CSC/ECE 573

Dynamic QoS for Multimedia Streaming Applications over SDN
(Final Report)

North Carolina State University

Instructor:
Dr. Rudra Dutta

Team Members:
Bharat Mukheja, Calvin Fernando, Kaustubh Gondhalekar, Ross Williams

Team WebSite:
https://sites.google.com/ncsu.edu/dynamicqos/home

Table of Contents

I. Introduction

Packet-based switching resides at the heart of the Internet, having beaten out circuit-based switching in the early days of its design. Best Effort packets carried the first information across the Information Superhighway; Transmission Control Protocol assured the eventual reception of the data. With the advent of multimedia streaming as the new dominant genus of Internet traffic, content providers are seeking to imitate real-time reliability of circuit-based network. Software defined QoS does a decent imitation of circuit-based networks, but is still far from industry standard service level desired.

II. Problem Statement

The Internet architecture works on a packet-switched, best-effort principle. This favors reliability over timely-delivery of packets. Multimedia-traffic, which now occupies a significant chunk of today's internet traffic, demands strict delay requirements. Though current QoS architectures sort-of achieve this, they are built on a routing architecture which is essentially hop-by-hop based and does not provide a global view of the network's state. The current architectures also lack adaptivity and real time configurability.

We aim to provide an end to end Quality of Service for multimedia flows in the network by providing minimum bandwidth guarantees to multimedia flows and dynamically routing these flows.

III. Resources

A. GENI

GENI (Global Environment for Network Innovations) is a virtual network testbed for testing out new network architectures and protocols. GENI provides 'slices' over their testbed to conduct experiments. Resources required for the experiment (switches, links, nodes, etc) can be reserved for the slice. A slice acts as an *isolated* unit of experiment which uses only the resources reserved for it. The word *isolated* here means that the experiment lives inside a slice and is not affected by other slices running on the GENI testbed.

We are using GENI testbed to build network topologies to test our experiment.

B. Openflow

OpenFlow is a Software Defined Networking (SDN) paradigm that decouples control and data forwarding layers of switching. The OpenFlow environment primarily includes OpenFlow Controller and OpenFlow switches. The OpenFlow Controller is the "brain" of the SDN network, relaying information to the OpenFlow switches (via Southbound APIs) and the application and business logic (via Northbound APIs). The OpenFlow Controller pushes down changes to the switch flow-table, thereby, allowing network administrators to partition traffic, control flows for optimal performance, etc.
The OpenFlow Switch is a software program or hardware device that forwards packets in a OpenFlow environment. The OpenFlow Controller and OpenFlow Switches communicate using OpenFlow protocol.

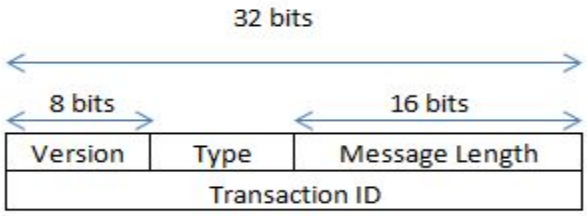


Figure 1: OpenFlow Protocol Header

In order to implement Dynamic QoS for multimedia applications in SDN, we are using OpenFlow environment.

C. Streaming Applications

Our test cases require a multimedia UDP stream to be transmitted from the servers to the client. We selected VideoLAN's VLC due to it meeting a few basic criteria. It is available for the GENI nodes, it is capable of streaming multimedia content in a standard UDP flow such that there is no reaction to lost packets, and it is able to send traffic at a controllable or predictable rate. The audio streaming must also be capable of visualizing the audio signal it is receiving. Ideally, the server application should be able to run headless (i.e through a Command Line Interface).

D. Traffic Generator

To create cross-traffic during our tests, we implemented a pseudorandom traffic generator on top of the linux utility iperf. The generator ensures that a few standard ports are open for iperf TCP/UDP traffic and generate traffic itself. To mimic real world conditions, it pseudorandomly selects from a list of IPs of other generators on the

network a target. It then transmits a pseudo-random amount of data for a pseudo random time period. Each generator will usually be transmitting at least 10 flows to different targets.

The generator must be configurable to provide both semi-continuous background traffic and bursty traffic when required. Similar configurations must be provided to scale the volume of traffic produced, including a linear increase during a test. Ideally it will be remotely accessible without an ssh connection, but that is not a requirement.

None of the free traffic generators we found would fit our testing needs, so we constructed it in house.

IV. Design

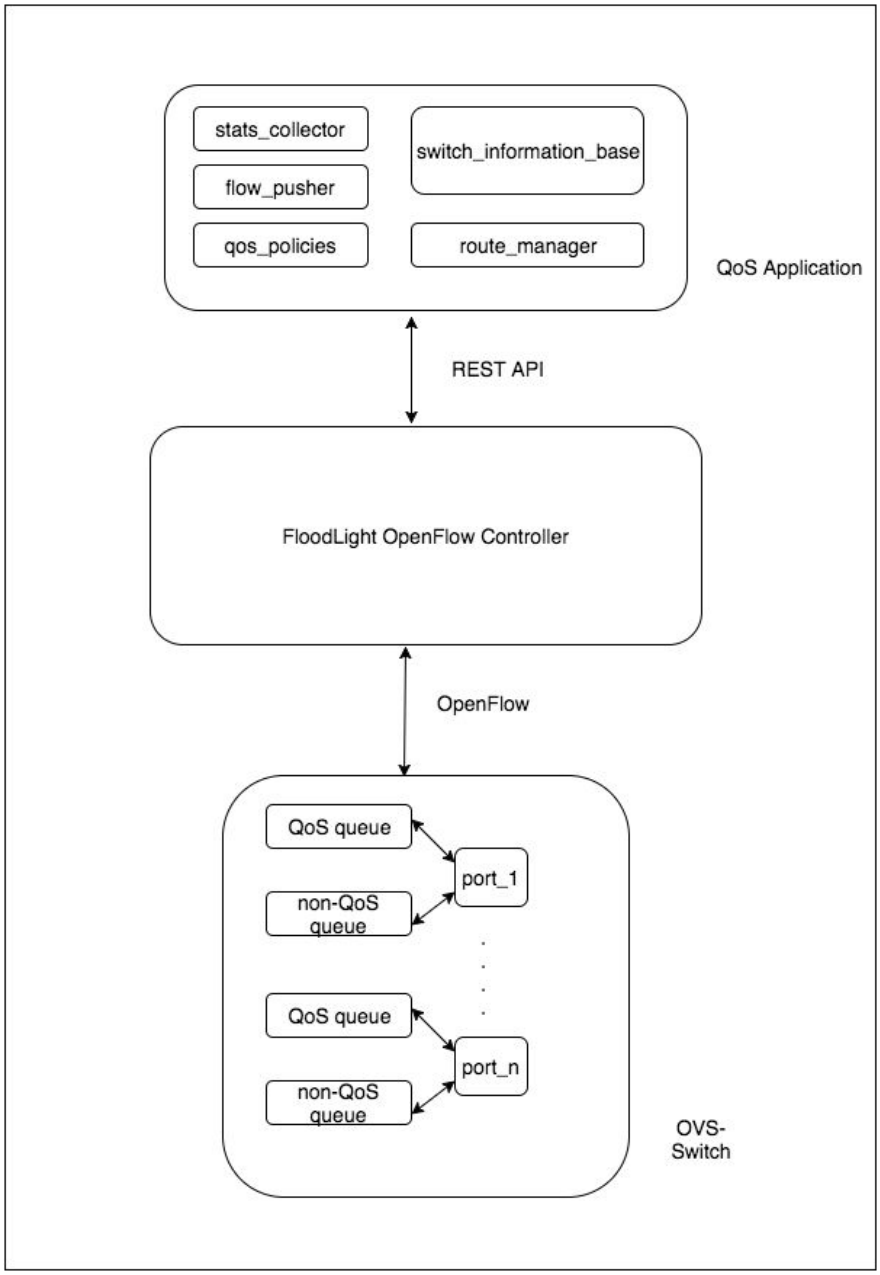


Fig. 1 High-Level view of System Architecture

A. Components

QoS Application

This is the parent application which handles all the QoS, provisioned in the network.
The application is divided into 5 parts.

- switch_information_base
- flow_pusher
- route_manager
- stats_collector
- qos_policies

At startup, the application fetches details for every switch connected to the controller via a GET call through the REST API exposed by the Floodlight Controller.
The data retrieved is stored in the switch_information_base.

Low Level Design

Application Flow

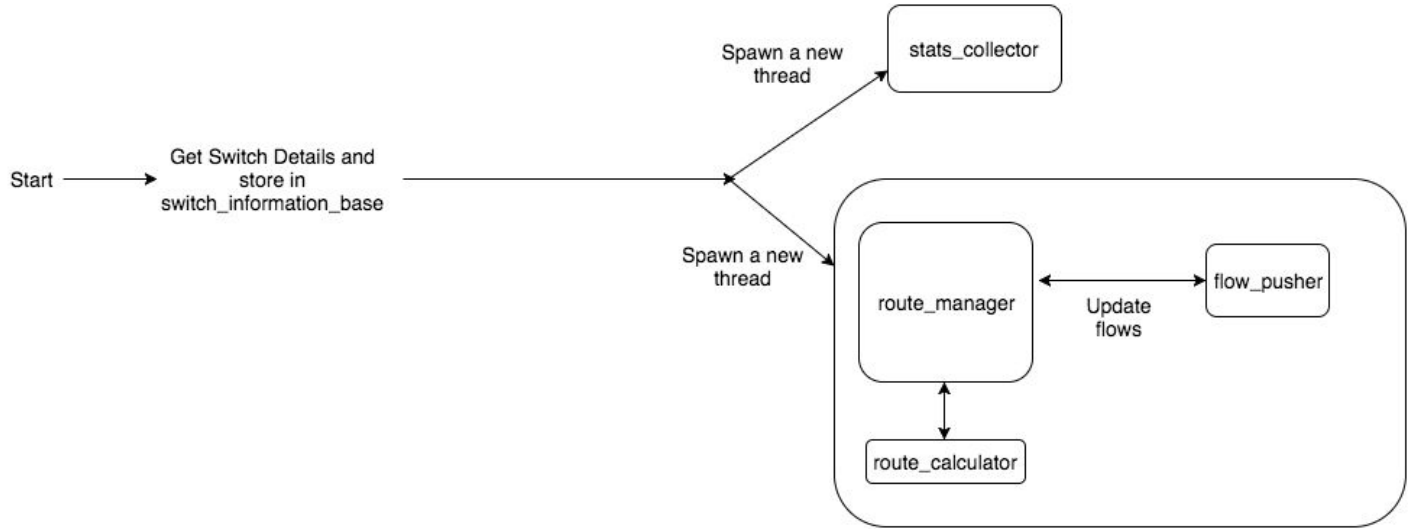


Fig. XX Application Flow

switch_information_base
The switch_information_base has two components switch_dict and topo_specs.
switch_dict in the code is a nested dictionary which stores statistics and flow information of all the switches.
The following is the structure of the switch_dict:

```
{ <switch_dpid> : {switch_details}}
```

switch_details is a dictionary which stores port-details and flows for each switch.
switch_details structure:

```
{ ports:{port_information}, flows:{<flow_1>, <flow_2>,..., <flow_n>}
}
```

port_information is a dictionary which stores port-specific information
port_information structure

```
{<port_no> : { mac:<mac_id>, bandwidth_util : <b>,
tx:<transmitted_bytes>, rx:<received_bytes>, time:<time_in_sec>,
latency:<l>}
```

