Dynamic Traffic Diversion in SDN

Comparison of Physical and Mininet Environments

April 24, 2015

Robert Barrett	100870624
Welile Nxumalo	100745051
Andre Facey	100863246
Phil Vatcher	100873821
Josh Rogers	100859025

Table of Contents

1 Introduction

1.1 Motivation

Software-defined networking (SDN) is one of the cutting-edge developments in Enterprise networking. The opportunity to do research and familiarize ourselves with this concept will inevitably help us in our future careers. Potential job applicants with knowledge of SDN are much more appealing to potential employers, as they are well versed in up and coming technologies. With this opportunity to explore and implement an actual test environment, we were fully able to further enhance our understanding of this concept and it's applications.

1.2 Objective

It is our objective to demonstrate a working implementation of a Software-defined Network that takes advantage of Quality of Service style traffic tagging in order to dynamically prioritize traffic through redundant flows. We were able to build an external application that can detect traffic patterns on a given interface. The application updates the controller when specific thresholds are reached in order for marked packets to be given alternative flows through the network. The controller then updates the switches with the new instruction set. Our goal is to provide a mechanism for dynamic traffic diversion within a software-defined network that is both scalable and viable in a production network. We will then emulate this solution in a mininet environment in order to compare performance statistics and scalability.

1.3 Background Information

The following subsections contain background information on both traditional and Software-defined networking.

1.3.1 Traditional Switching

The idea of network switching is that it divides the network into many small broadcast domains in an attempt to control congestion and broadcast spamming within the network.

Switches allow the creation of dedicated paths between groups of users and their destinations. Each frame arriving at a port has a destination address field identifying where they are being sent to. The switch examines each frame's Destination Address field on each hop and forwards it only to the port which has been determined the closest via a table.

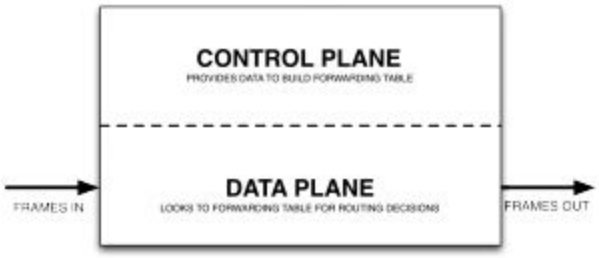

Figure 1: Traditional forwarding framework

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 (the routing table), it provides information used to build the forwarding table. The data plane then forwards traffic according to control plane logic.

The traditional structure has proven to be limiting. The devices are bound by the capabilities of 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.

1.3.2 Software Defined Networking

Software Defined Networking is a new method of designing and managing networks that is not only dynamic, but cost-effective as well. 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.

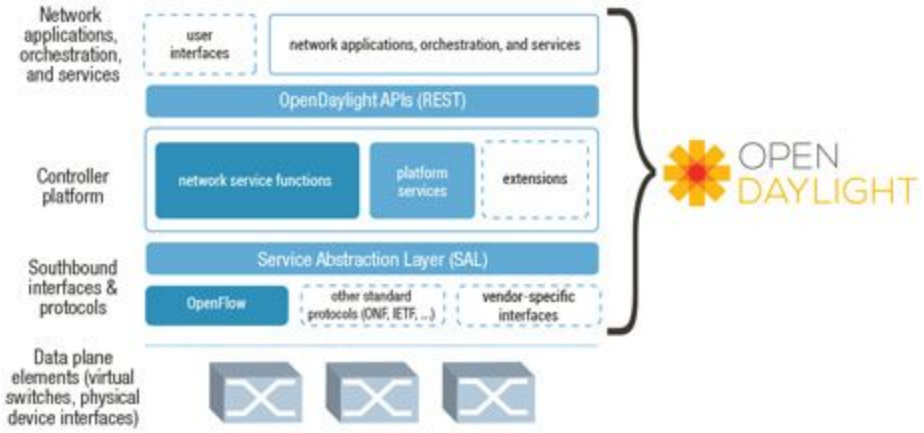

Figure 2: Basic OpenDaylight Framework [1]

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. You can provision the network to become more scalable, adaptable and less time consuming to manage. The day, when vendor dependence is no longer an issue, is soon approaching.

2 Design

In order to idealize a robust method of comparing the two test environments, we came up with a topology that could be easily recreated. With mininet, we had the option of using multiple switches, however in order to match our physical topology we chose to use 3 switches within mininet. We decided to have hosts connected to switches 1 and 3, and have two paths between them, which we could change dynamically.

The goal here was to represent a real world environment where there would be some traffic which was more important than other traffic. An example of this would be voice/video

application traffic, which are intolerant of delay. If the main internet connection for a company is congested, a second backup link will take the important traffic so the latency and jitter is kept to a minimum.

In order to meet these criteria, it was determined 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. The initial design had placed ‘hosts’ on each switch, though it was determined to be ineffective for testing and comparing purposes. Thus the design of two hosts on switches 1 and 3 was realized, as can be seen in Figure 2.1.

We chose to send several groups of traffic between our hosts. Some traffic would be categorised as background traffic in order to create congestion over the link. Some traffic would be categorised as unimportant traffic, which should see increased latency and jitter. Finally some traffic would be categorised as important traffic, which should see much less latency and jitter than the unimportant traffic.

