Agility of Tunnel-based and SDN-based Approaches in Rerouting-based Traffic Engineering

Pritesh Ranjan[1], Bharath Ravindranath[1], Balakundi Shravan Achar[2]
[1]*Computer Science*, [2]*Electrical and Computer Engineering*,
North Carolina State University
Raleigh, NC, USA
{pranjan, bharat3, bachar}@ncsu.edu

***Abstract*—A network policy can be enforced by network operators by introducing the policies at the edge nodes or by distributing them across the network. The goal of this project is to compare the two approaches and to find out which approach enables more capabilities in the network and the disruptions caused while changing the network-wide policies.**

I. INTRODUCTION

Network tunnels can be used to introduce the network smarts at the edge. In a network employing tunnels, the packets are assigned to a tunnel at the edge nodes and the core nodes simply forward the packets based on the encapsulated headers. In this project, we use MPLS as the tunneling technology.

On the other hand, distributing network policies in the network requires each node in the network to decide the fate of the incoming packet. This might incur processing overhead at the intermediate nodes but may also enable flexibility with the realization of fine-grained network policies. We use, OpenFlow as the technology of choice. The network policies are dictated at a centralized OpenFlow controller, which in turn can dictate the switches to act their part in enabling the policy.

Network traffic engineering is a capability that we shall be using to compare the two approaches. We reroute a flow on to an alternate path and measure the link utilization. We also measure the packet drops, latency induced, bandwidth fluctuations when the transition to a new path takes place.

In the above and in the rest of this paper, we have adopted the following terminology:

Software Defined Networks: The physical separation of the network control plane from the forwarding plane, and where a control plane controls several devices. [1]

Tunneling: Tunneling allows a network user to access or provide a network service that the underlying network does not support or provide directly. For our study, it is equivalent to encapsulation of one packet in another. Examples include IP packet encapsulated in MPLS header, IP packet encapsulated in IP header like GRE etc.

II. COMPONENTS

The project has two network setups. One is MPLS network and the other one is OpenFlow network. Both require an underlay network connecting the datapaths in the network. We are using the following platforms:

- Network Test Bed: We are using ExoGENI facility to create our network [4].
- Nodes Operating System: Fedora 22
- Controller (MPLS network): ODL Beryllium.
- Controller (OpenFlow network): RYU 4.13 [5]
- Datapaths: Open vSwitch 2.5
- Programming Language: Python
- Network Configuration: Ansible

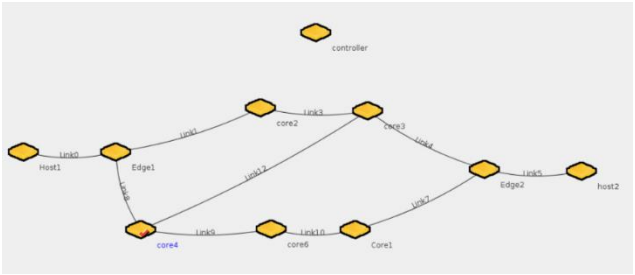

Fig. 1 Network Topology

The topology is constructed as shown in Figure 1. There are 7 nodes in the network, out of which 2 are considered as edge nodes. Each edge node is connected to one host node. Host nodes are the endpoints for the traffic flows.

- Connect the 2 host nodes to 2 Edge nodes. Assign IP addresses to the host nodes.
- Install OVS on the 5 network nodes. Connect the OVSs on network nodes to each other.
- Install controller at the controller node.
- Set each OVS to connect to the controller.

Start the controller. As the switches connect the controller shall discover the topology and install rules on datapaths to forward the traffic accordingly.

At this point, the nodes should have connectivity to each other.

III. Design and Development Details

The primary aim of this study is to analyze the difference in capabilities of a network when two contrasting approaches are used. One approach is the tunneling approach where much of the network-wide policies are enforced through tunnel end-points. In this way, only the end-points are 'policy aware'. The second approach is the distributed flow-based routing approach where, although a centralized controller exists, network policies can be enforced through multiple network nodes. In the second approach, nodes in the middle of the network can also be made 'policy aware' Figure 2.

