Dynamic Traffic Diversion in SDN: Testbed vs Mininet

Robert Barrett, Andre Facey, Welile Nxumalo, Josh Rogers, Phil Vatcher and Marc St-Hilaire
School of Information Technology
Carleton University, Ottawa, Ontario, Canada
Email: marc_st_hilaire@carleton.ca

***Abstract*—In this paper, we first propose a simple Dynamic Traffic Diversion (DTD) algorithm for Software Defined Networks (SDN). After implementing the algorithm inside the controller, we then compare the results obtained under two different test environments: 1) a testbed using real Cisco equipment and 2) a network emulation using Mininet. From the results, we get two key messages. First, we can clearly see that dynamically diverting important traffic on a backup path will prevent packet loss and reduce jitter. Finally, the two test environments provide relatively similar results. The small differences could be explained by the early field trial image that was used on the Cisco equipment and by the many setting parameters that are available in both environments.**

***Index Terms*—SDN, testbed, Mininet, comparison, dynamic traffic diversion.**

I. Introduction

In traditional switching, the networking device consists of two planes; the control plane (and a management plane which is within the control plane) and the data plane. These two planes are predefined with the capabilities and functions set out by the vendor for the specific device model. In traditional switching, both planes are implemented in the firmware of routers and switches. The control plane provides high level control and signalling for the switch. It populates, prunes and updates the routing information base (i.e. the routing table) and provides information used to build the forwarding table. The data plane then forwards traffic according to the control plane logic. The traditional structure has proven to be limiting. The devices are bound by the capabilities of the software shipped with the hardware. Thus, there is no room for improvement without investing in upgraded hardware, which can become costly in the long run. To overcome these limitations, the concept of Software Defined Networks (SDN) was introduced.

Software Defined Networking is a new method of designing and managing networks that is not only dynamic, but also cost-effective. This makes SDN an ideal solution for today's bandwidth hungry applications. The basic concept behind SDN is simply to separate the network's control plane from the data plane. By separating the two, the control plane can then be managed by a central controller, bypassing any need for proprietary control of individual devices. The central controller then allows for direct programming of the network control plane creating an agile, centrally controlled network. SDN addresses the main concerns of traditional networking, which is how does one make a network think and react as one? The answer is the underlying model of controller-based networking. The centralized controller will have a complete view of the network, as well as knowledge of all available paths. As a result, the controller can calculate paths based on a number of factors, like traffic type and source/destination. Since SDN does not depend on proprietary software, programmers can write programs that allow for quick and dynamic optimization of network resources that can be controlled in the manner which suites the necessity. The network can be provisioned to become more scalable, adaptable and less time consuming to manage. The day when vendor dependence is no longer an issue is soon approaching.

SDN is a hot topic and a lot of research was done in several different networking fields. For example, SDN was used in the area of cellular networks [1], sensor networks [2], optical networks [3], [4], for the convergence of packet and circuit switched networks [5], for virtualization [6], etc. However, when evaluating a new algorithm or a new method relying on SDN, most researchers rely on a network emulator called Mininet [7]. Mininet is a widely used tool but how does it compare to a real SDN testbed implementation? Are the emulated results going to hold in a real implementation? This is one of the goals of this paper. After proposing and implementing a simple Dynamic Traffic Diversion (DTD) algorithm, we want to compare the results that are obtained with a real SDN testbed and with Mininet.

The remaining of this paper is organized as follow. Section II presents the design of the network topology and how it was implemented using 1) a real testbed and 2) Mininet. Then, Section III introduces the dynamic traffic diversion algorithm that was implemented within the controller. Finally, the results and analysis are presented in Section IV followed by the conclusion in Section V.

