The *Data plane* is a range of physical networking devices such as routers and switches. These devices consist of highly efficient and programmable packet forwarding devices without any software to take autonomous decisions. On the other words, traffic's forwarding absolutely depends on the decision that the control plane makes.

The *Control plane* can be represented by a single entity: the SDN controller (SDNC). SDNC is able to execute directly its control over the Data plane. Such centralized logic entity is capable of providing some specific functionalities to simplify the network user's operation, e.g., network monitoring [ONF14-2]. A centralized controller must manage all the forwarding devices in the network and implements all control plane logic in a single location. Table 2.1 shows a brief overview of the implementation of current controllers.

Controller	Implementation	Open Source	Developer	Overview
POX	Python	Yes	Nicira	General, open-source SDN controller written in Python.
NOX	Python/C++	Yes	Nicira	The first OpenFlow controller written in Python and C++.
MUL	C	Yes	Kulcloud	OpenFlow controller that has a C-based multi-threaded infrastructure at its core. It supports a multi-level north-bound interface (see Section III-E) for application development.
Maestro	Java	Yes	Rice University	A network operating system based on Java; it provides interfaces for implementing modular network control applications and for them to access and modify network state.
Trema	Ruby/C	Yes	NEC	A framework for developing OpenFlow controllers written in Ruby and C.
Beacon	Java	Yes	Stanford	A cross-platform, modular, Java-based OpenFlow controller that supports event-based and threaded operations.
Jaxon	Java	Yes	Independent Developers	a Java-based OpenFlow controller based on NOX.
Helios	C	No	NEC	An extensible C-based OpenFlow controller that provides a programmatic shell for performing integrated experiments.
Floodlight	Java	Yes	BigSwitch	A Java-based OpenFlow controller (supports v1.3), based on the Beacon implementation, that works with physical- and virtual- OpenFlow switches.
SNAC	C++	No	Nicira	An OpenFlow controller based on NOX-0.4, which uses a web-based, user-friendly policy manager to manage the network, configure devices, and monitor events.
Ryu	Python	Yes	NTT, OSRG group	An SDN operating system that aims to provide logically centralized control and APIs to create new network management and control applications. Ryu fully supports OpenFlow v1.0, v1.2, v1.3, and the Nicira Extensions.
NodeFlow	JavaScript	Yes	Independent Developers	An OpenFlow controller written in JavaScript for Node.JS
ovs-controller	C	Yes	Independent Developers	A simple OpenFlow controller reference implementation with Open vSwitch for managing any number of remote switches through the OpenFlow protocol; as a result the switches function as L2 MAC-learning switches or hubs.
Flowvisor	C	Yes	Stanford/Nicira	Special purpose controller implementation.
RouteFlow	C++	Yes	CPqD	Special purpose controller implementation.

Table 2.1: Current controllers [Bruno14]

The *Application plane* can be described as the network brain. It executes the control functions that will be translated by the controller into commands to be installed in the data plane. It includes applications that allow network operators to develop their high-level policies of network. SDN applications can be grouped in to 5 different categories [Diego15]: traffic engineering, mobility and wireless, measurement, security, and data center networking.

For enabling information exchange between these three planes, *NorthBound API* and *SouthBound API* are defined between SDNC/applications and SDNC/devices (see Fig. 2.1).

The *SouthBound API* defines how the control and data planes can communicate and it also allows the SDNC instructs networking devices by the decisions of SDNC. OpenFlow [Nick08]

is the most popular protocol promoted by the Open Networking Foundation (ONF) [ONF] and it is also the most used protocol in the *SouthBound API*. An OpenFlow switch has one or more flow routing tables consisting of flow entries, flow tables can define how the packet will be processed depending on its flow entries . The controller can control traffic paths in the network by updating the flows entries from the flow tables of the switches.

The most important elements of flow entries consist of:

- *Matching rules/match*: set of rules used to match incoming packets. Matching starts at the first flow table and may continue to additional flow tables of the pipeline. It can be performed either on the packet header fields (e.g., Ethernet source address and IPv4 destination address) or against the ingress port, the metadata field, and other fields.

- *Actions*: an action defines how to handle a matching packet. Three basic actions are: forward this flow's packets to a given port, encapsulate and forward this flow's packets to a controller, and drop the packet.

For example, Table 2.2 shows a very simple match-action table for a routing policy that forwards packets based on destination IP address.

Match	Action
dstip=10.0.0.1/24	*fwd(p=3)*
dstip=10.0.0.2/24	*fwd(p=4)*
*	*drop*

Table 2.2: Example flow table for a routing policy

The *NorthBound API* connects the *Control plane* and the *Application plane* and defines the means of their communication. Its role consists in providing a high-level API between applications plane and the network infrastructure. Conversely, from the *SouthBound API*, which has OpenFlow as most used open source protocol, *NorthBound API* lacks such protocol standards. Defining a common *NorthBound API* is a critical task as the requirement of each

networking application can vary [Manish18].

2.2. Context of POX controller with Mininet

2.2.1.Mininet

Mininet is a Linux based emulation tool which creates a network of virtual hosts, switches, controllers, and links [Lantz10] . It uses container-based virtualization to make a single system behave and act as a complete network. Thus, Mininet is a robust and inexpensive tool to develop and test OpenFlow-based applications by running a simple or complex network topology without configuring real physical networks.

2.2.2. POX controller

POX is a Python based SDN controller that is inherited from the NOX controller [Gude08], it is also an open source controller [Pox] to implement the OpenFlow protocol for developing SDN-based functionalities and applications, such as load balancing, firewall, learning switch, etc. As shown in Fig. 2.2, controller inserts flow entries into flow table of switch to handle the incoming packets. Once these flow tables are defined, traffic belonging to flow table will be handled by the switches themselves without further intervention from the controller.

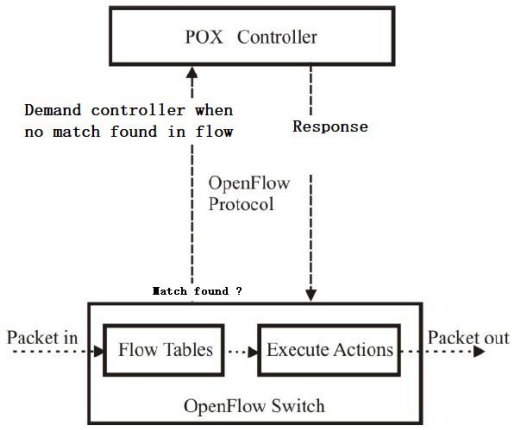


Fig. 2.2: POX Controller on SDN architecture [POX3]

2.2.3. Example: L2 learning switch using POX

This example shows a simple network using POX controller to control the behaviors of a L2 switch of IEEE802.1, the complete code can be found in appendix A. The POX controller would apply a L2 learning function [l2 learning] that allows the switch to updates automatically its flow table: it populates the table by observing traffic. When a packet from some source is coming from some port, one thing should be determined that source is out that port. Therefore, all hosts in the network applying L2 learning function can be discovered by ‘pingall’ command. The topology is illustrated in Fig. 2.3.

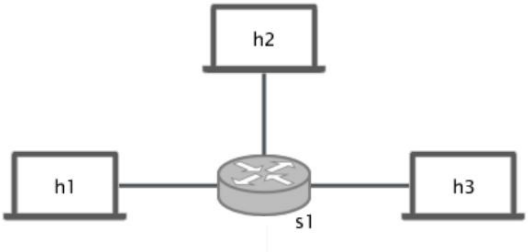


Fig. 2.3: Topology- POX

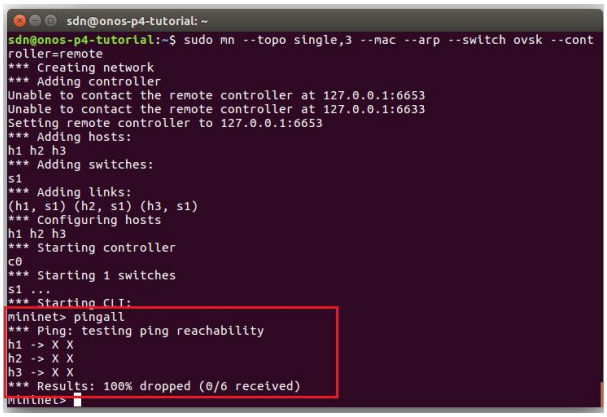


Fig. 2.4: Controller unconnected

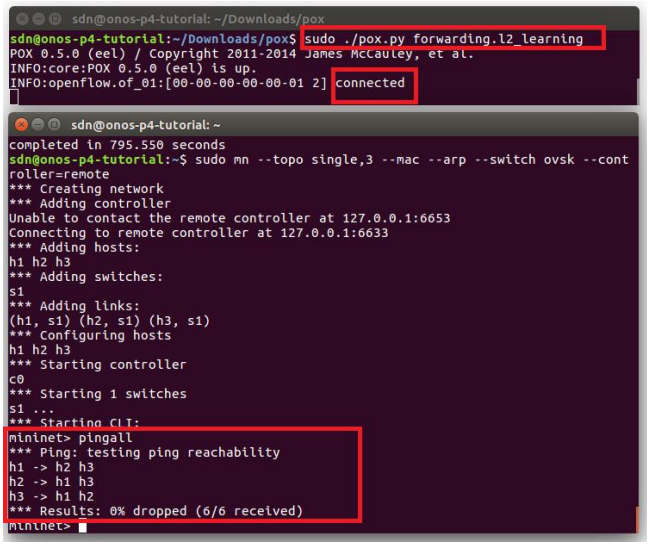


Fig. 2.5: Controller connected

From Fig. 2.4, the command 'sudo mn --topo single,3 --mac --arp --switch ovsk --contorller=remote' allows to run Mininet within a network emulation environment to emulate 1 switch with 3 hosts. In above command, there are some important keywords worth paying attention to: --mac: auto set MAC addresses; --arp: populate static ARP (Address Resolution Protocol) entries of each host in each other; --switch ovsk: ovsk refers to the Open vSwitch type of switch; --controller=remote: starting up a switch using a remote controller

The switch tries to connect to the controller at 127.0.0.1:6633, but there is no controller working cause no POX controller has been launched. Therefore, the 'pingall' command returns 100% packet dropped and 0 packet received result.

From Fig. 2.5, the controller of L2 learning switch has been launched and thus it can be connected to by the switch. Since the L2 learning function controller starts to work, the the 'pingall' command will discover all hosts with 0% packet dropped result. In this example, the Python based controller program is the original version of L2 learning function controller [Pox-L2] .

2.3. Context of P4 language

2.3.1. P4 overview

P4 (Programming Protocol-independent Packet Processors) is a programming language designed to allow programming of packet forwarding planes [Bosshart14]. Compared to the general programming language such as C or Python, P4 is a domain-specific language with a number of constructs optimized around network data forwarding. P4 is an open-source language and is maintained by a non-profit organization called the P4 Language Consortium [P4]. Fig.2.6 shows the relationship between P4 and existing APIs (such as OpenFlow) that are designed to populate the flow tables into switches.

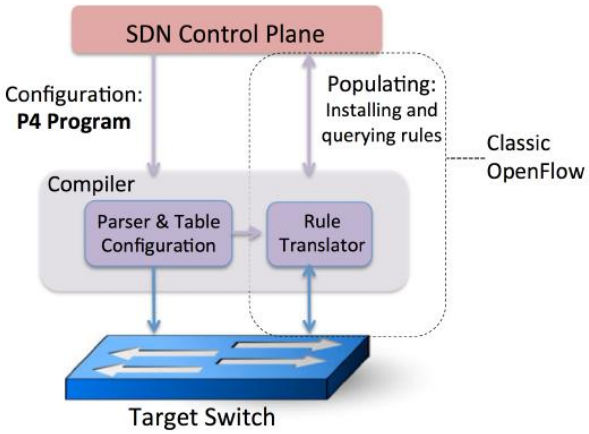


Fig. 2.6: Switch configuring language [P4]

2.3.2. P4 processing pipeline

A P4 program defines a pipeline for packet processing which is illustrated in Fig.2.7. It is

structured into 4 main parts: the parser, the ingress pipeline, the egress pipeline, and the deparser.

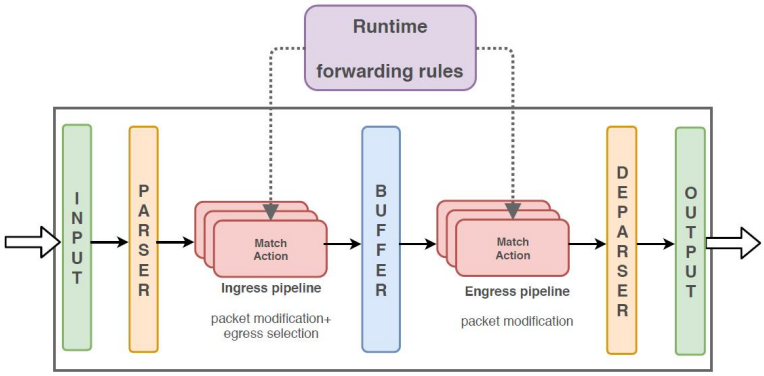


Fig. 2.7: P4 processing pipeline [Menth19]

The parser part is able to read and store packet header fields. These fields are carried through the entire processing with packets and standard metadata, e.g., the numbered ports. In addition, self-defined metadata can be stored as well during processing, e.g., time interval between two packets' arriving. The ingress and egress pipelines can modify, add or remove header fields, or even perform many more actions useful for flexible packet processing. Packets may be even processed several times by the ingress or egress pipeline. The ingress pipeline typically determines the output port for a packet. After completion of the egress pipeline, packets will be deparsed, this means their headers are assembled and then sent. In short: Parser recognizes and extracts fields from the header, these fields are then passed to the match+action tables which are divided between ingress and egress processing according to their own functionality. The deparser would assemble headers after all processing then send them.

2.3.3. Example: L2 learning switch using P4

This example presents the implementation of a L2 learning switch. Same to L2 learning function using POX controller, the P4 program would apply a L2 learning function[l2 learning] that allows the switch to updates automatically its flow table.

Now take a closer look at the P4 program, especially the *action* mac_learn: for every packet the switch receives, it checks if it has seen the src_mac address before. If its a new mac address, it sends to the controller a tuple with (mac_address, ingress_port). The controller receives the packet and adds two rules into the switch's tables. First it tells the switch that src_mac is known. Then, in another table it adds an entry to map the mac address to a port. The switch also checks if the dst_mac is known (using a normal forwarding table), if known the switch forwards the packet normally, otherwise it broadcasts it.

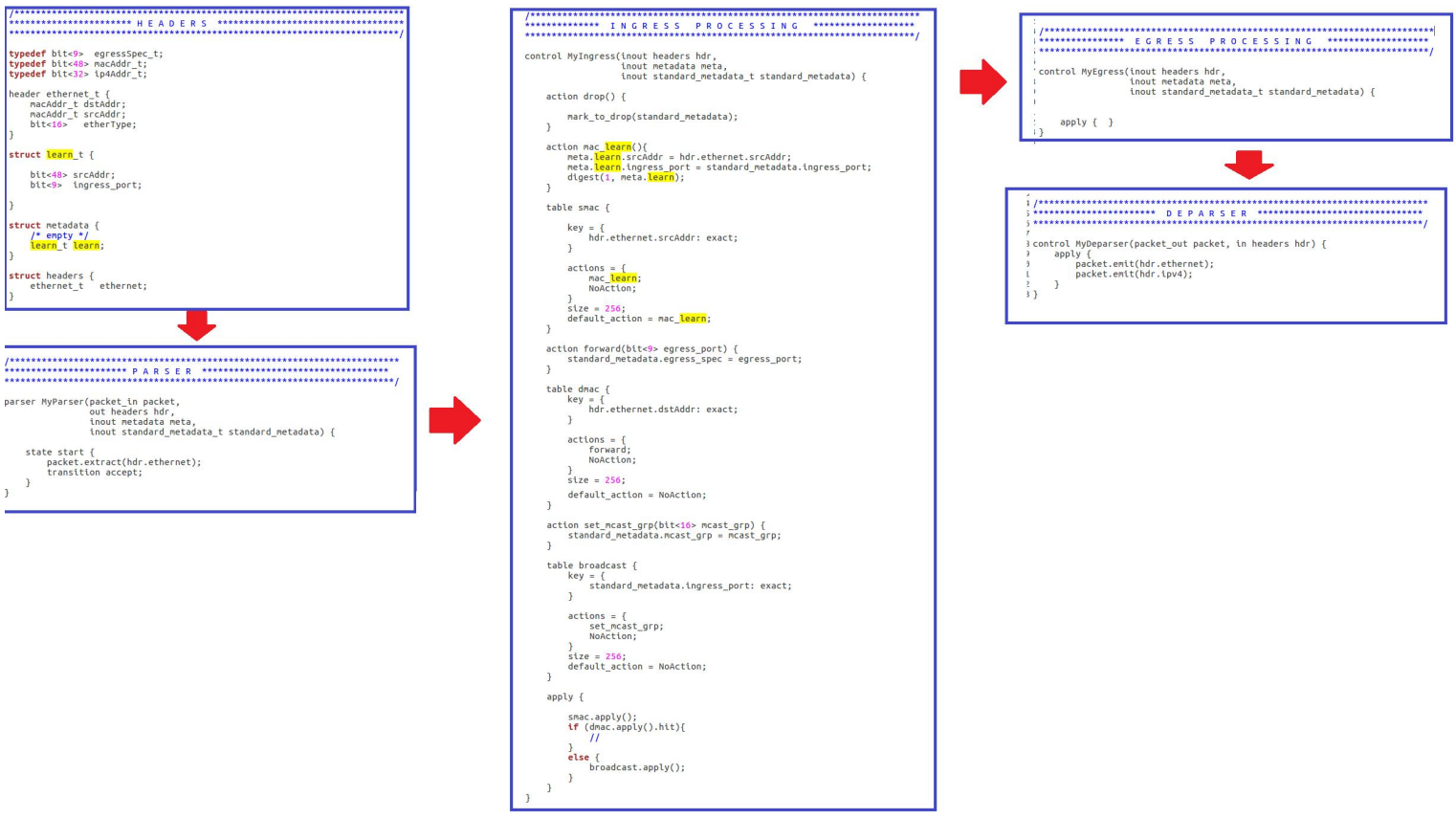
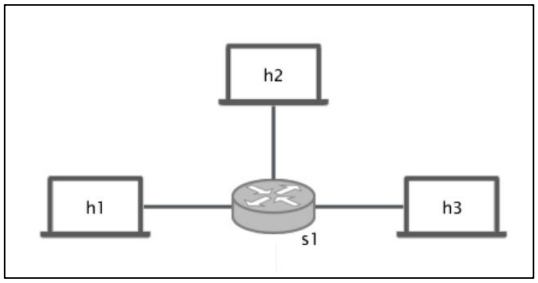


Fig. 2.8: Main L2 learning P4 program




```
"topology": {
  "assignment_strategy": "l2",
  "auto_arp_tables": false,
  "links": [["h1", "s1"], ["h2", "s1"], ["h3", "s1"]],
  "hosts": {
    "h1": {
    },
    "h2": {
    }
    ,
    "h3": {
    }
  },
  "switches": {
    "s1": {"cpu_port" : true
    }
  }
```

Fig. 2.9: Topology - P4

Fig. 2.9 shows the topology which is the same as the L2 learning switch using POX controller example, and the main parts of its description file named ‘topology.json’. At the end of thie file, ‘cpu_port’ is used to tell P4 program to add an extra port to connect with the controller. the complete code of controller can be found in appendix B.

```
sdn@onos-p4-tutorial: ~/Downloads/p4-learning-master/exercises/04-L2_Learning
sdn@onos-p4-tutorial:~/Downloads/p4-learning-master/exercises/04-L2_Learning$ sudo p4run --conf topology.json
*** Removing excess controllers/ofprotocols/ofdatapaths/pings/noxes
killall controller ofprotocol ofdatapath ping nox_corelt-nox_core ovs-openflowd ovs-controllerovs-testcontrolle
r udpbwtest mnexec ivs ryu-manager 2> /dev/null
killall -9 controller ofprotocol ofdatapath ping nox_corelt-nox_core ovs-openflowd ovs-controllerovs-testcontro
ller udpbwtest mnexec ivs ryu-manager 2> /dev/null
pkill -9 -f "sudo mnexec"
*** Removing junk from /tmp
rm -f /tmp/vconn* /tmp/vlogs* /tmp/*.out /tmp/*.log
```

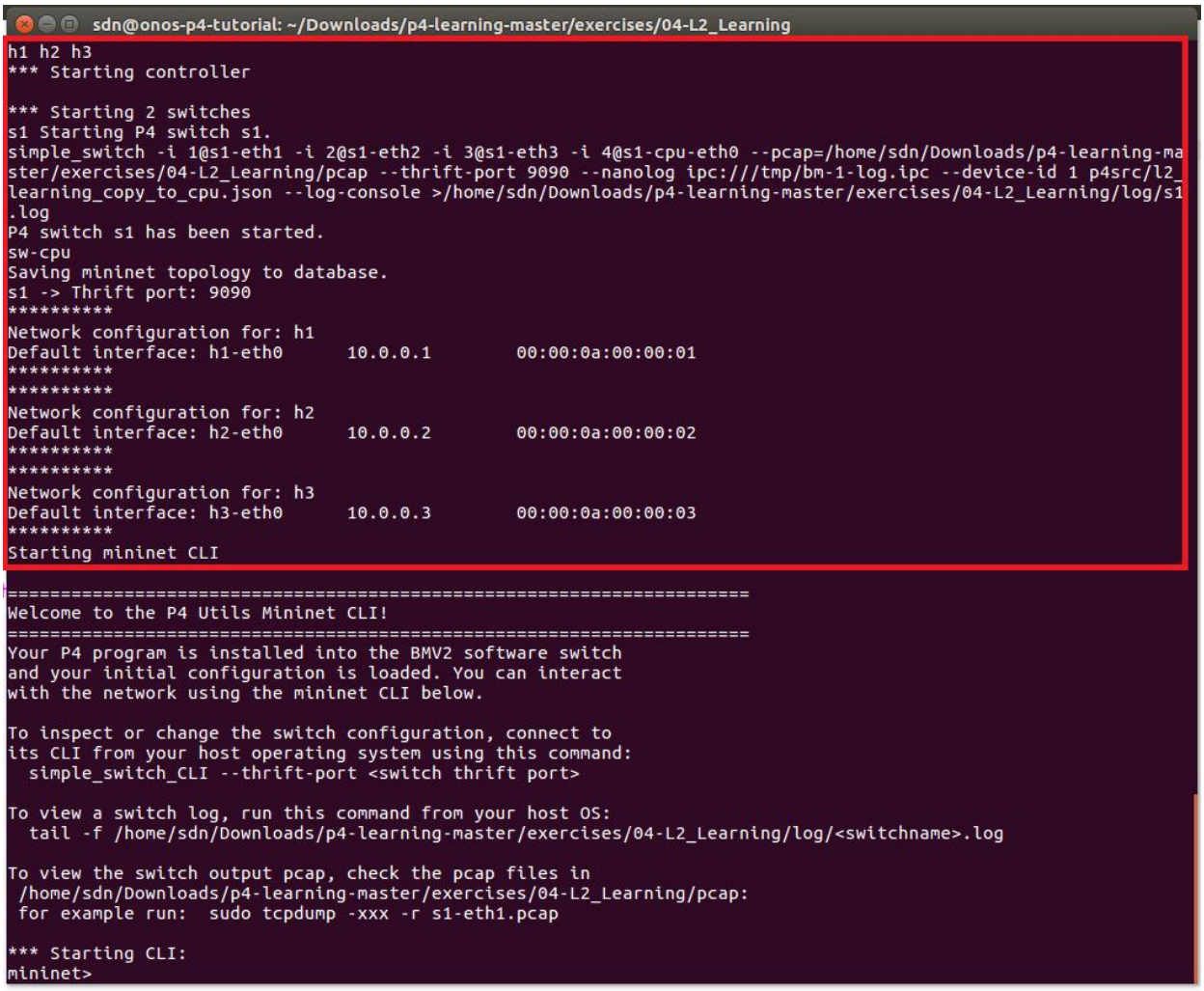


Fig. 2.10: Start the topology

Fig. 2.10 starts the topology by running the command: 'sudo p4run --conf topology.json'. This command is used to initialize the configured network using Mininet.

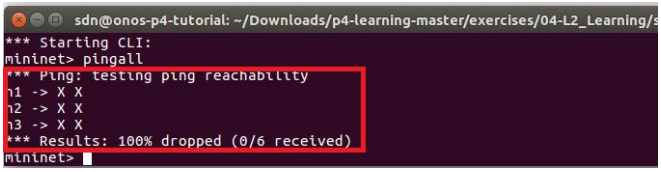


Fig. 2.11: No controller case

As shown in Fig. 2.11, there is no controller working cause no POX controller has been launched, so that the 'pingall' command returns 100% packet dropped and 0 packet received result.

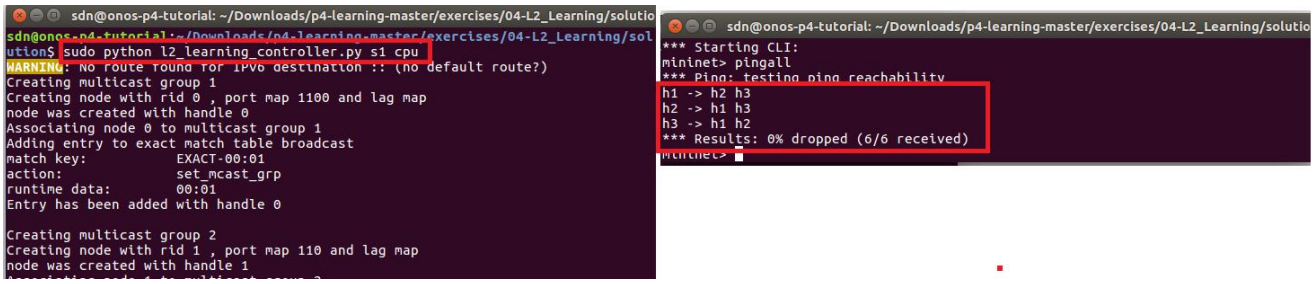


Fig. 2.12: No controller case

As shown in Fig. 2.12, L2 learning function controller is launched by running 'sudo python l2_learning_controller.py s1 cpu'. The controller has to be told from which switch listen from and the 'cpu' parameter tells the controller which port it has to connect to. Since the L2 learning function controller starts to work, the 'pingall' command will surely discover all hosts with 0% packet dropped results.