We build two networks.

- Network I: An MPLS network with ODL to distribute labels.
- Network II: An OpenFlow network with a centralized controller.

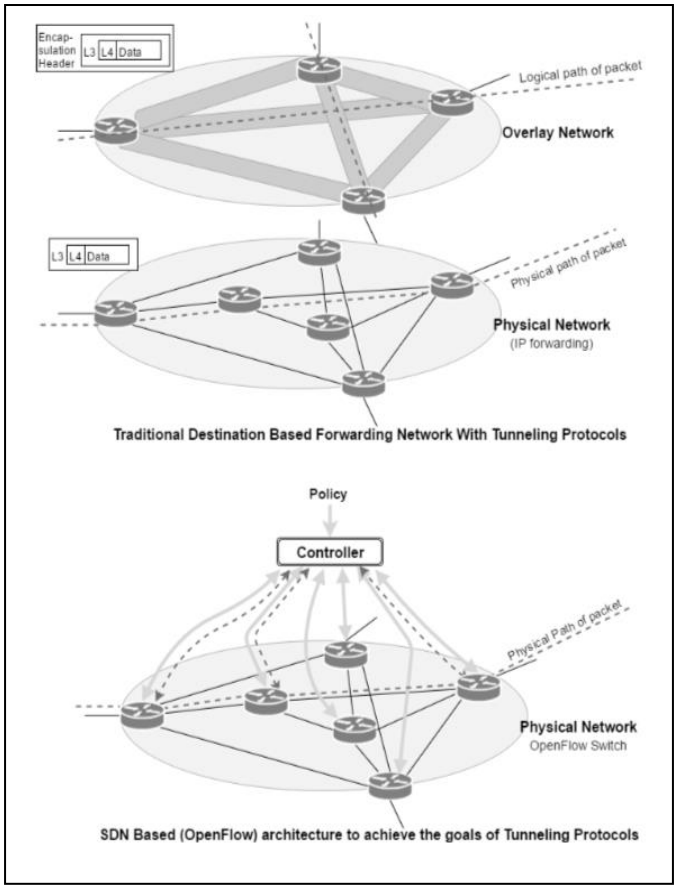

Fig. 2 Tunneling in traditional destination IP based routing vs SDN approach of Forwarding

A. High-level proposed design

We construct two networks (topology shown in Figure 1) with entry and exit points, albeit different from each other. The entry and exit points act as tunnel endpoints MPLS network.

A flow based on source and destination IP is added to the network in both the cases. At a particular point of time, the flow is changed to take a different path. In the case of MPLS, this decision is taken by the end node and in the case of OpenFlow, a centralized view of the network is available. Hence, with the MPLS, the entire LSP is changed. Whereas in OpenFlow network the rules need to be changed at only a subset of nodes.

We will be measuring the packet drop, latency and bandwidth of the system during the path change. The details of how this is done are given in the next section. The workflow for the project is giving in Figure 3

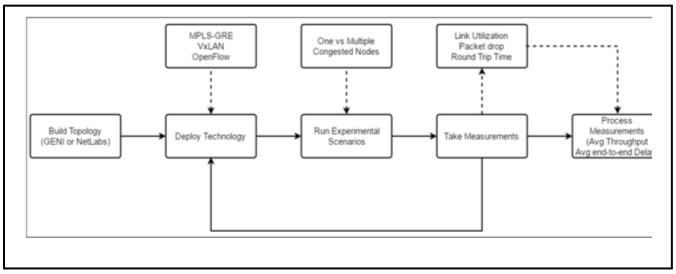

Fig. 3 Study Flow Chart

B. Per-component development phases

1. OpenFlow Network

Create the underlay network as specified in Figure 1

Create an initial configuration file describing the edge nodes at the network connected to it.

This file should be provided to the controller through a management channel. Sample configuration file can be found here.

