Development of a performance measurement tool for SDN

Marion MAUGENDRE

Supervisor : Pere Barlet Ros,
UPC-Computer Architecture Departement

Co-Supervisor : Patrice Laurençot, ISIMA

Date : 07/10/2015
Acknowledgement

I sincerely express my gratitude to ISIMA for providing me this opportunity to do this exchange at the UPC which was very rich intellectually, culturally and socially.

I would like to thank my project director M. Pere Barlet Ros for all his efforts in helping and guiding me throughout the course of this project. And also Elavazhagan Sethuraman for his help with the POX controller program.

A thought for my grandparents, Roger and Jeanine, who died respectively in December 2014 and June 2015.
Abstract

Software-Defined Networking is becoming more and more present in network due to the complexity and difficulty to manage traditional network. This new paradigm aims to separate control plane and data plane for more network programmability, serviceability, heterogeneity and maintainability.

SDN is most of the time associated with OpenFlow protocols, especially for cloud and enterprise infrastructures. But latest specification of this protocol does not take care about the latency monitoring in the networks. Those delay measurement are needed to make correct routing decisions or to efficiently apply QoS policies.

The project proposed wants to develop a tool to measure link latency from a OpenFlow controller. It uses Mininet, a software to emulate SDN computer network. Emulation deals with the process of mimicking the internal entities to obtain more knowledge on the real-time environment of the emulation. Also an emulated network can be sure of implementing successfully in the real-time environment. The controller chosen is POX, writing in python and easy to manage.

The delay is measured with a packet probe sending by the controller and returning by the switch once it achieved its goal. The times measure is compared to those from the ping values.
Summary

Acknowledgement i
Abstract ii
Summary iii
Figures v
Glossaire vi
Introduction 1

1 Software-Defined Networking 5
1.1 History 5
1.1.1 Active Networking 5
1.1.2 Separating control and data planes 6
1.1.3 OpenFlow 6
1.2 Architecture of SDN 7
1.3 OpenFlow 9
1.3.1 Overview 10
1.3.2 OpenFlow switches 10
1.3.3 Flow tables 11
1.3.4 Packet flow through an OpenSwitch 12

2 Tools used for this project 17
2.1 Mininet 17
2.1.1 Command Line Interface 18
2.1.2 Basic topologies 21
2.1.3 Custom topologies 23
2.2 POX Controller 26
2.3 OpenFlow messages 27
2.3.1 Controller-to-switch 27
2.3.2 Asynchronous 27
2.3.3 Symmetric 28

3 Measuring delay 31
3.1 Monitoring mechanism 31
3.2 Implementation 32
3.3 Results ......................................................... 33
Conclusion ................................................. 37

Annexes .............................................. I
Annexe A ..................................................... II
Annexe B ..................................................... VII
Table des figures

1  Gantt diagram ........................................ 3
2  The SDN architecture ............................... 8
3  OpenFlow deployment in the USA academic campus .... 9
4  OpenFlow components ............................... 10
5  OpenFlow switch components .................... 11
6  Flow table entry in OpenFlow 1.0 ................. 11
7  Flow table entry in OpenFlow 1.5 ................. 12
8  OpenFlow pipeline .................................. 13
9  Simplified flowchart detailing packet flow through an OpenFlow switch ......................... 14
10 Example of Mininet Virtual Network ............... 18
11 Command Line Interface ........................... 19
12 Comparative bandwidth between nodes ............. 21
13 Topology minimal ................................... 21
14 Linear topology ..................................... 22
15 Tree topology ....................................... 23
16 Example of script ................................... 25
17 Simple flowcharts: hub, layer2 learning switch and layer3 learning switch .................... 26
18 Types of OpenFlow messages ...................... 28
19 Monitoring mechanism ............................. 32
20 POX and Mininet terminals ......................... 33
21 Difference between the monitoring delays and the ping values with initial delay of 10ms ........ 34
22 Difference between the monitoring delays and the ping values without initial delay .......... 35
23 Time difference between monitoring and ping values ............................................. 35
**Glossaire**

**API** : Application Programming Interface : a set of routines, protocols, and tools for building software applications.

**ICMP** : Internet Control Message Protocol : used by network devices, like routers, to send error messages indicating.

**Mininet** : a linux based emulation software for rapid prototyping Software-Defined Networks.

**OpenFlow** : most popular SDN technology. It proposes to standardize the communication between the switches and the software based controller.