II. Design and Implementation

The design of the network topology was dictated by the number of switches that were available in the lab. Since only three SDN capable switches were available, we implemented the topology shown in Figure 1 in two different environments: 1) in a testbed using real Cisco equipment and 2) in Mininet. For testing purposes, we decided to have hosts connected to switches 1 and 3, and have two paths between them, which could be changed dynamically. The goal here was to represent a real world environment where there would be traffic from

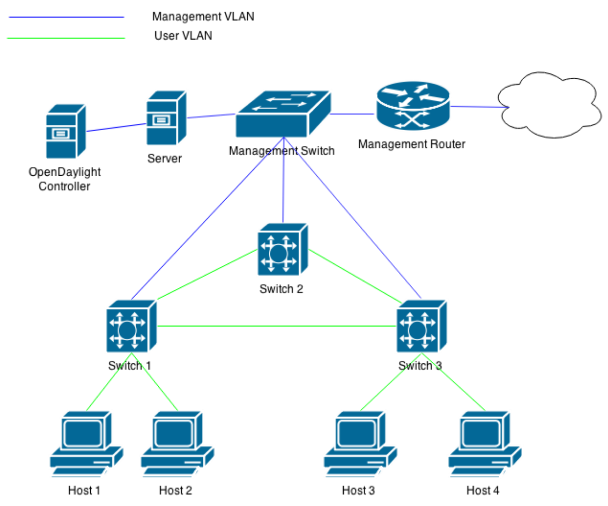

Fig. 1. Network topology used for testbed and Mininet

```
openflow
  switch 1
    of-port interface GigabitEthernet1/0/1
    of-port interface GigabitEthernet1/0/22
    of-port interface GigabitEthernet1/0/3
    of-port interface GigabitEthernet1/0/23
    of-port interface GigabitEthernet1/0/2
    of-port interface GigabitEthernet1/0/21
    of-port interface GigabitEthernet1/0/24
    pipeline 1
    protocol-version 1.0
    default-miss controller
    datapath-id 0x11
    controller ipv4 192.168.134.102 port 6633 security none
  exit-ofa-switch
exit
```

Fig. 2. Openflow configuration for switch 1 from Figure 1

different applications with different priorities. As an example, voice/video application traffic is well known to be delay intolerant compared to web of ftp traffic. If the main Internet connection for a company is congested, a second backup link will take the important traffic so the latency and jitter are kept to a minimum. In order to meet these criteria, it was determined that the optimal logical topology needed a link from one switch to each other switch, simulating a short path and a normally blocked redundant path to prevent looping. In the sections below, we describe each component in more details.

A. OpenDaylight Controller

Different open source controllers such as NOX [8] and POX [9] are available for many platforms and in many programming languages. However, for this work, OpenDaylight (hydrogen) [10] was selected. OpenDaylight is an open source controller platform implemented strictly in software and is contained within its own java virtual machine. Because it is built on Java, it can be used on any operating system that supports Java. We ran a virtual linux Ubuntu server to host our controller. The choice of controller was dependent mostly on the type of equipment we were using on the physical setup. While OpenDaylight worked seamlessly on our Mininet environment, we had to do a few tweaks to get it to work with our Cisco switched environment. Is it important to mention that the same controller was used to control the testbed and the Mininet topologies.

B. Testbed Environment

Despite the SDN popularity, it is still difficult to find SDN compatible hardware. To implement our testbed, we used three Cisco Catalyst 3650 switches all running an Early Field Trial (EFT) version of IOS-XE featuring Cisco's implementation of OpenFlow. After installing the Cisco plug-in for OpenFlow [11] on our Cisco 3650's, we initially configured OpenFlow [12] protocol 1.3 for functionality and communication between our switches and the controller. As per documentation, only protocol version 1.3 was to be supported, yet the controller had no communication with the switches. We ran a few tests and switched to version 1.0 which finally seemed to have connection to the Cisco switches which enabled the desired functionality.

After connecting the switches together, we needed to configure the switches to connect to the OpenDaylight controller. As an example, Figure 2 shows the OpenFlow configuration for switch 1. In addition, with our limits of traffic generation during testing, we set the bandwidth limit on all of the switch's ports to 100 Mbps.

Finally, each of the physical hosts were installed with Lubuntu as their operating system, as well with an additional NIC to provide access to both management and user VLANs. Also, on each host, Iperf [13] was installed for traffic generation, and gathering measurements for testing and analysis.

C. Mininet Environment

A pre-built Mininet virtual machine was used, which is available from mininet.org. This was useful as it meant that our configuration exactly matched the documentation from that website. The configuration of our Mininet topology can be seen in Figure 3.