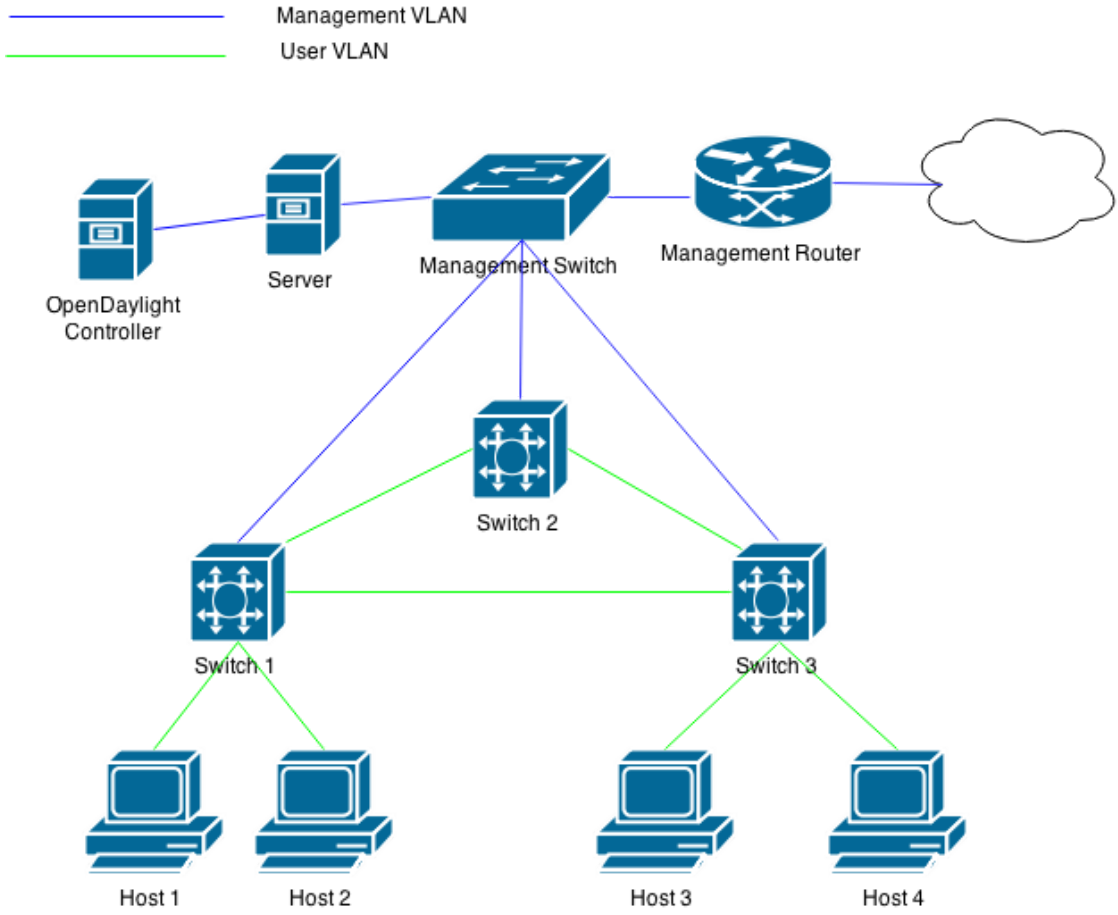

Figure 2.1: Detailed Network Topology

3 Implementation

For the server, we installed Lubuntu as the operation system, and installed Virtualbox, and Python. Virtualbox was used to host our virtual machines of Mininet, and our fourth host for our physical environment.

We created two VLANs, a management VLAN, and a user VLAN. The management VLAN which was accessible by all hosts and switches was used mainly for communication between the physical switches and the controller, as well as for remote access purposes. The user VLAN was used for the implementation and testing of the DTD application, its scope limited to ports controlled by openflow.