**Ping** : a computer network administration software utility used to test the reachability of a host on an Internet Protocol (IP) network and to measure the round-trip time for messages sent from the originating host to a destination computer and back.

**POX** : an open source controller for developing SDN applications.

**SDN** : Software-Defined Networking : a new network architecture which separates the control plane from the data plane for more network programmability, serviceability, heterogeneity and maintainability.
Introduction

The traditional IP networks, based on distributed control and transport network protocols running inside the routers and switches, are complex and hard to manage. The network operators need to configure each individual network device to express the desired high-level network policies. Also, network environments have to endure the dynamics of faults and adapt to load changes.

The current networks are vertically integrated which enforce the complexity. The control plane (decides how to handle the traffic) and the data plane (forwards traffic according to the control plane) are bundled inside the networking devices, reducing flexibility and hindering innovation and evolution of the networking.

Software-Defined Networking (SDN) is an emerging networking paradigm that gives hope to change the limitations of current network infrastructures. It breaks the vertical integration by separating the control plane from the data plane. Thank to this, network switches become simple forwarding devices and the policy enforcement and network configuration are simplifying with the use of a logically centralized controller.

SDN is most of the time associated with OpenFlow protocol. But the problem is that there are currently no ways to dynamically obtain the latency in a OpenFlow network to efficiently apply QoS policies.

The topic of the master thesis is to develop a performance measurement tool for SDN. In this report, I will present you what is SDN and the OpenFlow protocol in a first part. Then the different tools used to develop the tool. And finally, I will discuss about the results obtain by monitoring the delay between switches and controllers.

The organisation of this work is describe with the Gantt diagram presented on the figure 1. The first weeks were dedicated to read papers.
about traffic monitoring and analysis to define a topic. Next, my work was to discover and understand SDN and OpenFlow and the different tools used (Mininet, POX, python). And then, the scripts were difficult to writing cause I had trouble to find how and where implement them.
Figure 1 – Gantt diagram
1 Software-Defined Networking

The traditional computer networks are complex, difficult to manage, built from a large number of network devices... All of those drawbacks show the needed to find a new way to facilitate network evolution. In this context, it appeared the idea of “programmable network” [1]. Software-Defined Networking is one of the solution developed. It separates the control plane (which decides how to handle the traffic) from the data plane (which forwards traffic according to decisions that the control plane makes). This characteristic expects a more simplify network management, but also enables innovation and evolution. But, SDN is not appeared suddenly, it is a part of a long way of efforts to make network more programmable.

1.1 History

Software-Defined Networking relies on past research on active networking and work on separating the control plane and the data plane, as for example in the telephony networks, where the separation is clearly used to simplify network management and the deployment of new services [2].

1.1.1 Active Networking

The first work which contributed to the current SDN is the active networking (between the mid-1990s and the early 2000s). It introduced programmable functions in the network to enable innovation. Two programming models have been proposed by the active networking community: the capsule model and the programmable router/switch model. The intellectual contribution of active networks to SDN are:

— programmable functions in the network to lower the barrier to innovation;
— network virtualization, and the ability to demultiplex to software programs based on packet headers;
— the vision of a unified architecture for middlebox orchestration.
1.1.2 Separating control and data planes

In the early 2000s, the idea to separate the control and data planes has been developed, and two innovations appeared: an open interface between the control and data planes, such as the ForCES (Forwarding and Control Element Separation), and a logically centralized control of the network. Those two innovations have an intellectual contribution to SDN which is:

— logically centralized control using an open interface to the data plane;
— distributed state management.

One of the project developed during this period was the 4D project [3]. It describes an architecture in 4 layers:

— the data plane
— the discovery plane
— the dissemination plane
— the decision plane

Also, the Ethan [4] project defined a new architecture for enterprise network. It is considered as the OpenFlow predecessor.

1.1.3 OpenFlow

In the mid-2000s, a group of researchers of Stanford created OpenFlow switches [5]. To enable the creation of many new control application, the design of controller platforms has quickly followed. The intellectual contributions are:

— generalizing network devices and functions;
— the vision of a network operating system;
— distributed state management techniques.