2.4. A simple comparison between POX and P4

Name	Implementation Language	Open source	Developer	Overview
POX	Python	Yes	Nicira	One of the most used open-source SDN controller written in Python.
P4	Python/C/JavaScript	Yes	Stanford University	A programming language for controlling packet forwarding planes.

Table 2.3: Comparison between POX and P4

As shown in Table 2.3, both POX and P4 are open-source. The largest difference between them is, POX is an SDN controller written in Python and P4 is not an SDN controller. P4 is a programming language for controlling packet forwarding plane, which means the work of treating data plane will be much easier while programming the SDN controller. An obvious example is L2 learning function: the controller file in Python has nearly 250 lines [Pox-L2] using POX controller but P4's controller file has only about 130 lines [P4-L2]. Moreover, P4 language can implement headers and metadata and it has a clearer view of match-action, this is

also why P4 is able to simplify the controller programming. As the old saying goes every coin has two sides, although P4 simplifies the controller programming, it still requires a hard work to write the P4 program to specify useful headers fields or metadata fields such as Fig.2.8 shows. The comparison between these two tools with more technical aspects will be conducted in Chapter 4.

2.5. State of the art on current QoS management using SDN

2.5.1. General definition of QoS

Modern networking mechanism supports traffic beyond the traditional data types, such as email, information sharing. Increasingly, data networks share a common medium with more sensitive forms of traffic, like voice and video. These sensitive traffic types often require specific guaranteed or regulated service. QoS refers to the capability of a network to provide such service to selected network traffic. To quantitatively measure QoS, several related aspects of networking performance (NP) are often considered, such as:

Bandwidth - Describes the the maximum rate of data transfer across a given path.
Jitter - Measures the fragmentation that occurs when traffic arrives at irregular times or in the wrong order.
Data Loss - Defines the packet loss that occurs due to link congestion. A full queue will drop newly-arriving packets - an effect known as tail drop.

Delay - Defines the latency that occurs when traffic is sent end-to-end across a network. Delay will occur at various points on a network, and will be discussed in greater detail shortly. There exist various types of delay such as propagation delay, forwarding delay, QoS management in this internship focuses on the queuing delay.

2.5.2. Classification of QoS methodologies

There are three key methodologies for implementing QoS:

- **Best-Effort**
- **Integrated Services (IntServ)**
- **Differentiated Services (DiffServ)**

Best-Effort QoS is essentially **no** QoS. Traffic is routed on a first-come, first-served basis. Sensitive traffic is treated no differently than normal traffic. Best-Effort is the default behavior of routers and switches, and as such is easy to implement and very scalable. The Internet forwards traffic on a Best-Effort basis.

IntServ QoS is also known as end-to-end or hard QoS. IntServ QoS requires an application to signal that it requires a specific level of service. An Admission Control protocol responds to this request by allocating or reserving resources end-to-end for the application. If resources cannot be allocated for a particular request, then it is denied. Every device end-to-end must support the IntServ QoS protocols. IntServ QoS is not considered as a scalable solution for two reasons: (1)There is only a finite amount of bandwidth available to be reserved. (2)IntServ QoS protocols add significant overhead on devices end-to-end, as each traffic flow must be statefully maintained. In this sense IntServ provides per-flow QoS but suffers from the scalability issue.

DiffServ QoS was designed to be a scalable QoS solution. Traffic types are organized into specific classes, and then marked to identify their classification. Policies are then created on a

per-hop basis to provide a specific level of service to the traffic classes, depending on the traffic's classification.

DiffServ QoS is popular due to its scalability and flexibility in enterprise environments. However, DiffServ QoS is considered as soft QoS, as it does not provide per-flow guaranteed service, like IntServ QoS. This means that different application requirements classified into a same traffic class will get a same QoS level. Moreover, without end-to-end resource reservation, it can happen that the QoS provided by one hop (network node) will be compromised by another hop in an end-to-end path. DiffServ QoS does not employ signaling, and does not enforce end-to-end reservations.

2.5.3. Delay

2.5.3.1. Types of Delay

Delay can occur at any networking points, the 4 main types of Delay are:

- **Propagation Delay** refers to the time necessary for a single bit to travel end-to-end on a physical wire or radio channel. It can be computed as the ratio between the link length and the propagation speed over the specific medium (often approximated as 2/3 of the light wave speed).

- **Processing Delay** refers to the time necessary for a router or switch to move a packet between an ingress (input) queue and an egress (output) queue. Forwarding delay is affected by a variety of factors, such as the routing or switching method used, the speed of the device's CPU, or the size of the routing table, the time it takes to process the packet header. In most of commercially available switches and routers, the maximum processing delay is given as a constant value. It also called legacy latency.

- **Transmission Delay** is the amount of time required to push all the packet's bits into the wire. In other words, this is the delay caused by the data-rate of the link. Therefore, Transmission

Delay is a function of the packet length and has nothing to do with the distance between the two nodes. This delay is proportional to the packet length in bits.

- **Queuing Delay** refers to the time spent in an egress queue (most of existing switches are full line-speed, meaning that the cpu can process all the input ports within the maximum packet interarrival interval, so there is never queues at any input port. This allows to avoid the well-known "Head of line blocking" issue.), waiting for previously-queued packets to be serialized onto the wire. Output port queues that are too small can become congested, and start dropping newly arriving packets. This forces a higher-layer protocol (such as TCP) to resend data. Queues that are too large can actually queue too many packets, causing long queuing delays.

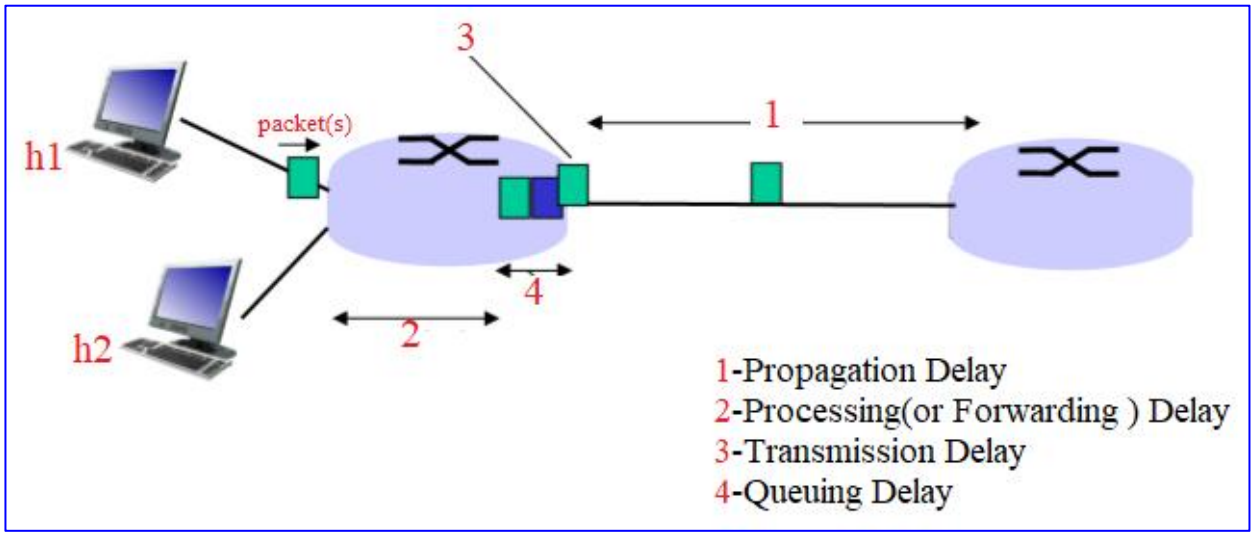


Fig. 2.13: Overview of 4 main types of Delay

Perceived QoS can be measured via a few key metrics, such as packet loss rate, throughput, delay and jitters. Various QoS provisioning schemes have been used to control the delivery delay from source to destination in order to fulfill the QoS requirements of the applications. Delay is considered to be one of the vital measures of QoS and it can be sub-divided into queuing delay, processing delay, transmission delay and propagation delay. **Of these delays, only the queuing delay is variable depending on the network load and is controllable. This is why the queuing delay has to be focused on as the key research point in this internship.**

2.5.3.3. Other QoS mechanisms- TSN

(1) Overview

IEEE 802.1 TSN (Time-Sensitive Networking) technology is one of the most important evolution of Ethernet switching for industrial Internet of things. To address the TSN timing constraints which can be used to ameliorate the delay management, ‘shapers’ and ‘scheduling’ have been introduced and standardized, they are principally: IEEE 802.1Qav Forwarding and Queuing Enhancements for Time-Sensitive Streams, IEEE 802.1Qbv Enhancements to Traffic Scheduling: Time-Aware Shaper (TAS). As one new key feature of TSN, TAS is supposed to be capable of accommodating hard real-time streams with deterministic end-to-end delays. TAS uses a pre-determined scheduling, which guarantees timely transmission of CDT (Control Data Traffic) within the scheduled time-slots. Besides, a guard-band (GB) is added to check that other traffic would not interfere with CDT traffic before each CDT time slot. TAS should be determined periodically by different time intervals.

(2) Time-Aware Shaper (TAS)

In a more practical word, that frames of TSN should be transmitted depending on their VLAN tag and priority. These frames are filtered by the TSN switch following a schedule described by one TAS configuration cycle, this cycle then includes at least:

A Guard-Band (GB), which s a period during which all gates would be closed. This period allows a frame that has been started to finish sending. For this reason, the duration of the Guard-Band is based on the time it takes to transmit the maximum-size frame exiting in the network.

The Control Data Traffic (CDT) slot, which is the period of time that the particular priority frame can be transmitted (e.g., priority 7 has been chosen as CDT priority). The CDT guarantees that periodical data has their own slot reserved on each cycle.

The rest of the cycle is used to pass the rest of the data, which has lower priorities (e.g.,

priorities between 0 to 6).

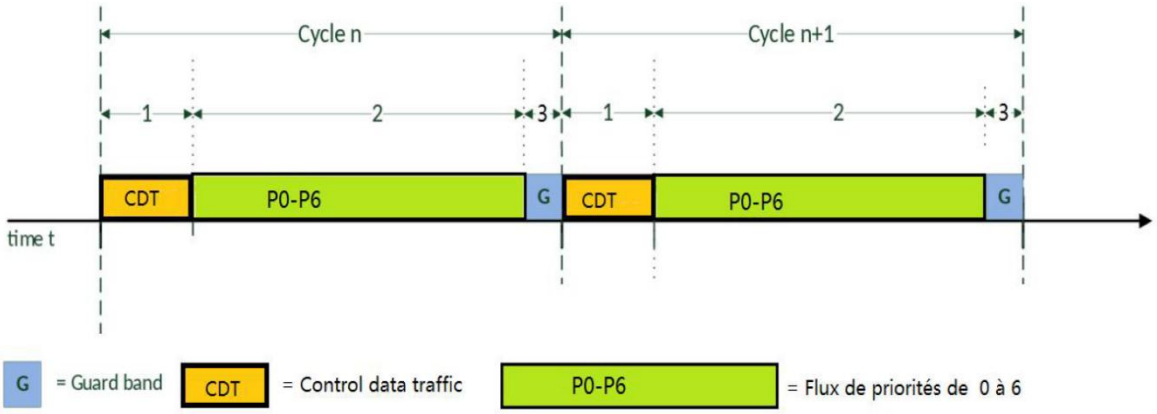


Fig. 2.12: Explanatory diagram of TAS configuration cycle

The time slot of CDT and the rest of cycle are able to be reserved for priority-based flows，e.g., the CDT for P7 and the rest of cycle for P0-P6, this means the priority-based flows' transmission through the switch can be deterministic with a very low delay, that is one of most important mechanisms for the time-sensitive transmission over TSN networks. TAS is applicable for low delay requirements according to the time slot configuration but needs to have all time-triggered windows synchronized (Time synchronization).

2.5.4. QoS using SDN

SDN enables separation of data and control planes, this enhances the SDNC with regard to control of the networks. Controllers can obtain a whole global network view and states, e.g. link states, statistics of flows, resource availability by sampling of packets. Using these information, networking policies/rules can be specified by network operators without reconfiguration thanks to the programmability of SDN. Thus, Networking policies/rules can be defined per flow and SDNC is able to apply these rules to Data plane, in other words, to the different devices. All these mechanisms make that QoS can benefit from SDN concept in different network functions [Raphael14].

As seen in Fig. 2.2 the related studies of QoS management using SDN have been organized into seven categories.

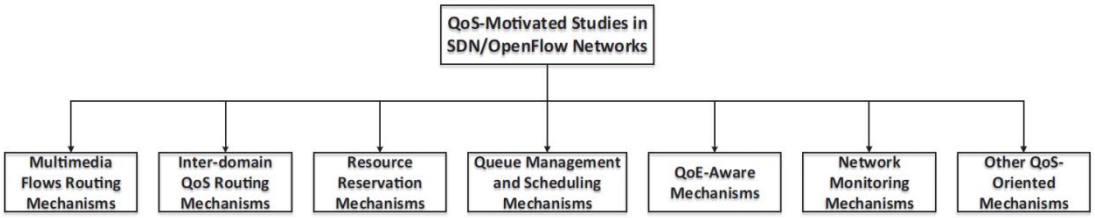


Fig. 2.2: The Organization of QoS-based studies using SDN [Murat17].

- **Multimedia flows routing mechanisms**: The routing mechanism for multimedia applications to address their QoS management requirements.

- **Inter-domain QoS routing mechanisms**: QoS flows routing between networks which have a distributed control plane employing multiple SDN controllers so that each single part is responsible for a its own part of network.

- **Resource reservation mechanisms**: This type of mechanism exploits flow classification and a rate-shaping through some implemented modules in SDN controller.

- **Queue management and scheduling mechanisms**: It establishes more complicated SDN services requiring complete control or management of the data plane in terms of configurations of ports, queues, etc.

- **QoE-aware mechanisms**: Main related work is to dynamically adapt QoE demands of the network operators to QoS parameters in the network. (*Quality of Experience (QoE) is a measure of the delight or annoyance of a customer's experiences with a service- Wikipédia*)

- **Network monitoring mechanisms**: It helps gather, process and deliver monitored data at requested aggregation levels from network components.

- **Other QoS-oriented mechanisms**: By taking advantage of SDN concept, studies have been

conducted in many broad areas such as **Virtualization-based QoS providing**, **QoS policy management**, **content delivery mechanisms**, etc.

Related QoS using SDN studies and their implementations illustrated in Table 2.4, this table also indicates the QoS parameters that these studies focus on and the main technologies utilized. From this table, it is apparent that QoS using SDN has been considerably advanced because of more attention payed from both industry and academia. However, some studies merely propose theoretical solutions or concepts instead of providing concrete implementations benefiting enough the advantages brought from SDN architecture. Hence there are still enormous challenges on QoS using SDN research.

Mechanism	Studies related	QoS	IntServ/DiffServ	Techniques
Multimedia flows routing mechanisms	[MM1]	B/D/L	N/Y	A QoS-enabled routing architecture for scalable video streaming
	[MM2]	B/D	N/Y	Design of HiQoS application for multi path routing and queueing mechanisms
	[MM3]	J/L	N/Y	A controller design, OpenQoS", for QoS-enabled routing of multimedia traffic delivery
	[MM4]	B/D/J	Y/N	A QoS-enabled (reliable) routing architecture (R-VSDN) for video streaming
	[MM5]			
	[MM6]	B	Y/N	A QoS routing framework to provide resource-guaranteed paths for multimedia applications
	[MM7]	B/D/L	N/Y	A QoS-enabled dynamic optimization-based routing architecture for scalable video streaming
	[MM8]	B/D/J/L		
	[MM9]	J/L	N/Y	
	[MM10]	J/L	N/Y	Server load balancing application that reroutes flows of video streams
Inter-domain QoS routing mechanisms	[ID 1]	J/L	N/Y	A distributed QoS routing architecture for scalable video streaming over multidomain OpenFlow networks
	[ID 2]	B/D/J/L		
	[ID 3]	B/D	Y/N	A hierarchic network architecture with an inter-AS QoS routing approach
	[ID 4]	B/D/L	Y/Y	Design of Broker-based FlowBroker architecture for QoS support
	[ID 5]			
	[ID 6]	B/D	Y/N	Design of MCTEQ model proposing a joint bandwidth allocation for trafffic classes
	[ID 7]			
	[ID 8]	D/L	Y/Y	Use of SDN and OPS nodes for QoS support
	[ID 9]	B/D	Y/Y	Design of Control Exchange Points (CXPs)" for QoS routing among ISPs
Resource reservation mechanisms	[RS 1]	B/D/L	Y/N	A network QoS control framework for management of converged network fabrics
	[RS 2]	B/D/J	Y/N	A Network Control Layer (NCL) based on SDN, OpenFlow, and NaaS for QoS requirements of applications
	[RS 3]	B/D	Y/Y	A framework to apply NaaS in SDN/OpenFlow networks to enable network service orchestration for supporting inter-domain end-to-end QoS
	[RS 4]			
	[RS 5]	B/D	Y/Y	A system, FlowQoS, enabling users to specify high-level application flow prioritization (e.g. VoIP etc.)
	[RS 6]			
	[RS 7]	B	N/Y	A QoS provisioning mechanisms for elephant flows
	[RS 8]			
Queue management and scheduling mechanisms	[QmS 1]	B	N/Y	A QoS control framework (QoSFlow) using multiple packet schedulers
	[QmS 2]	B	Y/Y	A QoS-motivated SDN architecture (OpenQFlow) for scalable and stateful SDN/OpenFlow networks
	[QmS 3]	B/D	N/Y	ToS/DSCP-based classification approach for QoS
	[QmS 4]	B/D/J		
	[QmS 5]	B/D	Y/Y	A hierarchical autonomic QoS model by adopting SDN
	[QmS 6]			
	[QmS 7]	B/D	Y/Y	A QoS configuration API using OVSDB protocol
QoE-aware mechanisms	[QoE 1]	B/D/J/L	N/Y	A QoE-Aware IPTV network architecture design over OpenFlow networks
	[QoE 2]	B	Y/Y	A system to improve user QoE by bandwidth allocation management framework at home networks
	[QoE 3]	B/D/J	Y/Y	
	[QoE 4]	B/D	N/Y	Design of Q-POINT, a QoE-driven path optimization model Georgopoulos
	[QoE 5]	B/D/L	Y/Y	
	[QoE 6]	B	N/Y	An OpenFlow-assisted QoE Fairness Framework (QFF) to maximize the QoE of clients in a shared network
	[QoE 7]	B/D	N/Y	A Northbound API design for online applications to increase QoE of users
	[QoE 8]			
	[QoE 9]	B	Y/Y	A study investigating how different kinds of information such as per-flowparameters, application signatures etc. can improve network management
Network monitoring mechanisms	[Netm 1]	NA	NA	Design and implementation of OpenNetMon monitoring framework
	[Netm 2]	NA	NA	Design and implementation of PayLess monitoring framework
	[Netm 3]	NA	NA	Design and implementation of an interactive network monitoring framework
	[Netm 4]	NA	NA	Design and implementation of traffic measurement framework
	[Netm 5]	NA	NA	Design and implementation of OpenSketch monitoring framework
	[Netm 6]	NA	NA	Design and implementation of OpenTM monitoring framework
	[Netm 7]	NA	NA	Design and implementation of OpenSAFE monitoring framework
Other QoS-related mechanisms	[Other 1]	B/D	N/Y	A language to express QoS requirements of applications when placing virtual network components
	[Other 2]	B	Y/Y	A QoS controller architecture, Q-Ctrl, for programmatically attaining requested QoS constraints by users in an SDN-based cloud infrastructure
	[Other 3]	B/D	N/Y	Design of a QoS policy management framework called PolicyCop
	[Other 4]	B/D	N/Y	A caching mechanism (OpenCache) to store content for VoD services
	[Other 5]			
	[Other 6]	B/D/J	Y/Y	An architectural extension for QoS-enabled experiments in Ofelia using OpenFlow
	[Other 7]	B	N/Y	Design of SoIP architecture showing interoperability of SDN and IP for better QoS
	[Other 8]	B/D	N/Y	Design of ACDPA architecture using SDN and Hadoop for better QoS support
	[Other 9]	NA	NA	Report of 2 years-running SDN network experiments on 3 different testbeds

Table 2.4: Classification of the studies surveyed in QoS using SDN

B: bandwidth / D: delay / J: jitter / L: loss rate

2.4. Challenges

QoS using SDN becomes an extreme hot research topic. Since the emerging applications in the Internet generate diverse flows which require different operations for each one, researchers have started exploiting the SDN paradigm and OpenFlow protocol because they bring centralized global network view, and more fine-granular flow management opportunities in networks. While SDN matures, QoS using SDN deserves more research efforts from both academia and industry. As shown in Table 2.4, a large part of new technologies has been implemented in QoS using SDN in order to improve the networking performance. However, QoS using SDN has still some challenging issues, e.g, the architectural challenges such as data plane challenge and control plane challenge [these QoS using SDN], and some particular technical challenges such as the absence of inter-AS QoS Provisioning or QoS signaling overhead [Murat17].

Taking a closer look at the QoS aspect, amount of excellent studies related to QoS using SDN has been conducted, but some of them did not invest enough in efficient solution benefiting the SDN advantages (e.g, QoS-aware rerouting) but propose architectural extension and theoretical optimization in SDN architecture. The delay management especially the queuing delay management using SDN has not been developed to a large extent. As mentioned before, queuing delay is the most controllable among all types of delay, it is also one of most significant QoS parameters. For these reasons, this internship proposes to investigate P4-Mininet and POX-Mininet solutions to see if they can be effectively used to address the delay management problem. These two solutions will be compared from different aspects (e.g, operability and complexity). To the best of our knowledge, there is not previous work on that direction.

3.Measurement design and methodology

3.1. Topology and traffic flows

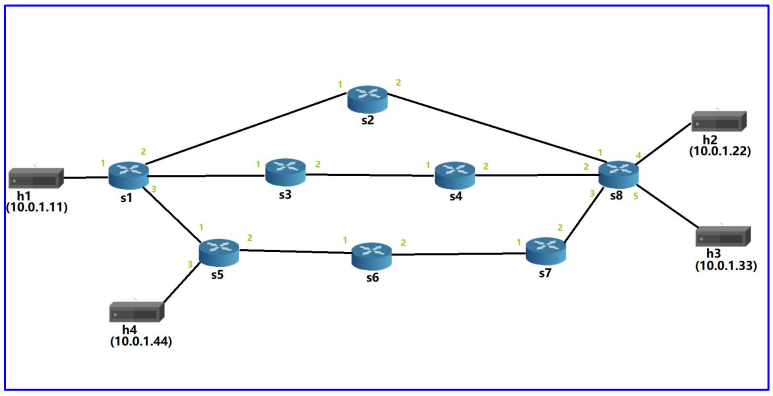


Fig. 3.1: Topology for delay management

The principal topology is illustrated in Fig.3.1. It includes 3 parallel paths:

High path: h1-s1-s2-s8-h2

Middle path: h1-s3-s4-s8-h2

Low path: h1-s5-s6-s7-s8-h2

h1 is the source host to send the packets, h2 and h3 are both destination host to receive the packets. h4 is added to observe the perturbation for networking when it starts to generate flows. The h1-h2 flows via the high path should be generated periodically, the h1-h3 flows will be the QoS-based flows to improve the networking performance. The advantage of such topology is its various numbers of switches setting up in each path, the results for delay management at any path can be easily shown.

3.2. Delay aware Dynamic Rerouting

The SDN centralized architecture allows the controller to update the flow table of network devices and also to carry out a real time QoS parameters monitoring. For this reason, SDN

controller is able to send the rerouting command and to collect the delay values of each path. Therefore, the delay aware dynamic rerouting functionality can be built using SDN.

The goal of delay aware rerouting is to allocate parallel paths according to the measured delay at each path in real time. For example, some paths are supposed to ensure the transmission of specific flows, but they can be in an overload situation if their queuing delays become extremely huge, then we can reroute a part of their current flows to other less occupied paths in order to balance their utilization. This delay management can help the network to involve the less occupied path to be in charge of flows which are initially transmitted at occupied path, so that the occupied path can have more capacity dedicated to the specific flows.

Scenario

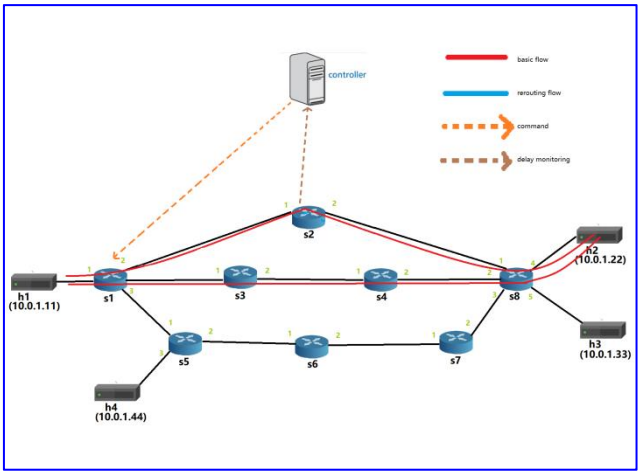


Fig. 3.2: Initial situation

Firstly, the high and low thresholds of each path should be determined. Once the real-time delay is more than the high threshold, part of current flow at this path should be moved to another less occupied path. On the contrary, that part of flow should be returned to this flow if the real-time delay is less than the low threshold. More precisely, in our case, high and middle paths are reserved to transmit h1-h2 flows, this means h1-h3 flows will be rerouted if the current path

exceeds the high threshold of delay. Of course, this rerouting flow will be moved back to high of middle paths if the delay is less than the low threshold. Be careful: the thresholds of each path are independent. The overload situation will be discussed in section 3.4.

(1)(2)-Initial situation:

(1)- h1-h2 flows via the high path will be generated periodically.

(2)- h1-h2 flows via the middle path will be generated periodically.

- First two points are illustrated in Fig.3.2. The red line represents the paths of h1-h2 flows.