3.1 OpenDaylight Controller

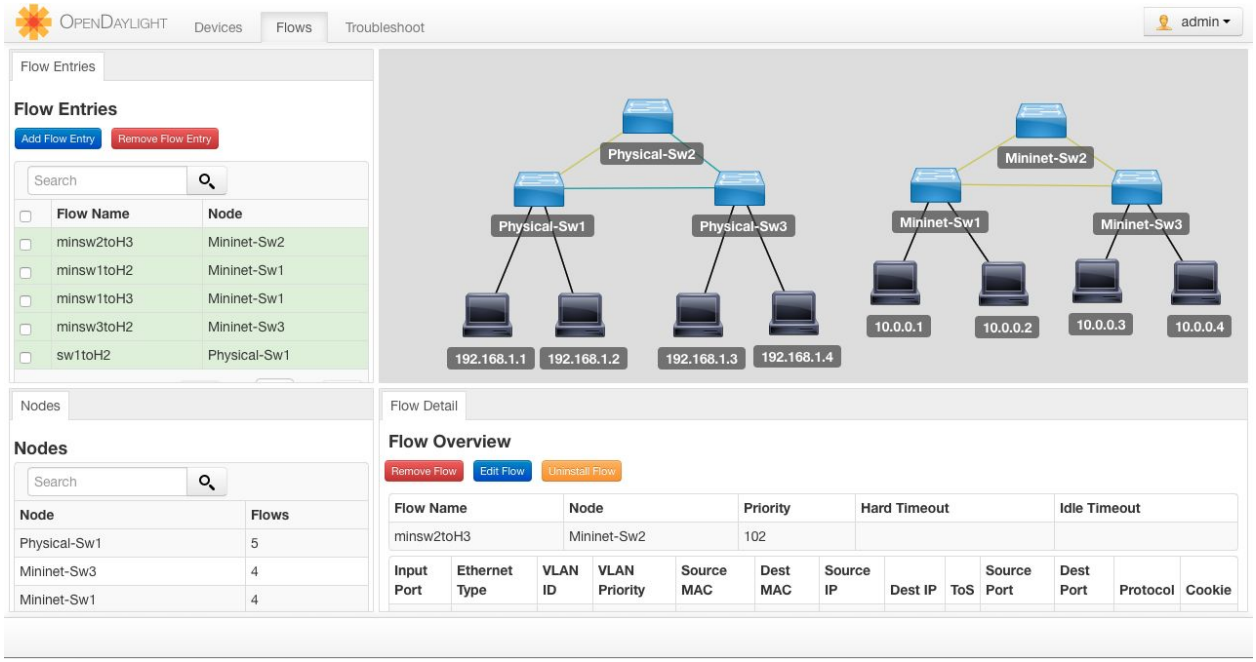

Figure 3.1 Main OpenDaylight Web UI with topology example

Background

OpenDaylight is an open source controller platform implemented strictly in software and is contained within it's own java virtual machine. Because it is built on Java, we had the flexibility of using any operating system that supported Java. We chose an Ubuntu platform as it allowed us to minimize resource allocation on our server.

We ran a virtual linux Ubuntu server to host our controller as well as on a mininet environment. 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.

Setup

After installing the Cisco plug-in for openflow on our Cisco 3650's, we initially configured OpenFlow 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.

Configuration

We downloaded and extracted the OpenDaylight controller zip file from our documentation package using the following series of commands:

```
wget https://github.com/opendaylight/controller/archive/stable/hydrogen.zip
unzip hydrogen.zip
cd controller-stable-hydrogen/opendaylight/distribution/opendaylight/
mvn clean install
cd ~
./controller-stable-hydrogen/opendaylight/distribution/opendaylight/target/distribution.opendaylight-osgipackage/opendaylight/run.sh
```

Upon completion of the installation of OpenDaylight we were able to use the web interface by navigating to **http://** 192.168.134.102**:8082** and externally through **http://** 134.117.92.76**:8082.**

The web interface is illustrated on figure 3.1 above. We were able to configure and view flows that were running in the controller. We were able to also have a view of the topology based on what was connected as seen in figure 3.1 which was useful in learning and understanding of what our flows were doing.

3.2 Physical Environment

To implement the physical testbed environment, we used three switches,three physical hosts, and one virtual host.

The three switches were Cisco Catalyst 3650 switches installed with an early field trial (EFT) image that enabled SDN capabilities to the switches. Furthermore, we needed to configure the switches to connect to the OpenDaylight controller using the configuration shown in Appendix 9.5. In addition, with our limits of traffic generation during testing, we set the bandwidth limit on all of the switch's ports to 100Mbps.

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 was installed for traffic generation, and gathering measurements for testing and analysis.

Configuration for each of the switches can be seen in the Appendix 9.5.

3.3 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 the Appendix. In general, mininet configuration was straightforward, however we did encounter an issue with the links between mininet switches were not obeying the bandwidth constraints we set in mininet. We discovered the issue here is that Open V Switch isn't aware of specifics of mininet. We had planned to run several other tests to compare the two platforms, however this issue made the results of the tests inconclusive.

3.4 Dynamic Traffic Diversion Application

Figure 3.4.1 below shows a high level overview of the controller application used to manage traffic in our 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.

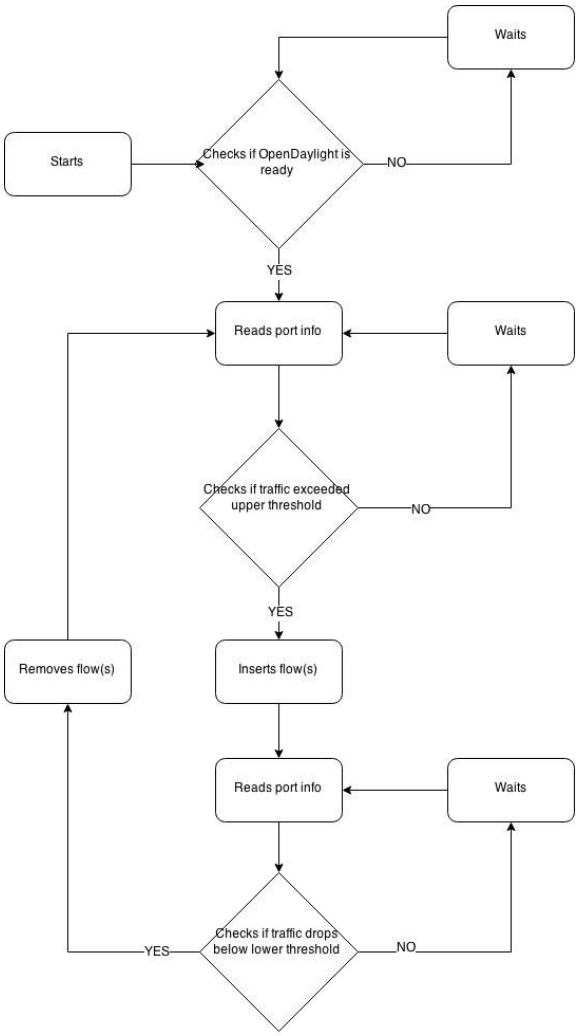

Figure 3.4.1: Flow chart of Dynamic Traffic Diversion Application.