Topo specs is a pre-created dictionary of network topology containing details which dont change over time, for example - the topology graph, the ethernet addresses and dpid's.
Following is the screenshot of topo_specs -

```
#from numpy import inf

global switch_port_mac_dict
switch_port_mac_dict ={'00:00:de:dc:b4:67:6b:4a': {'1': 'fa:16:3e:00:54:a1', '2': 'fa:16:3e:00:4a:a0', '3': 'fa:16:3e:00:56:5f',
                                                   '4': 'fa:16:3e:00:4f:57'},
 '00:00:06:a8:4b:ed:4c:4b': {'1': 'fa:16:3e:00:3d:27', '2': 'fa:16:3e:00:48:45', '3': 'fa:16:3e:00:40:9a',
                             '4': 'fa:16:3e:00:6e:17'},
 '00:00:f6:63:f0:0d:50:46': {'1': 'fa:16:3e:00:3f:f1', '2': 'fa:16:3e:00:3f:ba', '3': 'fa:16:3e:00:48:c3',
                             '4': 'fa:16:3e:00:2c:db'},
 '00:00:c6:ad:66:51:df:40': {'1': 'fa:16:3e:00:3f:dd', '2': 'fa:16:3e:00:75:61', '3': 'fa:16:3e:00:67:d1'},
 '00:00:de:dd:cf:4e:a4:46': {'1': 'fa:16:3e:00:4a:fc', '2': 'fa:16:3e:00:69:6c', '3': 'fa:16:3e:00:5a:06',
                             '4': 'fa:16:3e:00:58:27'},
 '00:00:32:cc:0e:23:09:46': {'1': 'fa:16:3e:00:64:72', '2': 'fa:16:3e:00:60:9e', '3': 'fa:16:3e:00:2e:fa',
                             '4': 'fa:16:3e:00:5c:b2'},
 }
sw1 = '00:00:de:dc:b4:67:6b:4a'
sw2 = '00:00:06:a8:4b:ed:4c:4b'
sw3 = '00:00:f6:63:f0:0d:50:46'
sw4 = '00:00:c6:ad:66:51:df:40'
sw5 = '00:00:de:dd:cf:4e:a4:46'
sw6 = '00:00:32:cc:0e:23:09:46'

eth_tg3 = 'fa:16:3e:00:7a:15'
eth_tg2 = 'fa:16:3e:00:84:4f'
eth_ms1 = 'fa:16:3e:00:29:84'
eth_ms2 = 'fa:16:3e:00:44:0e'
eth_client = 'fa:16:3e:00:6b:44'

global switch_port_mat
global switch_number_dpid
dpid_switch_number = {sw1: 0, sw2: 1, sw3: 2, sw4: 3, sw5: 4, sw6: 5}
switch_number_dpid = {0: sw1, 1: sw2, 2: sw3, 3: sw4, 4: sw5, 5: sw6}
switch_port_mat = [[0, '2', 0, 0, 0, '3'],
                   ['1', 0, '2', 0, '3', 0],
                   [0, '3', 0, '4', 0, '2'],
                   [0, 0, '2', 0, '3', 0],
                   [0, '2', 0, '4', 0, '1'],
                   ['4', 0, '1', 0, '2', 0],
                   ]
```

stats_collector
The stats_collector is a module that runs as an independent thread inside the QoS application. It queries the controller periodically for port statistics for every switch. Using this information it calculates the bandwidth utilization of every link.

The REST API exposed to fetch port statistics is a GET call to `http://<controller_ip>:8080/wm/core/switch/<switch_dpid>/port/json` [3]

This call returns a JSON object with the following port statistics.
(port_number, receive_packets, transmit_packets, receive_bytes, receive_dropped, transmit_dropped, receive_errors, trasmit_errors, receive_frame_errors, receive_overrun_errors, receive_CRC_errors, collisions)

To calculate bandwidth utilization we only need two statistics: receive_bytes and transmit_bytes.

Bandwidth utilization is calculated as follows:

at time t1 seconds: receive_bytes = rx1 and transmit_bytes = tx1
at time t2 seconds: receive_bytes = rx2 and transmit_bytes = tx2

bandwidth_util = (((rx2+tx2)-(rx1+tx1)) / (t2-t1)) bytes/sec

we store this statistic for every port of every switch.

Snippet of stat_collector running:

```
switch_dpid: 00:00:92:15:ed:4f:ff:49
port id 1       bandwidth_util  3.12    Mbits/sec
port id 2       bandwidth_util  3.12    Mbits/sec
port id local   bandwidth_util  0.0     Mbits/sec
```

route_manager
This module manages routing. It runs as an independent thread spawned from the QoS application. It has two components flow creator and cost matrix. It also maintains a flow counter to create a new name for every flow it pushes to switches. It then calls the route_calculator and computes paths for every pair of switches in the network. Using the routes, it creates flows for every switch in the path and then pushes them to the switches by calling flow_pusher module.

Following is a screenshot of its working -

```
{"switch":"00:00:ce:2b:ca:b2:20:43","name":"1684", "eth_dst":"fa:16:3e:00:44:1f", "actions":"output=1"}
{"switch":"00:00:ce:2b:ca:b2:20:43","name":"1685", "eth_dst":"fa:16:3e:00:7a:32", "actions":"output=4"}
{"switch":"00:00:da:e7:78:d1:30:44","name":"1688", "eth_dst":"fa:16:3e:00:44:1f", "actions":"output=3"}
{"switch":"00:00:0e:2b:55:4b:29:44","name":"1686", "eth_dst":"fa:16:3e:00:44:1f", "actions":"output=3"}
{"switch":"00:00:0e:2b:55:4b:29:44","name":"1687", "eth_dst":"fa:16:3e:00:7a:32", "actions":"output=4"}
{"switch":"00:00:22:67:d7:d5:d4:48","name":"1683", "eth_dst":"fa:16:3e:00:7a:32", "actions":"output=1"}
<Response [200]>
```

Above are the flows produced to by the route manager. Response [200] is the response from the REST api after pushing the flows.

route_calculator

This module calculates routes in the network. For non-QoS traffic, we calculate routes using Dijkstra's algorithm. For QoS traffic, routes are calculated using LARAC algorithm.

The route_ calculator module contains two methods which would be called for a pair of source and destination (s,t).
One is Dijkstra, and second is LARAC. LARAC will be called when calculating path for QoS traffic.

The route calculator returns a path between two switches to Route Manager which then calls flow_pusher which, in turn pushes the flows to switches via the controller.

Here is the screenshot of its working -

The route calculator is producing dynamic paths as the network traffic is changing. The counts are the seconds counters effectively showing the new path is re-calculated after every 5 seconds.

flow_pusher
Flow pusher is a module used to push flows into the switches connected to the controller. Each time the route calculator updates routes, the flow_pusher module is called and new routes are reflected on all the switches. While updating flows, we also pass a flag value to distinguish creation of QoS and non-QoS flow between two nodes.

A flow is pushed using a POST call to the REST API exposed by Floodlight controller at `http://<controller_ip>:8080/wm/staticentrypushers/json`[3]
The data field in the POST call is the actual flow we want to send out to the switch.
For example:

```
'{"switch":"00:00:00:00:00:00:00:01", "name":"flow-mod-1",
"in_port":"1","active":"true", "actions":"output=2"}'
```

qos_policies
The multimedia traffic in our network is marked with a value of 40 in the DSCP field of the IP packet.
Thus the QoS policy for multimedia traffic is all packets with DSCP field value = 40 (0x28) {i.e TOS field value = 160 (0xA0)}.

Here is how we are doing it -

```
"eth_type":"0x0800", "ip_dscp":"40", "actions":"output='
```

Controller

This is an OpenFlow controller used to manage OpenFlow enabled Switches.
We are using the FloodLight Controller to implement the controller [1].

Switches

These are OpenFlow enabled switches. We are using Open Virtual Switch to implement these [2].

Traffic Generator

Created as a Python extrapolation on the linux iperf utility. Valid targets and traffic model are defined prior to test.

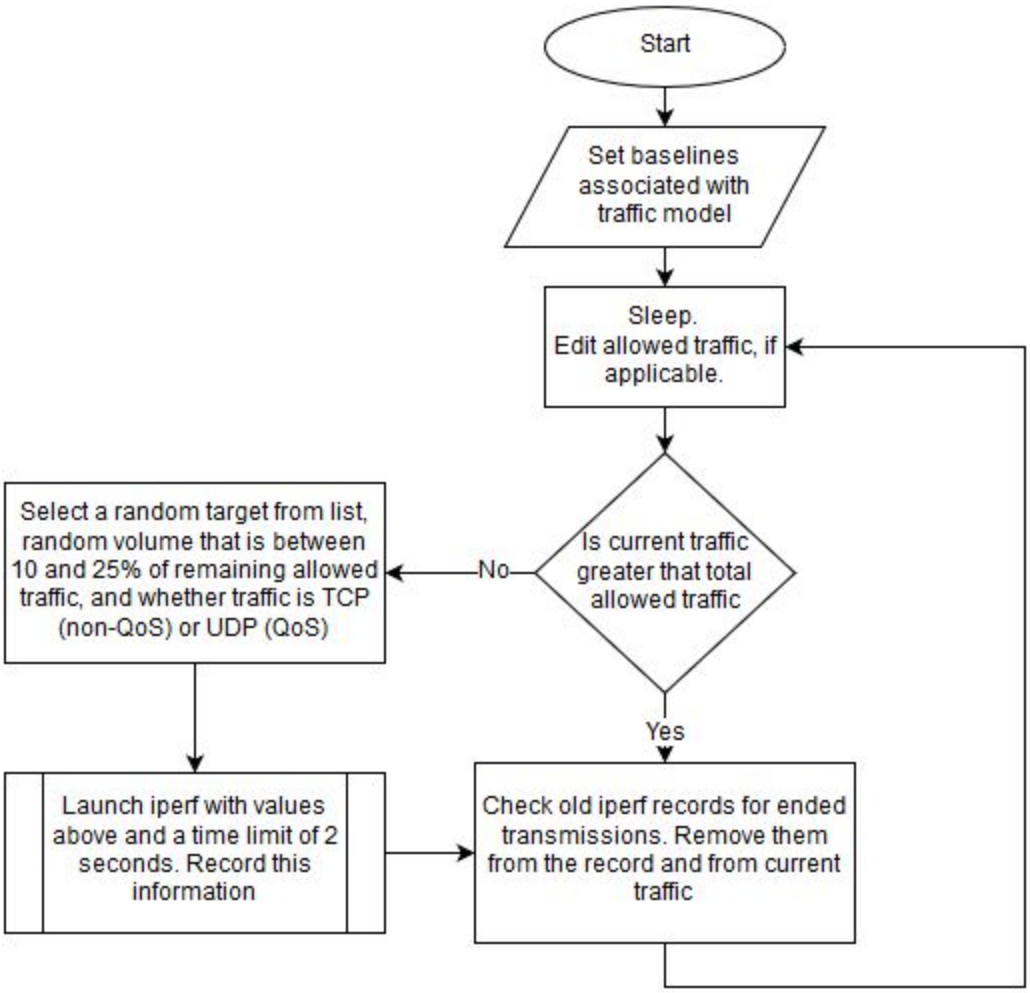


Figure XX: Traffic generator block diagram

QoS

QoS is achieved by computing favourable routes for QoS flows. Thus we essentially manage routing of packets to achieve QoS.

Routing-

Routing is done using both route management and queue management.

Route Management:
Route management for QoS packets is done by LARAC approximation as proposed in the paper[].

The link cost is modified as :

$$c(i,j) = k*bandwidth\_util(i, j) + d$$

where
c(i, j) is the modified link cost between switch i and switch j

bandwidth_util(i, j) is the bandwidth utilization for the link between switch i and switch j
d is the delay which the paper assumes as a constant = 1
On setting bandwidth utilisation threshold as 70% of link capacity,
k = (1-0.7*Link capacity)/(bandwidth_util(i, j) ^2)
Thus, $c(i, j) = k*bandwidth\_util(i, j) + 1$

The QoS path is the one with least total modified cost across all the links from a list of all possible paths.

For non-Qos packets, routing is handled by dijkstra's algorithm which assumes link costs to be constant. Thus non-QoS routes are essentially chosen based on hop-count.

Queue Management

Every switch is configured with two queues; one for QoS packets and one for non-QoS packets.
We assign rate guarantees to the QoS queue to ensure QoS packets always are given some assured bandwidth.
non-QoS queues are not assigned any rate guarantees.

Details of Queue configuration:

Below is the snap of the output queues configured on one of the OVSs

```
root@OVS1:/home/hw5# ovs-vsctl list queue
_uuid               : ec0b1083-b746-4678-9a6c-a6e975aa5a99
dscp                : 0
external_ids        : {}
other_config        : {max-rate="1000000", min-rate="1000000"}

_uuid               : 92a6c143-7dd6-421f-80a7-f883600fd111
dscp                : 40
external_ids        : {}
other_config        : {max-rate="10000000", min-rate="10000000"}
root@OVS1:/home/hw5#
```

Theory for Route Calculation

Routes are calculated differently for QoS and non-QoS traffic. For Non-QoS traffic, it is calculated using the Dijkstra's algorithm(Shortest Path problem).
For QoS traffic we propose to route through a different calculated path, and get better multimedia quality. We are routing the QoS traffic dynamically.