(3)- Add rerouting flows: h1-h3 flows via high path will be generated to have the increase of delay at the point of s2

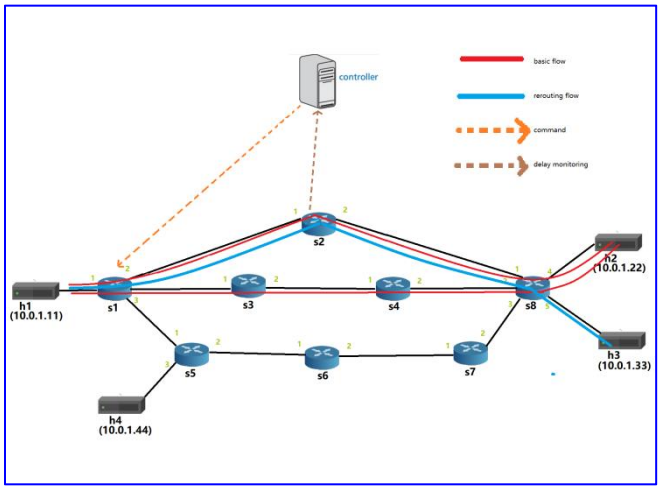


Fig. 3.3: Add rerouting flows

(3) is illustrated in Fig.3.3. The blue line represents the current path of h1-h3 flows, the controller is capable of sending commands to the s1 if **(4)** needs to be activated and receiving the delay occurs in s2 since h1-h3 flows increases the delay of high path.

(4)-High path to middle path: once the delay measured in s2 is more than the high threshold, the re-routing action which moves the flows h1-h3 from high path to middle path will be activated. **(4)** is illustrated in Fig.3.4.

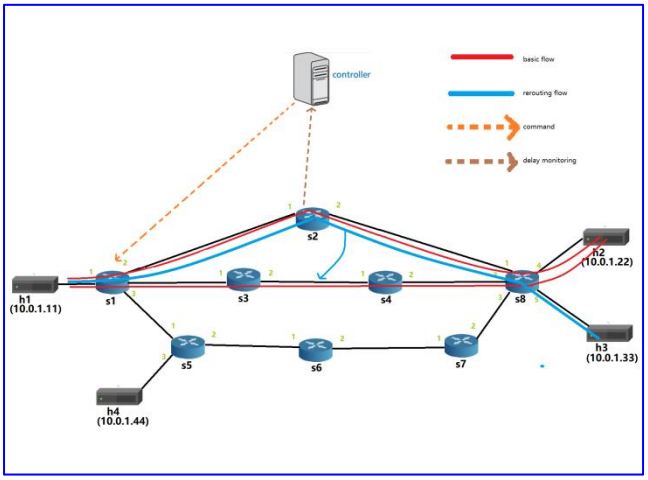


Fig. 3.4: High path to middle path

(5)-Middle path to low path: In a similar way, once the delay measured in s3-s4 is more than the high threshold, the re-routing action which moves the flows h1-h3 from middle path to low path will be activated. **(5)** is illustrated in Fig. 3.5.

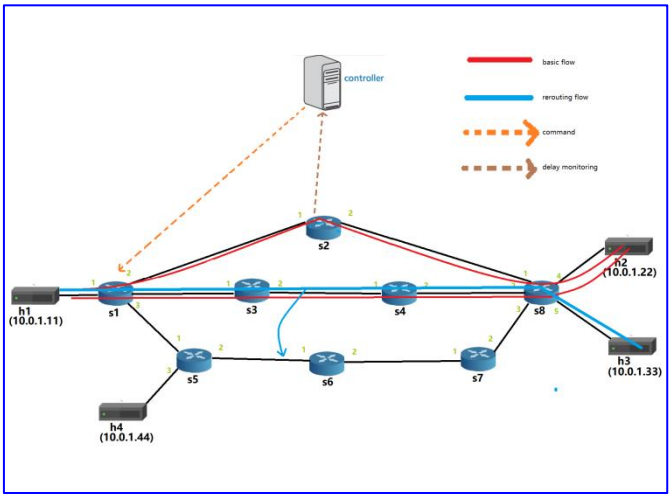


Fig. 3.5: Middle path to low path

(6)- Low path to high path

(5) will take place if delay measured in s2 is still not less than the low threshold of high path. This means, if the delay measured in s2 is less than its low threshold, even though the delay in s3-s4 is more than the high threshold of middle path, the rerouting to the low path will not be activated.

Another possibility is the delay measured at low path exceeds its high threshold, the rerouting flows should be moved from low path to high path whatever the delay at high path is. This means the rerouting flows return to the high path if low path is overloaded, then controller will decide where it would be rerouted to according to the delay measured at the high path.

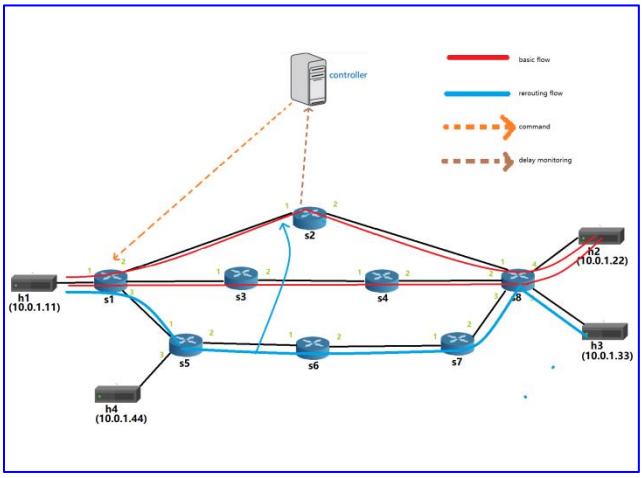


Fig. 3.5: Low path to high path

As shown in Fig.3.5, **(6)** describes how the h1-h3 flow will be moved directly from low path to high path if the delay of high path return to a low level.

(7) - **Low path to middle path:** Once the delays measured in s3-s4 is less than the low threshold, the rerouting of the flows h1-h3 returning to the middle path from low path will be activated.

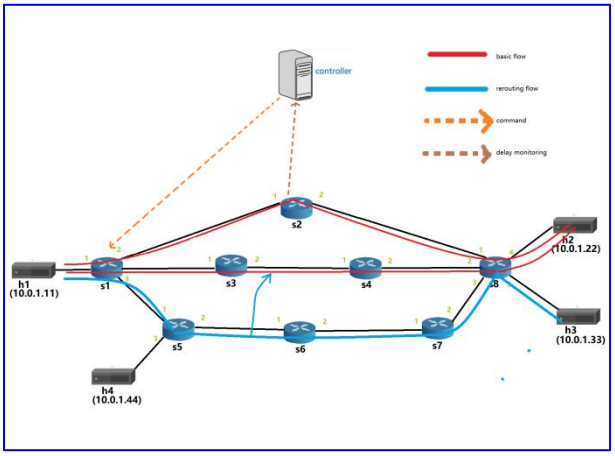


Fig. 3.6: Low path to middle path

As shown in Fig.3.6, **(7)** describes how the h1-h3 flow will be moved from low path to the middle path.

(8) -Middle path to high path: (5) will take place if delay measured in s2 is still not less than the low threshold. If the delay of high path returns to the low level, h1-h3 flow will be moved from the middle path to the high path whatever happened in middle path.

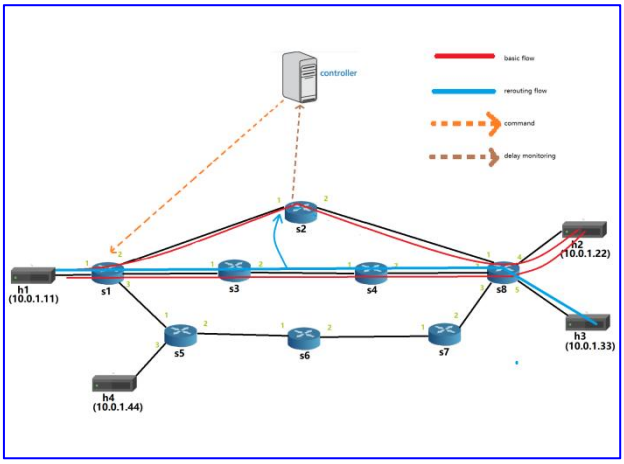


Fig. 3.7: Middle path to high path

As shown in Fig.3.8, **(8)** describes how the h1-h3 flow will be moved from the middle path to the high path.

(9)-Algorithm

```
1   Initialization
2   High and low thresholds of high path: hh ,hl ; High and low thresholds of middle path : mh,
3   ml ; High threshold of low path: lh
4   Delays at high, middle and low paths: vh, vm ,vl
5   Indicator of rerouting: indi
6   Processing
7   if   vh > hh
8       indi=1
9       if vm > mh
10          indi =2
11          if vl > lh
12              indi =0
13          end
14      elseif vm< ml
15          indi =1
16      elseif ml< vm < mh
17          indi=indi.current
18      end
19  elseif vh < hl
20      indi=0
21  elseif hl< vh < hh
22      indi=indi.current
23  end
24     Path decision
25  if indi= =0, high( )
26  elseif indi= =1, middle( )
27  elseif indi= =2, low( )
28  end
```

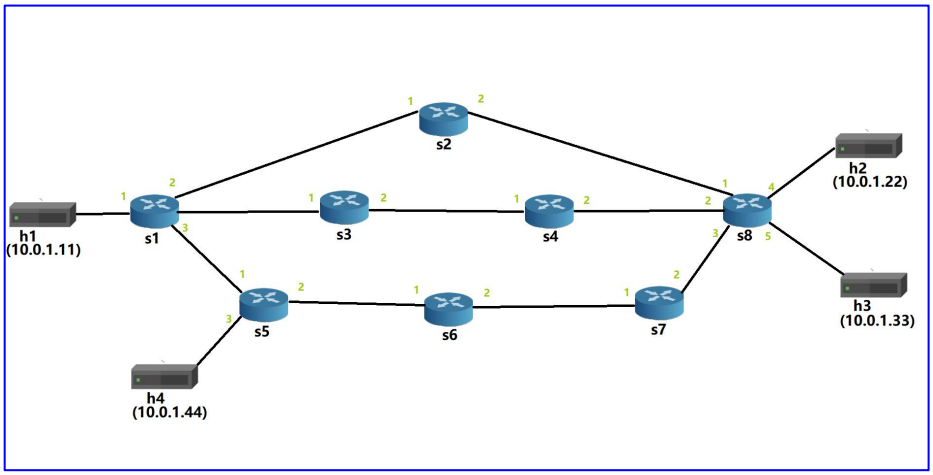
3.3. Priority-based traffic classification(witch strict priority queuing policy)

As discussed before, the implementation of QoS in both traditional network and SDN architecture often requires a high-level operation for flows. Implementing QoS via DiffServ is a good example [P4-QoS]: separating the ipv4_t header by splitting the TOS field into DiffServ and ECN fields, flows will be treated differently by the DiffServ values. Another example is VLAN priority utilization in TSN [TSN], priority-based flows will be transmitted through the switch deterministically by reserving time slots. Priority-based traffic classification can distinguish the importance of each flow, so that the delay aware rerouting mechanism has another possibility to deal with flows. Concretely, the rerouting function would be always implemented to the lower-priority flows what ever the current path is. Therefore, this method can ensure a relative stable transmission the high-priority flows .

Scenario

(1)-Definition of priorities

Define priorities to the flows hx-hy:



Flow	Priority
h1-h2	7
h1-h3	6
h4-h3	5

Fig. 3.8: Topology for delay management

(2)-Definition of thresholds: The high threshold and low threshold are the same for all three paths.

(3)-Rerouting rule: When the delay aware rerouting should be done, always the lower priority flow moves. E.g, flows of h1-h2 (P7), h1-h3 (P6) and h4-h3 (P5) are all at the high path, then the delay is already more than the threshold, so the h4-h3 (P5) flows should be moved to middle path firstly. If the delay is still more than the threshold after that, h1-h3 (P6) would thus be moved to middle path as well. The principal scenario is similar to the delay aware dynamic rerouting, the h1-h2 flows will be generated constantly at the high and middle paths. All other flows will be added to observe how this mechanism works and how this mechanism can be helpful in delay management.

(4)-Specific case: If there is only h1-h2(P7) flows at the high path, but the delay is more than the threshold, part of h1-h2 flows should treated as re-routing flows so these flows would also be moved to the middle path. To meet this case, the quantity of flows at high path will be accordingly increased .

(5)-Algorithm

---

1 **Initialization**
2 High and low thresholds of high path: **hh** ,**hl** ; High and low thresholds of middle path : **mh,**
3 **ml** ; High threshold of low path: **lh**
4 Delays at high, middle and low paths: **vh**, **vm** ,**vl**
5 Indicator of rerouting: **indi**
6 Lowest priority: **lp**, current priority: **fp**
7 **Processing**

```
8  if fp = = lp
9      if   vh > hh
10         indi=1
11         if vm > mh
12             indi =2
13             if vl > lh
14                 indi =0
15             end
16         elseif vm< ml
17             indi =1
18         elseif ml< vm < mh
19             indi=indi.current
20         end
21     elseif vh < hl
22         indi=0
23     elseif hl< vh < hh
24         indi=indi.current
25     end
26 end
27    Path decision
28 if indi= =0, high( )
   elseif indi= =1, middle( )
   elseif indi= =2, low( )
   end
```

3.4. Overload situation

Overview

With the implementation of dynamic rerouting and priority-based traffic classification, the network has still the possibility of being overloaded if there are too many flows enter into the network. This issue will totally paralyse the data transmission and the network might be forced to be shut off. For this reason, how to handle the overload situation has to be taken into consideration, this chapter proposes 3 main solutions: Drop tail, RED (Random Early Detection) and (m,k) -firm.

3.4.1.Drop tail

Tail drop is a passive queue management algorithm to make decision of dropping packets when the network has been overloaded (once the queue depth is more than the threshold). With this algorithm, the newly arriving packets will be dropped until the current network has enough capacity to accept incoming traffic.

Algorithm

```
1  while packets_arrive
2      if queue.depth < MaxTh
3          Enqueue_packet( )
4      else
5          Drop_packets( )
6      end
7  end
```

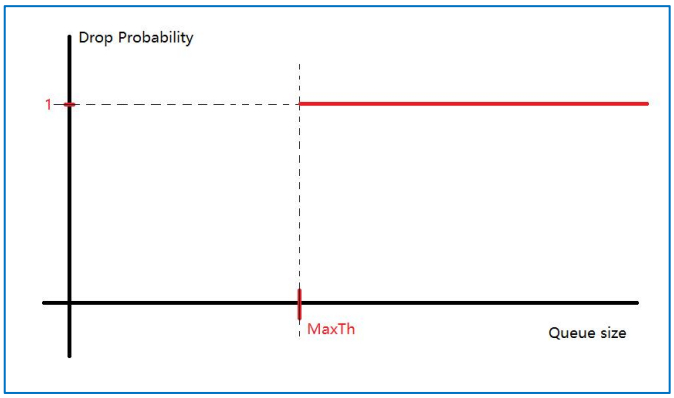


Fig. 3.9: Drop tail curve

3.4.2.RED

Compared with the Drop tail, RED is an active queue management algorithm, which detects incipient congestion early and randomly drops the packets with a probability [Floyd93]. RED can improve network performance and enhance network devices to detect the overload situation earlier.

Algorithm

```
1  while packets_arrive
2      if queue.depth > MinTh
3          Enqueue_packet( )
4      elseif   queue.depth < MaxTh
5          Enqueue_packet(p )
6      else
7          Drop_packets( )
8      end
9  end
```

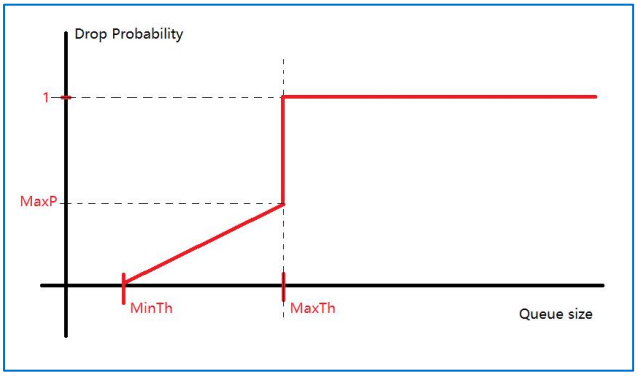


Fig. 3.9: RED curve

3.4.3.(m,k)-firm

At least m among any k consecutive invocations would meet their deadlines[Hamdaoui95]. (m,k)-firm real-time is one of the suitable ways to design the adaptive real-time system which provides dynamic QoS(Quality of Service) management [ARTIST03]

Example of (2,3)-firm

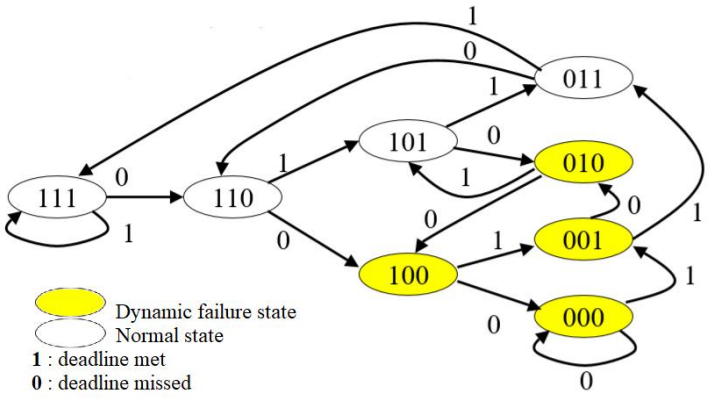


Fig. 3.10: State-transition of (2,3)-firm

Fig 3.10 shows the state transition diagram of (2,3)-firm: **1** indicates a deadline is met and **0** means missed. This diagram also shows the system is in dynamic failure state if there are more than 1 missed deadline. Otherwise in normal state.

4.Implementation

4.1. P4 and Mininet for delay management

4.1.1.Techniques and problems of P4 language

4.1.1.1. Technique applied for Time stamp

In the P4 program, one of most useful real time tools is the time stamp. Most of fields are part of the header named **standard_metadata**, which is predefined on the bmv2 framework. *bmv2 framework allows developers to implement their own P4-programmable architecture* [BMV2]. Besides, each P4 architecture usually defines its own **intrinsic_metadata** fields as well, which are used in addition to the metadata fields to offer more advanced features. The definition of **intrinsic_metadata** header is illustrated in Fig.4.1.

```
header_type intrinsic_metadata_t {
    fields {
        ingress_global_timestamp : 48;
        egress_global_timestamp : 48;
        mcast_grp : 16;
        egress_rid : 16;
    }
}
metadata intrinsic_metadata_t intrinsic_metadata;
```

Fig. 4.1: Definition of intrinsic metadata

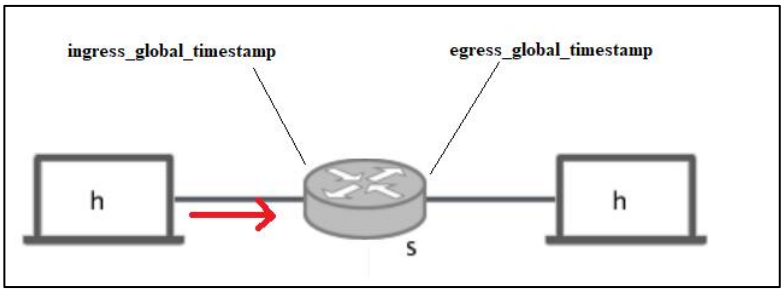


Fig. 4.2: Time stamps

As shown in Fig.4.2, the two time stamps to implement are:

Ingress_global_timestamp*:* a timestamp, in microseconds, set when the packet shows up on ingress. The clock is set to 0 every time the switch starts. This field can be read directly from either pipeline (ingress and egress) but should not be written to.

Egress_global_timestamp: a timestamp, in microseconds, set when the packet starts egress processing. The clock is the same as for ingress_global_timestamp. This field should only be read from the egress pipeline, but should not be written to.

4.1.1.2. Technique applied for Probe header

The most difficult in this part is to find out a suitable and useful header. This header shall have two functions: **(1)** it can record time stamps and to be used in receiver terminal to print out the delay measured, at least the ingress time and the egress time. **(2)** it has the numbered port and switch id variables which enables the routing of path. The best situation is that one header meets these two needs at the same time.

For these reasons, the PROBE header has been eventually chosen. In the P4 program, the PROBE header proposes two variables (type of time_t) to register the time stamps and also proposes switch-id and numbered port variables which simplify the rerouting of flows. This header has been used in link-monitoring tutorial, you can find more details here: https://github.com/p4lang/tutorials/tree/master/exercises/link_monitor. The header which can register the time stamp values are added, as usual, the specific header should be defined with following code :

```
from scapy.all import *

TYPE_PROBE = 0x812

class Probe(Packet):
    fields_desc = [ ByteField("hop_cnt", 0)]

class ProbeData(Packet):
    fields_desc = [ BitField("bos", 0, 1),
                    BitField("swid", 0, 7),
                    ByteField("port", 0),
                    IntField("byte_cnt", 0),
                    BitField("in_time", 0, 48),
                    BitField("out_time", 0, 48)]

class ProbeFwd(Packet):
    fields_desc = [ ByteField("egress_spec", 0)]

bind_layers(Ether, Probe, type=TYPE_PROBE)
bind_layers(Probe, ProbeFwd, hop_cnt=0)
bind_layers(Probe, ProbeData)
bind_layers(ProbeData, ProbeData, bos=0)
bind_layers(ProbeData, ProbeFwd, bos=1)
bind_layers(ProbeFwd, ProbeFwd)
```

Fig. 4.3: PROBE header definition

4.1.1.3. Problem of impossibility of measuring the whole delay of one way

The method of measuring the whole delay of one way can be described as follows:

(a) IngressTimeStamp of the 1st packet in the 1st switch at any path.

(b) EgressTimeStamp of the last packet in the last switch at this path.

Whole delay of one way = **(b) - (a)**

However, the P4 language has a systemic defect [BMV2-TS]: all time tamps will begin at 0 when the process begins, if one starts multiple switches processes, either on the same system, it is highly likely that their time tamps will have some small duration of each other. This means the switches generally start to work at different time. For example, Time stamp values from switch1 could easily be X microseconds later than, or earlier than switch 2. This problem also implied the impossibility of adding TSN mechanism into SDN architecture via P4 language because the time synchronization is the bedrock of the TSN implementation, all traffic shapers in TSN such as TAS (Time Aware shaper) or CBS (Credit Based shaper) should always validate the time synchronization at the beginning. The explication and demonstration of this issue are available in appendix C.

4.1.1.4. Problem of visualization of queuing delay and queue depth

According to the definition of queuing delay and queue depth in switch, there are at least three apparent facts:

(1) The queuing delay should be zero if the queue depth is empty.

(2) The queuing delay will be relatively stable since the queue depth has been a constant value.

(3) There should be a proportional relationship between these two parameters.

In the P4 program, the queue depth can be visualized using **deq_timedelta** of **queueing_metadata** header [BMV2-TS]: the time, in microseconds, that the packet spent in the queue. Also, the queuing delay can be visualized using **deq_timedelta**: the time, in microseconds, that the packet spent in the queue. The header's definition is illustrated in Fig.4.4.

```
header_type queueing_metadata_t {
    fields {
        enq_timestamp : 48;
        enq_qdepth : 16;
        deq_timedelta : 32;
        deq_qdepth : 16;
        qid : 8;
    }
}
metadata queueing_metadata_t queueing_metadata;
```

Fig. 4.4: Definition of queueing metad

However, the verification of all threes facts are failed via P4 program by implementing a simple topology. The queuing delay measured is never ever 0 (even close to 0) even though the queue depth is empty. The queuing delay never becomes stable since the queue depth has been constant. The worse thing is that there is no evidence can show there exists a proportional relationship between them. The explication and demonstration of this issue are available in appendix D.

Therefore, there is no reason to take the direct visualization of queuing delay (**deq_timedelta**) due to the unsuccessful verification for these threes points.

4.1.1.5. Priority-based traffic

As presented in section 3.3, priority-based traffic classification can distinguish the importance of each flow, so that the delay aware rerouting mechanism has another possibility to operate flows. The VLAN priority allows to assign a value to outbound packets containing the specified

VLAN-ID (VID). Packets containing the specified VID are marked with the priority level configured for the VID classifier, VLAN priority can thus be used to construct the priority-based traffic mechanism. By the way, the possible value of VLAN priority are 0-7 cause the IEEE 802.1Q standard proposes a 3-bit field which refers to different classes of traffic. The VLAN data frame is illustrated in following figure, it contains a 3-bit field corresponding to VLAN priority. The integration of VLAN priority in P4 program is available in appendix E.