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 information from an XML document provided and refreshed by the OpenDaylight Northbound REST interface.

An example of a statement to get the interface values would be:

```
curl -u admin:admin -H 'Accept: application/xml'
'http://localhost:8080/controller/nb/v2/flowprogrammer/default'
```

An example of a statement to insert a new flow would be

```
curl -u admin:project -H 'Content-type: application/json' -X PUT -d '{"installInHw":"true",
"name":"minsw1toH3-bak", "node": {"id":"00:00:00:00:00:00:00:01", "type":"OF"}, "etherType":
"0x800", "nwDst": "10.0.0.3", "priority":"152","actions":["OUTPUT=4"]}'
'http://localhost:8080/controller/nb/v2/flowprogrammer/default/node/OF/00:00:00:00:00:00:00:01
/staticFlow/minsw1toH3-bak'
```

Another method to achieve the same goal would be to make a plugin 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.

The code for our project is available here:
https://github.com/SDN-Network-Technology-Project/controller

4 Testing and Analysis

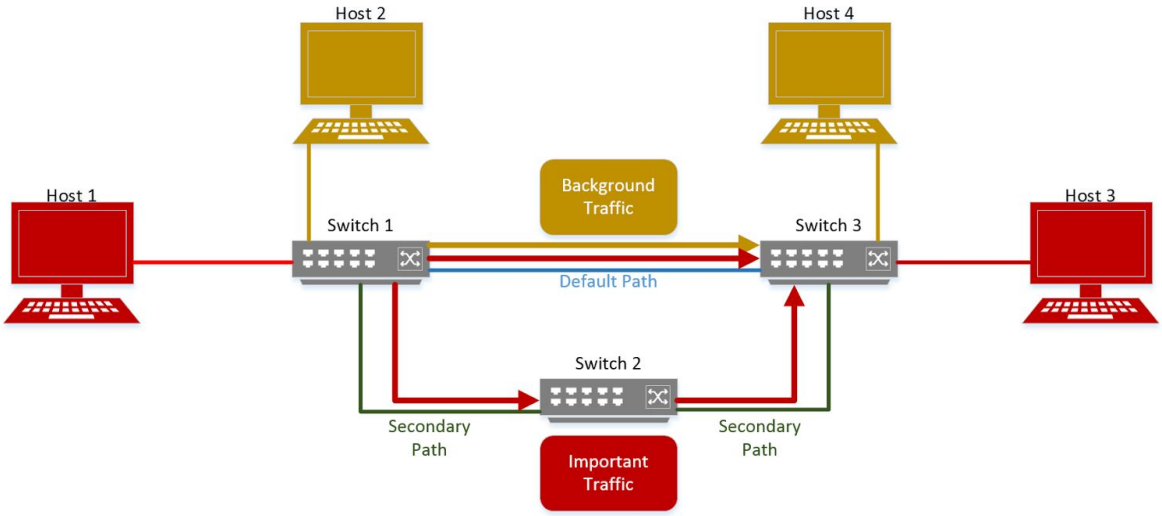

Figure 4.1: Topology of Test Environment for Both Physical and Mininet Networks

For the following tests, we set an upper threshold to mark Path 1 as congested once it has reached 90% capacity (90 Mbps) to trigger the DTD application to create flows that diverge the marked traffic to travel across Path 2 instead of Path 1. A lower threshold of 70% link capacity (70Mbps) is also set to trigger the DTD application to remove the created flow to diverge the traffic back to the no longer congested Path 1.

For each test, Host 1 was used to send traffic to Host 3, and Host 2 was used to send traffic to Host 4. The traffic between Host 2 and Host 4 was used only to create congestion on Path 1, while the traffic between Host 1 and Host 3 was used to test our DTD application. We used the traffic generation tool Iperf, to generate the traffic needed to run the tests, as well as gather the measurements of packet loss and jitter for analysis.

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 were able to determine whether our DTD application could decrease the delivery time and increase the rate of traffic if there was an alternative path to be taken.

Each test was repeated 10 times.

Test 1 - Baseline Testing

Scenario

Host 1 (H1) sent 600MB of UDP traffic at a data rate of 50Mbps to Host 3 (H3) through Path 1. The objective of this test was to use the statistics of the packet loss and jitter of the traffic to define a baseline for each environment with no congestion on Path 1.

Hypothesis

The traffic between H1 and H3 through Path 1 with no other traffic will result in no packet loss, and very low jitter.