III. Dynamic Traffic Diversion Application

Once the two environments were set up, we created a simple dynamic traffic diversion application. The goal behind this application is to be able to compare the results from the testbed with the results from Mininet. Figure 4 shows a high level overview of the controller application used to manage traffic in the network. At regular intervals, the application polls the transmitted bytes for that interval of the interface of the switch 1 that connects to switch 3, and turns it into a percentage of the maximum link capacity. If this value is more than a preset upper threshold, the link is considered to be congested, and the backup link takes precedence. If the value drops below a preset lower threshold, the link is considered to be no longer congested, and the main link takes precedence again.

We chose to use python to create our application, as much of our initial testing was done using cURL command line commands, and python was able to incorporate a lot of the same syntax. cURL was used to get the interface

```
from mininet.topo import Topo

class MyTopo( Topo):
  "Simple topology example."

  def __init__( self):
    "Create custom topo."

    #initialize topology
    Topo.__init__( self)

    # Add hosts and switches
    host1 = self.addHost( 'h1' )
    host2 = self.addHost( 'h2' )
    host3 = self.addHost( 'h3' )
    host4 = self.addHost( 'h4' )
    switch1 = self.addSwitch( 's1' )
    switch2 = self.addSwitch( 's2' )
    switch3 = self.addSwitch( 's3' )

    # Add links
    self.addLink( host1, switch1, bw=100 )
    self.addLink( host2, switch1, bw=100 )
    self.addLink( host3, switch3, bw=100 )
    self.addLink( host4, switch3, bw=100 )
    self.addLink( switch1, switch3, bw=100 )
    self.addLink( switch1, switch2, bw=100 )
    self.addLink( switch2, switch3, bw=100 )

topos = { 'mytopo': (lambda: MyTopo() )}
```

Fig. 3. Mininet topology configuration

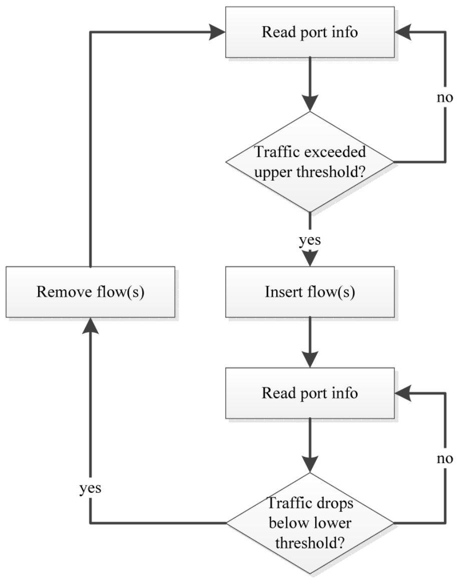
Read port info
no
Traffic exceeded upper threshold?
yes
Remove flow(s)
Insert flow(s)
Read port info
no
yes
Traffic drops below lower threshold?

Fig. 4. Flowchart of the dynamic traffic diversion application

information from an XML document provided and refreshed by the OpenDaylight Northbound REST interface. Another method to achieve the same goal would be to make a plug-in for OpenDaylight, so our program loaded as a part of the OpenDaylight controller. This was something we investigated, but did not pursue, as an external program was as efficient, and less complex.

IV. Testing and Analysis

In this section, we compare the Cisco testbed and Mininet in terms of packet loss and Jitter. Packet loss is the failure of transmitted packets arriving to their destination, while jitter is the measurement of the variance in time between packet delivery. With these measurements, we will be able to determine whether the dynamic traffic diversion application can decrease the delivery time and increase the rate of traffic if there was an alternative path to be taken. It will also allow us to compare the results between the two environments.

Three different scenarios are evaluated using the network shown in Figure 5. As can be seen, some traffic is categorized as background traffic in order to create congestion over the primary link (i.e. link between switch 1 and switch 3 and also referred to as path 1) and some traffic is categorized as important traffic, meaning that it should avoid congested paths.