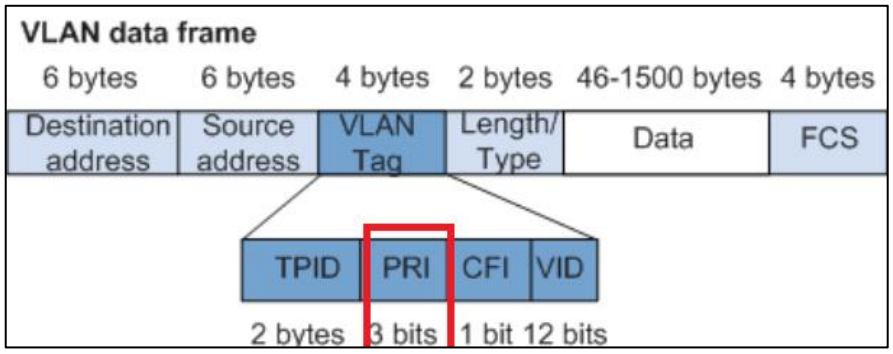


Fig. 4.5: VLAN data frame

4.1.1.6. Calculating the delay using Time stamps

As mentioned in section 2.5.3, the queuing delay is the most controllable of all types of delay, the queuing delay has been focused on as the research key point. However, there exists an innegligible problem that no evidence ensures that visualizations of queuing delay and queue depth are both correct. For this reason, the queuing delay is replaced by delay of switch with following equation with the help of time stamps.

$$\text{delay of switch} = \text{egress Time} - \text{ingress Time}$$

4.1.1.7. Technique applied for xterm

Xterm is the terminal emulator, it originated to the X Window System in 1984 (source: Wikipédia). The user can open many xterms on the same emulation system, each of terminals provides independent input and output for processing the network emulations, this means the every host and networking device can be running individually. This technique is also available and very useful in Mininet. There is an example that how the xterm technique works:

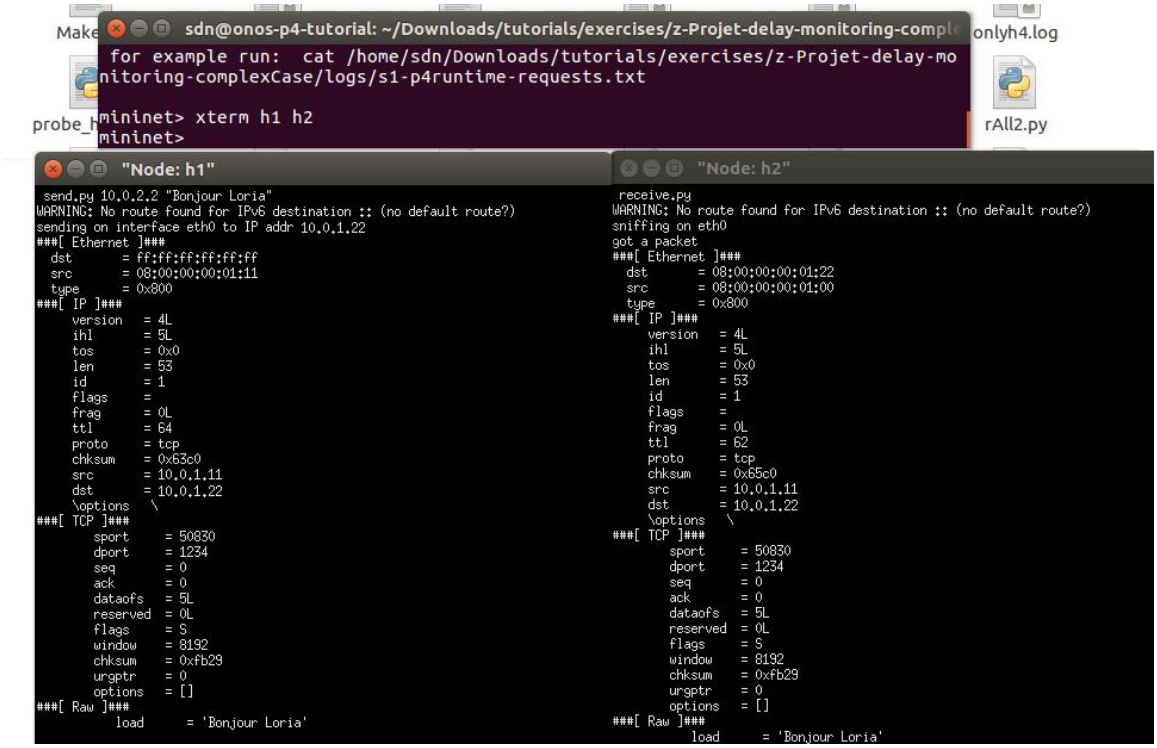


Fig. 4.6: Example of xterm

Run "mininet> xterm h1 h2", then you will have terminals for each host.

In h2's xterm, start the server: ./receive.py

In h1's xterm, send a message to h2: ./send.py 10.0.2.2 "Bonjour Loria"

As shown in Fig .4.6, h1 sent a message containing "Bonjour Loria" and h2 successfully received this message. Of course, other networking information are also printed, e.g., ethernet and IP fields. This example shows that hosts can be treated individually using xterm. Hence, this technique is widely used in networking emulations cause it can simplify the treatment of packets

in hosts such as inserting some specific headers into packets or visualizing some indicated values form packets.

4.1.2. Implementation

The scenarios of delay aware dynamic rerouting and priority-based traffic classification are both presented in Chapter 3. In this section, the results of implementations following the scenarios are performed to evaluate how the delay management can be benefited by the goodness of SDN architecture. First, we focus on the rerouting case. Then we added the priority-based traffic classification to measure how it can affect the delay management.

4.1.2.1. Delay aware dynamic rerouting

One of the primary goals of this report is to observe and balance the delay in real time by implementing the rerouting mechanism. Without this mechanism, one of most important QoS parameters, delay in the network will be extremely huge at one path since so many packets pass through the same path and other paths may not be enough occupied; Thus, the delay aware dynamic rerouting is capable of balancing the occupancy rate of each path from the perspective of real time delay measuring.

The network topology is the same which is shown in Fig.3.1. There are 4 hosts, each of them connected to one switch. Host h1 is generating packets towards hosts h2 and h3 constantly, only h1-h3 packets are rerouting packets. Hosts h2 and h3 are connected to s8 is able to receive the packets from h1 and collect the delay which is saved in the probe header (see in section 4.1.1.2). All links have the same configuration. The complete P4 code is available in appendix F and the

implementation steps are described in appendix G.

Every flow is sent with a speed of 30 packets/s, each packet contains 95 bytes. Cause the bandwidth of each link is configured to 10Mbps and the propagation delay on link is set to '0', there is thus no overload case on the link. We will calculate the average delay for every thirty packages to avoid the fluctuation sometimes caused by the measurement conducted in microseconds. An example of implementation is illustrated in Fig.

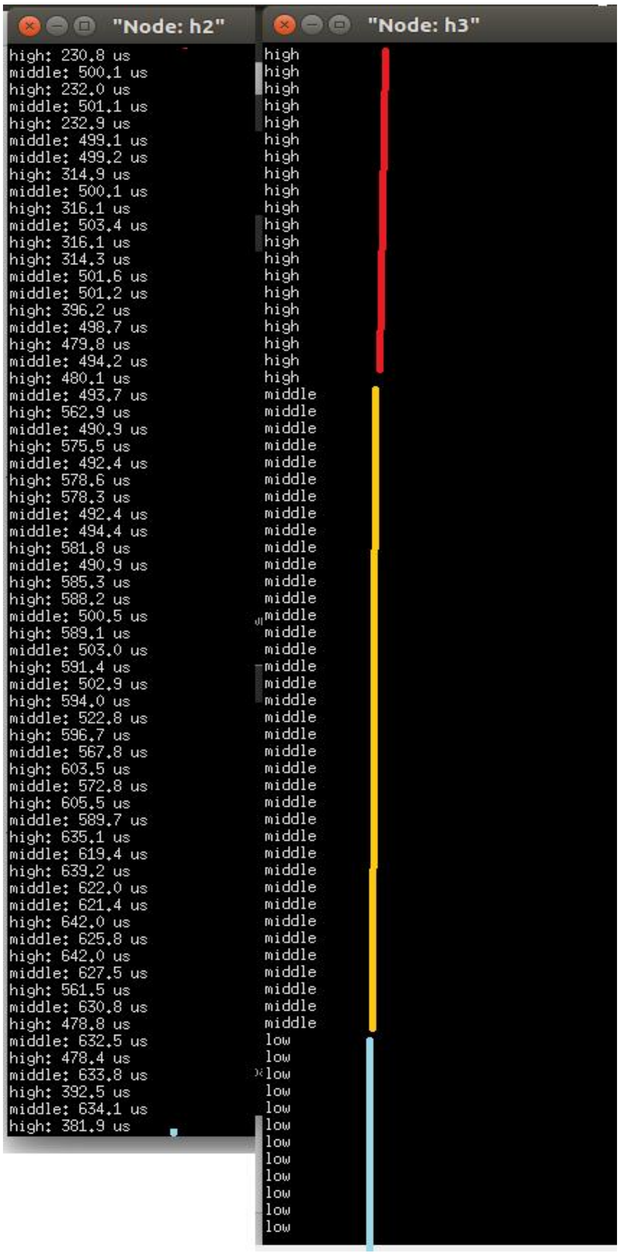


Fig. 4.7: Example of rerouting

There is an example of rerouting implementation at left. The "Node: h2" terminal window shows how the delay can be measured and visualized, the "Node: h3" terminal window gives the process that how the rerouting flows of h1-h3 goes to change its path from high (red) to middle (yellow) path, then middle to low (blue) path.

Be careful, the high and middle paths' delay can be observed in h2 cause there are always flows generated constantly from h1 to h2. The delay of low path has to be measured by the h1-h3 flow, the delay can be considered as zero if h1-h3 rerouting flow is not at the low path.

We did two implementations and 100 delays of each path are extracted randomly, these values are counted to draw bar graphs. Results of two implementations are illustrated as follows:

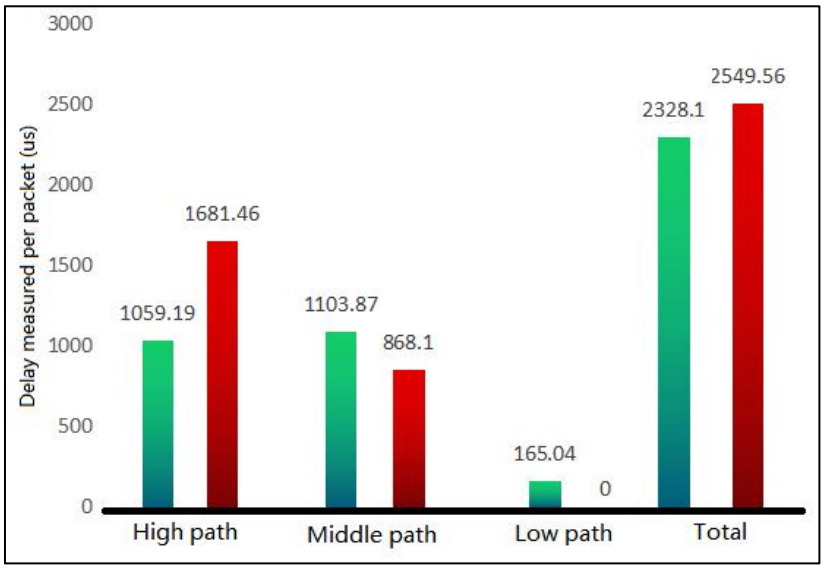


Fig. 4.8: Result 1 of delay management

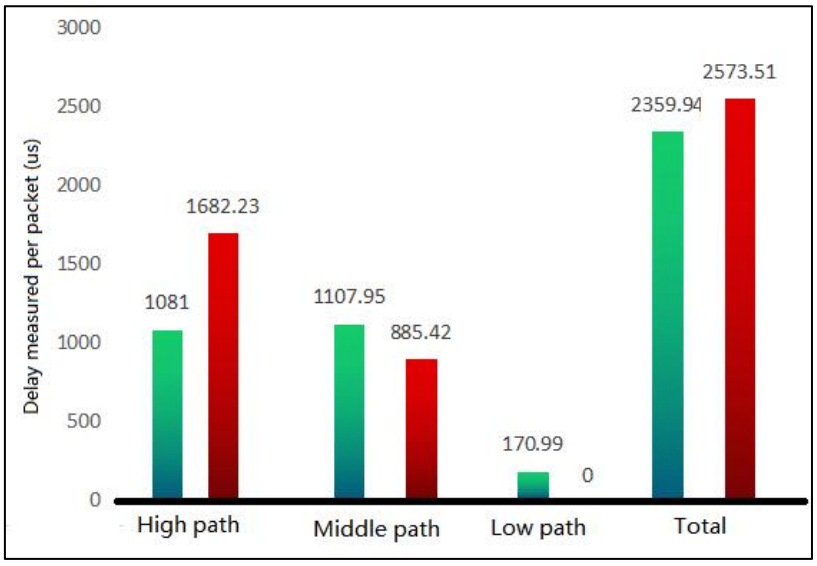


Fig. 4.9: Result 2 of delay management

From Figs. 4.8 and 4.9, the delays measured at high path of no-rerouting case is higher than rerouting case, this is because the h1-h3 and h1-h2 flows are initially both transmitted at the high path. The middle path obtains a lager delay in rerouting case due to the rerouting mechanism

moving some parts of h1-h3 flows from the high path to the middle path. The delay of low path without rerouting is zero cause there are no packets pass through, and its delay is no longer zero since the delay aware dynamic rerouting mechanism starts to work. We can see that the total delay of three paths are smaller in rerouting case, this means the rerouting network gives benefit to the delay management enabling more chance to load the middle and low paths. This result confirms that the implementation of delay aware dynamic rerouting matches the design of scenario in section 3.2 and it also shows the advantage of delay management using SDN.

4.1.2.3. Priority-based traffic classification

According to the scenarios in Chapter 3, the delay aware dynamic rerouting model does not provide the prioritization for each packet. In such model, all QoS flows are forwarded via single host to destination host without any indicator that distinguishes them. For this reason, an implementation of priority-based traffic classification is conducted to see how prioritization concept can improve the QoS management and how our model can solve problems related.

In our models, priorities will be defined and given to flows hx-hy (see its definition in section 3.3). These flows can be thus distinguished by their own priority. Section 3.3 shows how the scenario describes the implementation steps of such model. Compared to previous mechanism, in which for h1-h3 flows are supposed to be rerouted, the model presented in this section requires a prioritization configuration at the beginning. For every switch in any path, packets can be treated differently according to their priority. In this sense, the priority-based traffic classification can be considered as an improvement of rerouting mechanism. Topology and all configurations are the same to the previous implementations.

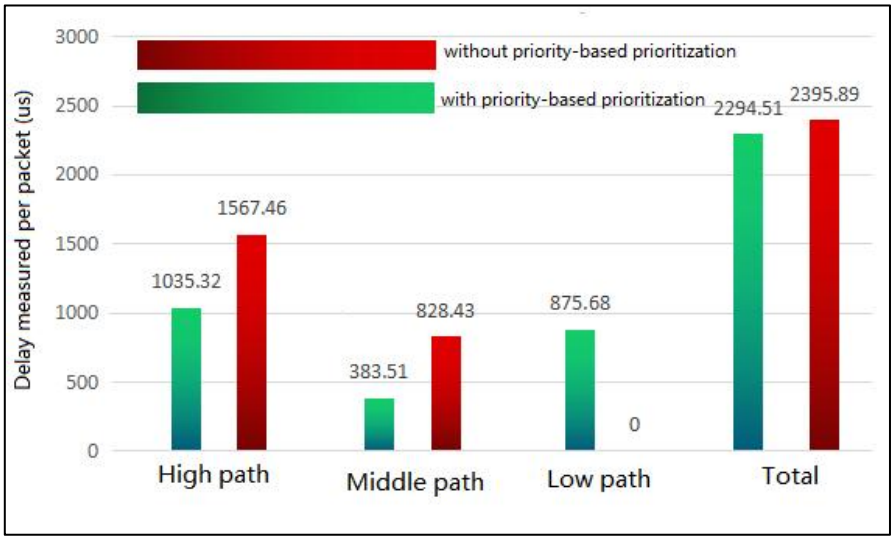


Fig. 4.10: Result of priority classification ranged by path

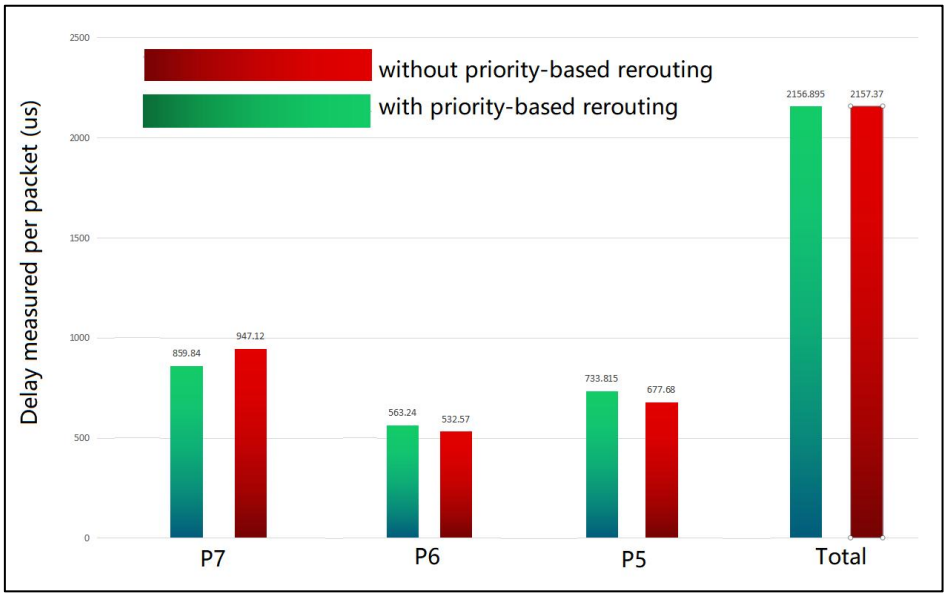


Fig. 4.11: Result of priority classification ranged by path

As shown in the Fig.4.8, the rerouting mechanism can still improve the performance of delay management if priority-based traffic classification has been applied into our model. The situation is quite similar to the no priority-based case, the delay becomes smaller by implementing the rerouting mechanism with the high and low thresholds at each path. One thing should be noticed that the high path's delay becomes smaller than no priority classification case because of multi-queue mechanism by different priorities, it is still bigger than the middle path cause the quantity of P7 flows is increased to conduct the P7 flows' rerouting ((4) in section 3.3). After

that the priority-based traffic classification is added, the observation delays by priority shows that the total delay of all priorities is basically unchanged from the Fig.4.9, and the high priority such as P7 gets effective delay management by moving low priority to other path since the high threshold is exceeded at every path.

It can be concluded that the priority-based traffic classification can not improve delay management for all priorities flows but the highest priority flow. And another conclusion is that the rerouting mechanism can always help to improve the performance of delay management in network whether the priority-based traffic classification mechanism is applied.

4.2. Pox/OpenFlow and mininet for delay management

4.2.1.Technique applied for and problems in POX

4.2.1.1. Technique applied for OpenFlow messages

The POX controller is able to configure and manages switches trough the OpenFlow channel, it can also receive events from them and sends packet out. There is an interface that connects switches and POX controller, as illustrated in the Fig. 4.12.

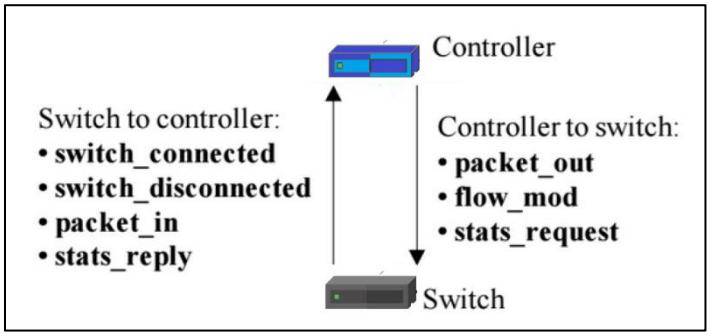


Fig. 4.12: OpenFlow message utilization

In our implementation, we need at least: packet_out, packet-in, and switch_connected.

These three OpenFlow messages allow the POX controller to respectively obtain and register their own time stamps at which each message was utilized, and to help to calculate the delay between switches (see in next section) .

4.2.1.2. Technique applied for Measuring the delay

The current POX controller does not directly offer a method to measure the delay. For this reason, we are going to look at an approach that monitors delay from inside the network[delay pox]. This method is able to capture the delay of path sending probe packets by means of OpenFlow messages, presented in previous section. The first step is to create a packet which will be used as a probe. Then, the controller, with a packet_out message, requests to the switch to send the packet through a particular port to the next one. Finally, when the next switch receives the packet, it sends a packet_in message to the controller in order to communicate the state of the packet. The Fig. 4.12 shows the mechanism.

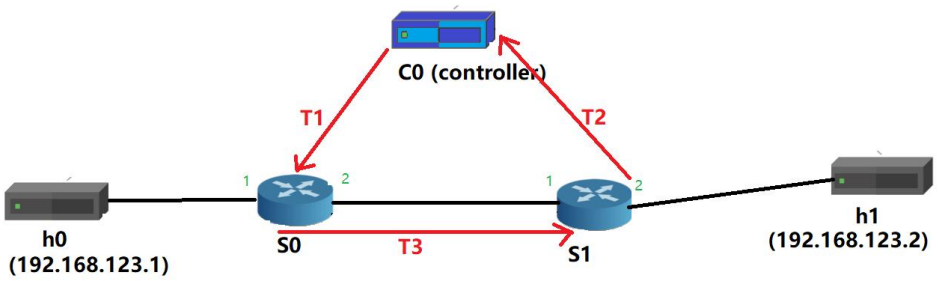


Fig. 4.12: Method of measuring the delay using POX controller

As shown in Fig. 4.12, the principal method to measure the latency is very simple: $T_3 = T_{total} - T_1 - T_2$

T_{total} is the total time.

T_3 corresponds to the latency we are going to measure,

T_1 and T_2 refer the time that the packets spent between the controller and switches.

So T_1= (RTT between the controller and s1)/2, the same to T2.

The code of controller is available in appendix H.

4.2.1.2. Problem of implementation of measuring the delay

The techniques presented in previous section is quickly and successfully implemented, the delay measured should be half of the 'ping' time using this technique. However, the result obtained shows there exists at least two problems using this technique.

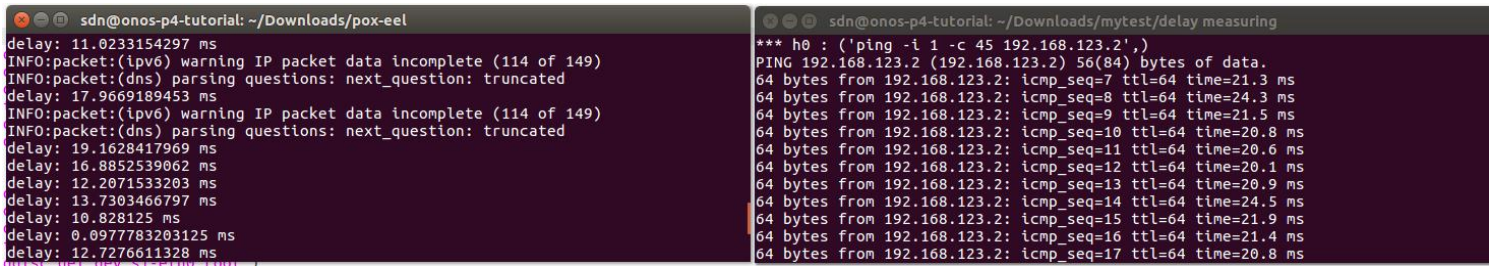


Fig. 4.13: Instability

As shown in Fig .4, the window at right gives the 'ping' result which should be double of the delay configured. but the window at left gives the delay measured by the controller using this method which is too unstable to be observed and some of them are even several times higher than others. After several tests, this situation dose not appear accidentally but is often observed from the implementation results.

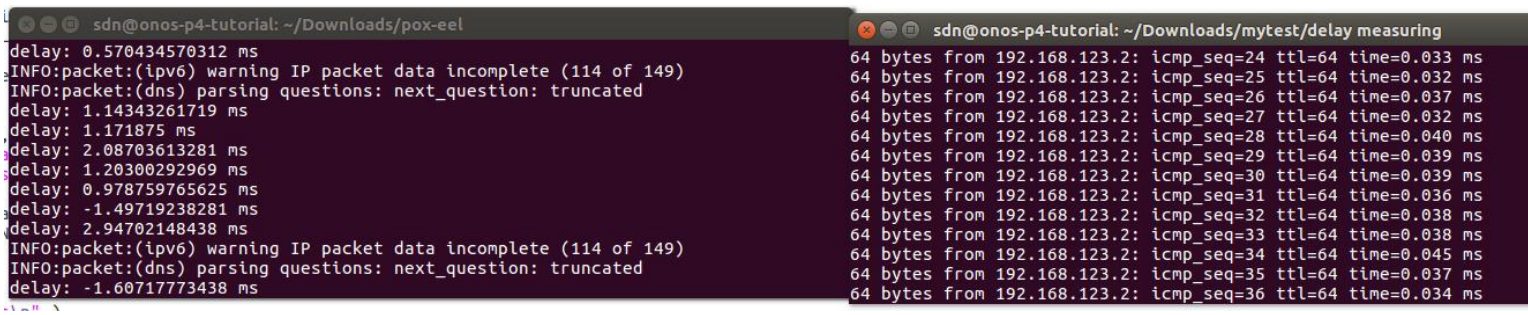


Fig. 4.14: Negativity

The ping's result in right window gives at least values close to zero, this means the delay should be very small and close to zero as well. However, it sometimes comes with negative values as shown in Fig. 4.14 and the delay measured stays very unstable. After several tests then, this issue dose not appear accidentally but is often observed from the implementation results.

After seriously checking the method presented in section 4.2.1.2 and all programming code of implementation, the source of these two problems is still not located. One reasonable and logical guess is that the computer's capacity affects the delay measuring, another one is that there exists the perturbation produced by the interaction between the POX controller and Mininet interface.

4.2.1.3. Technique applied for Printing the queue length

In view of the two main problems of the implementation delay measuring, another way to measure the delay has to be proposed. The queuing delay will be sometimes replaced by the queue length cause they have a linear relationship, other way to measure the delay is thus to use the queue length. This section presents the main idea that how to print the queue length with a simple example.