Start the RYU controller. Once, the controller starts it discovers the network topology and periodically monitors the network traffic. If at any point the controller determines that some link is oversubscribed then the controller adds more microflow rules on the datapaths to steer a portion of the traffic through other paths.

The detailed controller workflow is described below:

Controller workflow

1. Once all the datapaths (switches) are connected, the controller starts topology discovery by sending out LLDP packets. Afterward, the controller creates a network graph.
2. From each network node to every edge node the controller finds the shortest path (least number of hops) and installs rules on that datapath to forward the traffic to next hop.
3. At a periodic interval (TRAFFIC MONITOR INTERVAL) the controller queries the datapaths for port stats.
4. Then the controller calculates the average number of packets being handled by the ports and finds the overloaded ports. i.e. the ports handling a fraction (OVERLOAD FACTOR) of packets more than the average.

5. The controller samples the flow out of the overloaded ports and then finds the micro-flow that can be redirected at the datapaths.
6. The controller installs some ephemeral high priority micro-flow rules at the datapath. Afterward, the traffic matching the micro-flow rules take the new path while the other traffic follows the older paths.
7. Repeat from step 3.

Initial Configuration

To install the proactive rules the controller needs to know the networks connected at the edges. We specify the ingress network and the number of datapaths in the network through a `CONFIG.json` file. This file should be made available to the controller through a management channel. A sample config file is shown here.

```
{
   "node_ct": 3,
   "edges":
   [
     {
       "dpid": "000096406974cd44",
       "host": [
         {
           "address": "10.0.1.58",
           "netmask": "255.255.255.0",
           "port": 3
         }
       ]
     },
     {
      "dpid": "00007600a03e1648",
      "host": [
       {
         "address": "10.0.1.59",
         "netmask": "255.255.255.0",
         "port": 3
       }
       ]
     }
   ]
}
```

"`node_ct`" is the number of datapaths in the network. The Controller starts topology discovery after "`node_ct`" number of datapaths have connected.

"`edges`" has a list of dictionaries. Each dictionary has "`dpid`" of the datapath as the key and a list of "`host`" as values. Each entry in hosts list contains the network "`address`", "`netmask`" and "`port`" information. "`port`" is the OpenFlow port number on the OVS that connects to the hosts.

2. MPLS LSP setup

We use the OVS switch package on the Linux (Fedora 22) to set up the MPLS network (we need open vSwitch 2.4 or later version for MPLS support). To maintain the same packet processing overhead in both the networks we are using OVS as the switching element in both the networks. With OpenFlow 1.3 the action set supports push, pop and swap actions of MPLS shim header [2].

We use the OpenDaylight's REST APIs to push the flows on to the switch. Flows are defined as follows:

- At the ingress router push a label and forward to appropriate port.

```
cookie=0x379, duration=23.037s,
table=0, n_packets=0, n_bytes=0,
idle_age=23,
priority=5,mpls,in_port=2,mpls_la
bel=37
actions=pop_mpls:0x0800,output:1
```

- At the intermediate router match the label of the previous router and swap the label.

```
cookie=0x191, duration=29.888s,
table=0, n_packets=0, n_bytes=0,
idle_age=29,
priority=5,mpls,in_port=1,mpls_la
bel=27 actions=load:0x25-
>OXM_OF_MPLS_LABEL[],output:2
```

- At the egress router pop the label and forward to the appropriate destination as a IP packet.

```
cookie=0x191, duration=23.020s,
table=0, n_packets=0, n_bytes=0,
idle_age=23, priority=5,
ip,nw_dst=192.16.0.0/24
actions=push_mpls:0x8847,load:0x1
b->OXM_OF_MPLS_LABEL[],output:2
```

For simplicity, no penultimate hop popping is defined. These flows are pre-defined so as to maintain complete control over the path for an FEC. The labels are also decided statically. During the packet transfer, new LSPs are pushed on the switch

C. Metric collection

We use `qperf` tool to measure network latency. `qperf` provides average per-hop time taken by a UDP or a TCP packet. Packet drops are also obtained from the stats collected by the controller. These are obtained for every port on every node.