To trigger the dynamic traffic diversion application, we set an upper threshold to mark path 1 as congested once the traffic reaches 90% of the link capacity (i.e. 90 Mbps). Once the amount of traffic crosses that threshold, flows will be created to diverge the marked traffic to travel across path 2 (i.e. switch 1 - switch 2 - switch 3) instead of path 1. A lower threshold of 70% of the link capacity (i.e 70 Mbps) is also set to trigger the dynamic traffic diversion application to remove the created flow to diverge the traffic back to the no longer congested path 1.

For each test, Host 1 (H1) was used to send traffic to Host 3 (H3), and Host 2 (H2) was used to send traffic to Host 4 (H4). The traffic between H2 and H4 was used only to create congestion on path 1, while the traffic between H1 and H3 was used to test the dynamic traffic diversion application. We used Iperf to generate the traffic needed to run the tests and gather the measurements of packet loss and jitter for analysis.

Finally, for each scenario discussed below, 10 independent tests were evaluated and the results of each test are displayed on the x-axis of each graph.

A. Scenario 1 - Baseline Testing

The objective in this scenario is to define a baseline for each environment when there is no congestion on path 1. To achieve that, H1 sends 600 MB of UDP traffic at a data rate of 50 Mbps to H3 through path 1. As there was no traffic on path 1 other than H1's, there was, as expected, no packet loss in both environments. Similarly, as shown in Figure 6, there was very low jitter with a mean of 0.0097ms, and 0.0081ms for the testbed and the Mininet environments respectively. We can conclude that the two environments perform similarly.

B. Scenario 2 - Performance without Dynamic Traffic Diversion

The objective of this scenario is to determine a baseline while there was congestion on path 1. To accomplish that, H1 sends 600 MB of UDP traffic at a data rate of 50 Mbps to H3 through path 1, while H2 congests path 1 with a large amount of UDP traffic (to keep the link congested over a long period of time) at a data rate of 95 Mbps. In this scenario, the dynamic traffic diversion application is not running meaning that all the traffic must go through path 1.

As expected, the traffic between H1 and H3 was greatly affected by the congestion traffic between H2 and H4. As