OpenFlow will be more detailed in the part 1.3.
The term “SDN” has been first used to describe Stanford’s OpenFlow project, but now the definition is expanded to include a much wider array of technologies.

All those innovations permitted the definition of a new paradigm for network architecture, called Software-Defined Networking, which refers to a network architecture where the forwarding state in the data plane is managed by a remote control plane decoupled from the former [6].

1.2 Architecture of SDN

According to the Open Networking Foundation, SDN is defined as “an emerging architecture that is dynamic, manageable, cost-effective, and adaptable, making it ideal for the high-bandwidth, dynamic nature of today’s applications” [7].

SDN can be defined as a network architecture with four pillars:
(a) the control plane and the data plane are decoupled,
(b) forwarding decisions are flow-based, instead of destination-based,
(c) control logic is moved to an external entity (SDN controller or Network Operating System),
(d) the network is programmable through software applications running on the top of the controller that interacts with the underlying data planes devices.

This architecture consists of three layers, as illustrated in the figure 2.
The data plane, also called infrastructure layer in the figure 2, comprises the forwarding elements. This plane is programmed and managed by the control plane (control layer). Finally, the application layer contains network applications for the network features, such as network management and traffic engineering, security and network access control, network testing, debugging and verification...

The application layer communicate with the control plane with the northbound interface. For that, it uses APIs. So network programming language are needed to ease and automate the configuration and management of the network. They have to respect three important aspects, according to [8]:

— the network programming language has to provide the means for querying the network state;
— the language must be able to express network policies that define the packet forwarding behavior;
— the reconfiguration of the network is a difficult task, especially with various network policies.
The southbound interface, which connects the control and data planes, uses most of the time the OpenFlow protocol.

1.3 OpenFlow

OpenFlow is the most popular SDN technology, and it is now standardized by the ONF [7]. It proposes to standardize the communication between the switches and the software-based controller [5]. Even if some people consider SDN and OpenFlow as synonyms, there are differences. Indeed, SDN consists of decoupling the control and the data planes, while OpenFlow describes how a software controller and a switch should communicate in an SDN architecture [9].

OpenFlow has been initially deployed in academic campuses. The first one was Stanford, where it has been developed, and today, at least nine universities in the USA have deployed it, as shown in the figure 3 [10]. But, industry is also more and more interested in this new technology, SDN with OpenFlow, to increase the functionality of the network of the network, and so reducing costs and hardware complexity.

Figure 3 – OpenFlow deployment in the USA academic campus
1.3.1 Overview

An OpenFlow architecture consists of three basic concepts, see figure 4:
— the network is built up by OpenFlow-compliant switches that compose the data plane;
— the control plane consists of one or more OpenFlow controllers;
— a secure control channel connects the switches with the control plane.

Figure 4 – OpenFlow components

1.3.2 OpenFlow switches

OpenFlow switches consist of one or more flow tables, a group table which perform packet lookups and forwarding, a meter table consists of meter entries, defining per-flow meters, one or more OpenFlow channel to an external controller, and port to forward flow entries. The components of a OpenFlow switch are illustrated in the figure 5.
1.3.3 Flow tables

Each flow table in the switch contains a set of flow entries. In the specification 1.0 [11], each of them consists of match fields, counters, and a set of instructions to apply to matching packets as illustrated in figure 6. The header fields describe to which packet this entry is applicable. The counters are reserved for collecting statistics about flow. The actions specify how packet of that flow are handled.

<table>
<thead>
<tr>
<th>Header Fields</th>
<th>Counters</th>
<th>Actions</th>
</tr>
</thead>
</table>

Figure 6 – Flow table entry in OpenFlow 1.0

Other components have been added in the next specifications, until the 1.5 [12], the last one dated of March 2015. As show in the figure 7, the header fields have been replaced by match fields which consist of the ingress
port and packet headers, and optionally other pipeline fields such as metadata specified by a previous table. Priority is matching precedence of the flow entry. Timeout is the maximum amount of time or idle time before flow is expired by the switch. Cookies opaque data value chosen by the controller. And flags alter the way flow entries are managed.