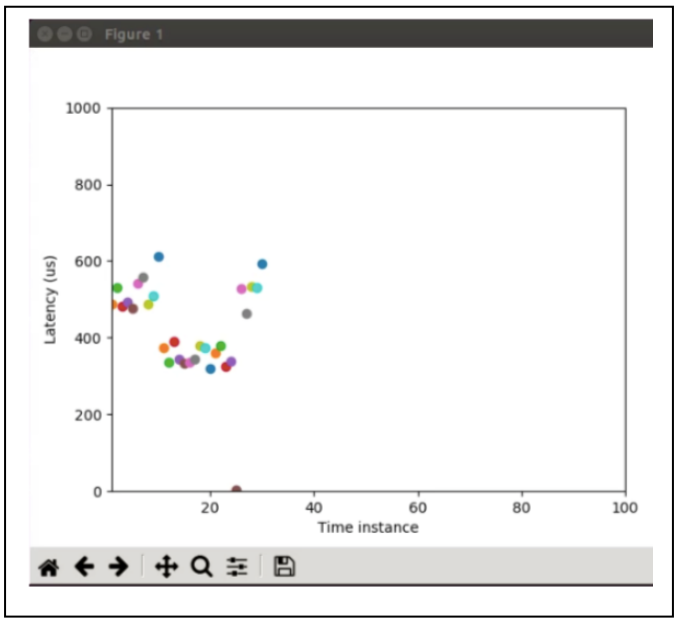

Fig. 4 Graph Showing the Variations in Latency due to path changes in OpenFlow Network

IV. Test Results

A. *Testing Conditions*

- OpenFlow network
  The primary path taken by packets from host1 to host2 is constructed as follows: `Host1- Edge1-Core3-Core4-Core5-Edge2-Host2`

  The secondary path(or backup path) taken by packets from host1 to host2 is constructed as follows: `Host1-Edge1-Core1-Core2-Edge2-Host2`

  The primary path is setup at time `t = 0`. A ping traffic is started at the same time. After time `t = 15`, the path is switched to the backup path. After time `t > 22`, the path is switched back to the primary path. TCP latency between the hosts is constantly measured. Packet drops in the network are constantly measured.

- MPLS network

  The primary path taken by packets from `host1` to `host2` is constructed as follows: `Host1-Edge1-Core3-Core4-Core5-Edge2-Host2`

  The secondary path (or backup path) taken by packets from `host1` to `host2` is constructed as follows: `Host1-Edge1-Core1-Core2-Edge2-Host2`

  The primary path is setup at time `t = 0`. A ping traffic is started at the same time. After time `t = 20`, the path is switched to the backup path. After `time > 30`, the path is switched back to the primary path. TCP latency between the hosts is constantly measured. Packet drops in the network are constantly measured.

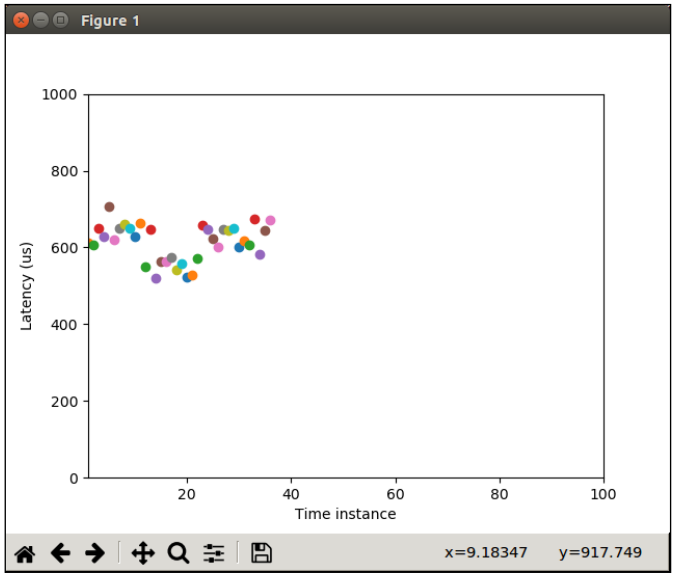

Fig. 5 Graph Showing the Variations in Latency due to path changes in MPLS Network

The time period for which the packets take the primary path is considered to be the ‘primary’ phase. The time period for which the packets take the secondary path is considered to be the ’secondary’ phase.