Results

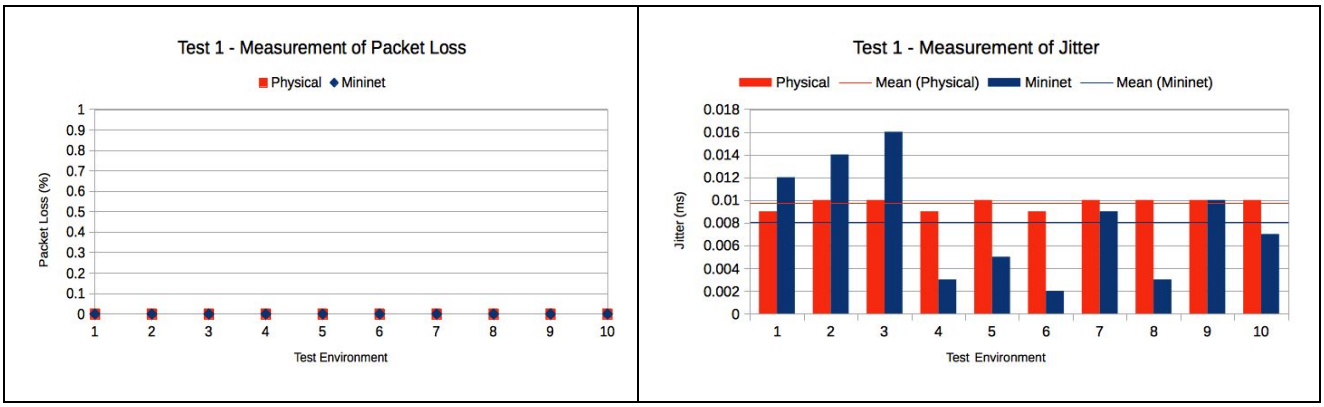

Figure 4.2: Test 1 results for packet loss and jitter measurements between the physical and mininet environments.

Analysis

As there was no traffic on Path 1 other than H1's there was no packet loss, as expected in both environments. As well, there was very low jitter with a mean of 0.0097ms, and 0.0081ms for the physical and mininet environments respectively.

Test 2 - Performance without Dynamic Traffic Diversion

Scenario

H1 sent 600MB of UDP traffic at a data rate of 50Mbps to H3 through Path 1, while H2 congested 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 95Mbps without the DTD application running. The objective of this test was to determine a baseline while there was congestion on Path 1.

Hypothesis

The transmission of the congestion traffic will greatly affect the transmission between H1 and H3 through Path 1, increasing the probability of packet loss and jitter.

Results

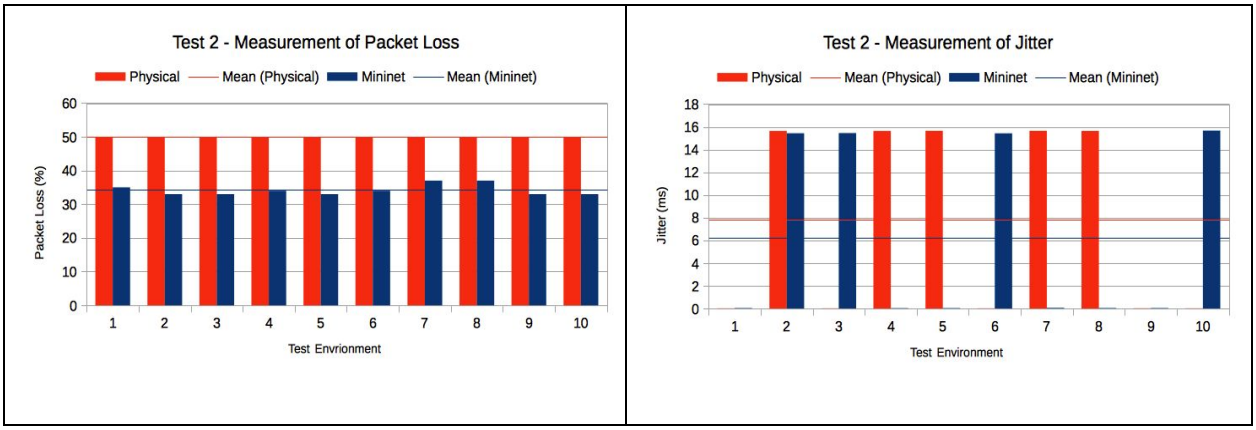

Figure 4.3: Test 2 results for packet loss and jitter measurements between the physical and mininet environments.

Analysis

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

Test 3 - Performance with Dynamic Traffic Diversion

Scenario

H1 sent 600MB of UDP traffic at a data rate of 50Mbps to H3 through Path 1, while H2 congested Path 1 with a large amount of UDP traffic at a data rate of 95Mbps with the DTD application running. Purpose of the test was to determine whether our DTD application will decrease the delivery time and increase the delivery rate of the marked traffic.

Hypothesis

The transmission of the congestion traffic will have no effect on the transmission between H1 and H3 through Path 1 as the traffic will diverge to Path 2 with no traffic to congest the link which will result in no packet loss, and very low jitter.

Results

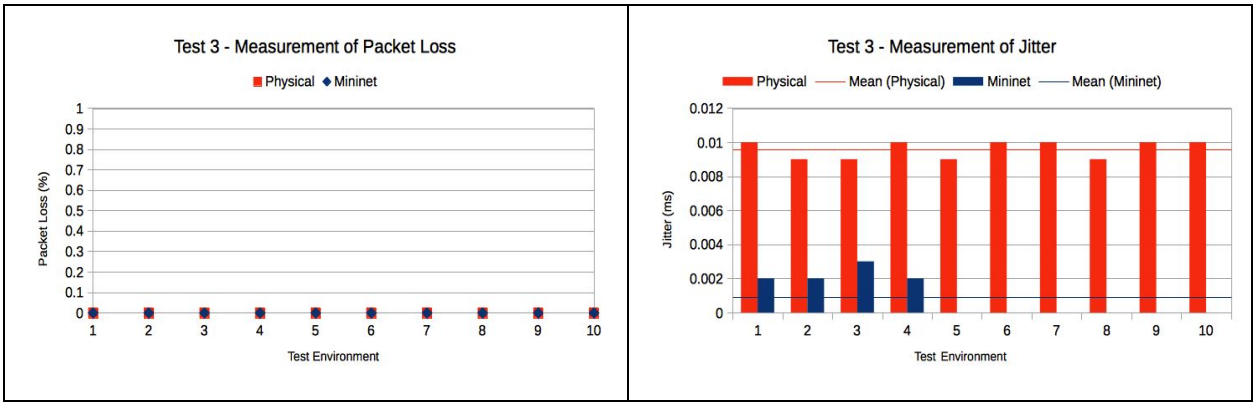

Figure 4.4: Test 3 results for packet loss and jitter measurements between the physical and mininet environments.