<table>
<thead>
<tr>
<th>Match Fields</th>
<th>Priority</th>
<th>Counters</th>
<th>Instructions</th>
<th>Timeouts</th>
<th>Cookie</th>
<th>Flags</th>
</tr>
</thead>
</table>

**Figure 7 – Flow table entry in OpenFlow 1.5**

### 1.3.4 Packet flow through an OpenSwitch

The figure 8 illustrates the packet processing in the OpenFlow pipeline. This processes in two stages, ingress processing and egress processing, which can be optional. The process always starts with the ingress processing at the first flow table. The packet is matched against the consecutive flow table from each of which the highest-priority matching flow table entry is selected. If a flow entry is found, the set of instruction of that flow entry is executed. Otherwise, if there is a table miss, its instruction are executed, or the packet is dropped.
The figure 9 summarizes the packet process through an OpenFlow switch. The process following by the packet is:

(a) the switch starts by performing a table lookup in the first flow table, and based on the pipeline processing, may perform table lookups in other flow tables.

(b) packet header fields are extracted and packet pipeline fields are retrieved.

(c) packet matches a flow entry if all the match fields of the flow entry are matching the corresponding header fields and pipeline fields from the packet.
Figure 9 – Simplified flowchart detailing packet flow through an OpenFlow switch

SDN is an architecture, combined with OpenFlow protocol, presents a new network paradigm easy to manage. Before to explain the work realises in this project, introduce the tools which permit to develop the pro-
gram. To emulate a network, it exists softwares, to create virtual network and test programs. The most famous is Mininet that can be associated with a remote controller, as POX. Before to explain the work realised in this project, the tools which permit to develop the program will be introduced.
2 Tools used for this project

2.1 Mininet

Implementing a SDN network is the real life a challenge because of the risks that can be involved. The topology can behave in a different way and it could be a great lost of time and costly. To avoid this, the better is to emulate the network.

Mininet is a Linux based emulation software for rapid prototyping Software-Defined Networks by using lightweight virtualization [13]. Some features of Mininet are to:

— Provide a simple and cheap way for testing networks for OpenFlow application development;
— Allow multiple researchers independently work on the same network topology;
— Allow the testing of a large and complex topology, without even the necessity of a physical network;
— Include tools to debug and run tests across the network;
— Support numerous topologies, and include a basic set of topologies;
— Provide simple Python API’s for creating and testing network.
The figure 10 shows an example of virtual network created by Mininet. It places host processes in network namespace and connecting them with virtual Ethernet pairs.

### 2.1.1 Command Line Interface

To control and manage the virtual network from a single console, Mininet includes a network-aware command line interface (CLI). To launch the CLI, the command is `sudo mn`. The example of the figure 11, creates a network with a single topology and 2 hosts.
The basics commands and their functionalities of the mininet environment are:

— **nodes**: lists all the nodes of the currently active mininet topology. Those nodes include controller, switches and hosts;

— **net**: displays all the links between the nodes in the currently invoked topology;

— **dump**: this command dumps information about all the nodes involved in the active topology. This provides the user with information such as IP address of the nodes and process identifier for each node;

— **sh**: this command is used to overcome the inabilities of the programmer to use the shell commands from the mininet environment. For instance, “clear” command cannot be interpreted by the mininet environment to clear the screen. It goes unrecognized as the command is not a local one. In such situations, the commands can be prefixed with “sh” to execute the command from the shell directly;

— **xterm**: this command provides independent terminal for a separate node in the topology. With this various tests can be done with the topology;
— **ping**: this command allows the nodes to ping between the nodes. Ping command is basically used to test the reachability of the nodes from one another. If we simply want to ping between nodes the following command helps: `host1 ping host2`. Irrespective of the number of packets ping command sends the packet one at a time. However using ping command with a specific number of packets can also be sent. This can be done by the command having the following syntax: `host1 ping –c number-of-packets host2`;

— **pingall**: unlike the previous command which pings between two nodes this command is used to ping between all the nodes in the topology. Each node pings all the other nodes in the topology one by one. This command is used to ensure the overall connectivity of the nodes in the topology and allows the programmer to make sure if the topology is configured in the intended way;