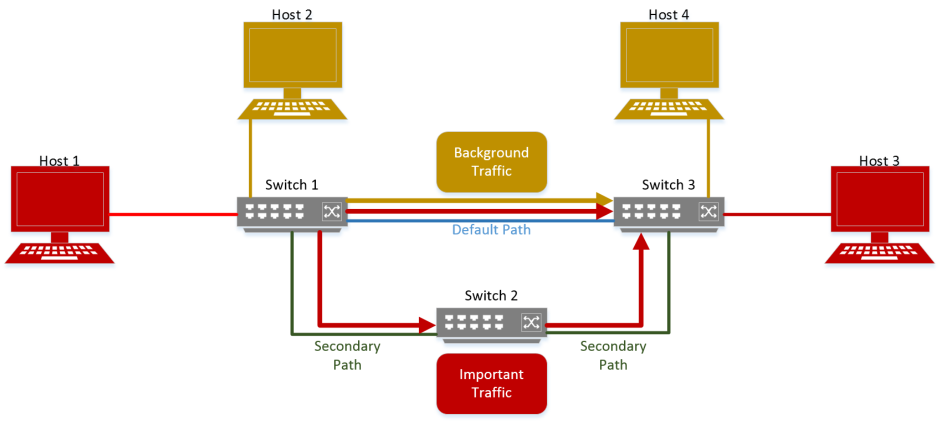

Fig. 5. Topology of the test environment for both testbed and Mininet networks

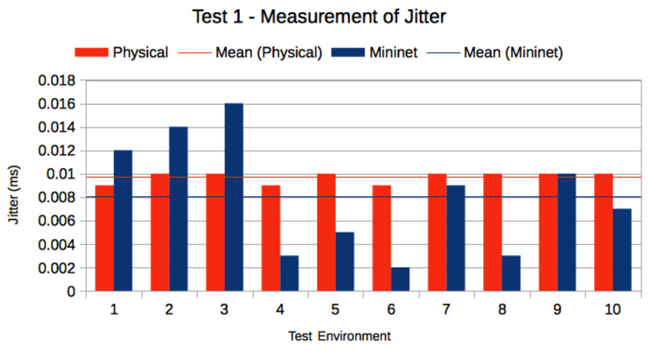

Fig. 6. Jitter measurements between testbed and Mininet for scenario 1

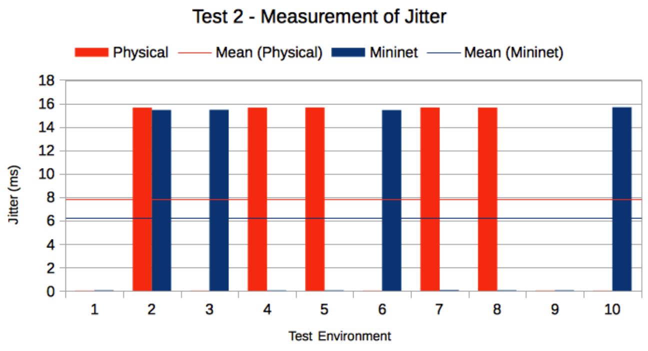

Fig. 8. Jitter measurements between testbed and Mininet for scenario 2

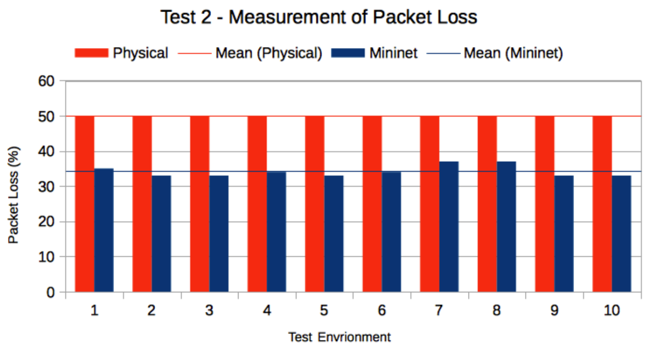

Fig. 7. Packet loss measurements between testbed and Mininet for scenario 2

shown in Figure 7 and Figure 8, the packet loss within the physical environment had a mean of 50% packet loss, and 7.829 ms of jitter, while the packet loss within the Mininet environment had a mean of 34% packet loss, and 6.2207 ms of jitter.

C. *Scenario 3 - Performance with Dynamic Traffic Diversion*

The purpose of this test was to determine whether the dynamic traffic diversion application will decrease the delivery time and increase the delivery rate of the marked traffic. In this scenario, H1 sends 600 MB of UDP traffic at a data rate of 50 Mbps to H3 through path 1, while H2 congested path 1 with a large amount of UDP traffic at a data rate of 95 Mbps with the DTD application running.

Our hypothesis of no packet loss in both environments was correct. As the congestion threshold was passed on path 1, and the DTD application changed the flow between H1 and H3 to pass through path 2 which has no congestion resulting in no packet loss. Also, as expected there was very low jitter in both environments during the test. However, the difference in jitter between the two environments was not expected. The lower jitter average of the Mininet environment compared to the physical environment is thought to be the result of Mininet being an integrated system.

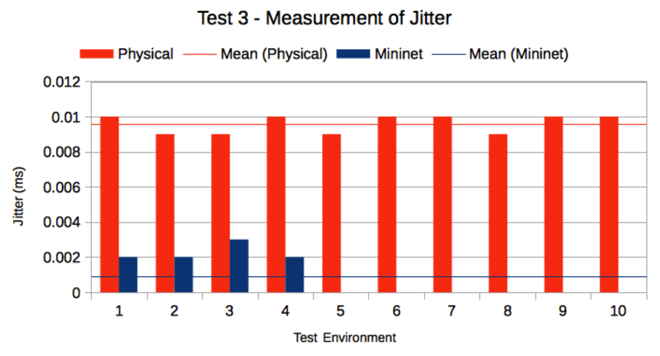

Fig. 9. Jitter measurements between testbed and Mininet for scenario 3

D. Overall Analysis

In comparison, our hypothesis was correct. Without the DTD application, the marked traffic experienced high packet loss and jitter, resulting in an increase in delivery time, and decrease in delivery rate. However, with the DTD application, the marked traffic decreased its delivery time and increased the delivery rate, with no packet loss, and low jitter. Furthermore, some of the results in the Mininet environment had better results than the physical environment which is believed to be caused by Mininet being an all-in-one box, and not having the traffic travel across actual physical links. Furthermore, a delay parameter may have been used to add on the Mininet links, but we did not have any way to accurately measure the delay of the physical links. Therefore, the algorithms for supporting real world testing are based within the implementation. Lastly, in comparison to the environments, from the results the Mininet environment is a suitable test environment if testing for scalability is an issue.