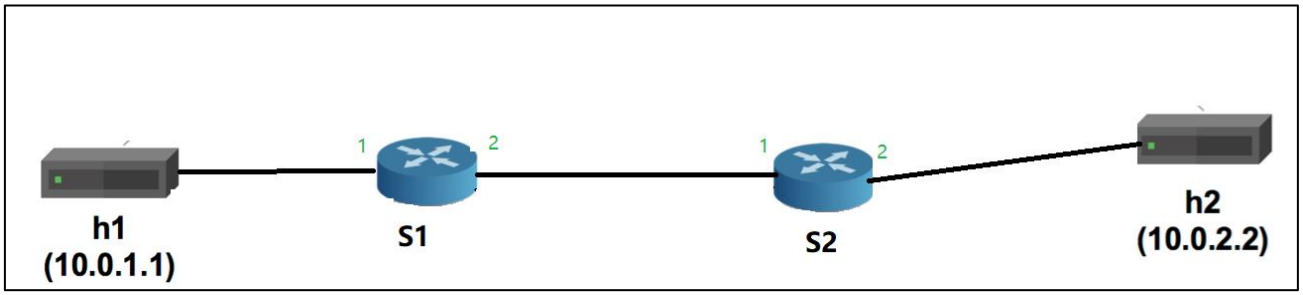


Fig. 4.15: Topology of the example to print the queue length

The topology of the example is illustrated in Fig. 4.15, some packets will be sent from h1 to h2 and theirs queue lengths can be thus printed. We focus on the port 2 of s1 in this example, its queue length should be increased since amount of packets are transmitted from h1 to h2.

```
def monitor_qlens1(interval_sec=0.01):
  pat_queued =re.compile(r'backlog\s[^\s]+\s([\d]+)p')
  cmd = "tc -s qdisc show dev s1-eth2"
  ret = []
  open("qlens1.txt","w").write('')
  t0 = "%f" % time()
  while 1:
    p = Popen(cmd, shell=True, stdout=PIPE)
    output = p.stdout.read()
    matches = pat_queued.findall(output)
    if matches and len(matches) > 1:
      ret.append(matches[1])
      t1 = "%f" % time()
      #print str(float(t1)-float(t0)), matches[1]
      open("qlens1.txt","a").write(str(float(t1)-float(t0))+' '+matches[1]+'\n')
    sleep(interval_sec)
```

Fig. 4.16: Program to print the queue length

This python program allows to extract the queue length of s1 per 0.01 second and to store the delays measured of s1 into a file named “qlens1.txt”. The same to s2, the result will be saved in “qlens2.txt”

```
519 6.9790430069 0
520 6.99190402031 0
521 7.00474095345 0
522 7.02026987076 0
523 7.03595280647 0
524 7.05556988716 0
525 7.07097196579 0
526 7.08358979225 0
527 7.09574794769 0
528 7.10821390152 0
529 7.1210539341 6
530 7.13343000412 17
531 7.14623498917 28
532 7.15907883644 40
533 7.1714630127 50
534 7.18642902374 62
535 7.19990491867 74
536 7.21285581589 85
537 7.22722792625 97
538 7.24005389214 109
539 7.25274181366 120
540 7.26535201073 131
541 7.27755093575 142
542 7.29011297226 152
543 7.30295395851 163
544 7.31595087051 174
545 7.32886791229 185
546 7.34122300148 194
547 7.35429787636 206
548 7.36790084839 217
549 7.38047981262 227
550 7.39324188232 238
551 7.40588498116 249
552 7.41857481003 259
553 7.43408489227 272
554 7.44695281982 283
555 7.45945000648 293
556 7.47230482101 303
557 7.48514795303 314
558 7.49986100197 326
559 7.51243591309 337
560 7.52684783936 349
561 7.53970885277 359
562 7.55216288567 369
563 7.56499695778 380
564 7.57963991165 392
```

Fig. 4.17: Result of qlen1.txt

From Fig. 4.17, the result of ‘qlen1.txt’ shows that the port 2 of s1 get some packets accumulated in its queue. As described before, this is normal cause its queue length should be increased since the packets are transmitted from h1 to h2.

The whole implementation and result are available in appendix I.

4.2.2. Implementation

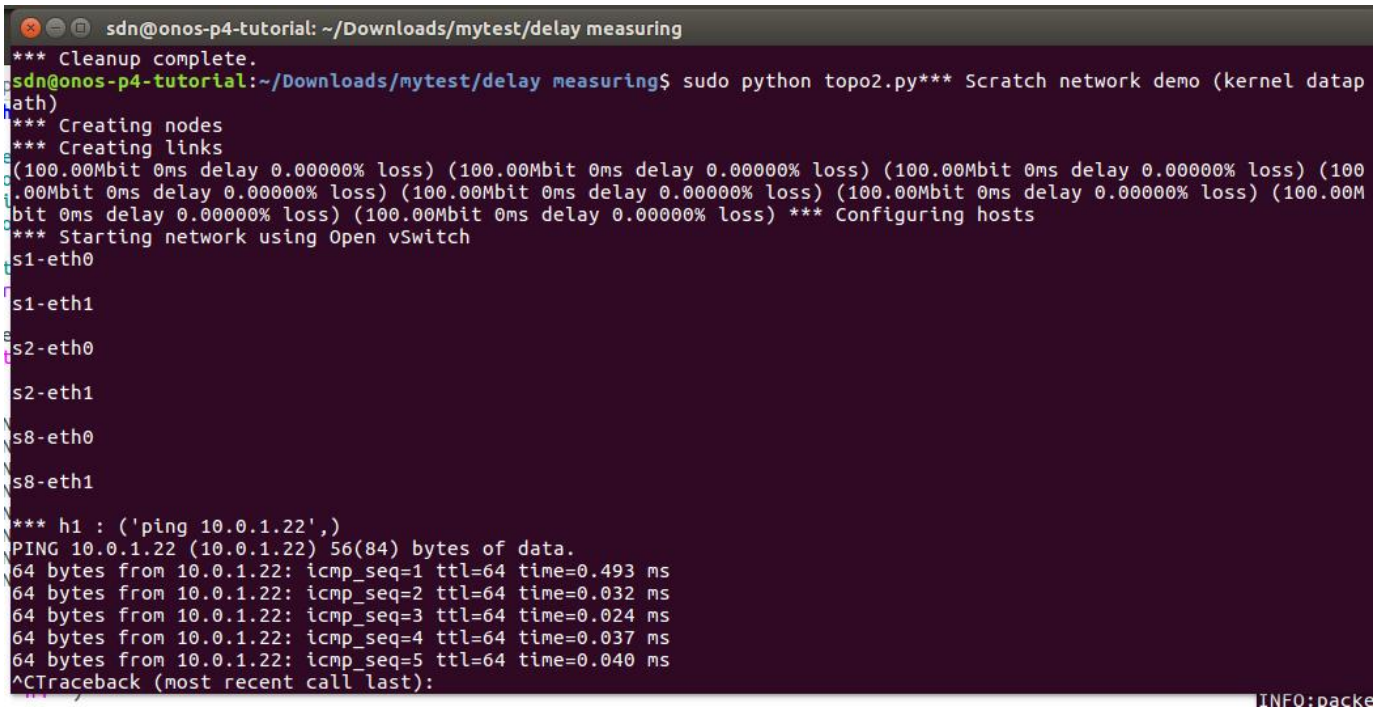


Fig. 4.18:

From Fig. 4.18, the topology is successfully initialized, then the h1: ('ping 10.0.1.22',) is going to test

4.3.Comparison between P4 and POX/OpenFlow

POX controller is inherited from NOX controller and is one of the oldest and most used controllers in the world. Contrary to this, P4 language is a very new recent, its updated version named P4-16 was released in 2017. Comparison between POX and P4 allows to discover the development and evaluation of technologies in SDN controllers and their simulations and emulations, and further measure the role of the POX and P4 in QoS using SDN especially the delay management by reviewing their highlights and the points that they can extend.

Aspect	P4	POX/OpenFlow
Duration of existence	*Less than 4 years*	*Almost 9 years*
Framework base	*Mininet*	*Mininet*
Open source	*Yes*	*Yes*
Developer	*Stanford University*	*Nicira*
Implementation Language	*Java/C/Javascript*	*Python*
Other APIs	*BMV2/PI*	*No*
Controller complexity	*Normal*	*Very high*
The richness of functionality	*Very high*	*High*
Packet-level operation	*Excellent*	*General*
Metadata utilization	*Yes*	*No*

Table 4.1: Comparison between POX and P4 globally

From the Table 4.1, we can see that POX and P4 are both open source projects based on Mininet framework, they are developed by different groups. POX was released a long time ago, it has accomplished an extraordinary work with the developments in the SDN functionalities by efforts of developers. P4 was created much later than POX, but its packet-level operation and use of metadata are very advanced, which made its SDN functionalities to be greatly expanded in a short period of time. It is worth mentioning that the P4 has three programming languages (Java/C/Javascript), the controller is still relatively easy to be programmed. However, POX has only one language, its controller is more complex to be programmed because it lacks data forwarding processing compared to P4. In addition, P4 has more APIs, it may cause some difficulties while debugging and implementing.

Aspect	P4	POX/OpenFlow
(1) Queue length visualization	*Using probe header*	*By Traffic control command*
Reliability of (1)	*Very unstable and illogical*	*Accurate*
(2) Queuing delay visualization	*Using probe header*	*By an indirect method*
Reliability of (2)	*Unstable*	*Problem of instability and negativity*
(3) Priority-traffic classification	*Adding VLAN header*	/
Reliability of (3)	*Reliable*	/

Table 4.1: Comparison between POX and P4 in Delay management

5. Conclusion and future work

This chapter is going to highlight the investigations and contributions of each previous chapter to addressing the initial research question for this internship (stated in introduction). Furthermore, the challenges and problems encountered during this internship are also underlined in this chapter. The initial research question is:

How can QoS management using SDN be applied? In particular, how can QoS management be implemented using different SDN platforms?

To answer this question, first thing to seek, is to choose the controller platform cause it conditions the success or unsuccess of the QoS management objectives. For this propose, sections 2.2 and 2.3 gave a complete overview of the POX controller and P4 language with an example of L2 learning switch. Chapter 2 presents also, firstly, the global context of SDN and its latest developments, e.g., the current SDN controllers [Bruno14]. Second, the state of art on QoS management using SDN especially the delay performance that we focus on in this internship. In addition, we also introduced the challenges of QoS management using SDN at the end of this chapter.

Since the traditional networking architectures have not been established enough to solve the QoS

issues due to its networking configuration static [Tomovic15], SDN is proposed as the best technique to simplify network management operations. Chapter 3 presents the measurement design and methodology, by demonstrating a dynamic delay aware automatic re-routing algorithm using SDN controller. The main contribution in this algorithm is to insert probe header into forwarding packets which are allowed to register the delay and the numbered ports to reroute. More than this, the priority-based traffic classification is proposed to measure how this mechanism can affect the delay management.

In Chapter 4, the concrete cechniques applied in both P4 and POX are presented and discussed. From the results of emulations, we see that the delay aware dynamic re-routing algorithm gives a better balance of the total traffic between high, middle and low paths. Though the priority-based traffic classification dose not improve the delay of all priories, the re-routing mechanism can still help to improve the delay performance, this shows great potentials of SDN in delay QoS management. Based on the comparisons between POX and P4 presented in chapter 4, our implementation revealed, first, some inaccuracy in queue length indication and the lack of explicit time management components in P4, second, the instability and negativity of indirect delay measuring in POX.

It is worth mentioning that the QoS using SDN is developing rapidly in both industry and academic [Murat17]. However, a lot of related researches and designs only put forward theoretical model and lack of practical operations and experiments support. Furthermore, QoS management is very sensitive and complex, the problems encountered during the internship may have already brought errors in some experiments. Thus, a further experiment needs to be conducted to locate the source of these problems. In addition, taking queue length into account to implement the dynamic delay aware rerouting algorithm in POX is also a future work to explore.

References

[Afaq15-1] Afaq, M., Rehman, S.U., Song, W.-C., 2015. A framework for classification and

visualization of elephant flows in sdn-based networks. Procedia Comput. Sci. 65 (2015), 672–681

[Afaq15-2] Afaq, M., Rehman, S., Song, W.-C., 2015. Visualization of elephant flows and qos provisioning in sdn-based networks. In: 17th Asia-Pacific Network Operations and Management Symposium (APNOMS), pp. 444–447

[Ayadi13] Ayadi, I., Diaz, G., Simoni, N., 2013. Qos-based network virtualization to future networks: An approach based on network constraints. In: Proceedings of the Fourth International Conference on the Network of the Future (NOF), pp. 1–5

[Ballard10] Ballard, J.R., Rae, I., Akella, A., 2010. Extensible and scalable network monitoring using opensafe. In: Proceedings of the 2010 Internet Network Management Conference on Research on Enterprise Networking, INM/WREN'10, USENIX Association, Berkeley, CA, USA, pp. 8–8

[Bari13] Bari, M., Chowdhury, S., Ahmed, R., Boutaba, R., 2013. Policycop: An autonomic qos policy enforcement framework for software defined networks. In: Future Networks and Services (SDN4FNS), 2013 IEEE SDN for, pp. 1–7

[BMV2-1] https://github.com/p4lang/behavioral-model/blob/master/docs/simple_switch.md

[BMV2-TS]https://github.com/p4lang/behavioral-model/blob/master/docs/simple_switch.md#BMv2-timesta mp-implementation-notes.

[Bosshart14] P. Bosshart, D. Daly, G. Gibb, M. Izzard, N. McKeown, J. Rexford,C. Schlesinger, D. Talayco, A. Vahdat, G. Varghese, and D. Walker,“P4: Programming Protocol-independent Packet Processors,” ACM SIGCOMMComput. Commun. Rev., vol. 44, no. 3, pp. 87–95, Jul. 2014.

[Broadbent12] Broadbent, M., Race, N., 2012. Opencache: exploring efficient and transparent content delivery mechanisms for video-on-demand. In: Proceedings of the 2012 ACM conference on CoNEXT student workshop, CoNEXT Student ’12, pp. 15–16.

[Broadbent15] Broadbent, M., King, D., Baildon, S., Georgalas, N., Race, N., 2015. Opencache: A software-defined content caching platform. In: Proceedings of the 1st IEEE Conference on Network Softwarization (NetSoft), pp. 1–5

[Bueno13] Bueno, I., Aznar, J., Escalona, E., Ferrer, J., Antoni, J., 2013. Garcia-Espin, An opennaas based sdn framework for dynamic qos control. In: IEEE SDN for Future Networks and Services (SDN4FNS), 2013 pp. 1–7

[Bruno14] Bruno Astuto A. Nunes, Marc Mendonca, Xuan-Nam Nguyen, Katia Obraczka, and Thierry Turletti, A Survey of Software-Defined Networking: Past, Present, and Future of Programmable Networks , IEEE CommunicationsSociety, 2014, 16 (3), pp.1617-1634

[Caba15] Caba, C., Soler, J., 2015. Apis for qos configuration in software defined networks. In: Proceedings of the 1st IEEE Conference on Network Softwarization (NetSoft), pp. 1–5

[Chowdhury14] Chowdhury, S., Bari, M., Ahmed, R., Boutaba, R., 2014. Payless: A low cost network monitoring framework for software defined networks. In: IEEE Network Operations and Management Symposium (NOMS), pp. 1–9

[Civanlar10] Civanlar, S., Parlakisik, M., Tekalp, A., Gorkemli, B., Kaytaz, B., Onem, E., 2010. A qosenabled openflow environment for scalable video streaming. In: IEEE GLOBECOM Workshops (GC Wkshps), pp. 351–356

[Desai15] Desai, A., Nagegowda, K., 2015. Advanced control distributed processing architecture (acdpa) using sdn and hadoop for identifying the flow characteristics and setting the quality of service(qos) in the network. In: IEEE International Advance Computing Conference (IACC), pp. 784–788

[Diego15] Diego Kreutz, Fernando MV Ramos, Paulo Verissimo, Christian Esteve Rothenberg, Siamak Azodolmolky, and Steve Uhlig. "Software-defined networking: A comprehensive survey". In: Proceedings of the IEEE 103.1 (2015), pp. 14–76 (cit. on pp. 3,4)

[Dobrijevic14] Dobrijevic, O., Kassler, A.J., Skorin-Kapov, L., Matijasevic, M., 2014. Q-point: Qoedriven path optimization model for multimedia services. In: Wired/Wireless Internet Communications. Springer International Publishing, pp. 134–147.

[Duan14] Duan, Q., 2014. Network-as-a-service in software-defined networks for end-to-end qos provisioning. In: Wireless and Optical Communication Conference (WOCC), 2014 23rd, pp. 1–5

[Duan15] Duan, Q., Wang, C., Li, X., End-to-end service delivery with qos guarantee in software defined networks, 2015. arXiv preprint,

[Egilmez11] Egilmez, H., Gorkemli, B., Tekalp, A., Civanlar, S., 2011. Scalable video streaming over openflow networks: an optimization framework for qos routing. In: Proceedings of the 18th IEEE International Conference on Image Processing (ICIP), pp. 2241–2244

[Egilmez12-1] Egilmez, H., Dane, S., Bagci, K., Tekalp, A., 2012. Openqos: An openflow controller design for multimedia delivery with end-to-end quality of service over softwaredefined networks. In: Signal Information Processing Association Annual Summit and Conference (APSIPA ASC), Asia-Pacific, pp. 1–8.

[Eglimez12-2] Egilmez, H., Civanlar, S., Tekalp, A., 2012. A distributed qos routing architecture for scalable video streaming over multi-domain openflow networks. In: Proceedings of the 19th IEEE International Conference on Image Processing (ICIP), pp. 2237–2240

[Egilmez13] Egilmez, H., Civanlar, S., Tekalp, A., 2013. An optimization framework for qos-enabled adaptive video streaming over openflow networks. IEEE Trans. Multimed. 15 (3), 710–715.

[Egilmez14] Egilmez, H., Tekalp, A., 2014. Distributed qos architectures for multimedia streaming over software defined networks. IEEE Trans. Multimed. 16 (6), 1597–1609

[Fernandez13] Fernandez, Marcial. "Evaluating OpenFlow controller paradigms." In ICN 2013, The Twelfth International Conference on Networks, pp. 151-157. 2013.

[Floyd93] Floyd, S.,andJacobson,V.,"RandomEarlyDetectionGatewaysfor Congestion Avoidance".InACM/IEEETransactionsonNetworkingg,3(1),August1993

[Georgopoulos13] Georgopoulos, P., Elkhatib, Y., Broadbent, M., Mu, M., Race, N., 2013.

Towards networkwide qoe fairness using openflow-assisted adaptive video streaming. In: Proceedings of the 2013 ACM SIGCOMM workshop on Future human-centric multimedia networking, FhMN'13, pp. 15–20.

[Gorlatch14] Gorlatch, S., Humernbrum, T., Glinka, F.,2014. Improving qos in real-time internet applications: from best-effort to software-defined networks. In: International Conference on Computing, Networking and Communications (ICNC), pp. 189–193

[Gorlatch15] Gorlatch, S., Humernbrum, T., 2015. Enabling high-level qos metrics for interactive online applications using sdn. In: 2015 International Conference on Computing, Networking and Communications (ICNC), pp. 707–711

[Govindarajan14] Govindarajan, K., Meng, K.C., Ong, H., Tat, W.M., Sivanand, S., Leong, L.S., 2014. Realizing the quality of service (qos) in software-defined networking (sdn) based cloud infrastructure. In: Proceedings of the 2nd International Conference on Information and Communication Technology (ICoICT), pp. 505–510

[Gude08] N. Gude, T. Koponen, J. Pettit, B. Pfaff, M. Casado, N. McKeown,and S. Shenker. Nox: towards an operating system for networks. ACM SIGCOMM Computer Commun. Review, 38(3):105–110, 2008.

[Heleno14] Heleno Isolani, P., Araujo Wickboldt, J., Both, C., Rochol, J., Zambenedetti Granville, L., 2014. Interactive monitoring, visualization, and configuration of openflow-based sdn. In: IFIP/IEEE International Symposium on Integrated Network Management (IM), pp. 207–215

[Hu15] Hu, C., Wang, Q., Dai, X., 2015. Sdn over ip: enabling internet to provide better qos guarantee. In: Proceedings of the Ninth International Conference on Frontier of Computer Science and Technology (FCST), pp. 46–51

[Ishimori13] Ishimori, A., Farias, F., Cerqueira, E., Abelem, A., 2013. Control of multiple packet schedulers for improving qos on openflow/sdn networking. In: Proceedings of the Second European Workshop on Software Defined Networks (EWSDN), pp. 81–86

[ITU-T08] ITU-T. Definitions of terms related to Quality of Service. Recommendation E800. Telecommunication Standardization Sector of ITU, Sept. 2008 (cit. on pp. 1, 2).

[Jarschel13] Jarschel, M., Wamser, F., Hohn, T., Zinner, T., Tran-Gia, P., 2013. Sdn-based application-aware networking on the example of youtube video streaming. In: Proceedings of the Second European Workshop on Software Defined Networks (EWSDN), pp. 87–92

[Jinyao15] Jinyao, Y., Hailong, Z., Qianjun, S., Bo, L., Xiao, G., 2015. Hiqos: an sdn-based multipath qos solution. China Commun. 12 (5), 123–133.

[Jose11] Jose, L., Yu, M., Rexford, J., 2011. Online measurement of large traffic aggregates on commodity switches. In: Proceedings of the 11th USENIX Conference on Hot Topics in Management of Internet, Cloud, and Enterprise Networks and Services, Hot- ICE'11, pp. 13–13.

[Karakus15] Karakus, M., Durresi, A., 2015. A scalable inter-as qos routing architecture in software defined network (sdn). In: IEEE Proceedings of the 29th International Conference on Advanced Information Networking and Applications (AINA), pp. 148–154

[Kassler12] Kassler, A., Skorin-Kapov, L., Dobrijevic, O., Matijasevic, M., Dely, P., 2012. Towards qoe-driven multimedia service negotiation and path optimization with software defined networking. In: Proceedings of the 20th International Conference on Software, Telecommunications and Computer Networks (SoftCOM), pp. 1–5.

[Kim10] Kim, W., Sharma, P., Lee, J., Banerjee. S. , Tourrilhes, J., Lee, S.-J., Yalagandula, P.,2010. Automated and scalable qos control for network convergence. In: Proceedings of the 2010 Internet Network Management Conference on Research on Enterprise Networking, INM/WREN'10, pp. 1–1.

[Kim13] Kim, Hyojoon, and Nick Feamster. "Improving network management with software defined networking." Communications Magazine, IEEE 51, no. 2 (2013): 114-119.

[Kotronis16] Kotronis, V., Klöti, R., Rost, M., Georgopoulos, P., Ager, B., Schmid, S., Dimitropoulos, X., 2016. Stitching inter-domain paths over ixps. In: Proceedings of the Symposium on SDN Research, SOSR '16, ACM, New York, NY, USA, pp. 17:1–17:12

[Kumar13] Kumar, H., Gharakheili, H., Sivaraman, V., 2013. User control of quality of experience in home networks using sdn. In: IEEE International Conference on Advanced Networks and Telecommuncations Systems (ANTS), pp. 1–6

[Lantz10] Lantz, Bob, Brandon Heller, and Nick McKeown. "A network in a laptop: rapid prototyping for software-defined networks." In Proceedings of the 9th ACM SIGCOMM Workshop on Hot Topics in Networks, p. 19. ACM, 2010.

[Manish18] Manish Paliwal, Deepti Shrimankar, and Omprakash Tembhurne. "Controllers in SDN: A review report". In: IEEE Access6 (2018), pp. 36256–36270 (cit. on p. 6).

[Marconett15-1] Marconett, D., Yoo, S.J., 2015. Flowbroker: a software-defined network controller architecture for multi-domain brokering and reputation. J. Netw. Syst. Manag. 23 (2), 328–359

[Marconett15-2] Marconett, D., Yoo, S.J.B., 2015. Flowbroker: market-driven multi-domain sdn with heterogeneous brokers. In: Optical Fiber Communications Conference and Exhibition (OFC), pp. 1–3

[Menth19] Menth, Michael; Mostafaei, Habib; Merling, Daniel; Häberle, Marco
Implementation and Evaluation of Activity-Based Congestion Management Using P4 (P4-ABC), Future Internet volume 11, issue 7 (2019)

[Miao15] Miao, W., Agraz, F., Peng, S., Spadaro, S., Bernini, G., Perello, J., Zervas, G., Nejabati, R., Ciulli, N., Simeonidou, D., Dorren, H., Calabretta, N., 2015. Sdn-enabled ops with qos guarantee for reconfigurable virtual data center networks. IEEE/OSA J. Opt. Commun. Netw. 7 (7), 634–643

[Middleton15] Middleton, S., Modafferi, S., 2015. Experiences monitoring and managing qos using sdn on testbeds supporting different innovation stages. In: 2015 Proceedings of the 1st IEEE Conference on Network Softwarization (NetSoft), pp. 1–5

[Murat17] Murat Karakus and Arjan Durresi, Quality of Service (QoS) in Software Defined Networking (SDN) J. Netw.Comput. Appl., Vol. 80, pp200-218, 2017.

[Nam-Seok13] Nam-Seok, K., Hwanjo, H., Jong-Dae, P., Hong-Shik, P., 2013. Openqflow: Scalable openflow with flow-based qos. IEICE Trans. Commun. 96 (2), 479–488.

[Nick08] Nick McKeown, Tom Anderson, Hari Balakrishnan, Guru Parulkar, Larry Peterson, Jennifer Rexford, Scott Shenker, and Jonathan Turner. "OpenFlow: enabling innovation in campus networks". In: ACM SIGCOMM Computer Communication Review 38.2 (2008), pp. 69–74 (cit. on pp. 2, 4)

[ONF] Open Networking Foundation. url: https://www.opennetworking/ about (cit. on p. 4).

[ONF14-1] Open Networking Foundation, Software Defined Networking: the new norm for networks, White Paper, Retrieved Apr. 2014.

[ONF14-2] Open Networking Foundation (ONF), SDN architecture, 2014. Tech. rep. , June 2014.

[OVS controller] OVS controller at http://yuba.stanford.edu/~casado/of-sw.html.

[Owens13] Owens, H., Durresi, A., 2013. Video over software-defined networking (vsdn). In: Proceedings of the 16th International Conference on Network-Based Information Systems (NBiS), pp. 44–51

[Owens14] Owens, H., Durresi, A., Jain, R., 2014. Reliable video over software-defined networking (rvsdn). In: 2014 IEEE Global Communications Conference, pp. 1974–1979

[P4-2] P4 at https://p4.org/.

[P4-L2]https://github.com/nsg-ethz/p4-learning/blob/master/exercises/04-L2_Learning/solution/l2_learning_controller.py

[P4-QoS] https://github.com/p4lang/tutorials/blob/master/exercises/qos/solution/qos.p4

[Pox] POX at https://github.com/noxrepo/pox

[Pox-L2]https://github.com/CPqD/RouteFlow/blob/master/pox/pox/forwarding/l2_learning.py

[Raphael14] Raphael Durner, Andreas Blenk, Wolfgang Kellerer, Performance Study of Dynamic QoS Management for OpenFlow-enabled SDN Switches IEEE 23rd International Symposium on Quality of Service (IWQoS 2015), Portland, OR, USA, 15-16 June 2015

[Seddiki14] Seddiki, M.S., Shahbaz, M., Donovan, S., Grover, S., Park, M., Feamster, N., Song, Y.-Q., 2014. Flowqos: Qos for the rest of us. In: Proceedings of the Third Workshop on Hot Topics in Software Defined Networking, HotSDN'14, ACM, New York, NY, USA, pp. 207–208

[Seddiki15] Seddiki, M.S., Shahbaz, M., Donovan, S., Grover, S., Park, M., Feamster, N., Song, Y.-Q., 2015. Flowqos: per-flow quality of service for broadband access networks. Georgia Institute of Technology.