Dynamic QoS Routing[9]-

Dynamic QoS routing differs from shortest path problem in that the shortest path might not be best path due to presence of other traffic, or link congestion.

In our setup, a network is represented as a undirected simple graph G(N,A), where N is the set of nodes and A is the set of all arcs (also called links), so that arc *(i,j)* are sets of vertices, which each set representing an existing edge with nodes(vertices) *i* and *j* as end points.

The route calculation,for QoS-multimedia traffic, is computed using Lagrangian Relaxation Based Aggregated Cost (LARAC) algorithm which minimizes the cost function over a given delay constraint . Non QoS traffic routes are computed using Dijkstra's Shortest Path algorithm with respect to link_cost.

Let R_{st} be the set of all routes (subsets of A) from **source node *s*** to **destination node *t***. For any **path *p*** $\in R_{st}$ we define **cost *c(p)*** and **delay *d(p)*** measures as,

$$c(p) = \sum_{(i,j)\in r} c_{ij}$$

$$d(p) = \sum_{(i,j)\in r} d_{ij}$$

where c_{ij} and d_{ij} are cost and delay coefficients for the arc (i,j), respectively. The CSP problem can then be formally stated as below

$$r* = arg_r \min \{c(p) | p \in R_{st}, d(p) \leq D_{max}\}$$

that is, finding a route *r* *which minimizes the cost function c*(p)* subject to the delay variation d(p) to be less than or equal to a specified value D_{max}.

The cost function is the addition of bandwidth utilization of the link and the delay(dij).

As mentioned in [6], considering that a link is congested if that link has utilized 70% bandwidth, the congestion measure is found as

$$g_{ij} = \begin{cases} \frac{T_{ij} - 0.7 \times B_{ij}}{T_{ij}}, & 0.7 \times B_{ij} < T_{ij} \\ 0, & 0.7 \times B_{ij} \geq T_{ij} \end{cases}$$

where Tij is the total measured traffic amount in bps and Bij is the max achievable bandwidth in bps on link (i, j). The non-congested links have 0 congestion measure value. The delay parameter,as mentioned in [6], corresponds to hop count.

We have assumed *Dmax* as the number of hops in the largest path in a topology, a packet can take from Multimedia Server 1 and Client.

Marking QoS packets

QoS packets are identified by the IP header's DSCP field. This value is set to 40 which translates to a TOS value "0xa0" for a QoS packet. All the non-QoS packets will have a default DSCP value of "000000" which translates to a TOS value "0". In our network topology, Multimedia Server1 is the streaming server to which client has subscribed for high and guaranteed service. Therefore, Multimedia Server1 will mark the packets with DSCP value 40,post routing, just before sending it out on the output interface. The below snap shows the IPTable rule for the same

```
root@MultimediaServer1:/home/hw5# iptables -L -t mangle --line-numbers
Chain PREROUTING (policy ACCEPT)
num  target     prot opt source               destination

Chain INPUT (policy ACCEPT)
num  target     prot opt source               destination

Chain FORWARD (policy ACCEPT)
num  target     prot opt source               destination

Chain OUTPUT (policy ACCEPT)
num  target     prot opt source               destination

Chain POSTROUTING (policy ACCEPT)
num  target     prot opt source               destination
1    DSCP       udp  --  192.168.1.1          192.168.1.3          DSCP set 0x28
root@MultimediaServer1:/home/hw5#
```

The below tcpdump confirms the marking of all QoS Packets originating from Multimedia Server1

```
root@MultimediaServer1:/home/hw5# tcpdump -i any host 192.168.1.1 -v
tcpdump: listening on any, link-type LINUX_SLL (Linux cooked), capture size 65535 bytes
17:28:59.022547 IP (tos 0xa0, ttl 64, id 39181, offset 0, flags [DF], proto UDP (17), length 1498)
    192.168.1.1.34537 > 192.168.1.3.5001: UDP, length 1470
17:28:59.033779 IP (tos 0xa0, ttl 64, id 39184, offset 0, flags [DF], proto UDP (17), length 1498)
    192.168.1.1.34537 > 192.168.1.3.5001: UDP, length 1470
17:28:59.044970 IP (tos 0xa0, ttl 64, id 39185, offset 0, flags [DF], proto UDP (17), length 1498)
    192.168.1.1.34537 > 192.168.1.3.5001: UDP, length 1470
17:28:59.056184 IP (tos 0xa0, ttl 64, id 39188, offset 0, flags [DF], proto UDP (17), length 1498)
    192.168.1.1.34537 > 192.168.1.3.5001: UDP, length 1470
```

V. Development Phases.

Setting up the OpenFlow Virtual Switches

The network topology demands a maximum of six OpenVSwitches to be deployed on the GENI testbed. Configuration of these switches were successfully completed and was done using 'ovs-vsctl' command line tool [7].

Setting up the Controller

Controller Development

Our test cases require an SDN controller for the OpenVswitch management and setup. This must be available or pre-installed in the GENI environment vswitches. We are using Floodlight,which is an OpenFlow controller implementation in Java, as the controller .

Development of QoS Application

Additional modules like switch_information_base, stats_collector, route_manager, route_calculator, flow_pusher,qos_policies ; as mentioned above in the Low Level Design under the title Components are being developed to make available the required functionality into the controller.

Setup the Servers and Client

The Servers and Client hosts the streaming server and client respectively. Apart from that the server hosts the multimedia files to be streamed. The client holds the software which captures the multimedia it received and is able to capture it in a way so as to enable us to view it in the demo playbook.
The network topology demands the presence of two multimedia servers and a client. The server is Command line based while the client is Graphical User Interface based. The servers and client are linux ubuntu 14.04LTS VMs. All devices here will be using VideoLAN's VLC as a streaming/receiving system.

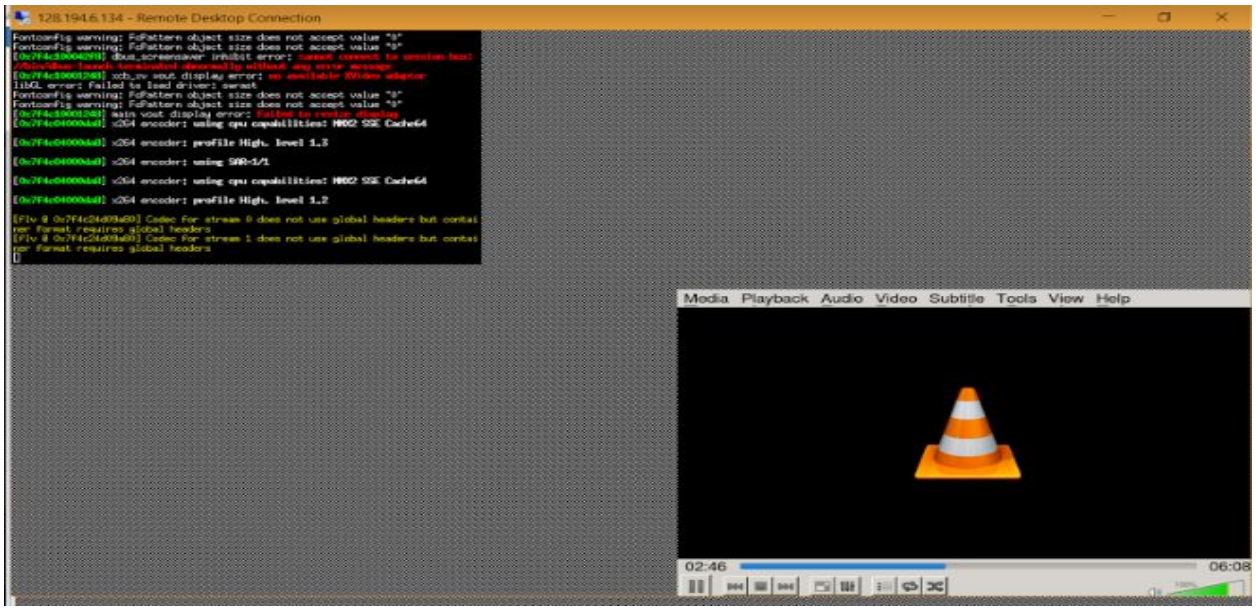
Cross Traffic Generation

Cross Traffic is generated in a pseudo random manner to mimic Internet traffic. Using the iperf tool, a wrapper application handles generation of pseudo random cross traffic and push it into the network.

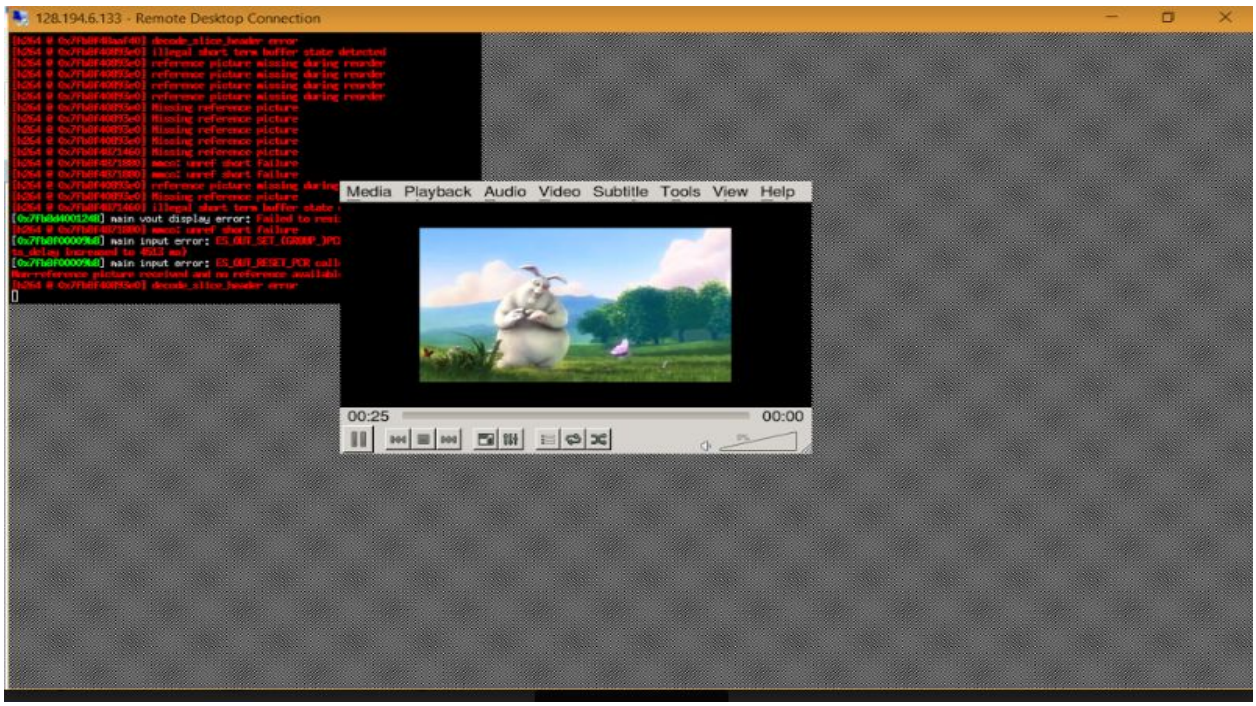
Test the Data Flow without implementing QoS

Testing was successfully carried out without QoS and below is the Multimedia Server1 snap and that of Client receiving the video snap respectively

Multimedia Server 1



Client:



Test Variable Link Costs

Our test cases require some paths which can be preferred over their alternatives, and so we might need to setup topology containing variable cost paths. Hence, at this step of development we would be implementing changes in link costs by modifications in link speed, load etc. to test the automatic routing based on the link metrics. This would be done to ensure that we are as near to a real world scenario as possible.

Setup QoS policy

Add configuration and algorithm to the controller, Setup controller-switch automatic route management, setup queue management in the system.

Test the setup

Final test where all the test cases would be tested and results taken.