V. Conclusion and Future Work

In this paper, we came up with a software defined network that could dynamically divert traffic through a layer 2 network. This enabled us to have a basis for testing and comparing statistical differences between two different test environments: a physical Cisco network run off of beta software and an emulated Mininet network.

As expected and demonstrated by the three scenarios described above, dynamically deviating important traffic on a backup path can prevent packet loss and minimize jitter. By comparing the two environments, we were able to ascertain that the emulated Mininet network is suitable for testing purposes on the basis that all applicable metrics were encoded into the algorithms that produce the network performance statistics. The Mininet environment, excels at design and scalability testing. Different factors such as the early field trial image from Cisco and the various parameter settings available in both environments could explain the slight differences.

There are several areas that could see future research and have the potential for improvement. First, we want to do a deeper investigation into the sources of the similarities and discrepancies between the two approaches. Also, our program currently works with a fixed network topology, with all flow options predetermined. The future could see this program become more diverse, allowing for scalable networks and a wide range of traffic filters. Finally, we are also planning to work with more recent version of Openflow and Cisco IOS-XE.

References

[1] D. Venmani, D. Zeghlache, and Y. Gourhant, "Demystifying link congestion in 4G-LTE backhaul using openflow," in *5th International Conference on New Technologies, Mobility and Security (NTMS)*, 2012, pp. 1–8.

[2] T. Luo, H.-P. Tan, and T. Quek, "Sensor openflow: Enabling software-defined wireless sensor networks," *IEEE Communications Letters*, vol. 16, no. 11, pp. 1896–1899, 2012.

[3] S. Gringeri, N. Bitar, and T. Xia, "Extending software defined network principles to include optical transport," *IEEE Communications Letters*, vol. 51, no. 3, pp. 32–40, 2013.

[4] M. Shirazipour, W. John, J. Kempf, H. Green, and M. Tatipamula, "Realizing packet-optical integration with SDN and openflow 1.1 extensions," in *IEEE International Conference on Communications (ICC)*, 2012, pp. 6633–6637.

[5] S. Das, G. Parulkar, N. McKeown, P. Singh, D. Getachew, and L. Ong, "Packet and circuit network convergence with openflow," in *Conference on Optical Fiber Communication (OFC), collocated with the National Fiber Optic Engineers Conference, (NFOEC)*, 2010, pp. 1–3.

[6] R. D. Corin, M. Gerola, R. Riggio, F. D. Pellegrini, and E. Salvadori, "Vertigo: Network virtualization and beyond," in *European Workshop on Software Defined Networking (EWSDN)*, 2012, pp. 24–29.

[7] Mininet, Online: http://mininet.org/.

[8] 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 Communication Review*, vol. 38, no. 3, pp. 105–110, 2008.

[9] M. McCauley, NOXREPO, Online: http://www.noxrepo.org/.

[10] Linux Foundation, OpenDaylight, Online: http://www.opendaylight.org/.

[11] Cisco, Cisco Plug-in for OpenFlow Configuration Guide 1.1.5, San Jose, Cisco Press, 2014.

[12] N. McKeown, T. Anderson, H. Balakrishnan, G. Parulkar, L. Peterson, J. Rexford, S. Shenker, and J. Turner, "Openflow: enabling innovation in campus networks," *ACM SIGCOMM Computer Communication Review*, vol. 38, no. 2, pp. 69–74, 2008.

[13] Iperf - The TCP/UDP Bandwidth Measurement Tool, Online: http://sourceforge.net/projects/iperf2/.