B. *Results*

OpenFlow network:

Figure 4 shows the variation of latency due to path changes in the OpenFlow network. In the primary phase, the latency varies from approx. 500 us to 600us. During secondary phase, the latency reduces to a band of 300us to 400us. This is due to the fact that the secondary path has lesser number of hops from source to destination.

Figure 6 shows packet drops due to the path switching in the OpenFlow network. When the switch to secondary path took place, packet drops were seen on `Edge1`, `Edge2`, `Core3`, `Core4`, and `Core5`. `Edge1` and `Edge2` had drops of around 50 packets. Core1 and Core2 had no drops during the secondary phase. The drops seen on Core1 and Core2 are when the path was switched back to the primary from the secondary path. Though more drops were seen on the edges, non-zero drops are seen on intermediate nodes as well. This is probably due to the fact that the path setup in the intermediate nodes took longer than the switching rate at Edge1. However, it can be said the path setup was considerably fast to allow less than 10 drops on Core3, Core4, and Core5. The switches are connected to the controller via the Internet. Delay variation on the Internet at different time intervals is also a factor that impacts packet drops on intermediate nodes.

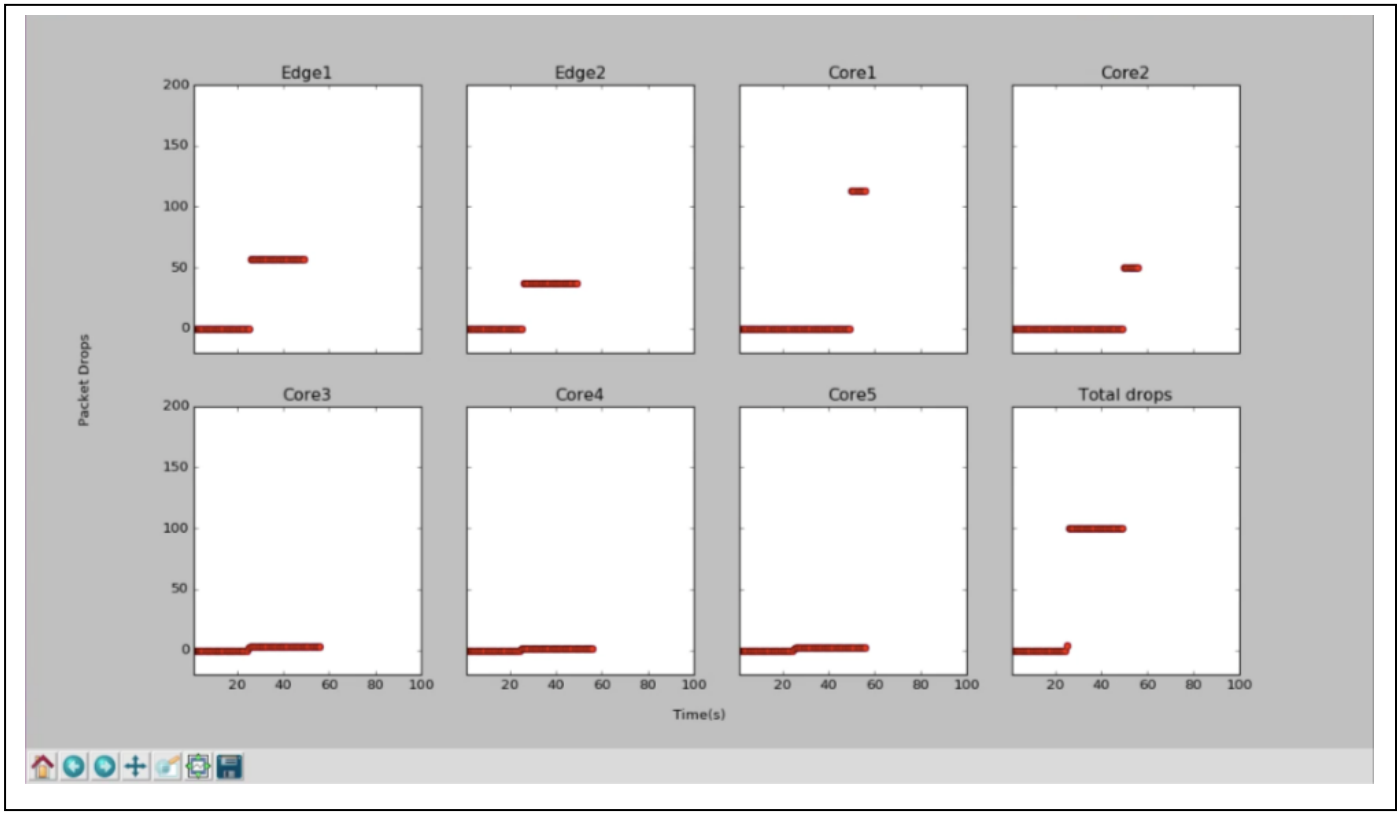

Fig. 6 Graph Showing Packet drops due to path changes in OpenFlow Network

MPLS network:

Figure 5 shows the variation of latency due to path changes in the MPLS network. In the primary phase, the latency varies from 600 - 700us. This range is slightly higher than the latency range observed in the primary phase of OpenFlow network. This may be explained due to the encapsulation overhead incurred by all the packets traversing the label switched path. This may also be due to the implementation of MPLS being less efficient than the implementation of OpenFlow by OpenvSwitches. During the