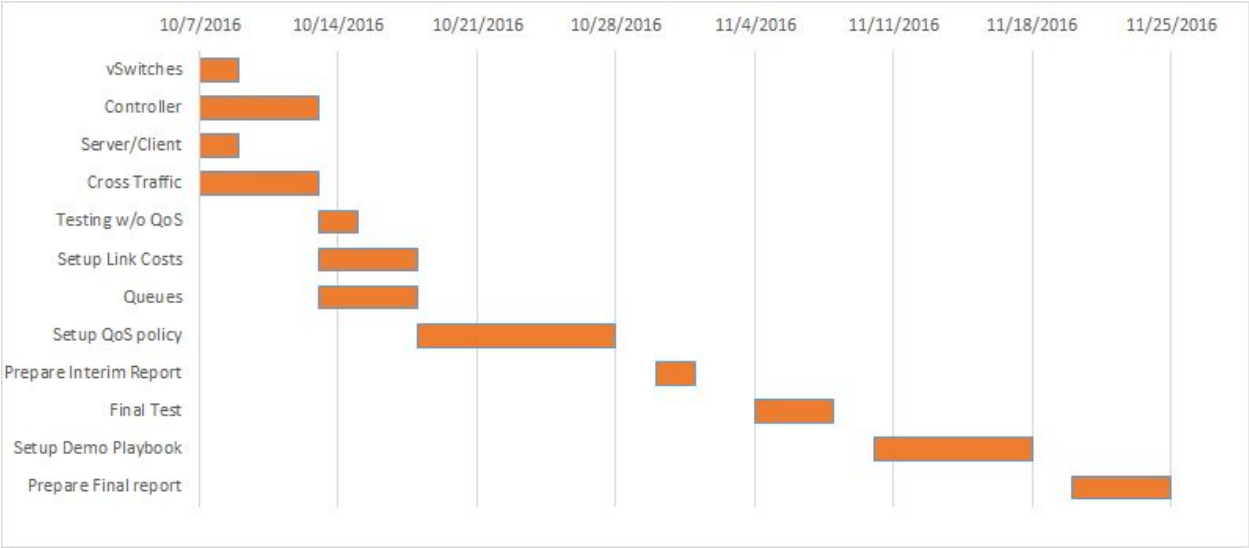
Make Demo Playbook

Programming an API so as to run a demo for the final presentation.

VI. Team Responsibilities.

Phase	Tasks	Time(Man Hours)	Start Date	Complete Date	Contributors	Status
vSwitches	1. Setup of topology 2. Configure queues 3. Test the topology	10	10/7/2016	10/9/2016	Calvin, Bharat	DONE
Controller	1. Setup dummy controller 2. Testing of dummy controller 3. Program the final controller 4. Test final controller	30	10/7/2016	10/13/2016	Kaustubh, Bharat	DONE
Server/Client	1. Setup server and client systems 2. Test streaming and stream capture	10	10/7/2016	10/9/2016	Calvin, Ross	DONE
Cross Traffic	1. Setup manual cross traffic 2. Setup programmatically handled cross traffic	10	10/7/2016	10/13/2016	Ross, Kaustubh	DONE
Testing w/o QoS	1. Run system tests	5	10/13/2016	10/15/2016	Ross, Calvin	DONE
Setup Link Costs	1. Run system tests using different link costs.	5	10/13/2016	10/18/2016	Kaustubh, Calvin	DONE
Queues	1. Experiment using different queue setups	10	10/13/2016	10/18/2016	Ross, Bharat	DONE
Setup QoS policy	1. Algorithm Provisioning 2. Route management 3. Queue management	30	10/18/2016	10/28/2016	Kaustubh, Bharat, Calvin	DONE
Prepare Interim Report	1. Write a report on the state of the project, and finalize all components	15	10/30/2016	11/1/2016	All	DONE
Final Test	1. Execute all tests as listed in section V. Record and Analyse data gathered	10	11/4/2016	11/8/2016	All	DONE
Setup Demo Playbook	1. Setup the system for final demo	20	11/10/2016	11/18/2016	All	DONE
Prepare Final report	Write up results and analysis of the results	20	11/20/2016	11/28/2016	All	DONE

Timeline


10/7/2016
10/14/2016
10/21/2016
10/28/2016
11/4/2016
11/11/2016
11/18/2016
11/25/2016
vSwitches
Controller
Server/Client
Cross Traffic
Testing w/o QoS
Setup Link Costs
Queues
Setup QoS policy
Prepare Interim Report
Final Test
Setup Demo Playbook
Prepare Final report

Network Topologies

Two network topologies, viz. General Topology and Direct Topology, are used to evaluate the success of our implementation. In both the topologies, a pair of multimedia servers are present which are capable of streaming multimedia content. A destination computer acts as the client to these servers. A selection of OpenFlow-enabled switches, under the guidance of the OpenFlow controller, transmits the content over the links. Moreover,additional nodes called Traffic Generators(TG1 and TG2) are connected to some of the switches for the purpose of injecting cross-traffic. The third topology, Binary Topology, is primarily used to demonstrate a special use case as described later.
The details of the nodes common to all the three topologies are as follows

Multimedia Server 1/2: These servers are Virtual Machines(VM) running Ubuntu 14.04LTS . VideoLAN(VLC) media player is installed in each of these servers and further configured to multicast video and audio streams to desired client.

```
root@MultimediaServer1:/home/hw5# uname -a
Linux MultimediaServer1 3.13.0-68-generic #111-Ubuntu SMP Fri Nov 6 18:17:06 UTC 2015 x86_64 x86_64 x86_64 GNU/Linux
root@MultimediaServer1:/home/hw5#
```

Multimedia Server 1 is configured to mark the multimedia packets with DSCP=40.

```
root@MultimediaServer1:/home/hw5# iptables -L -t mangle --line-numbers
Chain PREROUTING (policy ACCEPT)
num  target     prot opt source               destination

Chain INPUT (policy ACCEPT)
num  target     prot opt source               destination

Chain FORWARD (policy ACCEPT)
num  target     prot opt source               destination

Chain OUTPUT (policy ACCEPT)
num  target     prot opt source               destination

Chain POSTROUTING (policy ACCEPT)
num  target     prot opt source               destination
1    DSCP       udp  --  192.168.1.1          192.168.1.3          DSCP set 0x28
root@MultimediaServer1:/home/hw5#
```

OVS(Open vSwitch): OVS is a virtual switch installed in a Virtual Machine(VM) running Ubuntu 14.04LTS. The OVS is installed by executing *apt-get install openvswitch-switch*.
The successful configuration of OVS will result into the below output

```
root@OVS1:/home/hw5# ovs-vsctl show
86225181-6c9e-4d86-b833-71070a60bc06
    Bridge "Sw1"
        Controller "tcp:192.168.2.2:6653"
        fail_mode: secure
        Port "eth4"
            Interface "eth4"
        Port "Sw1"
            Interface "Sw1"
                type: internal
        Port "eth1"
            Interface "eth1"
        Port "eth3"
            Interface "eth3"
        Port "eth5"
            Interface "eth5"
    ovs_version: "2.0.2"
root@OVS1:/home/hw5#
```

Moreover, each OVS has been configured to have two output queues for every output interface. One queue will hold packets with DSCP=40(high priority multimedia packets) and the other queue will hold the rest of the traffic. Below is the snap stating the same

```
root@OVS1:/home/hw5# ovs-vsctl list queue
_uuid               : ec0b1083-b746-4678-9a6c-a6e975aa5a99
dscp                : 0
external_ids        : {}
other_config        : {max-rate="1000000", min-rate="1000000"}

_uuid               : 92a6c143-7dd6-421f-80a7-f883600fd111
dscp                : 40
external_ids        : {}
other_config        : {max-rate="10000000", min-rate="10000000"}
root@OVS1:/home/hw5#
```

Traffic Generator(TG) : These servers are Virtual Machines(VM) running Ubuntu 14.04LTS. They run the python traffic generator described above.

```
root@TG1:/home/hw5# uname -a
Linux TG1 3.13.0-68-generic #111-Ubuntu SMP Fri Nov 6 18:17:06 UTC 2015 x86_64 x86_64 x86_64 GNU/Linux
root@TG1:/home/hw5#
```

Controller: The Controller used is Floodlight Controller that is running on a Ubuntu 14.04LTS Virtual Machine(VM) .

Client: The Client is a Virtual Machines(VM) running Ubuntu 14.04LTS. VideoLAN(VLC) media player is installed and further configured to receive video and audio streams from the server.

Links: The links connecting any two nodes in each of the topology is 100Mbps

1. General Topology

This topology is designed to mimic, in a small-scale, the pathing options available on the wider Internet. Four direct paths and numerous indirect paths provide sufficient complexity to offer non-trivial test cases.

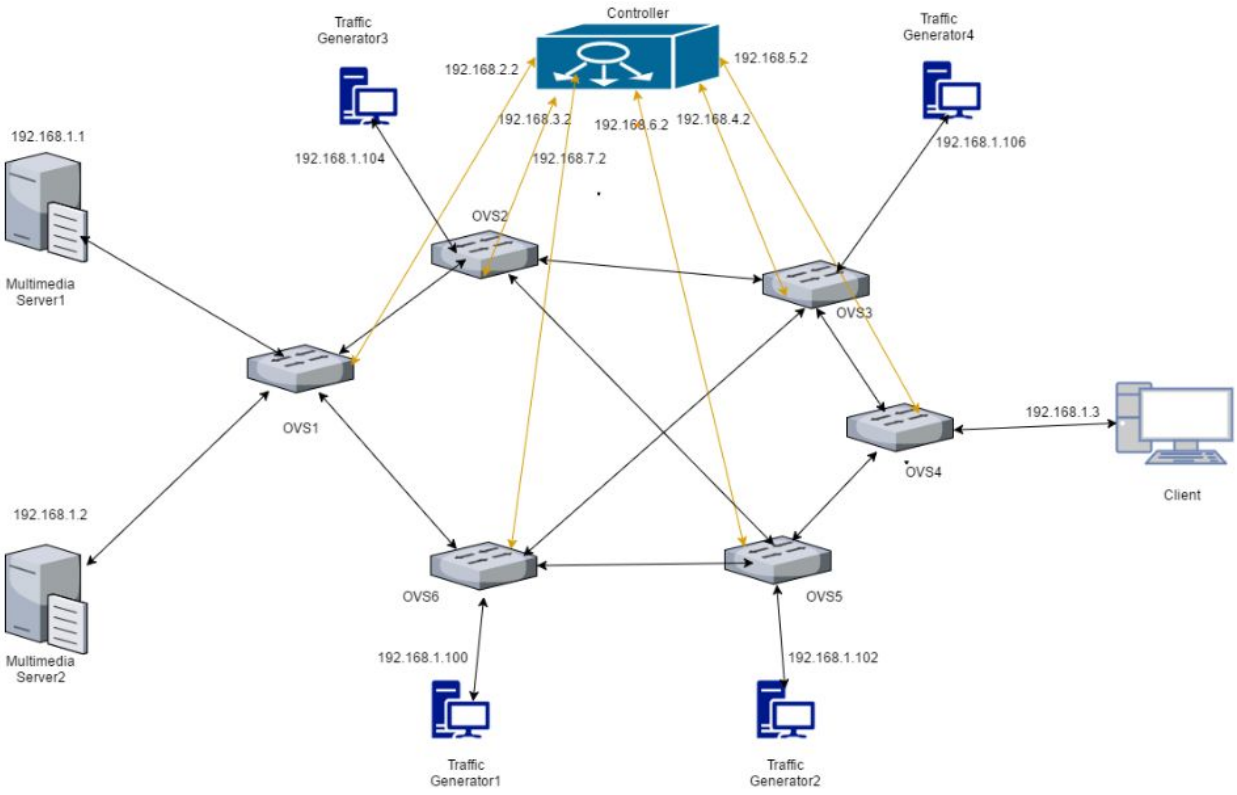


Figure 2: General Topology

2. Direct Topology

The Direct topology allows for direct evaluation of transmission quality in a known situation, that is, a single link with precisely defined traffic. Thus, it is focused on the link between the lone switches without any other influences. This topology makes no attempt to mimic the layout or behavior of the wider Internet and is used for fully controlled experiments.