[Sonkoly12] Sonkoly, B., Gulyas, A., Nemeth, F., Czentye, J., Kurucz, K., Novak, B., Vaszkun, G., 2012. On qos support to ofelia and openflow. In: European Workshop on Software Defined Networking (EWSDN), pp. 109–113

[Tomovic14] Tomovic, S., Prasad, N., Radusinovic, I., 2014. Sdn control framework for qos

provisioning. In: 22nd Telecommunications Forum Telfor (TELFOR), pp. 111–114

[Tomovic15] S. Tomovic, I. Radusinovic, and N. Prasad, “Performance comparison of QoS routing algorithms applicable to large-scale SDN networks,” in IEEE EUROCON 2015 - International Conference on Computer as a Tool (EUROCON), 2015, Conference Proceedings, pp. 1–6.

[Tootoonchian10] Tootoonchian, A., Ghobadi, M., Ganjali, Y., 2010. Opentm: traffic matrix estimator for openflow networks. In: Proceedings of the 11th International Conference on Passive and Active Measurement, PAM'10, pp. 201–210.

[van14] van Adrichem, N., Doerr, C., Kuipers, F., 2014. Opennetmon: network monitoring in openflow software-defined networks. In: Proceedings of the IEEE Network Operations and Management Symposium (NOMS), pp. 1–8

[Wallner13] Wallner, R., Cannistra, R., 2013. An sdn approach: Quality of service using big switchs floodlight open-source controller. In: Proceedings of the Asia-Pacific Advanced Network, vol. 35, pp. 14–19

[Wang14] Wang, J., Wang, Y., Dai, X., Bensaou, B., 2014. Sdn-based multi-class qos-guaranteed inter-data center traffic management. In: Proceedings of the IEEE 3rd International Conference on Cloud Networking (CloudNet), pp. 401–406

[Wang,J15] Wang, J., Wang, Y., Dai, X., Benasou, B., 2015. Sdn-based multi-class qos guarantee in inter-data center communications. IEEE Trans. PP Cloud Comput. 99, 1.

[Wang,W15] Wang, W., Tian, Y., Gong, X., Qi, Q., Hu, Y., 2015. Software defined autonomic qos model for future internet. J. Syst. Softw. 110, 122–135.

[Xu15] Xu, C., Chen, B., Qian, H., 2015. Quality of service guaranteed resource management dynamically in software defined network. In: Journal of Communications, Vol. 10, pp. 843–850

[Yiakoumis12] Yiakoumis, Y., Katti, S., Huang, T.-Y., McKeown, N., Yap, K.-K., Johari, R., 2012. Putting home users in charge of their network. In: Proceedings of the 2012 ACM Conference on Ubiquitous Computing, UbiComp'12, pp. 1114–1119.

[Yilmaz15] Yilmaz, S., Tekalp, A., Unluturk, B., 2015. Video streaming over software defined networks with server load balancing. In: Proceedings of the 2015 International Conference on Computing, Networking and Communications (ICNC), pp. 722–726.

[Yu13] Yu, M., Jose, L., Miao, R., 2013. Software defined traffic measurement with opensketch.In: Proceedings of the 10th USENIX Symposium on Networked Systems Design and Implementation (NSDI 13), Lombard, IL, pp. 29–42.

[Yu15] Yu, T.-F., Wang, K., Hsu, Y.-H., 2015. Adaptive routing for video streaming with qos support over sdn networks. In: 2015 International Conference on Information Networking (ICOIN), pp. 318–323

Appendix

Appendix A: L2 learning switch controller in POX

File: /home/sdn/Downloads/pox-eel/pox/forwarding/l2_learning.py

```
 # Copyright 2011-2012 James McCauley #
# Licensed under the Apache License, Version 2.0 (the "License"); # you
may not use this file except in compliance with the License. # You may obtain
a copy of the License at:
#
#     http://www.apache.org/licenses/LICENSE-2.0
#
# Unless required by applicable law or agreed to in writing, software #
distributed under the License is distributed on an "AS IS" BASIS,
# WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied. # See
the License for the specific language governing permissions and
# limitations under the License.

"""
An L2 learning switch.

It is derived from one written live for an SDN crash course. It is
somwhat similar to NOX's pyswitch in that it installs exact-match
rules for each flow.
"""

from pox.core import core
import pox.openflow.libopenflow_01 as of
from pox.lib.util import dpid_to_str, str_to_dpid from
pox.lib.util import str_to_bool
import time

log = core.getLogger()
_flood_delay = 0

class LearningSwitch (object): """
  The learning switch "brain" associated with a single OpenFlow switch.

  When we see a packet, we'd like to output it on a port which will eventually
  lead to the destination.              To accomplish this, we build a table
  that maps addresses to ports.

  We populate the table by observing traffic.  When we see a packet from
  some source coming from some port, we know that source is out that port.

  When we want to forward traffic, we look up the desintation in our table.
          If we don't know the port, we simply send the message out all ports
  except the one it came in on.             (In the presence of loops, this
  is bad!).
  """
  def_init (self, connection, transparent):
    # Switch that is adding L2 learning switch capabilities to
    self.connection = connection
    self.transparent = transparent

    # table to save the ports
    self.macToPort = {}

    # listen to the connection to hear PacketIn messages
    connection.addListeners(self)


  def _handle_PacketIn (self, event):
```

```
    #Handle packet in messages from the switch to implement above algorithm. packet =
    event.parsed

    def flood (message = None):
      #Floods the packet
      msg = of.ofp_packet_out()
      # Wait for a little while to flood
      if time.time() - self.connection.connect_time >= _flood_delay:
        if self.hold_down_expired is False:
          self.hold_down_expired = True
          log.info("%s: Flood hold-down expired -- flooding",
              dpid_to_str(event.dpid))

        if message is not None: log.debug(message)
        # OFPP_FLOOD is optional; on some switches you may need to change this to OFPP_ALL.
        msg.actions.append(of.ofp_action_output(port = of.OFPP_FLOOD))
      else:
        pass
      msg.data = event.ofp msg.in_port =
      event.port self.connection.send(msg)

    def drop (duration = None):

      #Drops this packet and optionally installs a flow to continue #dropping
      similar ones for a while
      if duration is not None:
        if not isinstance(duration, tuple):
          duration = (duration,duration)
        msg = of.ofp_flow_mod()
        msg.match = of.ofp_match.from_packet(packet)
        msg.idle_timeout = duration[0] msg.hard_timeout =
        duration[1] msg.buffer_id = event.ofp.buffer_id
        self.connection.send(msg)
      elif event.ofp.buffer_id is not None:
        msg = of.ofp_packet_out() msg.buffer_id =
        event.ofp.buffer_id msg.in_port =
        event.port self.connection.send(msg)
    #Use source address and switch port to update address/port table
    self.macToPort[packet.src] = event.port

    #Is transparent = False and either Ethertype is LLDP or the packet's #destination
    address is a Bridge Filtered address?
    if not self.transparent:
      if packet.type == packet.LLDP_TYPE or packet.dst.isBridgeFiltered(): drop() # if
        yes?
        return

    # Is destination multicast? # if
    yes ?Flood the packet if
    packet.dst.is_multicast:
      flood()
    else:
      #Port for destination address in our address/port table?
      if packet.dst not in self.macToPort:

        flood("Port for %s unknown -- flooding" % (packet.dst,)) # no? Flood the
packet
      else:
        port = self.macToPort[packet.dst]
        if port == event.port: # Is output port the same as input port? # if

          yes ? Drop packet and similar ones for a while

          log.warning("Same port for packet from %s -> %s on %s.%s. Drop."

              % (packet.src, packet.dst, dpid_to_str(event.dpid), port)) drop(10)
          return
```

```
          # Install flow table entry in the switch so that this #flow
          goes out the appopriate port log.debug("installing flow
          for %s.%i -> %s.%i" %
                    (packet.src, event.port, packet.dst, port)) msg =
          of.ofp_flow_mod()
          msg.match = of.ofp_match.from_packet(packet, event.port)
          msg.idle_timeout = 10
          msg.hard_timeout = 30
          msg.actions.append(of.ofp_action_output(port = port)) msg.data =
          event.ofp # Send the packet out appropriate port
          self.connection.send(msg)


  class l2_learning (object):

    Waits for OpenFlow switches to connect and makes them learning switches.

    def_init #Initialize

(self, transparent, ignore = None):

      #See LearningSwitch for meaning of 'transparent' #'ignore' is
      an optional list/set of DPIDs to ignore
      core.openflow.addListeners(self)
      self.transparent = transparent
      self.ignore = set(ignore) if ignore else ()

    def _handle_ConnectionUp (self, event):
      if event.dpid in self.ignore:
        log.debug("Ignoring connection %s" % (event.connection,))
        return
      log.debug("Connection %s" % (event.connection,))
      LearningSwitch(event.connection, self.transparent)


  def launch (transparent=False, hold_down=_flood_delay, ignore = None): #Starts the
    L2 learning switch.
    try:
      global _flood_delay
      _flood_delay = int(str(hold_down), 10)
      assert _flood_delay >= 0
    except:
      raise RuntimeError("Expected hold-down to be a number")

    if ignore:
      ignore = ignore.replace(',', ' ').split()
      ignore = set(str_to_dpid(dpid) for dpid in ignore)
    core.registerNew(l2_learning, str_to_bool(transparent), ignore)
```

Appendix B: L2 learning switch controller in P4

File: /home/sdn/Downloads/p4-learni···ning/l2_learning_controller.py

```
import      nnpy
import struct
from p4utils.utils.topology import Topology
from p4utils.utils.sswitch_API import SimpleSwitchAPI from
scapy.all import Ether, sniff, Packet, BitField

class   CpuHeader(Packet):
    name = 'CpuPacket'
    fields_desc = [BitField('macAddr',0,48), BitField('ingress_port', 0, 16)]

class   L2Controller(object):
    #configuration
    def_init_(self, sw_name):

self.topo = Topology(db="topology.db") # the topology self.sw_name = sw_name
# the topology
self.thrift_port  =  self.topo.get_thrift_port(sw_name)  #  the  ports  self.cpu_port  =
                        self.topo.get_cpu_port_index(self.sw_name) # the portSDN of
 sw
self.controller = SimpleSwitchAPI(self.thrift_port)# portSDN of controller self.init()
    # initialization
    init(self):      self.controller.reset_state()
self.add_boadcast_groups()      self.add_mirror()
#self.fill_table_test()

       def add_mirror(self):
           if self.cpu_port:
             self.controller.mirroring_add(100, self.cpu_port)

    def add_boadcast_groups(self):
interfaces_to_port = self.topo[self.sw_name]["interfaces_to_port"].copy() #filter lo and cpu
port
interfaces_to_port.pop('lo',None)
interfaces_to_port.pop(self.topo.get_cpu_port_intf(self.sw_name), None)

mc_grp_id = 1
rid = 0
for ingress_port in interfaces_to_port.values():

             port_list = interfaces_to_port.values()[:]
             del(port_list[port_list.index(ingress_port)])

             #add              multicast              group
             self.controller.mc_mgrp_create(mc_grp_id)

             #add multicast node group
             handle = self.controller.mc_node_create(rid, port_list)

             #associate              with              mc              grp
             self.controller.mc_node_associate(mc_grp_id, handle)

             #fill broadcast table
             self.controller.table_add("broadcast", "set_mcast_grp", [str (ingress_port)],
 [str(mc_grp_id)])

             mc_grp_id +=1
             rid +=1
```

```
     #the table to test the ports of sw
     def fill_table_test(self):
self.controller.table_add("dmac", "forward", ['00:00:0a:00:00:01'], ['1'])
self.controller.table_add("dmac", "forward", ['00:00:0a:00:00:02'], ['2'])
self.controller.table_add("dmac", "forward", ['00:00:0a:00:00:03'], ['3'])
self.controller.table_add("dmac", "forward", ['00:00:0a:00:00:04'], ['4'])

     # L2 learning
     def learn(self, learning_data):
for mac_addr, ingress_port in           learning_data:
             print  "mac:  %012X  ingress_port:  %s  "  %  (mac_addr,  ingress_port)
             self.controller.table_add("smac", "NoAction", [str(mac_addr)]) # add the
 source mac to the input port
             self.controller.table_add("dmac",    "forward",    [str(mac_addr)],    [str
 (ingress_port)]) # add the destination mac to the output port

     # receiver of cpu
 recv_msg_cpu(self, pkt): packet =
Ether(str(pkt))
if packet.type == 0x1234:
             cpu_header             =             CpuHeader(packet.payload)
             self.learn([(cpu_header.macAddr, cpu_header.ingress_port)])

     def run_cpu_port_loop(self):

         cpu_port_intf  =  str(self.topo.get_cpu_port_intf(self.sw_name).replace("eth0",
 "eth1"))
sniff(iface=cpu_port_intf, prn=self.recv_msg_cpu)


 if_name == "_main__":
     import sys
     sw_name     =     sys.argv[1]
     receive_from = sys.argv[2] if
     receive_from == "digest":
controller = L2Controller(sw_name).run_digest_loop()
     elif receive_from == "cpu":
controller = L2Controller(sw_name).run_cpu_port_loop()
```

Appendix C: Problem for measuring the whole one way delay in P4

The initial method of measuring the delay of flow is simple:

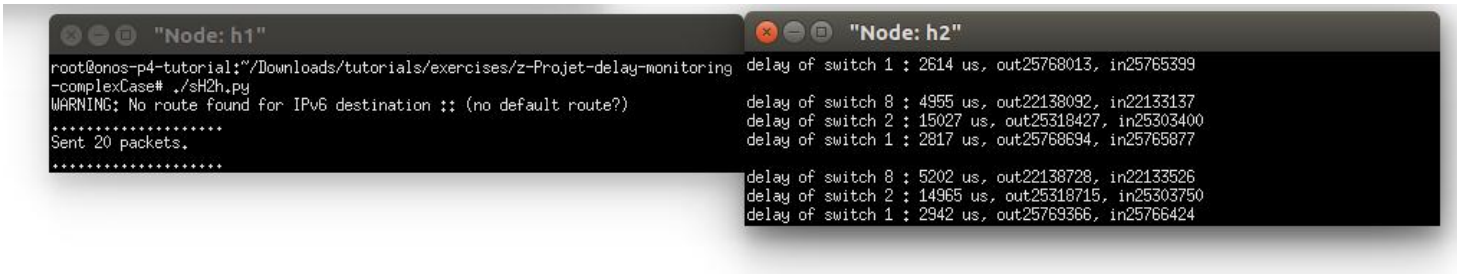
1. Obtain the ingressTimeStamp (a) of the 1st packet in the 1st switch at any path.
2. Obtain the egressTimeStamp (b) of the last packet in the last switch at this path.
3. Delay of flow = (b) - (a)

However, the P4 language has a systematic problem: all time tamps will begin at 0 when the process begins, if one starts multiple switches processes, either on the same system, it is highly likely that their time tamps will have some small duration of each other. This means the switches generally start to work at different time. For example, Timestamp values from switch1 could easily be X microseconds later than, or earlier than switch 2.

For further information:

https://github.com/p4lang/behavioral-model/blob/master/docs/simple_switch.md#BMv2-timestamp-implementation-notes

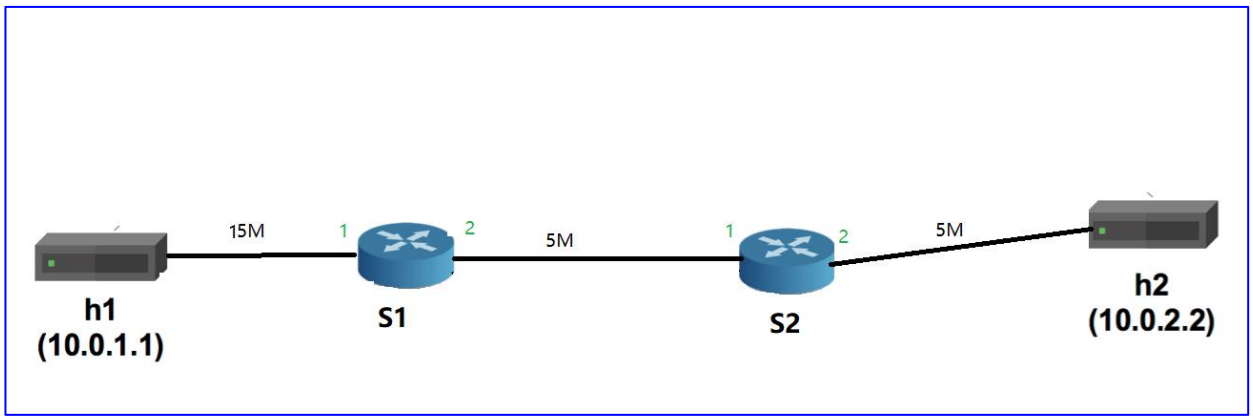
Example:



According to the topology, the h1-h2 packets sending will pass the high path so s1-s2-s8. As shown in the figure, the egress time stamps are always bigger than the ingress time stamps, but the egress time stamp of s8 is even much smaller than the ingress time stamps of s1, this is what we discussed before, the switches are turned on at different time point, the delay of flows thus can not be measured accurately, even some of them would be minus because of this issue.

Appendix D: Problem of visualization of delay and queue length in P4

The goal of this log is to deal with the problem of visualization of queuing delay and queue depth. The topology:



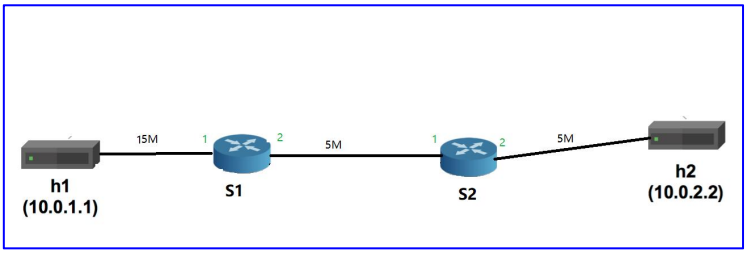
The bandwidth are set to 15Mbps(h1-s1), 5Mbps(s1-s2) and 5Mbps(s2-h2).

Large number of packets would be send from h1 to h2, so the port 2 of s1 should have some packets into the queue and the port 2 of s2 should have no packets in the queue cause the bandwidth of link s1-s2 and s2-h2 are the same.

Topology

The topology illustrated before has been built as follows:

```
{
    "hosts": {
        "h1": {"ip": "10.0.1.1/24", "mac": "08:00:00:00:01:11",
               "commands":["route add default gw 10.0.1.10 dev eth0",
                           "arp -i eth0 -s 10.0.1.10 08:00:00:00:01:00"]},
        "h2": {"ip": "10.0.2.2/24", "mac": "08:00:00:00:02:22",
               "commands":["route add default gw 10.0.2.20 dev eth0",
                           "arp -i eth0 -s 10.0.2.20 08:00:00:00:02:00"]}
    },
    "switches": {
        "s1": { "runtime_json" : "pod-topo/s1-runtime.json" },
        "s2": { "runtime_json" : "pod-topo/s2-runtime.json" }
    },
    "links": [
        ["h1", "s1-p1","0",15], ["h2", "s2-p2","0",5], ["s1-p2", "s2-p1","0",5]
    ]
}
```



How to check the mentioned problem

According to the definition of queuing delay and queue depth in switch, there are at least three apparent facts:

(4) The queuing delay should be zero if the queue depth is empty.

(5) The queuing delay will be relative stable since the queue depth has been a constant value.

(6) There should be a proportional relationship between these two parameters.

In order to check this problem, some works have been done:

- In P4 program, the header of queue utilization's definition is illustrated in following figure:

```
header_type queueing_metadata_t {
    fields {
        enq_timestamp : 48;
        enq_qdepth : 16;
        deq_timedelta : 32;
        deq_qdepth : 16;
        qid : 8;
    }
}
metadata queueing_metadata_t queueing_metadata;
```

-In P4 program:

```
// update the time stamps
hdr.probe_data[0].in_time = (bit<48>)standard_metadata.enq_qdepth;
hdr.probe_data[0].out_time = (bit<48>)standard_metadata.deq_timedelta;
```

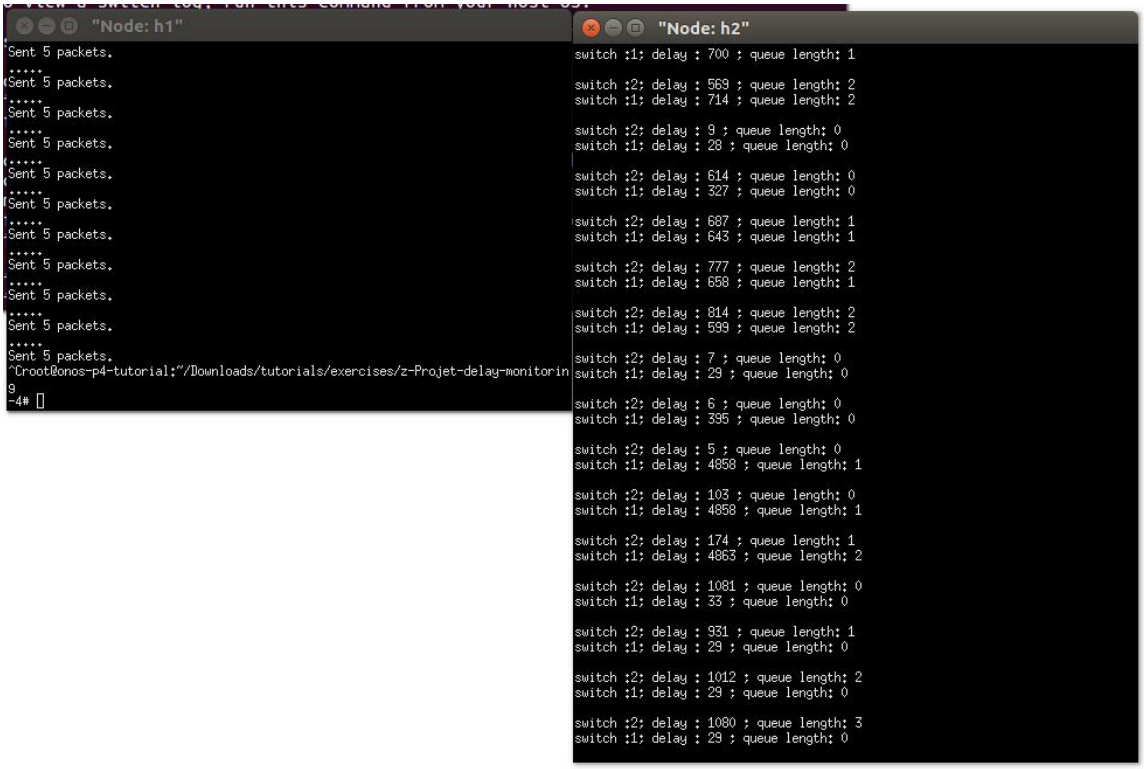
The field in_time is used to register the value of queue length and the out_time is used to register the queuing delay.

-For visualizing the queue length, run the python file r.py:

```
def handle_pkt(pkt):
    if ProbeData in pkt:
        data_layers = [l for l in expand(pkt) if l.name=='ProbeData']
        print ""
        for sw in data_layers:
            delay = 0 if sw.out_time == sw.in_time else (sw.out_time - sw.in_time)
            print "switch :{}; delay : {} ; queue length: {}".format(sw.swid, sw.out_time, sw.in_time)
```

Observation

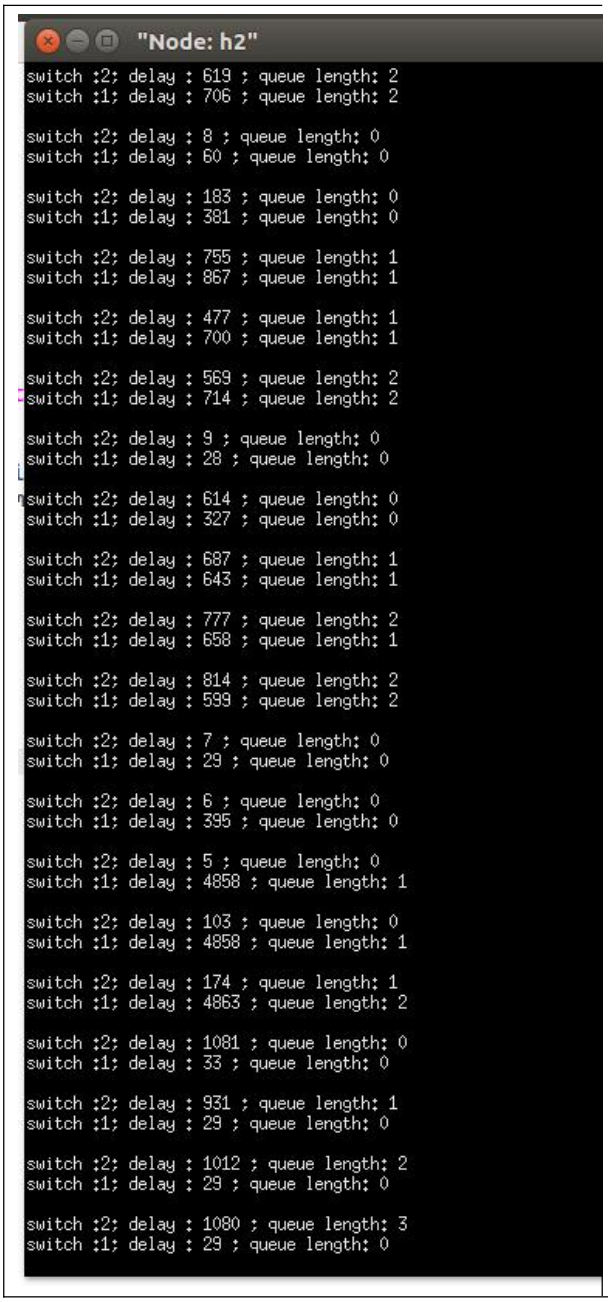
h1 and h2



h1 plays the role of sender and h2 is the receiver. Based on the queue utilization's header and receiver file in Python, h2 is able to visualize the queuing delay and queue depth of each switch .