— **iperf**: iperf is actually a tool that is used to measure the network performance. It can measure both TCP and UDP bandwidth performance, as show in the figure 12, which compares the bandwidth if the nodes are used like hubs or like switch on a single topology which one switch and three hosts. Using iperf, a client-server connection can be created and the packets can be sent between them. In iperf, various detailed information about the packet such as the type of connection, bandwidth, port number, number of packets can be specified. Another advantageous possibility of iperf is that the time interval between two consecutive packets can be specified. The client node in iperf is connected to the iperf server by using the IP address of the server;
— info, debug, output: those commands are used to set the verbosity level. The default verbosity level is “info”. This verbosity level shows in the mininet window what is happening when the startup and tear down of network. The verbosity level “debug” provides the user with a detailed information. It displays all the packages invoked during the mininet. This level is helpful when the programmer wants to know what is happening when the mininet is invoked. For those who are just concerned with the output in the terminal and wants no additional information to be displayed there is a verbosity level called “output”. All these levels are passed as arguments in the commands with the following syntax: `sudo mn –v verbosity-level`.

2.1.2 Basic topologies

The default topology invoked when the command `sudo mn` or `sudo mn –topo minimal` is launch from the terminal, the topology creating is composed by one switch and two hosts, as shown with the figure 13.

![Figure 13 – Topology minimal](image-url)
The single topology is like the default one but the number of host can be selected. For example, the command `sudo mn –topo single,5` will create a topology with one switch connected to 5 hosts.

The linear topology os launch with `sudo mn –topo linear,4` and the network created is presented on the figure 14. It can be noticed that, such a command creates links between each switch to the nearest host. In addition to this the links are also created between the nearest switches among them.

![Linear topology diagram](image)

**Figure 14 – Linear topology**

But mininet command line topologies are not limited to simple linear topologies. Tree topologies can also be created using mininet. The tree topology command for mininet takes two arguments namely : depth and fanout. `sudo mn –topo tree, depth = 3, fanout = 2` will create the topology of the figure 15.
2.1.3 Custom topologies

The Python API allows to create custom topologies based on scripts. Various packet can be imported from mininet and directly used in python. The main important to create a new topology are:

- from mininet.net import Mininet
- from mininet.topo import Topo (or the type of topology that will be used: linear, single, tree...)

To add hosts and switches to the topology, the functions are:

```python
addHost("host name", mac = MAC Address, ip = IP Address, "inNamespace":True)
addSwitch("switch name", switch id, protocol)
```

The options, like the MAC or IP addresses, are optional and will be generated according to the default configuration of mininet.

Once the nodes are created, they must be connected. The links can be between hosts, switches and both. For that, the following command is used:
addLink(node1, node2, port no. of 1st node, port no. of 2nd node, delay, bandwidth)

There are two kinds of controller: local or remote. The remote controller is programmed as a separate module that contains the definitions and methods for how to control the entire network. To add a remote controller there is two possibilities. The first one is to import it with:

```
from pox import POX
addController("controller name", controller=POX)
```

where POX is the name of the module to launch the controller.

The second way, is to used the following command:

```
RemoteController("controller name", IP Address, port number)
addController("Remote Controller name")
```

where specifying the IP Address and port number of the controller integrates the controller with the rest of the nodes of mininet.

Specifying the necessary topology with switches, hosts, controllers and links does not mean that the custom topology has been deployed in mininet. So for deploying the topology several steps are to be followed. An object, called net most of the time, is created for Mininet class as following:

```
net = Mininet( topo, link, switch, autoSetMacs, build)
```

It contains the configuration of the topology. Once, the topology is built, the deployment is started with the command `net.start()` and `CLI(net)` to launch the Command Line Interface. `net.stop()` stops and deletes the topology.

An example of script to create a topology with 2 switches, 2 hosts and a remote controller, POX, is shown on the figure 16.
Figure 16 – Example of script

The controllers used can be anywhere on the real or simulated network. But if Mininet runs on a virtual machine, the controller could run inside the VM, natively on the host machine, or in the cloud.
2.2 POX Controller

The controller defines the nature of the SDN paradigm. It can be a local or a remote controller and programmable in different platforms (C++, Java, Python...). Some of the most popular are:

- NOX/POX
- OpenDayLight
- FloodLight

For this project, the more convenient is POX.

POX is an open source controller for developing SDN applications. It is a python based SDN controller that is inherited from the NOX controller [14]. It comes with three network devices: hub, layer2 learning switch and layer3 learning switch. The figure 17 shows simple flowcharts, from Python code in POX, corresponding to the three network devices of POX.