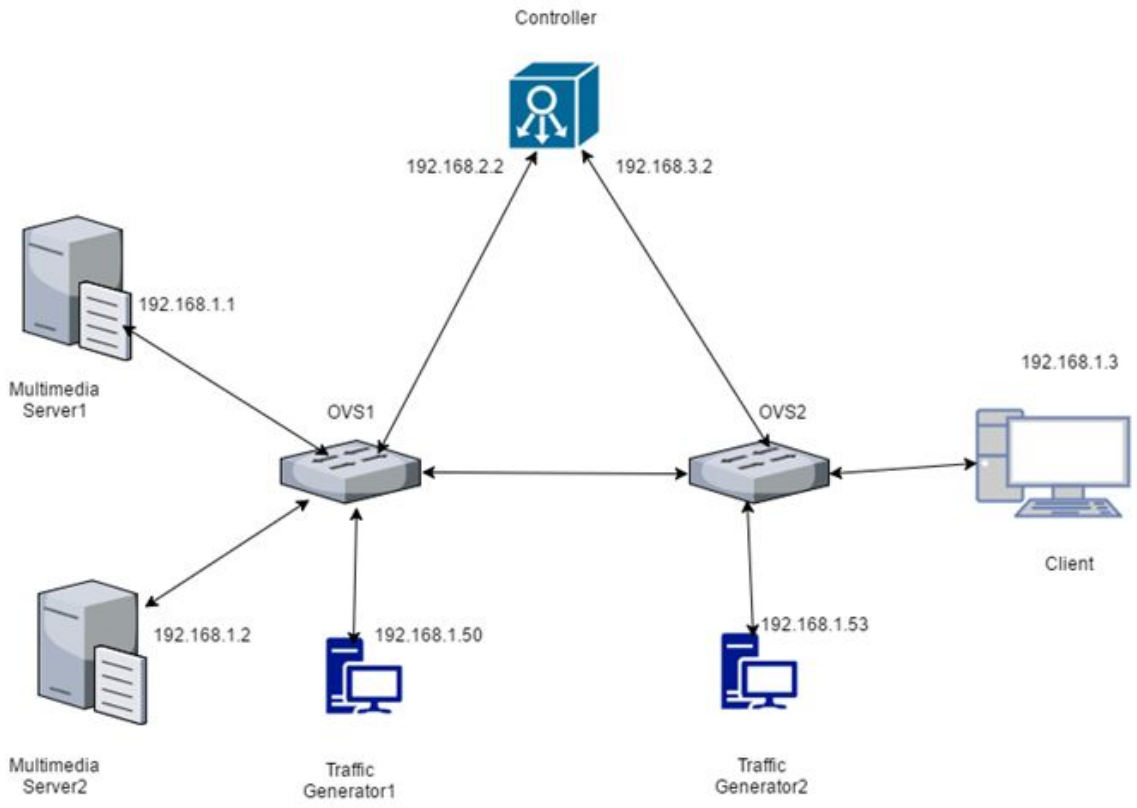


Figure 3: Direct Topology

3. Binary Topology

Similarly to Direct, the Binary topology is designed to show a specific use case. Here the traffic may be routed over either a long high capacity route or a short low capacity route. This topology will be used to test base decision and logic for two circumstances: a link going down and a link becoming full of QoS traffic (See Unit Tests 3 and 4 below for more details). It will not be used for general system tests.

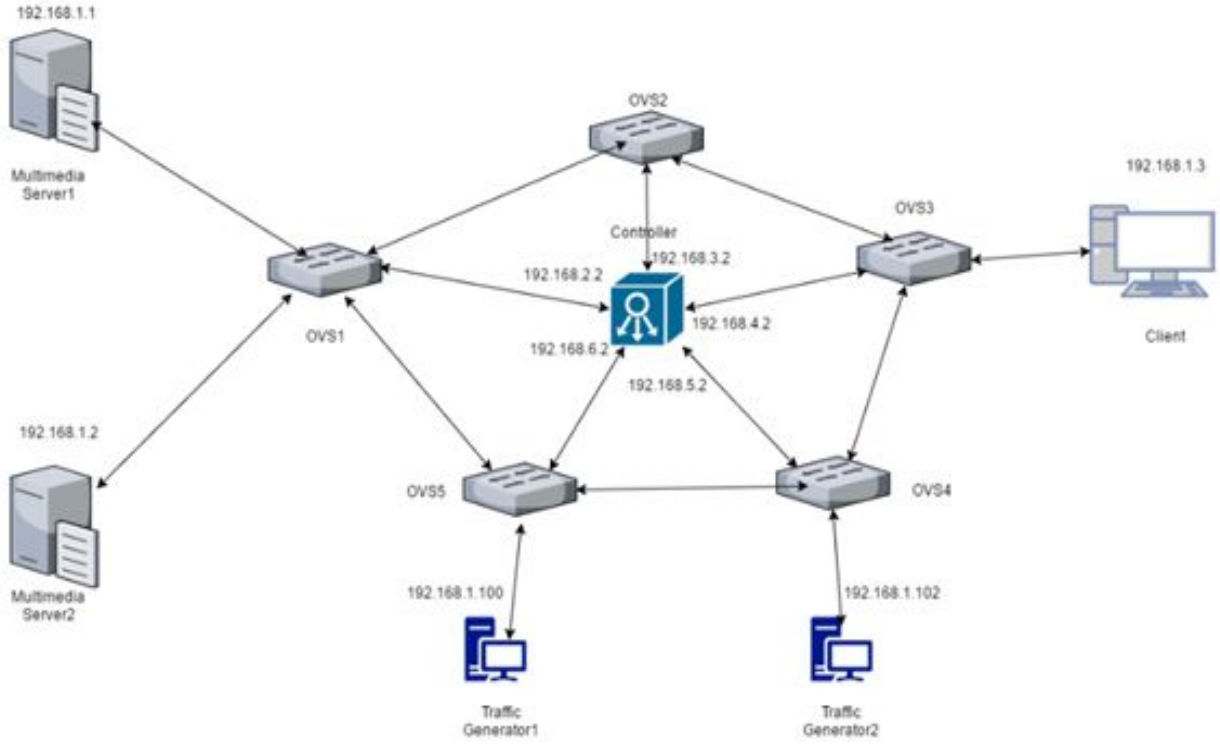


Figure 4: Binary Topology

VII. Testing Framework

A. Testing Procedures (Unit)

1. Ping Test

Actions	Expected observations	Conclusion	Status
Slices are entered into GENI according to the Direct topology and brought online.	Slices are now accessible to SSH.	The simulated network is online.	Done
OpenFlow is configured and launched.	OpenFlow should be running.	The network is prepared for the test.	Done
Pings are sent from nodes to each other.	OpenFlow should provide the forwarding information	OpenFlow is configured and functional.	Done

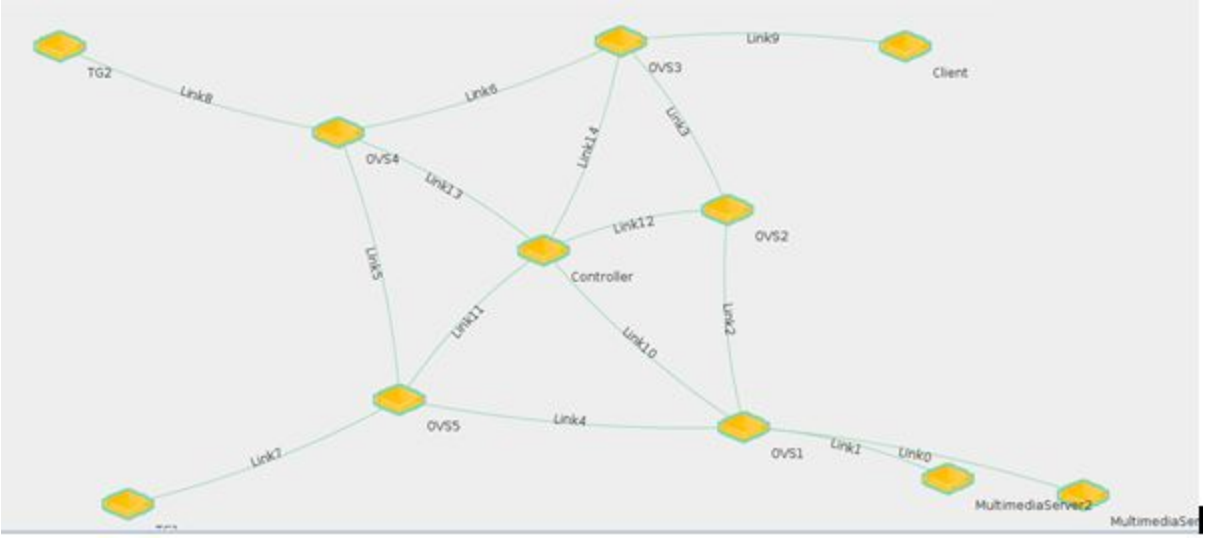
Logs of Ping Test

Multimedia Server1: 192.168.1.1

Multimedia Server2:192.168.1.2

Client:192.168.1.3

Binary Topology



Ping from MultimediaServer1 to Client

```
root@MultimediaServer1:/home/hw5# ping 192.168.1.3
PING 192.168.1.3 (192.168.1.3) 56(84) bytes of data.
64 bytes from 192.168.1.3: icmp_seq=1 ttl=64 time=71.0 ms
64 bytes from 192.168.1.3: icmp_seq=2 ttl=64 time=8.96 ms
64 bytes from 192.168.1.3: icmp_seq=3 ttl=64 time=1.48 ms
64 bytes from 192.168.1.3: icmp_seq=4 ttl=64 time=1.59 ms
^C
--- 192.168.1.3 ping statistics ---
4 packets transmitted, 4 received, 0% packet loss, time 3002ms
rtt min/avg/max/mdev = 1.487/20.783/71.084/29.199 ms
root@MultimediaServer1:/home/hw5#
```

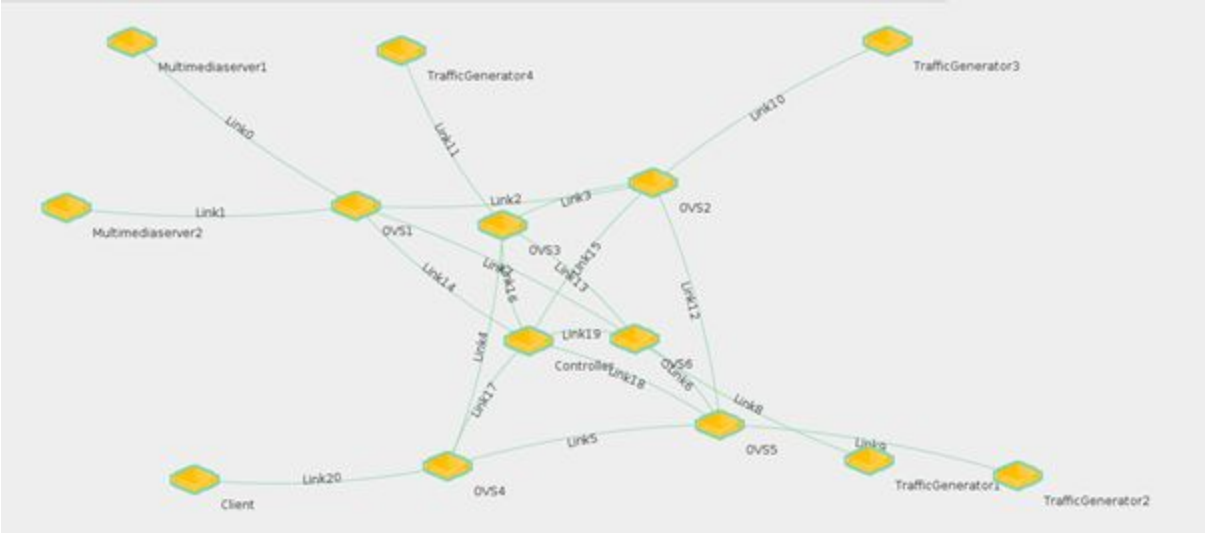
Ping from MultimediaServer2 to Client

```
root@MultimediaServer2:/home/hw5# ping 192.168.1.3
PING 192.168.1.3 (192.168.1.3) 56(84) bytes of data.
64 bytes from 192.168.1.3: icmp_seq=1 ttl=64 time=19.1 ms
64 bytes from 192.168.1.3: icmp_seq=2 ttl=64 time=1.64 ms
64 bytes from 192.168.1.3: icmp_seq=3 ttl=64 time=1.89 ms
64 bytes from 192.168.1.3: icmp_seq=4 ttl=64 time=1.65 ms
^C
--- 192.168.1.3 ping statistics ---
4 packets transmitted, 4 received, 0% packet loss, time 3004ms
rtt min/avg/max/mdev = 1.641/6.088/19.158/7.546 ms
root@MultimediaServer2:/home/hw5#
```