Analysis

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 Host 1 and Host 3 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.

Final Analysis

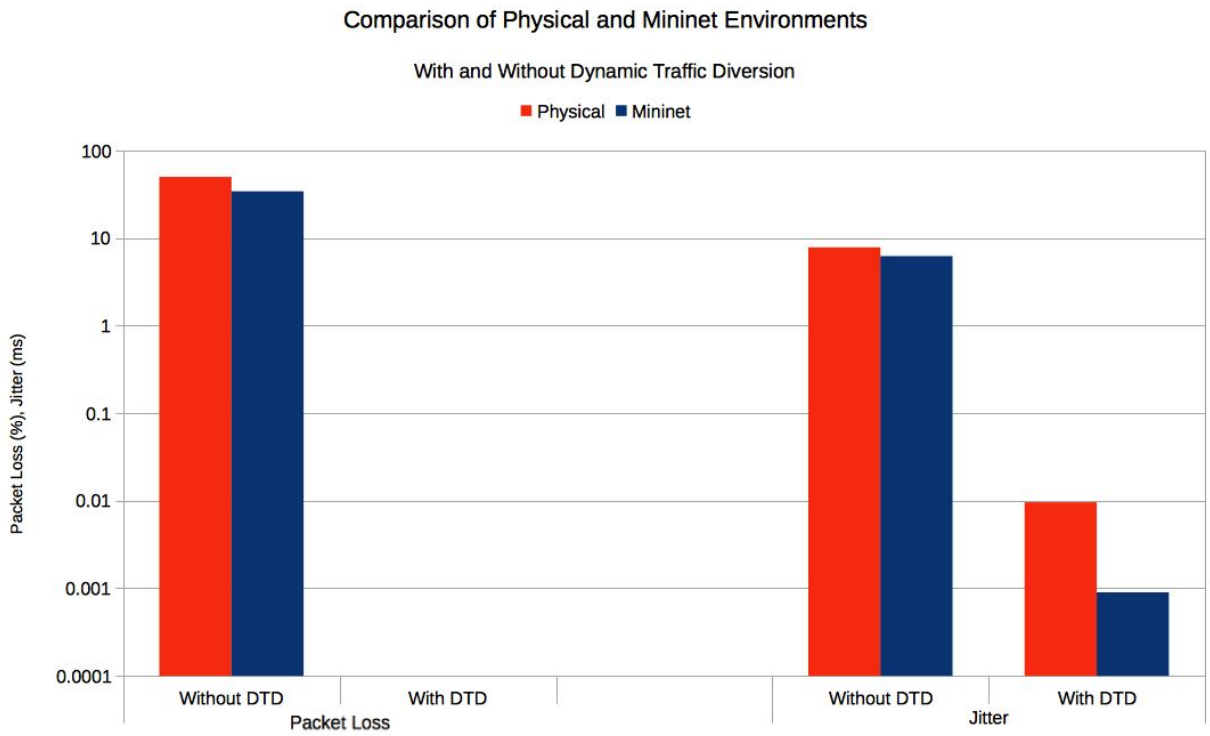

Figure 4.5: Packet Loss and Jitter with and without DTD

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 the 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.

5 Obstacles Faced

While OpenDaylight works out of the box with mininet, tweaks had to be made to work with the openflow plugin on the Cisco switches. By default, the switches forward unmatched packets to the controller, the controller extracts the source IP and MAC addresses from the packet and generates a flow matching those details to the source port that generated the packet. It assigns these flows a standard default priority and assigns a timeout, similar to mac-address-table inactivity.

The version of the openflow plugin we were provided with cannot have multiple flows with the same priority, causing them to be ignored. In this case we needed to tweak the default flow generation method. An option attempted was to include a small random number generator to keep priorities separate. This however did not behave consistently and often resulted in multiple copies of default flows with different priorities. In addition, the openflow plugin does not support the inactive flow timeout feature, this part of the flow generation had to be removed.

During the creation and testing phase of our DTD program, we noticed the values that were being displayed via the XML file did not add up. More specifically, the program was not able to trigger a change-over event effectively. The default refresh time of the XML document was 5 seconds. The only way to remedy this situation was to find the hard-coded XML refresh time within the code of OpenDaylight in order to have a more responsive change-over call. After locating the necessary configuration file, we were able to lower the refresh time to 1 second.

While not directly related to the configuration of our controller or environments, we suffered a hard drive failure in our main server and were forced restore everything. This resulted in a delay of approximately two weeks near the end of December.

6 Team Structure

The breakdown of our team is as follows:

Welile setup the server and virtual machines,
Phil the physical setup,
Robert the Dynamic Traffic Diversion application,
Josh the documentation,
Andre the Test Planning.

All members of the team were involved with testing and analysis.

7 Conclusion

After eight months of research, planning and work, we ended 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 within two test environments. A physical cisco network run off of beta software and an emulated mininet network were the focus of the comparison.