Result

A piece of result from h2 has been randomly extracted.



delay of s1	delay of s2	queue length of s1	queue length of s2
706	619	2	2
60	8	0	0
381	183	0	0
867	755	1	1
700	477	1	1
714	569	2	2
28	9	0	0
327	614	0	0
643	687	1	1
658	777	1	2
599	814	2	2
29	7	0	0
395	6	0	0
4858	5	1	0
4858	103	1	0
4863	174	2	1
33	1081	0	0
29	931	0	1
29	1012	0	2
29	1080	0	3

(1) The queuing delay is supposed to be zero or close to zero if the queue depth is empty.

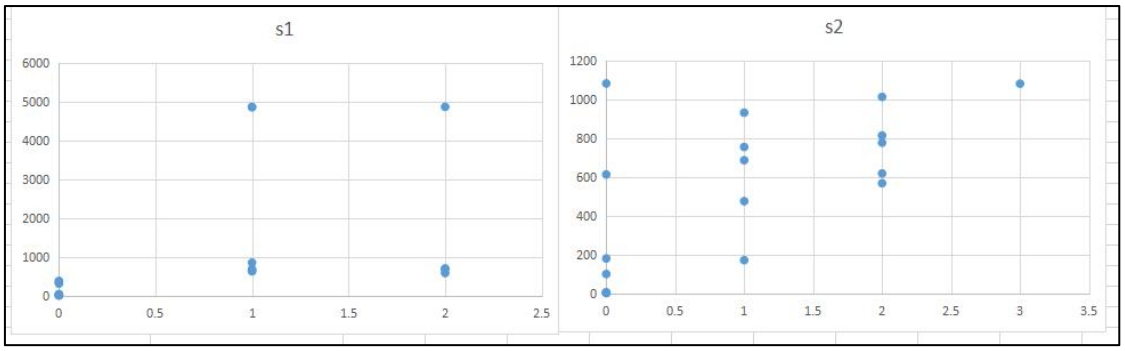
Form the table illustrated on the right, the queuing delays of both s1 and s2 is never zero while the queue length is empty, some of them are even quite huge value.

(2) The queuing delay should be stable since the queue depth has been a constant value.

Easy to see that basically all delays vary irregularly while the queue length is relatively stable. If we take a closer look at the table, s1 has some delays already more than 4800 us, which is extremely huge than the other delays measured.

(3) There should be a proportional relationship between these two parameters.

Either to check this point with the table illustrated before or with the following figure: the abscissa corresponds to queue length and the ordinate corresponds to queuing delay.



Their is no proportional relationship between queuing delay and queue length, one queuing delay sometimes has several queuing delay. This is totally illogical.

Conclusion

As described before, there are at least three apparent points to check according to the definition of queuing delay and queue depth in switch.

However, the verification for all threes facts are failed via P4 program by implementing a same topology to POX controller, the queuing delay measured is never ever 0 even though the queue depth is empty. The queuing delay never becomes stable since the queue depth has been

relatively constant. The worse thing is that there is no evidence can show there exists a proportional relationship between these two parameters. Therefore, we can conclude that the there is an apparent problem that queue length and queuing delay can not be visualized correctly via P4 language.

-Another conclusion can be made: Mininet supports both POX controller and P4 language, the incorrect queue length-visualization in P4 is thus not caused by Mininet since there is no problem in queue length visualization by POX controller.

Appendix E: Priority mechanism in P4

Headers added

```python
from scapy.all import *

TYPE_PROBE = 0x812
TYPE_VP =  0x1212;
class VP(Packet):
    name = "VP"
    fields_desc = [
       ShortField("prio", 0),
        ShortField("type", 0)]


class Probe(Packet):
   fields_desc = [ ByteField("hop_cnt", 0)]

class ProbeData(Packet):
   fields_desc = [ BitField("bos", 0, 1),
                   BitField("swid", 0, 7),
                   ByteField("port", 0),
                   IntField("byte_cnt", 0),
                   BitField("in_time", 0, 48),
                   BitField("out_time", 0, 48)]

class ProbeFwd(Packet):
   fields_desc = [ ByteField("egress_spec", 0)]


bind_layers(Ether, VP, type=TYPE_VP)
bind_layers(VP,Probe, type=TYPE_PROBE)
bind_layers(Probe, ProbeFwd, hop_cnt=0)
bind_layers(Probe, ProbeData)
bind_layers(ProbeData, ProbeData, bos=0)
bind_layers(ProbeData, ProbeFwd, bos=1)
bind_layers(ProbeFwd, ProbeFwd)
```

partie vlan

il faut aussi ajouter les bind_layers

The headers of Probe header and vlan header should be defined as header.py at the local folder.

The basic P4 codes should be modified to add the Vlan layer as well.

```
header vp_t {
    bit<16> prio;
    bit<16> proto_id;
}
```

```
struct headers {
    ethernet_t              ethernet;
    ipv4_t                  ipv4;
    vp_t                    vp;
    probe_t                 probe;
    probe_data_t[MAX_HOPS]  probe_data;
    probe_fwd_t[MAX_HOPS]   probe_fwd;
}
```

```
    transition parse_ethernet;
}

state parse_ethernet {
    packet.extract(hdr.ethernet);
    transition select(hdr.ethernet.etherType) {
        TYPE_IPV4: parse_ipv4;
        TYPE_VP: parse_vp;
        default: accept;
    }
}


state parse_ipv4 {
    packet.extract(hdr.ipv4);
    transition accept;
}

   state parse_vp {
    packet.extract(hdr.vp);
    transition select(hdr.vp.proto_id) {
        TYPE_PROBE: parse_probe;
        default: accept;
    }
}
```

```
/*************************************************************************
**********************  D E P A R S E R  ***********************************
*************************************************************************/

control MyDeparser(packet_out packet, in headers hdr) {
    apply {
        packet.emit(hdr.ethernet);
        packet.emit(hdr.ipv4);
        packet.emit(hdr.vp);
        packet.emit(hdr.probe);
        packet.emit(hdr.probe_data);
        packet.emit(hdr.probe_fwd);
    }
}
```

To send a packet containing the Vlan priority:

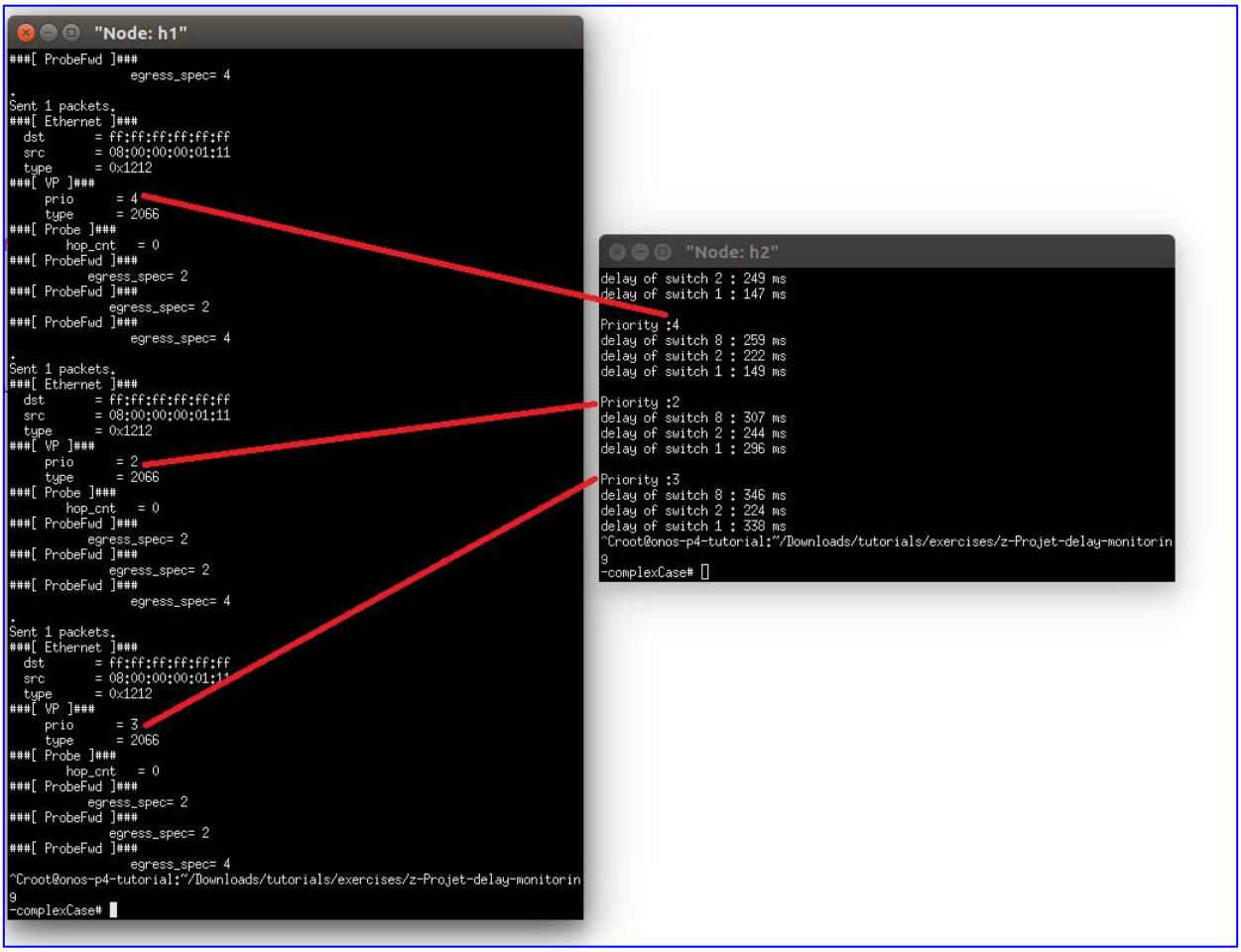
```
probe_pkt = Ether(dst='ff:ff:ff:ff:ff:ff', src=get_if_hwaddr('eth0'))/ \
    VP(prio=a)/ \
    Probe(hop_cnt=0) / \
    ProbeFwd(egress_spec=2) / \
    ProbeFwd(egress_spec=2) / \
    ProbeFwd(egress_spec=4)
```

The a is the priority given to the packet and it should be into [0,7].

To print the vlan.priority:

```
if VP in pkt:
    print ""
    dl = [l1 for l1 in expand(pkt) if l1.name=='VP']
    for sw1 in dl:
        print "Priority :{}".format(sw1.prio)
```

Implementation



As shown in this figure, the vlan priority has been successfully integrated into the packets sending and we can also receive the vlan priority from the destination host.

Attention, this vlan priority is not the real vlan priority, cause the switch would delete the vlan ticket , the Vlan part has been added manually.

Appendix F: P4 Code of delay measuring

File: /home/sdn/Downloads/tutorials⋯complexCase/delay_measuring.p4

```
/* -*- P4_16 -*- */ #include
<core.p4>          #include
<v1model.p4>

const bit<16> TYPE_IPV4 = 0x800; const
bit<16> TYPE_PROBE = 0x812; const bit<16>
TYPE_VP =  0x1212;

#define MAX_HOPS 10
#define MAX_PORTS 8

/*************************************************************************
********************** H E A D E R S  ***********************************
*************************************************************************/

typedef bit<9>   egressSpec_t;
typedef bit<48> macAddr_t; typedef
bit<32> ip4Addr_t;
typedef bit<48> time_t; header
ethernet_t {
     macAddr_t dstAddr;
     macAddr_t       srcAddr;
     bit<16>   etherType;
}

header vp_t {
     bit<16> prio; bit<16>
     proto_id;


}


header ipv4_t {
     bit<4>    version;
     bit<4>    ihl;
     bit<8>    diffserv;   bit<16>
               totalLen;   bit<16>
               identification;
     bit<3>    flags;
     bit<13>   fragOffset;
     bit<8>    ttl;
     bit<8>    protocol;
     bit<16>   hdrChecksum;
     ip4Addr_t         srcAddr;
     ip4Addr_t dstAddr;
}

// Top-level probe header, indicates how many hops this probe
//  packet  has  traversed  so  far.  header
probe_t {
     bit<8> hop_cnt;
}

// The data added to the probe by each switch at each hop. header
probe_data_t {
     bit<1>    bos;
     bit<7>    swid;
     bit<8>    port; bit<32>
               byte_cnt;
```

```
     time_t    in_time; // use for ingress_timestamp (in microseconds) time_t
               out_time;  //used fo egress_timestamp (in microseconds)
}

// Indicates the egress port the switch should send this probe
// packet out of. There is one of these headers for each hop. header
probe_fwd_t {
     bit<8>   egress_spec;
}

 struct       parser_metadata_t
     { bit<8>   remaining;
}

 struct  metadata  {  bit<8>
     egress_spec;
     parser_metadata_t parser_metadata;
}

struct headers {
     ethernet_t              ethernet;
     ipv4_t                  ipv4;
     vp_t                    vp;
     probe_t                 probe;
     probe_data_t[MAX_HOPS]  probe_data;
     probe_fwd_t[MAX_HOPS]   probe_fwd;
}

/*************************************************************************
********************** P A R S E R  ***********************************
*************************************************************************/

parser MyParser(packet_in packet,
                out  headers  hdr,  inout
                metadata meta,
                inout standard_metadata_t standard_metadata) {

     state start {
         transition parse_ethernet;
     }

     state                  parse_ethernet
         { packet.extract(hdr.ethernet);
         transition       select(hdr.ethernet.etherType)
             { TYPE_IPV4: parse_ipv4;
             TYPE_VP: parse_vp;
             default: accept;
         }
     }


     state                      parse_ipv4
         {       packet.extract(hdr.ipv4);
         transition accept;
     }

        state                     parse_vp
         { packet.extract(hdr.vp);
         transition       select(hdr.vp.proto_id)
             { TYPE_PROBE:  parse_probe;  default:
             accept;
         }

     }
```

```
    State                       parse_probe
        { packet.extract(hdr.probe);
        meta.parser_metadata.remaining  =  hdr.probe.hop_cnt  +  1;
        transition select(hdr.probe.hop_cnt) {
            0: parse_probe_fwd;
            default: parse_probe_data;
        }
    }

    state                                   parse_probe_data
        {  packet.extract(hdr.probe_data.next);  transition
        select(hdr.probe_data.last.bos) {
            1: parse_probe_fwd;
            default: parse_probe_data;
        }
    }

    state                                  parse_probe_fwd
        { packet.extract(hdr.probe_fwd.next);
        meta.parser_metadata.remaining = meta.parser_metadata.remaining - 1;
        // extract the forwarding data
        meta.egress_spec = hdr.probe_fwd.last.egress_spec; transition
        select(meta.parser_metadata.remaining) {
            0: accept;
            default: parse_probe_fwd;
        }
    }
}

/*************************************************************************
************    C H E C K S U M    V E R I F I C A T I O N    ************
*************************************************************************/

 control MyVerifyChecksum(inout headers hdr, inout metadata meta) { apply
    {        }
}


/*************************************************************************
**************   I N G R E S S   P R O C E S S I N G   *******************
*************************************************************************/

control MyIngress(inout headers hdr,
                  inout metadata meta,
                  inout standard_metadata_t standard_metadata) {

    action                              drop()
        { mark_to_drop(standard_metadata);
    }

    action   ipv4_forward(macAddr_t   dstAddr,   egressSpec_t   port)
        { standard_metadata.egress_spec = port; hdr.ethernet.srcAddr
        = hdr.ethernet.dstAddr; hdr.ethernet.dstAddr = dstAddr;
        hdr.ipv4.ttl = hdr.ipv4.ttl - 1;
    }

    table ipv4_lpm { key =
        {
            hdr.ipv4.dstAddr: lpm;

        }
```

```
        actions = {
            ipv4_forward;
            drop; NoAction;
        }
        size = 1024; default_action =
        drop();
    }

    apply {
        if      (hdr.ipv4.isValid())
            { ipv4_lpm.apply();
        }
        else if (hdr.probe.isValid()) {
            standard_metadata.egress_spec = (bit<9>)meta.egress_spec; hdr.probe.hop_cnt =
            hdr.probe.hop_cnt + 1;
        }
    }
}

/*************************************************************************
****************   E G R E S S   P R O C E S S I N G   *******************
*************************************************************************/

control MyEgress(inout headers hdr,
                 inout metadata meta,
                 inout standard_metadata_t standard_metadata) {

    // count the number of bytes seen since the last probe
    register<bit<32>>(MAX_PORTS) byte_cnt_reg;


    action      set_swid(bit<7>      swid)
        { hdr.probe_data[0].swid = swid;
    }

    table swid {
        actions = {
            set_swid;
            NoAction;
        }
        default_action = NoAction();
    }

    apply {
        bit<32> byte_cnt; bit<32>
        new_byte_cnt;
        // increment byte cnt for this packet's port byte_cnt_reg.read(byte_cnt,
        (bit<32>)standard_metadata.egress_port);    byte_cnt    =    byte_cnt    +
        standard_metadata.packet_length;
        // reset the byte count when a probe packet passes through new_byte_cnt
        = (hdr.probe.isValid()) ? 0 : byte_cnt;
        byte_cnt_reg.write((bit<32>)standard_metadata.egress_port, new_byte_cnt);

        if (hdr.probe.isValid()) {
            //    fill    out    probe    fields
            hdr.probe_data.push_front(1);
            hdr.probe_data[0].setValid();    if
            (hdr.probe.hop_cnt == 1) {
                hdr.probe_data[0].bos = 1;
            }
            else {
                hdr.probe_data[0].bos = 0;
            }
            //  set  switch  ID  field
            swid.apply();
            hdr.probe_data[0].port       =       (bit<8>)standard_metadata.egress_port;
            hdr.probe_data[0].byte_cnt = byte_cnt;
```

```
            // update the time stamps
            hdr.probe_data[0].in_time    =    standard_metadata.ingress_global_timestamp;
            hdr.probe_data[0].out_time = standard_metadata.egress_global_timestamp;
        }
    }
}

/*************************************************************************
*************    C H E C K S U M    C O M P U T A T I O N   **************
*************************************************************************/

 control MyComputeChecksum(inout headers  hdr, inout metadata meta) { apply
     {
        update_checksum( hdr.ipv4.i
            sValid(),
            {            hdr.ipv4.version,
              hdr.ipv4.ihl,
              hdr.ipv4.diffserv,
              hdr.ipv4.totalLen,
              hdr.ipv4.identification,
              hdr.ipv4.flags,
              hdr.ipv4.fragOffset,
              hdr.ipv4.ttl,
              hdr.ipv4.protocol,
              hdr.ipv4.srcAddr,
              hdr.ipv4.dstAddr },
            hdr.ipv4.hdrChecksum,
            HashAlgorithm.csum16);
    }
}

/*************************************************************************
***********************  D E P A R S E R  *******************************
*************************************************************************/

 control MyDeparser(packet_out packet, in headers hdr) { apply {
        packet.emit(hdr.ethernet);
        packet.emit(hdr.ipv4);
        packet.emit(hdr.vp);
        packet.emit(hdr.probe);
        packet.emit(hdr.probe_data);
        packet.emit(hdr.probe_fwd);
    }
}

/*************************************************************************
***********************  S W I T C H  *******************************
*************************************************************************/

V1Switch( MyParser()MyVe
rifyChecksum(),
MyIngress(),
MyEgress(),
MyComputeChecksum(),
MyDeparser()
) main;
```

Appendix G: Implementation of delay aware re-routing algorithm in P4

Receiver

We defined three rerouting cases, the packet is sent via the high/middle/low path.

```
if indi==1:
   calculateH()
elif indi==2:
      calculateM()
elif indi ==3:
      calculateL()
```

Each case will activate the corresponding function to calculate the average delay of them respectively the calculateH( ), calculateM( ) and calculateL( );

For example, in calculateH( ):

We determined a number (nbH) that how many of packets will be involved in average delay of flows calculating, basically the number of packets in the flows.

Then the average delay of this flow(packets) would be saved in the variable named totalDelayH. Then the function saveH( ) would be activated.

```
def calculateH():
    global totalDelayH
    global nbH
    global delay2
    if nbH <= (nbph-1):
       valuesH[nbH] = delay2
       nbH = nbH + 1
    elif nbH > (nbph-1):
         totalDelayH = (sum(valuesH)/nbph)
         valuesH[0]=delay2
         nbH = 1
         print "high: %.1f us" %(totalDelayH)
         saveH()
    delay2=0
```

All those average delays of flows would be saved in valuesHH, and nbHH is the number that how many averages will be involved in average delay of path calculating. So the size of valuesHH is nbHH. Than this average of path will be saved in variable totalDelayHH and be written into the txt file to let the program read the delay of path.

```
def saveH():
    global totalDelayH,totalDelayHH,nbHH,valuesHH
    if nbHH <=(nbs-1):
       valuesHH[nbHH] = totalDelayH
       nbHH = nbHH + 1
    elif nbHH > (nbs-1):
         for i in range(1):
             valuesHH.insert(len(valuesHH),valuesHH[0])
             valuesHH.remove(valuesHH[0])
             valuesHH[nbs-1] = totalDelayH
         totalDelayHH = sum(valuesHH)/nbs
         with open('h-s2.txt', "w+") as f:
              print >> f, int(totalDelayHH)
         print "avg h: %.1f us" %(totalDelayHH)
    totalDelayH=0
```

Example: we send flows containing 30(nbH) packets, the calculateH( ) allows to calculate the average delay of each 30 packets, so the totalDelayH. If we have enough average delay of each 30 packets, 20(nbHH) averages for example, we can determined the average delay of path by these averages of packets, so the totalDelayHH. All the delay measured will be saved respectively into the corresponding text file.

Sender

```
def high():

    probe_pkt = Ether(dst='ff:ff:ff:ff:ff:ff', src=get_if_hwaddr('eth0')) / \
                Probe(hop_cnt=0) / \
                ProbeFwd(egress_spec=2) / \
                ProbeFwd(egress_spec=2) / \
                ProbeFwd(egress_spec=5)



    sendp(probe_pkt, iface='eth0')
    time.sleep(0.050)


def middle():

    probe_pkt = Ether(dst='ff:ff:ff:ff:ff:ff', src=get_if_hwaddr('eth0')) / \
                Probe(hop_cnt=0) / \
                ProbeFwd(egress_spec=3) / \
                ProbeFwd(egress_spec=2) / \
                ProbeFwd(egress_spec=2) / \
                ProbeFwd(egress_spec=5)



    sendp(probe_pkt, iface='eth0')
    time.sleep(0.050)

def low():

    probe_pkt = Ether(dst='ff:ff:ff:ff:ff:ff', src=get_if_hwaddr('eth0')) / \
                Probe(hop_cnt=0) / \
                ProbeFwd(egress_spec=4) / \
                ProbeFwd(egress_spec=2) / \
                ProbeFwd(egress_spec=2) / \
                ProbeFwd(egress_spec=2) / \
                ProbeFwd(egress_spec=5)



    sendp(probe_pkt, iface='eth0')
    time.sleep(0.050)
```

As presented in section 4.1.1.2, the probe header has the numbered port and switch id variables which enables the routing of path. With the help of probe header, we can add the port to reroute at the end of packets. High(), middle() and low() function is capable of sending flows by indicating the out ports of switches.