Ping from MultimediaServer1 to MultimediaServer2

```
root@MultimediaServer1:/home/hw5# ping 192.168.1.2
PING 192.168.1.2 (192.168.1.2) 56(84) bytes of data.
64 bytes from 192.168.1.2: icmp_seq=1 ttl=64 time=5.67 ms
64 bytes from 192.168.1.2: icmp_seq=2 ttl=64 time=2.27 ms
64 bytes from 192.168.1.2: icmp_seq=3 ttl=64 time=0.531 ms
64 bytes from 192.168.1.2: icmp_seq=4 ttl=64 time=0.600 ms
^C
--- 192.168.1.2 ping statistics ---
4 packets transmitted, 4 received, 0% packet loss, time 3002ms
rtt min/avg/max/mdev = 0.531/2.269/5.674/2.085 ms
root@MultimediaServer1:/home/hw5#
```

General Topology



Ping from Multimedia Server1 to Client

```
root@Multimediaserver1:/home/hw5# ping 192.168.1.3
PING 192.168.1.3 (192.168.1.3) 56(84) bytes of data.
64 bytes from 192.168.1.3: icmp_seq=1 ttl=64 time=17.6 ms
64 bytes from 192.168.1.3: icmp_seq=2 ttl=64 time=4.72 ms
64 bytes from 192.168.1.3: icmp_seq=3 ttl=64 time=1.59 ms
64 bytes from 192.168.1.3: icmp_seq=4 ttl=64 time=1.30 ms
64 bytes from 192.168.1.3: icmp_seq=5 ttl=64 time=1.46 ms
64 bytes from 192.168.1.3: icmp_seq=6 ttl=64 time=1.70 ms
64 bytes from 192.168.1.3: icmp_seq=7 ttl=64 time=2.21 ms
^C
--- 192.168.1.3 ping statistics ---
7 packets transmitted, 7 received, 0% packet loss, time 6010ms
rtt min/avg/max/mdev = 1.303/4.374/17.610/5.512 ms
root@Multimediaserver1:/home/hw5#
```

Ping from MultimediaServer2 to Client

```
root@Multimediaserver2:/home/hw5# ping 192.168.1.3
PING 192.168.1.3 (192.168.1.3) 56(84) bytes of data.
64 bytes from 192.168.1.3: icmp_seq=1 ttl=64 time=2.53 ms
64 bytes from 192.168.1.3: icmp_seq=2 ttl=64 time=1.21 ms
64 bytes from 192.168.1.3: icmp_seq=3 ttl=64 time=1.31 ms
64 bytes from 192.168.1.3: icmp_seq=4 ttl=64 time=1.48 ms
64 bytes from 192.168.1.3: icmp_seq=5 ttl=64 time=2.03 ms
64 bytes from 192.168.1.3: icmp_seq=6 ttl=64 time=2.00 ms
^C
--- 192.168.1.3 ping statistics ---
6 packets transmitted, 6 received, 0% packet loss, time 5007ms
rtt min/avg/max/mdev = 1.212/1.762/2.531/0.467 ms
root@Multimediaserver2:/home/hw5#
```

Ping from MultimediaServer1 to MultimediaServer2

```
root@Multimediaserver1:/home/hw5# ping 192.168.1.2
PING 192.168.1.2 (192.168.1.2) 56(84) bytes of data.
64 bytes from 192.168.1.2: icmp_seq=1 ttl=64 time=1.94 ms
64 bytes from 192.168.1.2: icmp_seq=2 ttl=64 time=7.81 ms
64 bytes from 192.168.1.2: icmp_seq=3 ttl=64 time=1.78 ms
64 bytes from 192.168.1.2: icmp_seq=4 ttl=64 time=1.52 ms
64 bytes from 192.168.1.2: icmp_seq=5 ttl=64 time=1.85 ms
64 bytes from 192.168.1.2: icmp_seq=6 ttl=64 time=1.92 ms
64 bytes from 192.168.1.2: icmp_seq=7 ttl=64 time=1.75 ms
^C
--- 192.168.1.2 ping statistics ---
7 packets transmitted, 7 received, 0% packet loss, time 6007ms
rtt min/avg/max/mdev = 1.524/2.655/7.818/2.112 ms
root@Multimediaserver1:/home/hw5#
```

2. Routing Test

Actions	Expected observations	Conclusion	Status
Floodlight's Forwarding module is off. Routing module is off.	Ping shouldn't work	Currently there are no routes on the switches.	Done
Turn on the routing module	Ping should start working	Flows are getting pushed on the switches	Done

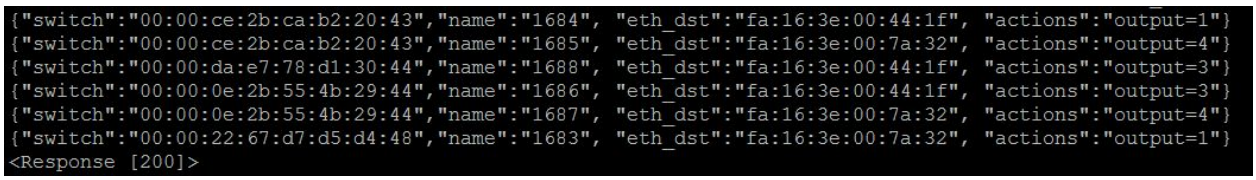
Routing Test Logs:

Initially ping doesn't work

Log:

```
root@Multimediaserver1:/home/hw5# ping 192.168.1.3
PING 192.168.1.3 (192.168.1.3) 56(84) bytes of data.
^C
--- 192.168.1.3 ping statistics ---
4 packets transmitted, 0 received, 100% packet loss, time 2999ms
```

Now we push flows on the switches.
flow pusher in action:



Ping should now start working.

Log:

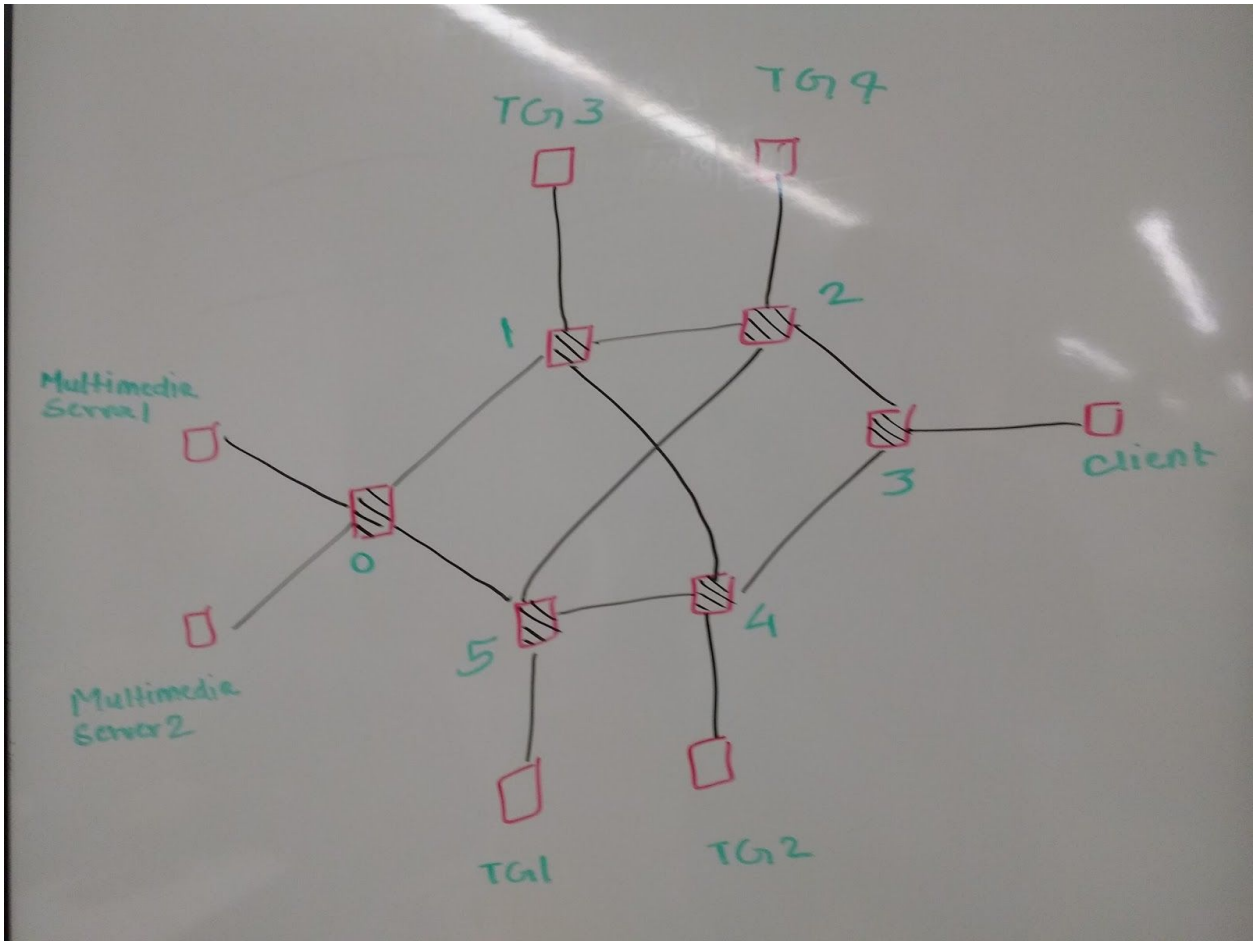
```
root@Multimediaserver1:/home/hw5# ping 192.168.1.3
PING 192.168.1.3 (192.168.1.3) 56(84) bytes of data.
64 bytes from 192.168.1.3: icmp_seq=1 ttl=64 time=3.98 ms
64 bytes from 192.168.1.3: icmp_seq=2 ttl=64 time=2.42 ms
64 bytes from 192.168.1.3: icmp_seq=3 ttl=64 time=1.49 ms
^C
--- 192.168.1.3 ping statistics ---
3 packets transmitted, 3 received, 0% packet loss, time 2002ms
rtt min/avg/max/mdev = 1.499/2.635/3.981/1.024 ms
```

3. Dynamic Routing Test (General Topology)

Action	Expected Observations	Conclusion	Status
Route Manager is brought up on the topology	Calculates and pushes QoS routes on the switches based on bandwidth utilization	Route management is working as expected.	Done
Link on the currently chosen path is flooded with traffic by the traffic generator.	Route manager should dynamically select a different path to route QoS flows	QoS routes are always guaranteed to be routed along the path of least bandwidth utilization	Done

Dynamic Routing Test Logs:

Given below is the general topology
The nodes numbered as switches. We assume all the switches are connected to the controller for the sake of not messing up the diagram.