![Figure 17 – Simple flowcharts: hub, layer2 learning switch and layer3 learning switch](image)

POX allows to create your own network device. In a python file, save in the folder home/pox/ext, it is possible to write how the controller must
work. For example it can send probe packets through the network for re-
trieving delay.

2.3 OpenFlow messages

The controller configures and manages switches, receives events from
them and sends packet out trough the OpenFlow channel, an interface
that connects OpenFlow switch and OpenFlow controller, as illustrated
in the figure 5. The OpenFlow switch protocol supports three messages
types.

2.3.1 Controller-to-switch

Controller-to-switch messages are initiated by the controller and used
directly to manage or inspect the state of the switch. A response from the
switch may not be required. Those messages are :

— features : identity and basic capabilities of a switch,
— configuration : to set and query configuration parameters in the switch,
— modify-state : to manage state on the switch,
— read-state : to collect various information from the switch,
— packet-out : to send packets out of a specified port on the switch,
— barrier : they are request/reply messages use to ensure message de-
dendencies have been met or to receive notifications for completed
operation,
— role-request : to set the role of the OpenFlow channel, set the Control-
er ID, or query them,
— asynchronous-configuration : to set additional filter on the asynchro-
nous messages.

2.3.2 Asynchronous

Asynchronous messages are sent from the switch without a controller
soliciting. Those messages informing the controller are :

— packet-in : transfer the control of a packet,
— flow-removed: removal of a flow entry from a flow table,
— port-status: change on a port,
— role-status: change of the role of the controller,
— controller-status: the status of a OpenFlow channel changes,
— flow-monitor: change in a flow table:

2.3.3 Symmetric

Symmetric messages are sent without any solicitation from the controller and switch. They are:
— Hello: messages exchanged between the switch and controller upon connection startup,
— Echo: request/reply messages to verify the liveness of a controller-switch connection, and as well can be used to measure its latency or bandwidth,
— error: to notify a problem to the other side of the connection,
— experimenter: provide a standard way for OpenFlow switches to offer additional functionality within the OpenFlow message type space.

![Diagram of OpenFlow messages](image)

**Figure 18** – Types of OpenFlow messages
The figure 18 summarizes the main messages used to measure delay between the switch and controller.

The way to measuring the delay relies on those technologies. The POX controller includes the possibility to create your own controller and Mininet your own topology. With that, a packet probe is sent to measure the delay needed for a packet to go from one switch to the other.
3 Measuring delay

3.1 Monitoring mechanism

The current applications have several distributed components and need to communicate between them with the lower latency networks path to reduce their response times. Monitoring path latency is most of the time doing form the edge, it means that an ICMP requests (probes) are sent and the response time is measuring.

Some papers have proposed solutions as OpenNetMon [15], DevoFlow [16], OpenSketch [17], SLAM [18]...

The solution proposed here, monitors latency from inside the network. In other words, the information about path is captured directly from network devices. The main idea is to use the OpenFlow messages, presented in the part 2.3, in order to measure the delay between switches.

The first step is to create a packet which will be used as a probe. Then, the controller, with a PacketOut 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 PacketIn message to the controller in order to communicate the state of the packet. The figure 19 shows the mechanism.
The delay needed corresponds to the time $T_3$, meaning the time between the two switches. When the controller receives the PacketIn messages, the total time can be determined: \[ T_{total} = T_1 + T_2 + T_3 \].

$T_1 = 0.5 \times (T_b - T_a)$ where $T_a$ is the time when sending out ports-stats-request packet and $T_b$ is the time when receiving port-stats-received packet. The same method can be applied to get $T_2$. As a consequence:

$T_3 = T_{total} - T_1 - T_2$.

### 3.2 Implementation

As said in the part 2.2 about the POX controller, it is possible to implement its own program to the controller and launch it from a terminal with the command: 

`./pox.py your-file`

In this, the packet probe is created by defining its source, destination and payload with the port number and the timestamp. The packet probe is sent each 2 seconds. Each time that this packet will be detected on the network, the delay calculated will be printing on a file and on the terminal. If the packet detected is not the probe, it is forward.