secondary phase, the latency reduces to a band ranging from 500us - 600us. This is because the secondary path has lesser number of hops from source to destination. The latency times in the secondary phase of MPLS is seen to be higher than that of the times in secondary phase of OpenFlow network. This may again be due to encapsulation overhead in the MPLS network.

Figure 7 shows packet drops due to the path switching in the MPLS network. During the switch, packet drops were predominantly seen on Edge1. Non-zero drops were also seen on Core3 and Core5. Less than 40 packet drops were observed on Edge1 and no drops were observed on Edge2. Lower packet drops overall in the MPLS network may be due to the faster setup time of MPLS labels as opposed to OpenFlow rules which consist of multiple fields as match criteria.

V. Conclusions

We compared tunnel-based versus SDN-based methods for traffic engineering. Overall, it can be concluded that the MPLS network had higher end-to-end latencies than OpenFlow network, but had lower packet drops than OpenFlow network. As we have mentioned above, this may well be a characteristic of the two methodologies, since tunnel-based methods require all traffic engineering decisions to be expressed by means to end-to-end tunnels (but intermediate nodes have simpler forwarding), but SDN-based methods allow incremental traffic engineering decisions to be realized by only manipulating a few nodes, (but each forwarding node is involved in more complicated forwarding decisions). However, to be sure that our results are significant, and to rule out the possibility thatspecific characteristics of our specific platforms affected the results, more extensive experimentation would be required.

Acknowledgment

This work was performed as part of advanced coursework in the graduate curriculum of the North Carolina State University, using NCSU equipment, and other resources made available by NCSU.