The route manager is brought up and initially it calculates the route [0, 1, 2, 3] for QoS packets.
The traffic generators then kick in and flood link between switches 1 and 2.
The route manager now switches the path to [0, 1, 4, 3]
Then, the link between switch 4 and 1 is flooded and the route is switched to [0, 5, 4, 3]
Now link between switch 5 and 4 is flooded and the route is switched to [0, 5, 2, 3]

Test Logs

```
switch     00:00:56:40:08:c1:cd:43
     port id         1     bandwidth_util  0.0   Mbits/sec
     port id         3     bandwidth_util  0.0   Mbits/sec
     port id         2     bandwidth_util  0.0   Mbits/sec
     port id         4     bandwidth_util  0.0   Mbits/sec
     port id         local      bandwidth_util  0.0   Mbits/sec
switch     00:00:1e:96:5d:c9:b0:49
     port id         1     bandwidth_util  0.0   Mbits/sec
     port id         3     bandwidth_util  0.0   Mbits/sec
     port id         2     bandwidth_util  0.0   Mbits/sec
     port id         local      bandwidth_util  0.0   Mbits/sec
switch     00:00:da:f5:a7:c4:3a:49
     port id         1     bandwidth_util  0.0   Mbits/sec
     port id         3     bandwidth_util  0.0   Mbits/sec
     port id         2     bandwidth_util  0.0   Mbits/sec
     port id         4     bandwidth_util  0.0   Mbits/sec
     port id         local      bandwidth_util  0.0   Mbits/sec
switch     00:00:4a:ff:6e:89:e4:42
     port id         1     bandwidth_util  0.0   Mbits/sec
     port id         3     bandwidth_util  0.0   Mbits/sec
     port id         2     bandwidth_util  0.0   Mbits/sec
     port id         4     bandwidth_util  0.0   Mbits/sec
     port id         local      bandwidth_util  0.0   Mbits/sec
switch     00:00:1a:9b:f5:c7:10:47
     port id         1     bandwidth_util  0.0   Mbits/sec
     port id         3     bandwidth_util  0.0   Mbits/sec
     port id         2     bandwidth_util  0.0   Mbits/sec
     port id         4     bandwidth_util  0.0   Mbits/sec
     port id         local      bandwidth_util  0.0   Mbits/sec
switch     00:00:72:40:b2:ac:47:4b
     port id         1     bandwidth_util  0.0   Mbits/sec
     port id         3     bandwidth_util  0.0   Mbits/sec
     port id         2     bandwidth_util  0.0   Mbits/sec
     port id         4     bandwidth_util  0.0   Mbits/sec
     port id         local      bandwidth_util  0.0   Mbits/sec
<Response [200]>
[0, 1, 2, 3]
<Response [200]>
```

```
switch    00:00:56:40:08:c1:cd:43
     port id         1    bandwidth_util  11.16 Mbits/sec
     port id         3    bandwidth_util  11.16 Mbits/sec
     port id         2    bandwidth_util  0.0  Mbits/sec
     port id         4    bandwidth_util  0.0  Mbits/sec
     port id         local     bandwidth_util  0.0  Mbits/sec
switch    00:00:1e:96:5d:c9:b0:49
     port id         1    bandwidth_util  0.0  Mbits/sec
     port id         3    bandwidth_util  0.0  Mbits/sec
     port id         2    bandwidth_util  0.0  Mbits/sec
     port id         local     bandwidth_util  0.0  Mbits/sec
switch    00:00:4a:ff:6e:89:e4:42
     port id         1    bandwidth_util  0.0  Mbits/sec
     port id         3    bandwidth_util  0.0  Mbits/sec
     port id         2    bandwidth_util  0.0  Mbits/sec
     port id         4    bandwidth_util  0.0  Mbits/sec
     port id         local     bandwidth_util  0.0  Mbits/sec
switch    00:00:da:f5:a7:c4:3a:49
     port id         1    bandwidth_util  11.16     Mbits/sec
     port id         3    bandwidth_util  0.0  Mbits/sec
     port id         2    bandwidth_util  11.16     Mbits/sec
     port id         4    bandwidth_util  0.0  Mbits/sec
     port id         local     bandwidth_util  0.0  Mbits/sec
switch    00:00:1a:9b:f5:c7:10:47
     port id         1    bandwidth_util  0.0  Mbits/sec
     port id         3    bandwidth_util  0.0  Mbits/sec
     port id         2    bandwidth_util  0.0  Mbits/sec
     port id         4    bandwidth_util  0.0  Mbits/sec
     port id         local     bandwidth_util  0.0  Mbits/sec
switch    00:00:72:40:b2:ac:47:4b
     port id         1    bandwidth_util  0.0  Mbits/sec
     port id         3    bandwidth_util  0.0  Mbits/sec
     port id         2    bandwidth_util  0.0  Mbits/sec
     port id         4    bandwidth_util  0.0  Mbits/sec
     port id         local     bandwidth_util  0.0  Mbits/sec
[0, 1, 4, 3]
<Response [200]>
switch    00:00:56:40:08:c1:cd:43
     port id         1    bandwidth_util  10.09     Mbits/sec
     port id         3    bandwidth_util  10.09     Mbits/sec
     port id         2    bandwidth_util  0.0  Mbits/sec
```

```
     port id         4    bandwidth_util  0.0  Mbits/sec
     port id         local     bandwidth_util  0.0  Mbits/sec
switch    00:00:1e:96:5d:c9:b0:49
     port id         1    bandwidth_util  0.0  Mbits/sec
     port id         3    bandwidth_util  0.0  Mbits/sec
     port id         2    bandwidth_util  0.0  Mbits/sec
     port id         local     bandwidth_util  0.0  Mbits/sec
switch    00:00:4a:ff:6e:89:e4:42
     port id         1    bandwidth_util  0.0  Mbits/sec
     port id         3    bandwidth_util  0.0  Mbits/sec
     port id         2    bandwidth_util  0.0  Mbits/sec
     port id         4    bandwidth_util  0.0  Mbits/sec
     port id         local     bandwidth_util  0.0  Mbits/sec
switch    00:00:da:f5:a7:c4:3a:49
     port id         1    bandwidth_util  10.14     Mbits/sec
     port id         3    bandwidth_util  10.0 Mbits/sec
     port id         2    bandwidth_util  20.14     Mbits/sec
     port id         4    bandwidth_util  0.0  Mbits/sec
     port id         local     bandwidth_util  0.0  Mbits/sec
switch    00:00:1a:9b:f5:c7:10:47
     port id         1    bandwidth_util  0.0  Mbits/sec
     port id         3    bandwidth_util  0.0  Mbits/sec
     port id         2    bandwidth_util  9.97 Mbits/sec
     port id         4    bandwidth_util  9.97 Mbits/sec
     port id         local     bandwidth_util  0.0  Mbits/sec
switch    00:00:72:40:b2:ac:47:4b
     port id         1    bandwidth_util  0.0  Mbits/sec
     port id         3    bandwidth_util  0.0  Mbits/sec
     port id         2    bandwidth_util  0.0  Mbits/sec
     port id         4    bandwidth_util  0.0  Mbits/sec
     port id         local     bandwidth_util  0.0  Mbits/sec
[0, 5, 4, 3]
<Response [200]>
switch    00:00:56:40:08:c1:cd:43
     port id         1    bandwidth_util  10.1 Mbits/sec
     port id         3    bandwidth_util  10.1 Mbits/sec
     port id         2    bandwidth_util  0.0  Mbits/sec
     port id         4    bandwidth_util  0.0  Mbits/sec
     port id         local     bandwidth_util  0.0  Mbits/sec
switch    00:00:1e:96:5d:c9:b0:49
     port id         1    bandwidth_util  0.0  Mbits/sec
```

```
     port id         3    bandwidth_util  0.0  Mbits/sec
     port id         2    bandwidth_util  0.0  Mbits/sec
     port id         local     bandwidth_util  0.0  Mbits/sec
switch    00:00:4a:ff:6e:89:e4:42
     port id         1    bandwidth_util  10.48     Mbits/sec
     port id         3    bandwidth_util  0.0  Mbits/sec
     port id         2    bandwidth_util  10.48     Mbits/sec
     port id         4    bandwidth_util  0.0  Mbits/sec
     port id         local     bandwidth_util  0.0  Mbits/sec
switch    00:00:da:f5:a7:c4:3a:49
     port id         1    bandwidth_util  10.09     Mbits/sec
     port id         3    bandwidth_util  10.42     Mbits/sec
     port id         2    bandwidth_util  20.51     Mbits/sec
     port id         4    bandwidth_util  0.0  Mbits/sec
     port id         local     bandwidth_util  0.0  Mbits/sec
switch    00:00:1a:9b:f5:c7:10:47
     port id         1    bandwidth_util  0.0  Mbits/sec
     port id         3    bandwidth_util  10.47     Mbits/sec
     port id         2    bandwidth_util  10.43     Mbits/sec
     port id         4    bandwidth_util  20.9 Mbits/sec
     port id         local     bandwidth_util  0.0  Mbits/sec
switch    00:00:72:40:b2:ac:47:4b
     port id         1    bandwidth_util  0.0  Mbits/sec
     port id         3    bandwidth_util  0.0  Mbits/sec
     port id         2    bandwidth_util  0.0  Mbits/sec
     port id         4    bandwidth_util  0.0  Mbits/sec
     port id         local     bandwidth_util  0.0  Mbits/sec
[0, 5, 2, 3]
<Response [200]>
```

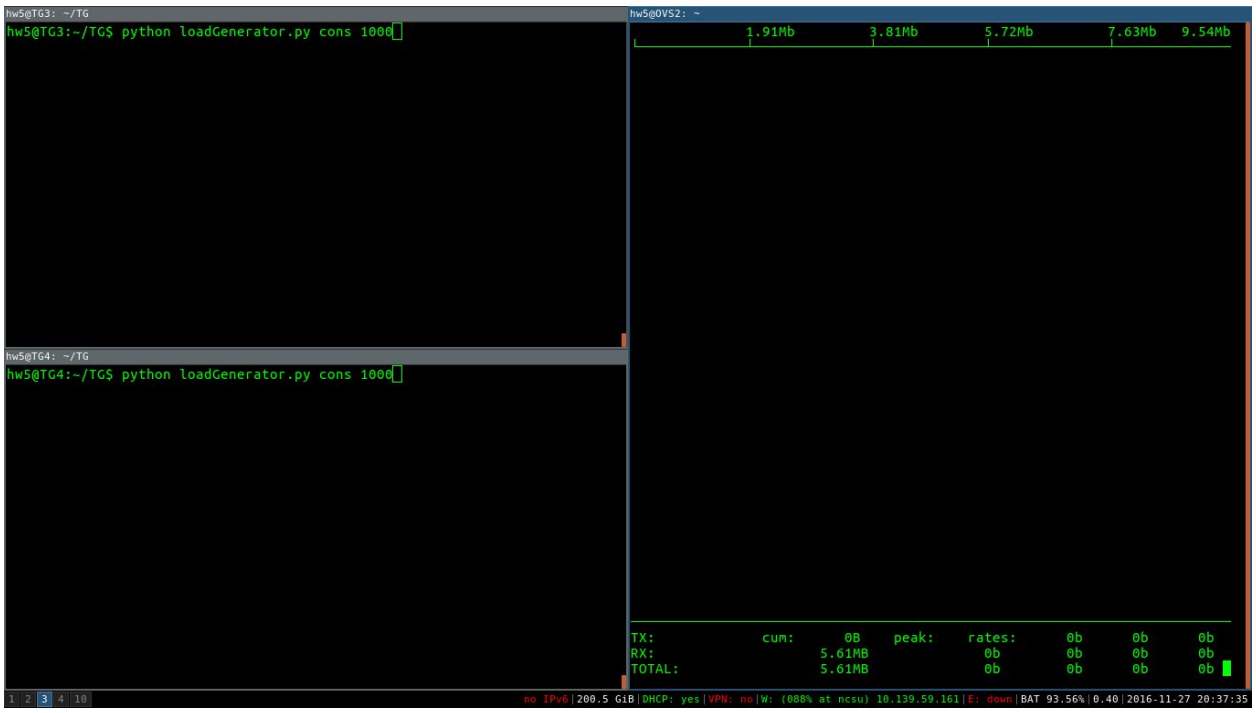
4. Traffic Generation Test

Actions	Expected observations	Conclusion	Status
The Traffic Generator is configured to run different packet volumes (ie 1 Mb/s, 10 Mb/s, 100 Mb/s)	The generator sends approximately that much traffic over the network, allowing for randomization.	The Traffic Generator is functioning as expected.	Done