The network is simulated with Mininet. It is composed of 2 Open vS-witches and 2 hosts, as on the figure 19. At the beginning, the delay bet-
ween the host and the switches is 1ms and between switches is 10ms. The host1 pings the host2 45 times. The delay between switch is increased to 50ms after 15s, and 200ms after 30s.

### 3.3 Results

![POX and Mininet terminals](image.png)

**Figure 20 – POX and Mininet terminals**

POX and Mininet are launched in two different terminals as we can see on the figure 20. POX on the left, show less measures of delay than Mininet on the right. This is due to it takes care only of the probe packets sent and then are sent every 2 seconds. The first ping is always higher than the next.
one because it is the first time that a packet browses the network and so the path is discovered.

The graphic 21 shows the difference between the monitoring delays and the ping values. We can see that at the beginning, they are pretty closed, but when the delay is 200ms, there is a gap between both. And after having repeated the experiment numerous times, it was the same result.

![Figure 21 – Difference between the monitoring delays and the ping values with initial delay of 10ms](image)

In a second experimentation, the delay between switches at the beginning is 0ms. After 15s, it becomes 10ms, after 30s it is of 30ms, after 45s 50ms and then after 60s it is of 20ms. I obtained the following graphic 22:
The values of the time difference are shown on the graphic 23. In the first times, ping is above the monitoring delay, but after 10ms, it changes. The time difference between both is no more than 14ms but it is still huge, and it reaches when the delay between switch is 50ms.
An explanation of this difference could be the calibration of the controller. As we can see on the graphic 22, the delay at the beginning should be closed to 0ms, and not to 2ms.
Conclusion

The main purpose of this project was to work SDN and OpenFlow instead of traditional networks. The first main step has been to discover and learn more about SDN and OpenFlow protocol, and also Mininet and POX. And the second one, was to see how to integrate a monitoring function directly on the controller.

This project show that it is possible to monitoring a SDN network to measure the delay. After, there are some improvements to bring on the program to considering it as efficient and reliable. For example, the controller’s calibration or the number of packet lost. The bandwidth could also been integrated to the monitoring. Moreover, the topology used on this work is linear with 2 switches and 2 hosts. It could be interesting to see what it happens with a topology more complexe.

Thus, the SDN environment has been emulated with Mininet, but it could have differences with a real network, even if Mininet certifies an emulation pretty closed to the reality.
References


Annexes
Annexe A

Script for POX controller
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.lib.util import dpid_to_str
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
from datetime import datetime
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
OWD1 = 0.0
OWD2 = 0.0
postfix = datetime.now().strftime("%Y%m%d%H%M%S")
f2 = open("delay\%s.csv" % postfix, "w")
f2.write("Type,Source,Destination,Delay\n")
f2.flush()

#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(self, event):
        global mytimer
        print "ConnectionDown: ", dpidToStr(event.connection.dpid)
        mytimer.cancel()
f2.close()

    def _handle_ConnectionUp(self, event):

III
global src_dpid, dst_dpid, mytimer

print "ConnectionUp: ", dpidToStr(event.connection.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

if src_dpid<>0 and dst_dpid<>0:
    mytimer=Timer(2, _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, OWD1, OWD2

    received_time = time.time() * 1000 - start_time

    #measure T1
    if event.connection.dpid == src_dpid:
        OWD1=0.5*(received_time - sent_time1)
        #print "OWD1: ", OWD1, "ms"
    #measure T2
    elif event.connection.dpid == dst_dpid:
        OWD2=0.5*(received_time - sent_time1)
        #print "OWD2: ", OWD2, "ms"

def _handle_PacketIn (event):
    global start_time, OWD1, OWD2

    packet = event.parsed
    #print packet
    received_time = time.time() * 1000 - start_time

    if packet.type==0x5577 and event.connection.dpid==dst_dpid:
        c=packet.find('ethernet').payload
        d,=struct.unpack('!I', c)
        print "delay:", received_time - d - OWD1-OWD2, "ms"
        f2.write("Packet-In,%s,%s,%s,\n%(src_dpid,dst_dpid,(received_time - d - OWD1-OWD2))
        f2.flush()