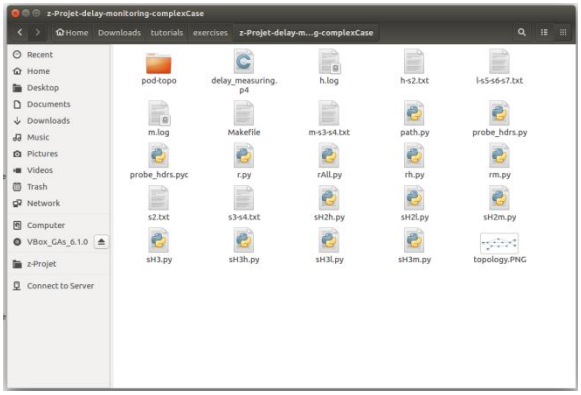
We should also read the current delays of high, middle and low paths then start the rerouting algorithm.

```
def main():
    while True:
        try:
            with open('h-s2.txt', "r+") as f1:
                read_data1 = f1.read()
                vh=float(read_data1)

            with open('m-s3-s4.txt', "r+") as f2:
                read_data2 = f2.read()
                vm=float(read_data2)

            with open('l-s5-s6-s7.txt', "r+") as f3:
                read_data3 = f3.read()
                vl=float(read_data3)


                global indi,hh,hl,mh,ml,lh,ll
                if vh > hh:
                   indi=1
                   if vm > mh:
                      indi=2
                   elif vm < ml:
                         indi=1
                   elif vm>=ml and vm <=mh:
                        if indi ==1:
                           indi=1
                        elif  indi ==2:
                             indi=2

                elif vh < hl:
                     indi=0

                elif vh >= hl and vh <= hh:
                     if indi ==0:
                        indi=0
                     elif indi==1:
                          indi=1


            if indi == 0:
               high()
            elif indi ==1:
                 middle()
            elif indi ==2:
                 low()
```

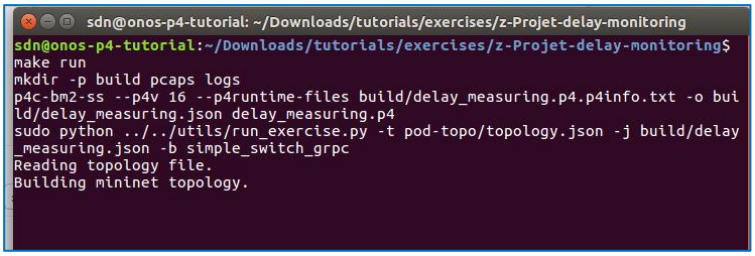
Implementation

Open the terminal in the local folder :

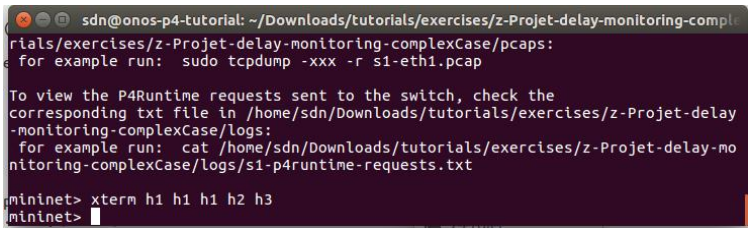
Open the terminal in the local folder :



Then run : make run



In Mininet run : xterm h1 h1 h1 h2 h3

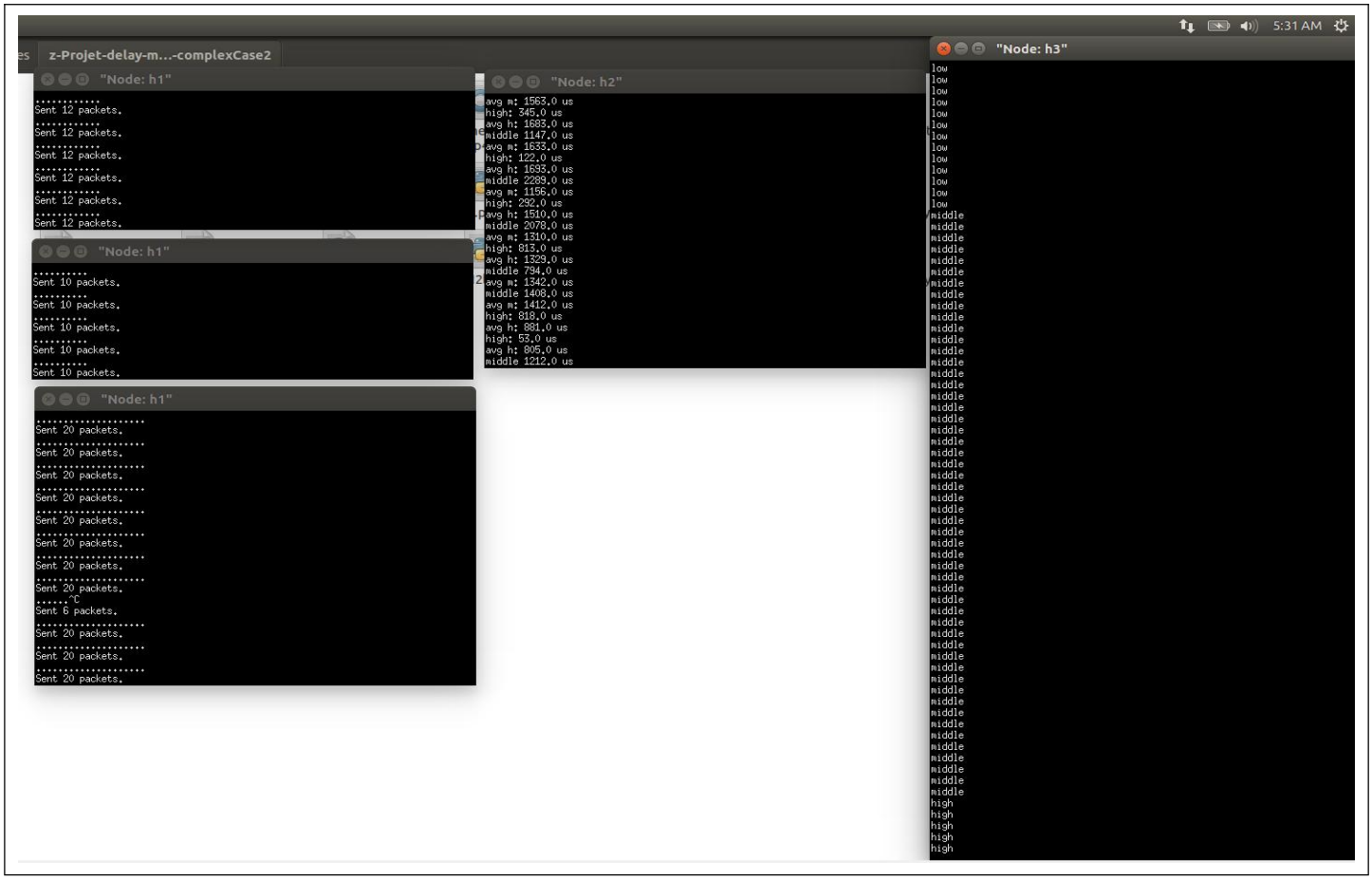


The first two h1 terminals will send flows to h2 via high and middle path, the third h1 terminals will send flows to h3 (initially the high path). The h2 will print the average of last 20 delays and the h3 will show which path that the flows h1-h3 pass，respectively ‘high’ ,‘middle’, ‘low’.

```
 6 hh= 1400.0
 7 hl = 1000.0
 8
 9 mh= hh
10 ml= hl
11
12 lh= hh
13 ll= hl
14
```

The thresholds of all paths are set up to the same values.

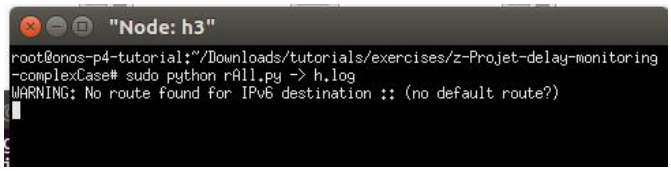
Run ./rAll.py on h2 print the average results of delay switch 2.
Run ./path.py on h3 terminals to print path of flows h1-h3.
Run ./sH2h.py in first h1 and ./sHhm.py in second h1.
Run ./sH3.py in last h1.
The flows h1-h2 via high path and middle will be generated during the whole implementation.
The flows h1-h3 can be added in ./sH3.py by doubling the send function.

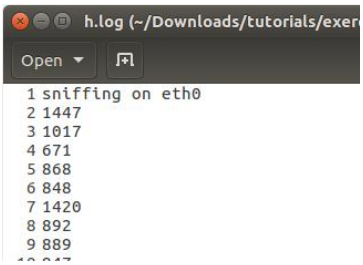


As shown in this figure, h1 generates flows to h2 constantly via high and middle paths, and the h2 terminal will print the average delays of packets and also print the average delay of each path. And h1 generates rerouting flows to h3, then the h3 terminal will indicate which path flows pass through.

All results can be saved running: file.py -> result.log.

For example, to save the result of high path in h3:





As shown at right, the values of delay can be found in the log you named. Then we can extract these values randomly (e.g., 100 values) to check if the result follows the scenario.

Appendix H: Delay measuring controller in POX

File: /home/sdn/Downloads/pox-eel/ext/measure_delay.py

```
from pox.core import core
from pox.lib.util import dpidToStr import
pox.openflow.libopenflow_01 as of
from pox.lib.addresses import IPAddr, EthAddr import
pox.lib.packet as pkt
from pox.openflow.of_json import * from
pox.lib.recoco import Timer import time
from pox.lib.packet.packet_base import packet_base from
pox.lib.packet.packet_utils import *
import struct

log = core.getLogger()

#global variables
start_time = 0.0
sent_time1=0.0
sent_time2=0.0
received_time1 = 0.0
received_time2 = 0.0
src_dpid=0 dst_dpid=0
mytimer = 0
T1=0.0 T2=0.0

#probe protocol, only timestamp field
class myproto(packet_base): "My
  Protocol packet struct"

  def__init__(self): packet_base.
     init_(self) self.timestamp=0

  def hdr(self, payload):
     return struct.pack('!I', self.timestamp)

def _handle_ConnectionDown (event):
  global mytimer
  print "ConnectionDown: ", dpidToStr(event.connection.dpid) mytimer.cancel()

def _handle_ConnectionUp (event):
  global src_dpid, dst_dpid, mytimer
  print "ConnectionUp: ", dpidToStr(event.connection.dpid)

  #remember the connection dpid for switch to controller (src_dpid) and switch1 to
controller(dst_dpid)
  for m in event.connection.features.ports:
if m.name == "s0-eth0":
      src_dpid = event.connection.dpid
    elif m.name == "s1-eth0": dst_dpid =
      event.connection.dpid

  # when the controller knows both src_dpid and dst_dpid, the probe packet is sent out every
2 seconds
  if src_dpid<>0 and dst_dpid<>0:
mytimer=Timer(10, _timer_func, recurring=True)
mytimer.start()

def _handle_portstats_received (event):
   global start_time, sent_time1, sent_time2, received_time1, received_time2,
```

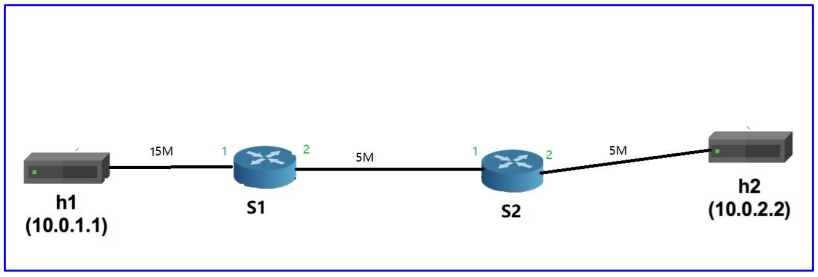
```
  src_dpid, dst_dpid,T1,T2

     received_time  =  time.time()  *  1000  -  start_time
     #measure T1
     if event.connection.dpid == src_dpid:
       T1=0.5*(received_time   -   sent_time1)
       #print "T1: ", T1, "ms"
     #measure T2
     elif   event.connection.dpid   ==   dst_dpid:
       T2=0.5*(received_time - sent_time1) #print
       "T2: ", T2, "ms"

  def   _handle_PacketIn   (event):
    global start_time,T1,T2 packet =
    event.parsed #print packet

    received_time = time.time() * 1000 - start_time
f packet.type==0x5577 and event.connection.dpid==dst_dpid: c=packet.find('ethernet').payload
 d,=struct.unpack('!I', c)
 print ""
      print  "delay{},  Tt:{},T1:{},T2:{},c:{}".format(  received_time  -  d  -  T1-T2,
  received_time,T1,T2,d,)

    a=packet.find('ipv4')
    b=packet.find('arp') if a:
      #print "IPv4 Packet:", packet msg
      = of.ofp_flow_mod() msg.priority
      =1
      msg.idle_timeout = 0
      msg.match.in_port =1 msg.match.dl_type=0x0800
      msg.actions.append(of.ofp_action_output(port

                                                = 2))
      event.connection.send(msg)

      msg = of.ofp_flow_mod()
      msg.priority =1
      msg.idle_timeout = 0
      msg.match.in_port =2 msg.match.dl_type=0x0800
      msg.actions.append(of.ofp_action_output(port

                                                = 1))
      event.connection.send(msg)

    if b and b.opcode==1:
 #print  "ARP  Request  Packet:",  packet  msg  =
 of.ofp_flow_mod()
 msg.priority =1
 msg.idle_timeout = 0
 msg.match.in_port =1 msg.match.dl_type=0x0806
 msg.actions.append(of.ofp_action_output(port = 2))
      if event.connection.dpid == src_dpid: #print
        "send              to              switch"
        event.connection.send(msg)
 elif event.connection.dpid == dst_dpid:
        #print      "send      to      switch1"
        event.connection.send(msg)

    if b and b.opcode==2:
 #print "ARP Reply Packet:", packet

 msg      =      of.ofp_flow_mod()
 msg.priority =1
 msg.idle_timeout = 0
 msg.match.in_port =2 msg.match.dl_type=0x0806
 msg.actions.append(of.ofp_action_output(port = 1))
```

```
      if event.connection.dpid == src_dpid: #print
        "send              to              switch"
        event.connection.send(msg)
elif event.connection.dpid == dst_dpid:
        #print      "send      to      switch1"
        event.connection.send(msg)

 def _timer_func ():
   global start_time, sent_time1, sent_time2, src_dpid, dst_dpid

   if src_dpid <>0:
sent_time1=time.time() * 1000 - start_time #print
"sent_time1:", sent_time1
#send         out         port_stats_request         packet         through         src_dpid
core.openflow.getConnection(src_dpid).send(of.ofp_stats_request
 (body=of.ofp_port_stats_request()))

f = myproto()
f.timestamp = int(time.time()*1000 - start_time) #print
f.timestamp
e = pkt.ethernet() e.src=EthAddr("0:0:0:0:0:2")
e.dst=EthAddr("0:1:0:0:0:1") e.type=0x5577
e.payload = f
msg = of.ofp_packet_out() msg.data =
e.pack()
msg.actions.append(of.ofp_action_output(port=2))
core.openflow.getConnection(src_dpid).send(msg)

   if dst_dpid <>0:
      sent_time2=time.time() * 1000 - start_time #print
      "sent_time2:", sent_time2
      #send        out        port_stats_request        packet        through        dst_dpid
      core.openflow.getConnection(dst_dpid).send(of.ofp_stats_request
 (body=of.ofp_port_stats_request()))

 def launch ():
   global start_time
   start_time = time.time() * 1000
   print "start_time:", start_time core.openflow.addListenerByName("ConnectionUp",
   _handle_ConnectionUp)        core.openflow.addListenerByName("ConnectionDown",
   _handle_ConnectionDown) core.openflow.addListenerByName("PortStatsReceived",
     _handle_portstats_received)
   core.openflow.addListenerByName("PacketIn",
_handle_PacketIn)
```

Appendix I: The queue length printing in POX

Introduction

The goal of this log is to test the queue length utilization by POX controller since the queue length has been visualized incorrectly in P4. The topology :



The bandwidth are all set to 15Mbps(h1-s1), 5Mbps(s1-s2) and 5Mbps(s2-h2).

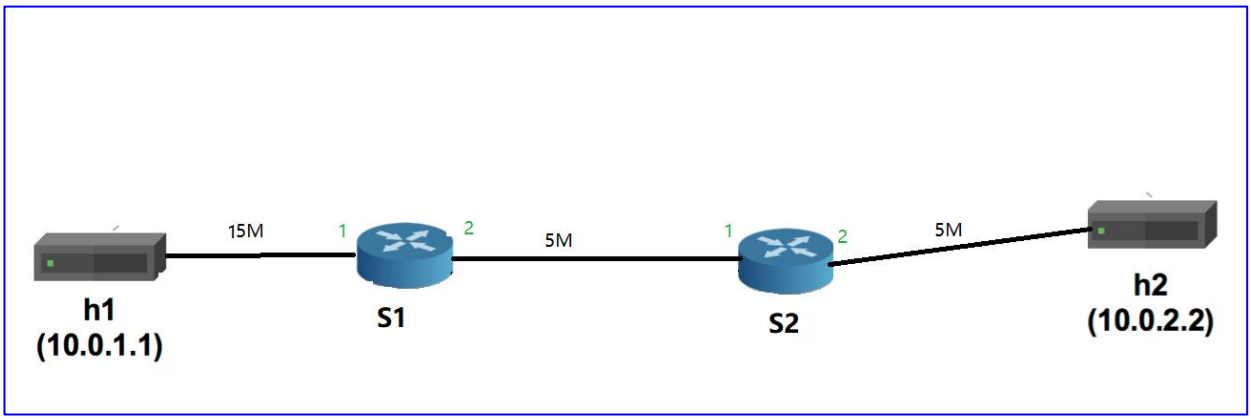
Large number of packets would be send from h1 to h2, so the port 2 of s1 should have some packets into the queue and the port 2 of s2 should have no packets in the queue cause the bandwidth of link s1-s2 and s2-h2 are the same.

Topology

The topology illustrated before has been built as follows :

```
#!/usr/bin/env python
from mininet.cli import CLI
from mininet.net import Mininet
from mininet.link import Link,TCLink,Intf
from mininet.node import RemoteController

if '__main__' == __name__:
  net = Mininet(link=TCLink)
  h1 = net.addHost('h1')
  h2 = net.addHost('h2')
  s1 = net.addSwitch('s1')
  s2 = net.addSwitch('s2')
  c0 = net.addController('c0', controller=RemoteController, ip='127.0.0.1', port=6633)

  linkopts0=dict(bw=15, delay='0ms', loss=0, max_queue_size=1000, use_tbf=True)
  linkopts1=dict(bw=5, delay='0ms', loss=0, max_queue_size=1000, use_tbf=True)
  linkopts2=dict(bw=5, delay='0ms', loss=0, max_queue_size=1000, use_tbf=True)

  net.addLink(h1, s1, cls=TCLink, **linkopts0)
  net.addLink(s1, s2, cls=TCLink, **linkopts2)
  net.addLink(s2, h2, cls=TCLink, **linkopts1)


  net.build()
  c0.start()
  s1.start([c0])
  s2.start([c0])

  CLI(net)
  net.stop()
```

```
sdn@onos-p4-tutorial: ~/Downloads/Mininet+Pox/queue length
sdn@onos-p4-tutorial:~/Downloads/Mininet+Pox/queue length$ sudo python topology.
py
mininet> links
h1-eth0<->s1-eth1 (OK OK)
s1-eth2<->s2-eth1 (OK OK)
s2-eth2<->h2-eth0 (OK OK)
mininet>
```



Observation

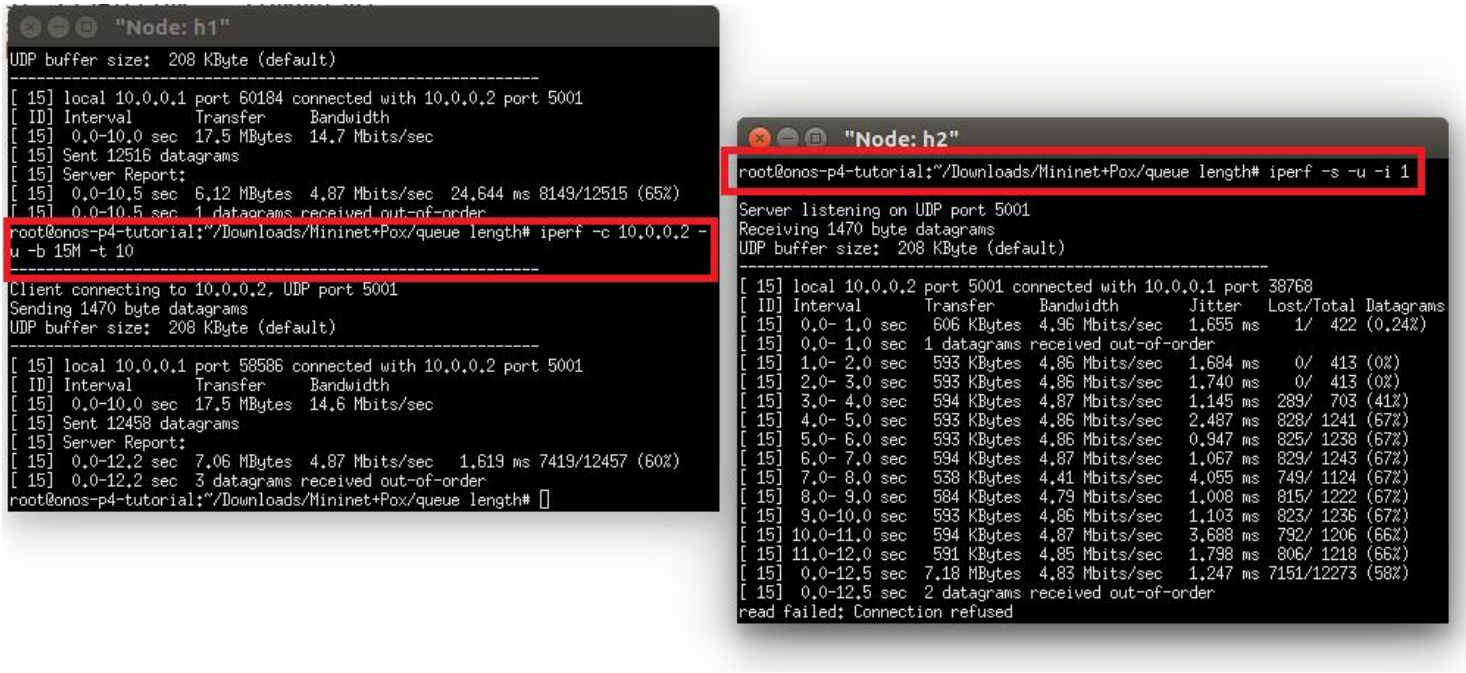
For visualize the queue length, run the command:

tc -s qdisc show dev s1-eth2

```
def monitor_qlens1(interval_sec=0.01):
  pat_queued =re.compile(r'backlog\s[^\s]+\s([\d]+)p')
  cmd = "tc -s qdisc show dev s1-eth2"
  ret = []
  open("qlens1.txt","w").write('')
  t0 = "%f" % time()
  while 1:
    p = Popen(cmd, shell=True, stdout=PIPE)
    output = p.stdout.read()
    matches = pat_queued.findall(output)
    if matches and len(matches) > 1:
      ret.append(matches[1])
      t1 = "%f" % time()
      #print str(float(t1)-float(t0)), matches[1]
      open("qlens1.txt","a").write(str(float(t1)-float(t0))+' '+matches[1]+'\n')
    sleep(interval_sec)
```

This python program allows to extract the queue length of s1 per 0.01 second, and to store the values into a file named “qlens1.txt”. The same to s2, the result will be illustrated in “qlens2.txt”

h1 and h2



As shown in this figure, the commands are respectively :

h1 : iperf -c 10.0.0.2 -u -b 15M -y 10

h2 : iperf -s -u -i 1

h1's command is able to test the network by sending iperf traffic with 15Mbps rate in UDP mode.

h2's command is able to report UDP data transmission every second.

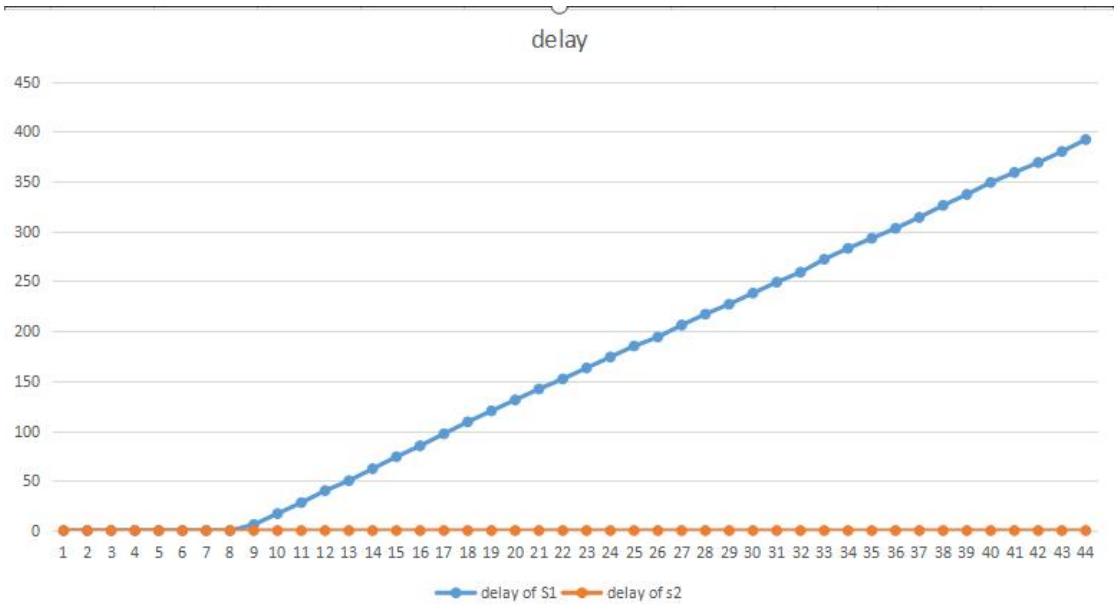
Result

```
519 6.9790430069 0
520 6.99190402031 0
521 7.00474095345 0
522 7.02026987076 0
523 7.03595280647 0
524 7.05556988716 0
525 7.07097196579 0
526 7.08358979225 0
527 7.09574794769 0
528 7.10821390152 0
529 7.1210539341 6
530 7.13343000412 17
531 7.14623498917 28
532 7.15907883644 40
533 7.1714630127 50
534 7.18642902374 62
535 7.19990491867 74
536 7.21285581589 85
537 7.22722792625 97
538 7.24005389214 109
539 7.25274181366 120
540 7.26535201073 131
541 7.27755093575 142
542 7.29011297226 152
543 7.30295395851 163
544 7.31595087051 174
545 7.32886791229 185
546 7.34122300148 194
547 7.35429787636 206
548 7.36790084839 217
549 7.38047981262 227
550 7.39324188232 238
551 7.40588498116 249
552 7.41857481003 259
553 7.43408489227 272
554 7.44695281982 283
555 7.45945000648 293
556 7.47230482101 303
557 7.48514795303 314
558 7.49986100197 326
559 7.51243591309 337
560 7.52684783936 349
561 7.53970885277 359
562 7.55216288567 369
563 7.56499695778 380
564 7.57963991165 392
```

s1 get some packets accumulated in the queue.

```
3705 51.5359561443 0
3706 51.5487031937 0
3707 51.5633511543 0
3708 51.5758590698 0
3709 51.5889279842 0
3710 51.6012461185 0
3711 51.6172761917 0
3712 51.630562067 0
3713 51.6433050632 0
3714 51.656167984 0
3715 51.6688010693 0
3716 51.6814889908 0
3717 51.6940681934 0
3718 51.7068610191 0
3719 51.7195501328 0
3720 51.7318232059 0
3721 51.7440299988 0
3722 51.7568340302 0
3723 51.7697470188 0
3724 51.7827250957 0
3725 51.7954850197 0
3726 51.8075680733 0
3727 51.8201980591 0
3728 51.8329629898 0
3729 51.8454670906 0
3730 51.8582379818 0
3731 51.8709471226 0
3732 51.883562088 0
3733 51.8963160515 0
3734 51.9094572067 0
3735 51.921683073 0
3736 51.9362130165 0
3737 51.9481780529 0
3738 51.9618780613 0
3739 51.9745841026 0
3740 51.9871640205 0
3741 51.9998061657 0
3742 52.0120370388 0
3743 52.0247399807 0
3744 52.0376961231 0
3745 52.0500950813 0
3746 52.0628211498 0
3747 52.0756120682 0
3748 52.088668108 0
3749 52.1008081436 0
3750 52.1134860516 0
3751 52.1265890598 0
3752 52.1389660835 0
```

The queue length of s2 is always zero



As shown in this figure, the delay of s2 remains at 0 whatever the delay of s1 varies.

Conclusion

As described in introduction, the s2 should have no packets accumulated in its queue cause the s1-s2 link and s2-h2 link has the same bandwidth. According to the results obtained, the s1 has a queue length more than 0, and the the queue length always remains at 0. Therefore, we can conclude that the queue length can be visualized correctly by POX controller.

-Another conclusion can be made : Mininet supports the POX controller and this log shows there is no problem in queue length visualization by POX controller, the incorrect queue length-visualization in P4 is thus not caused by Mininet.