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 although not perfectly suited for statistical analysis, it excels at design and scalability testing.

There are several areas that could see future research and have the potential for improvement. Currently, our program runs 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.

As we further our understanding in SDN and it's applications, there will be a greater shift from expensive hardware to a more cost effective and efficient manner of networking. Dynamic Traffic Diversion is only in it's infancy with use in SDN and we can already see the potential. Furthermore, we can confidently say that Mininet is a suitable emulation tool that inherently will work with this novel approach to network design.

8 References

[1] J. Metzler (2012, Aug 29). What is software defined networking(SDN)?.[Online] Available: http://www.networkworld.com/article/2159545/software/what-is-software-defined-networking--sdn--.html

[2] OpenDaylight Project. Technical Overview. [Online]
Available: http://www.OpenDaylight.org/project/technical-overview

[3] Cisco, Cisco Plug-in for OpenFlow Configuration Guide, San Jose, Cisco Press, 2014

[4] OpenDaylight Application Developers' Tutorial. [Online]
Available: http://sdnhub.org/tutorials/OpenDaylight/

[5] F. Durr, OpenDaylight: Programming Flows with the REST Interface and cURL, [Online]
Available: http://www.frank-durr.de/?p=68

[6] F. Durr, Developing OSGi Components for OpenDaylight, [Online]
Available: http://www.frank-durr.de/?p=84

[7] Cisco, Release notes for Cisco Plug-In for Open Flow Release 2.0, Cat4K/3850/3650 EFT 08/22/14, San Jose, Cisco Press, 2014

[8] Mininet Team, Get started with mininet, [Online]
Available:http://mininet.org/download/

[9] F.Durr, OpenDaylight: Programming Flows with the REST Interface and cURL, [Online]
Available: http://www.frank-durr.de/?p=68

9 Appendix

9.1 Project Planning

Start	Completion	Weeks	Stage
12/10/14	26/10/14	2	- Design of network topology / addressing scheme - Establish weekly project work schedule
27/10/14	02/11/14	1	- Implement physical network
03/11/14	23/11/14	3	- Build & test an initial SDN controller
24/11/14	14/12/14	3	- Manipulate flows with ToS specifications, examine routes for statistics gathering
15/12/14	28/12/14	2	- Break (Exams/Christmas)
29/12/14	18/01/15	3	- Begin major development of external application - Prepare for preliminary progress demonstration
19/01/15	08/02/15	3	- Finish major development of external application. - Adding a web interface for program control - Begin debugging and testing
09/02/15	01/03/15	3	- Test performance, compare with mininet scenarios with scalable networks

9.2 All Required Hardware/Software

Hardware:

- Three Cisco Catalyst 3650 Switches with EFT image enabling SDN-capabilities
- Router for remote access
- Three physical hosts
- Server
- Additional Network Cards
- Cabling

Software

- Mininet
- OpenDaylight
- Lubuntu
- Virtualbox
- Tmux
- iPerf
- Python
- Bash

9.3 Raw Test Data

Table 1: Test 1 Measured Statistics for Packet Loss and Jitter

TEST 1	Packet Loss (%)			Jitter (ms)	
	Mininet	Physical		Mininet	Physical
	0	0		0.012	0.009
	0	0		0.014	0.01
	0	0		0.016	0.01
	0	0		0.003	0.009
	0	0		0.005	0.01
	0	0		0.002	0.009
	0	0		0.009	0.01
	0	0		0.003	0.01
	0	0		0.01	0.01
	0	0		0.007	0.01
Average	0	0		0.0081	0.0097
Std. Dev	0	0		0.0046572524	0.0004582576

Table 2: Test 2 Measured Statistics for Packet Loss and Jitter

TEST 2	Packet Loss (%)			Jitter (ms)	
	Mininet	Physical		Mininet	Physical
	35	50		0.038	0.017
	33	50		15.432	15.637
	33	50		15.445	0.017
	34	50		0.031	15.637
	33	50		0.035	15.648
	34	50		15.426	0.015
	37	50		0.055	15.645
	37	50		0.041	15.639
	33	50		0.039	0.018
	33	50		15.665	0.017
Average	34.2	50		6.2207	7.829
Std. Dev	1.6193277069	0		7.570251793	7.8122006759

Table 3: Test 3 Measured Statistics for Packet Loss and Jitter

TEST 3	Packet Loss (%)			Jitter (ms)	
	Mininet	Physical		Mininet	Physical
	0	0		0.002	0.01
	0	0		0.002	0.009
	0	0		0.003	0.009
	0	0		0.002	0.01
	0	0		0	0.009
	0	0		0	0.01
	0	0		0	0.01
	0	0		0	0.009
	0	0		0	0.01
	0	0		0	0.01
Average	0	0		0.0009	0.0096
Std. Dev	0	0		0.0011357817	0.0004898979

Table 4: Test Averages for the Measured Statistics of Packet Loss and Jitter

TEST AVERAGES		Mininet	Physical
Packet Loss	TEST 1	0	0
	TEST 2	34.2	50
	TEST 3	0	0
Jitter		Mininet	Physical
	TEST 1	0.0081	0.0097
	TEST 2	6.2207	7.829
	TEST 3	0.0009	0.0096