    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)
Annexe B

Script to create the topology for Mininet
#!/usr/bin/python

from mininet.net import Mininet
from mininet.node import Node
from mininet.link import TCLink
from mininet.log import setLogLevel, info
from threading import Timer
from mininet.util import quietRun
from time import sleep

def myNet(cname='controller', cargs='-v ptcp: '):
    """Create network from scratch using Open vSwitch.""
    info( "*** Creating nodes\n"
    )
    controller = Node( 'c0', inNamespace=False )
    switch = Node( 's0', inNamespace=False )
    switch1 = Node( 's1', inNamespace=False )
    h0 = Node( 'h0' )
    h1 = Node( 'h1' )

    info( "*** Creating links\n"
    )
    linkopts0 = dict(bw=100, delay='1ms', loss=0)
    linkopts1 = dict(bw=100, delay='10ms', loss=0)
    link0 = TCLink( h0, switch, **linkopts0)
    link1 = TCLink( switch, switch1, **linkopts1)
    link2 = TCLink( h1, switch1, **linkopts0)

    link0.intf2.setMAC("0:0:0:0:0:1")
    link1.intf1.setMAC("0:0:0:0:0:2")
    link1.intf2.setMAC("0:1:0:0:0:1")
    link2.intf2.setMAC("0:1:0:0:0:2")

    info( "*** Configuring hosts\n"
    )
    h0.setIP( '192.168.123.1/24' )
    h1.setIP( '192.168.123.2/24' )
    h0.setMAC("a:a:a:a:a:a")
    h1.setMAC("8:8:8:8:8:8")

    info( "*** Starting network using Open vSwitch\n"
    )
    switch.cmd( 'ovs-vsctl del-br dp0' )
    switch.cmd( 'ovs-vsctl add-br dp0' )
    switch1.cmd( 'ovs-vsctl del-br dp1' )
    switch1.cmd( 'ovs-vsctl add-br dp1' )

    controller.cmd( cname + ' ' + cargs + ' &' )
    for intf in switch.intfs.values():
        print intf
        print switch.cmd( 'ovs-vsctl add-port dp0 %s' % intf )
    for intf in switch1.intfs.values():
        print intf
        print switch1.cmd( 'ovs-vsctl add-port dp1 %s' % intf )

    switch.cmd( 'ovs-vsctl set-controller dp0 tcp:127.0.0.1:6633' )
    switch1.cmd( 'ovs-vsctl set-controller dp1 tcp:127.0.0.1:6633' )
while 'is_connected' not in quietRun('ovs-vsctl show'):
    sleep( 1 )
    info( '.' )

def cDelay1():
    switch.cmdPrint('ethtool -K s0-eth1 gro off')
    switch.cmdPrint('tc qdisc del dev s0-eth1 root')
    switch.cmdPrint('tc qdisc add dev s0-eth1 root handle 10: netem delay 50ms')
    switch1.cmdPrint('ethtool -K s1-eth0 gro off')
    switch1.cmdPrint('tc qdisc del dev s1-eth0 root')
    switch1.cmdPrint('tc qdisc add dev s1-eth0 root handle 10: netem delay 50ms')

def cDelay2():
    switch.cmdPrint('ethtool -K s0-eth1 gro off')
    switch.cmdPrint('tc qdisc del dev s0-eth1 root')
    switch.cmdPrint('tc qdisc add dev s0-eth1 root handle 10: netem delay 200ms')
    switch1.cmdPrint('ethtool -K s1-eth0 gro off')
    switch1.cmdPrint('tc qdisc del dev s1-eth0 root')
    switch1.cmdPrint('tc qdisc add dev s1-eth0 root handle 10: netem delay 50ms')

# 15 seconds later, the delay from switch to switch 1 will change to 50ms
    t1=Timer(15, cDelay1)
    t1.start()
# 30 seconds later, the delay from switch to switch 1 will change to 200ms
    t2=Timer(30, cDelay2)
    t2.start()

    #info( '*** Running test
    h0.cmdPrint( 'ping -i 1 -c 45 ' + h1.IP() )
    sleep( 1 )
    info( '*** Stopping network
    controller.cmd( 'kill %' + cname )
    switch.cmd( 'ovs-vsctl del-br dp0' )
    switch.deleteIntfs()
    switch1.cmd( 'ovs-vsctl del-br dp1' )
    switch1.deleteIntfs()

if __name__ == '__main__':
    setLogLevel( 'info' )
    info( '*** Scratch network demo (kernel datapath)
Mininet.init()
myNet()