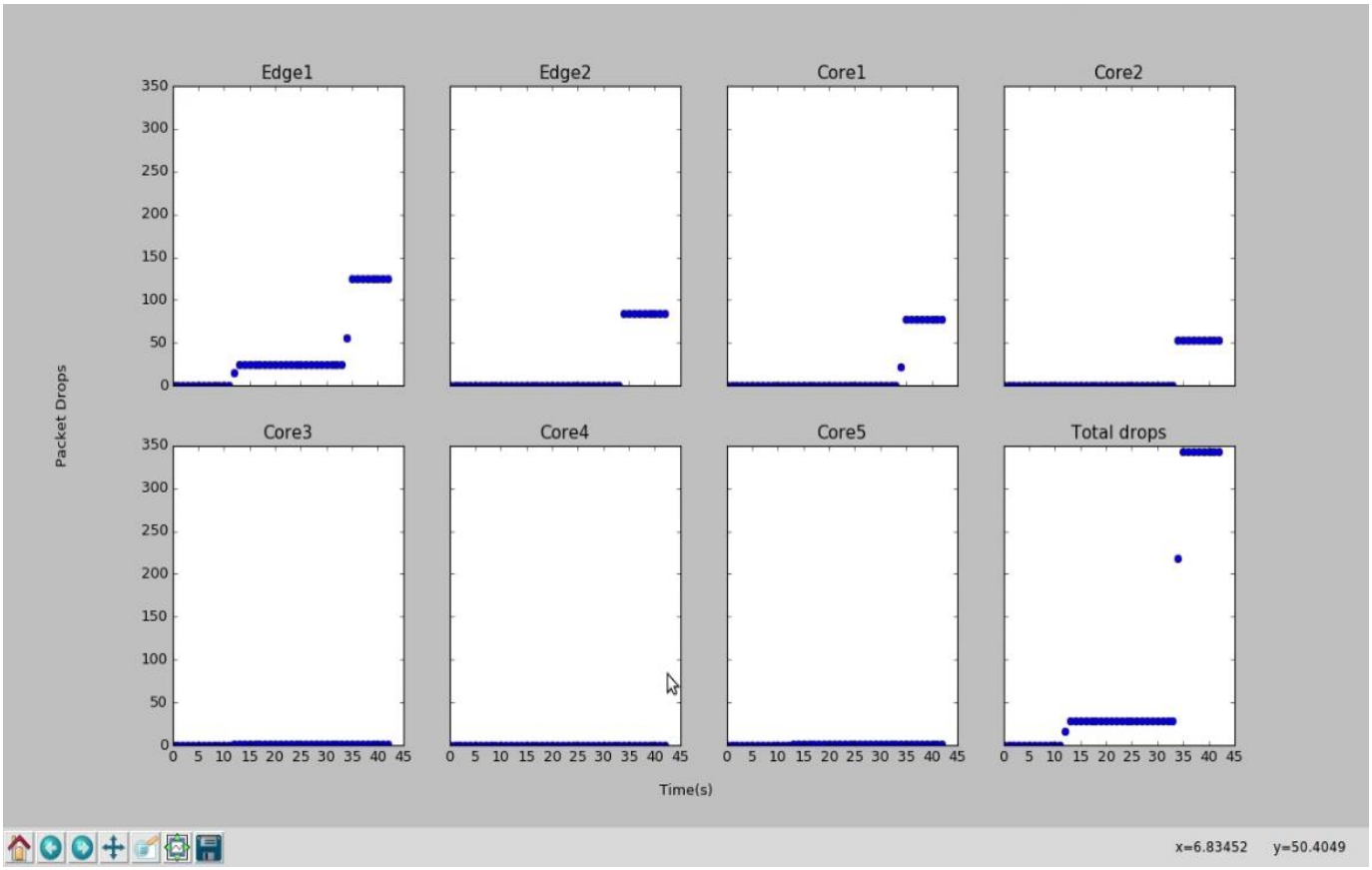

Fig. 7 Graph Showing Packet drops due to path changes in MPLS Network

References

[1] Open Network Foundation. SDN overview, 20-March-2017. https://www.opennetworking. org/sdn-resources/sdn-definition.

[2] Open Network Foundation. Openflow switch specification, 6-September-2012. https://www.opennetworking.org/images/stories/downloads/sdn-resources/onf- specifications/openflow/openflow-spec-v1.3.1.pdf.

[3] Merit Network Inc. NSFNET T3 Backbone Map Fall 1992, Created 2011. [https://en.wikipedia.org/wiki/Backbone_network#/media/File:NSFNET-backbone-T3.png[Online; accessed via Wikipedia 26-March-2017].

[4] http://www.exogeni.net/

[5] http://ryu.readthedocs.io/en/latest/