9.4 Program Code and Logic

```
import pycurl
from StringIO import StringIO
import xml.etree.ElementTree as ET
import time
from subprocess import call
import sys

#----------------------THRESHOLDS-----------------------------------------------------------------------------------------------------------------------------------------------------------
maxLinkSpeed = 63700000
upperThreshold = 0.9
lowerThreshold = 0.7

standardLink = 0
backupLink = 1
state = standardLink


#---------------------ENVIRONMENT SETUP-----------------------------------------------------------------------------------------------------------------------------------------------------------
environment = "/home/localadmin/PHY/"


# Physical Environment XML Positions WITHOUT Mininet Environemnt
'''
xmlPos1 = 0
xmlPos2 = 6
xmlPos3 = 3
'''

# Physical Environment XML Positions WITH Mininet Environment
xmlPos1 = 2
xmlPos2 = 3
xmlPos3 = 3

# XML Positions for Testing Mininet Environment
if 'mininet' in str(sys.argv):
   xmlPos1 = 5
   xmlPos2 = 1
   xmlPos3 = 3
   environment = "/home/localadmin/MN/"
   maxLinkSpeed = maxLinkSpeed / 5

# delete any flows that have been previously created before startup
call([environment+"backup05.sh"])
call([environment+"backup05-tos.sh"])

                                                                                                                                                                                    7,0-1         Top

#----------------------FUNCTIONS-------------------------------------------------------------------------------------------------------------------------------------------------------------

# poll controller for port statistics of switches
def getInterfaceBytes():
    c = pycurl.Curl()
    c.setopt(pycurl.URL, 'http://localhost:8080/controller/nb/v2/statistics/default/port/')
    c.setopt(pycurl.HTTPHEADER, ['Accept: application/xml'])
    c.setopt(pycurl.VERBOSE, 0)
    c.setopt(pycurl.USERPWD, 'admin:project')
    c.setopt(c.WRITEDATA, buffer)
    c.perform()
    c.close()
    body = buffer.getvalue()
    tree = ET.fromstring(body)
    buffer.truncate(0)
    # retrieve port specific statistics
    return int(tree[xmlPos1][xmlPos2][xmlPos3].text) + int(tree[xmlPos1][xmlPos2][xmlPos3+1].text)

# switch traffic between links based on traffic thresholds
def switchLink(state):
        if 'tos' not in str(sys.argv):
                if state is standardLink: #switch traffic to standard link
                        call([environment+"backup05.sh"])
            print (". Changed Flow to Standard Link")
                if state is backupLink: #switch traffic to alternative link
                        call([environment+"backup21.sh"])
            print (". Changed Flow to Backup Link")
        else:
                if state is standardLink:
                        call([environment+"backup05-tos.sh"])
            print (". Changed Flow to Standard Link")
                if state is backupLink:
                        call([environment+"backup21-tos.sh"])
            print (". Changed Flow to Backup Link")

#-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------

buffer = StringIO()
totalPackets = getInterfaceBytes() # get the total amount of received bytes on interface
n = 0
while True:
    oldTotalPackets = totalPackets
    totalPackets = getInterfaceBytes()
    latestPackets = totalPackets - oldTotalPackets

                                                                                                                                                                                    90,1          67%

    if (state == backupLink):
        stStr = "Alternative Path"
    else:
        stStr = "Standard Path"

    if (totalPackets != oldTotalPackets) or (n>15):
        n = 0
        # "XML Updated. Bytes since last update:",latestPackets,
        print "Link Capacity (%):",round(float(latestPackets)/float(maxLinkSpeed)*100,2), "Current State:", stStr
        if latestPackets > (maxLinkSpeed * upperThreshold) and state is standardLink: # check for upper threshold
            print "Alert, Upper Threshold reached"
            state = backupLink
            switchLink(state) #switch traffic to backup link
        if latestPackets < (maxLinkSpeed * lowerThreshold) and state is backupLink: # check for lower threshold
            print "Alert, Lower Threshold reached"
            state = standardLink
            switchLink(state) #switch traffic to standard link

    buffer.truncate(0)
    n = n + 1
    time.sleep(0.5)
```

9.5 Cisco Switch Configuration

Openflow Configuration

Switch 1

```
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
```

Switch 2

```
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 0x12
   controller ipv4 192.168.134.102 port 6633 security none
  exit-ofa-switch
exit
```

Switch 3

```
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 0x13
   controller ipv4 192.168.134.102 port 6633 security none
  exit-ofa-switch
exit
```

9.6 Mininet Topology Configuration

```
14 from mininet.topo import Topo
15
16 class MyTopo( Topo ):
17     "Simple topology example."
18
19     def __init__( self ):
20         "Create custom topo."
21
22         # Initialize topology
23         Topo.__init__( self )
24
25         # Add hosts and switches
26         host1 = self.addHost( 'h1' )
27         host2 = self.addHost( 'h2' )
28         host3 = self.addHost( 'h3' )
29         host4 = self.addHost( 'h4' )
30         switch1 = self.addSwitch( 's1' )
31         switch2 = self.addSwitch( 's2' )
32         switch3 = self.addSwitch( 's3' )
33
34         # Add links
35         self.addLink( host1, switch1, bw=100 )
36         self.addLink( host2, switch1, bw=100 )
37         self.addLink( host3, switch3, bw=100 )
38         self.addLink( host4, switch3, bw=100 )
39         self.addLink( switch1, switch3, bw=100 )
40         self.addLink( switch1, switch2, bw=100 )
41         self.addLink( switch2, switch3, bw=100 )
42
43
44 topos = { 'mytopo': ( lambda: MyTopo() ) }
```