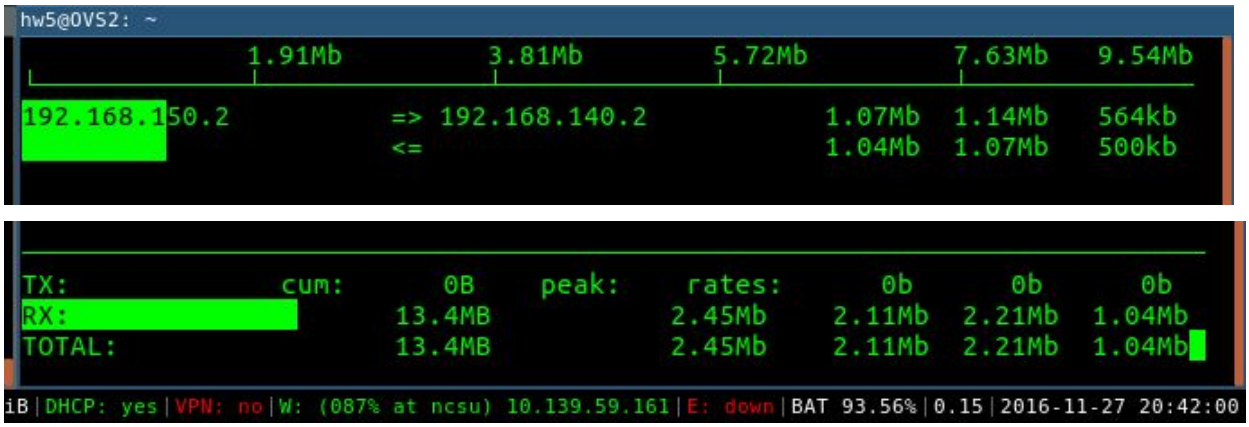
Traffic Generation Test Logs

Peer to Peer

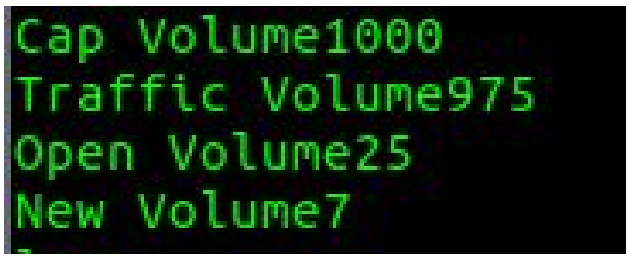
A portion of the network is used from the general topology: Traffic Generators 3&4, and Switches 2&3. A network monitor tool (iftop, although any other would do) is set up on the link between TG 3 and OVS 2. No other traffic is running.



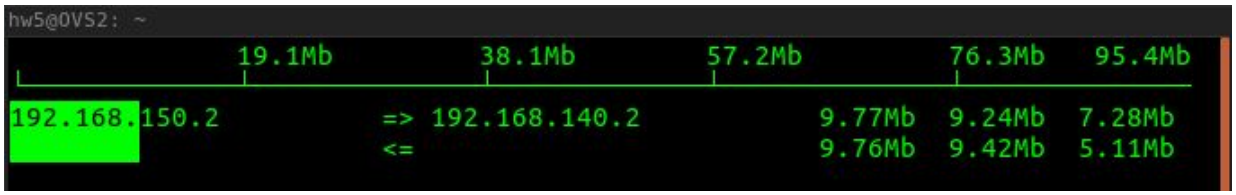
The Traffic Generators are setup to generate approximately 1Mb of traffic in each direction. They are launched, and allowed to normalize, which takes less than 5 seconds. The network monitor then reads as follows, showing about 1Mb/s to and from these two Generators, and about 2Mb/s overall:

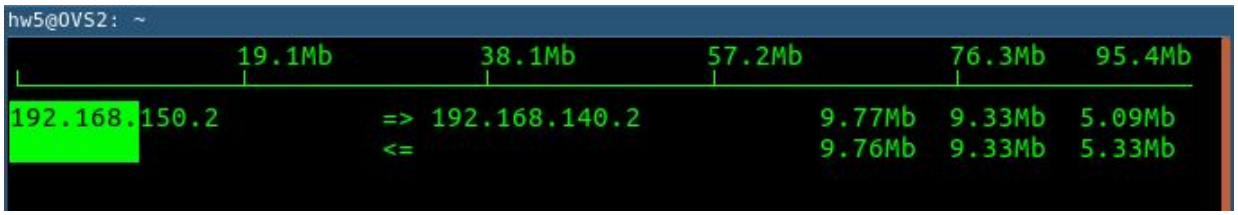


The Traffic Generators output continual statistics as well, where Cap Volume is the maximum amount allowed, Traffic Volume is the current amount sent, Open Volume is the amount of bandwidth left, and New Volume is the amount the will be sent in a new stream. All of these are in kb.



The tests are repeated at 10Mb/s, 100Mb/s:



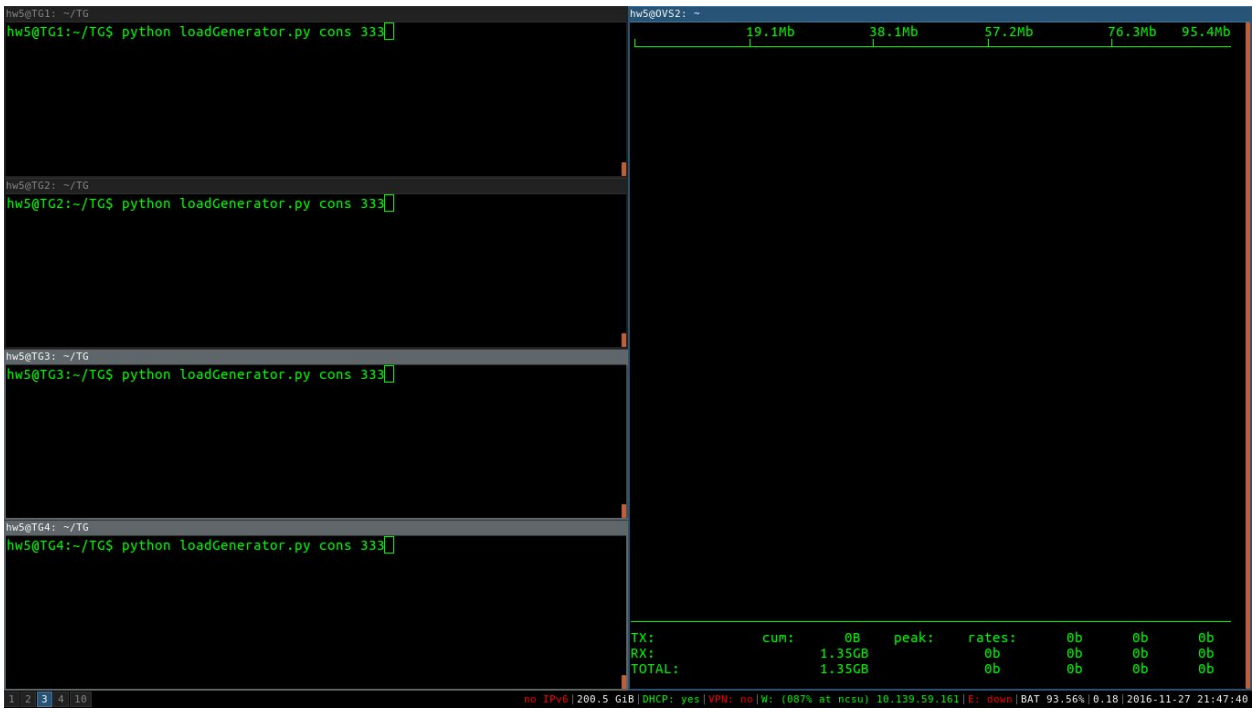


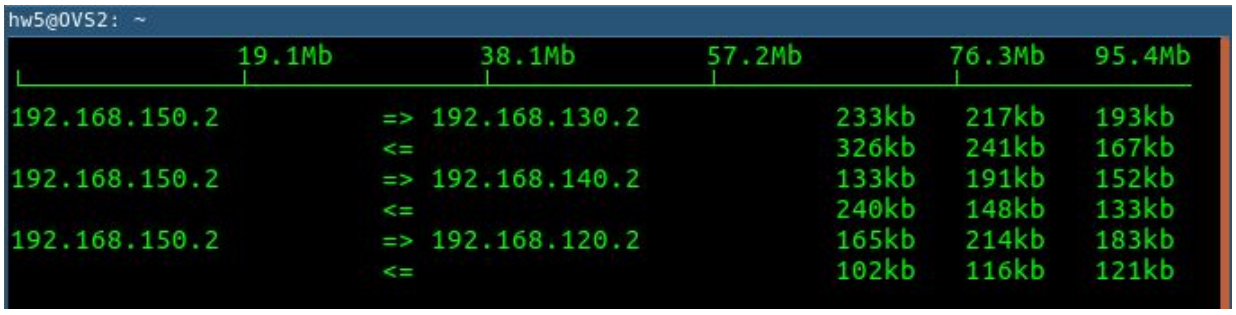
The test at 100Mb/s confirms that the links have a maximum capacity of 10Mb/s, and that even the 10Mb/s test fills the bandwidth as much as reasonably possible.

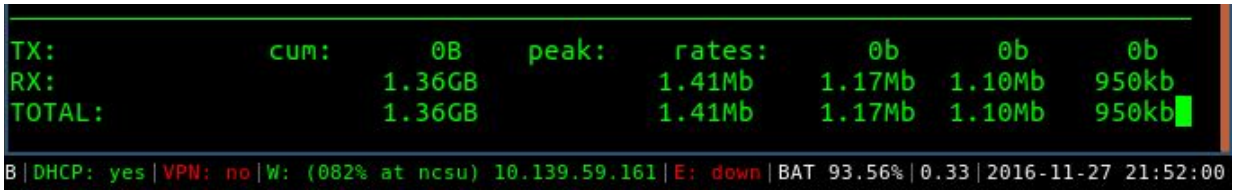
This cap is later found to be in error, but as the tests satisfied the testing conditions anyway, we leave the results.

Network

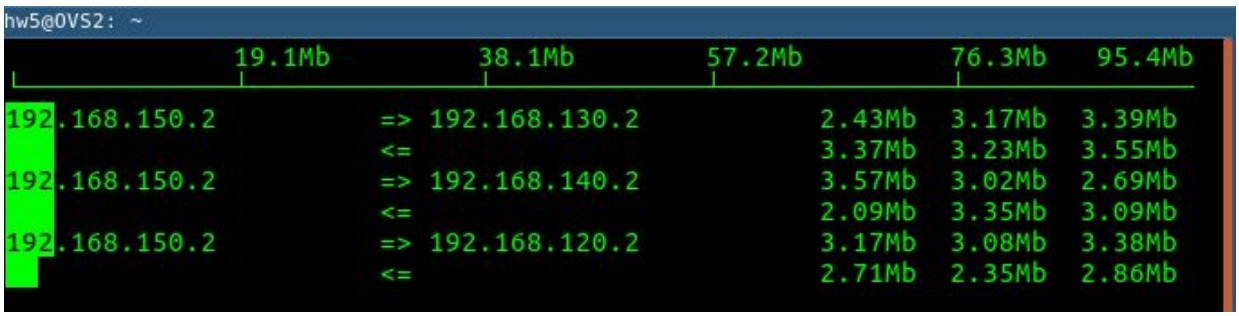
All the Traffic Generators and the rest of the switches are brought online, bringing our usage up to TGs 1-4, as well as Switches 1-6. The Traffic Generators are again configured to generate 1Mb/s on each link between the four of them.

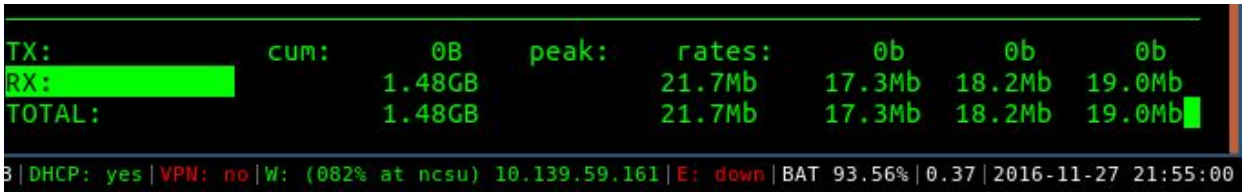






Results from the 10 Mb/s test:





5. Call Admission/ Route Calculation Test

Actions	Expected observations	Conclusion
Slices are entered into GENI according to the Binary topology and brought online. The shorter path is configured to be preferable (high bandwidth, low latency) to the other.	Slices are now accessible to SSH	The simulated network is online.
Create an iperf QoS-UDP stream from the server to the client	Data flows over the shorter path.	The controller is routing packets optimally.
Create enough QoS-UDP flows from S1 to S3 to fill the link.	All streams are added to the shorter path.	The controller is able to fill the link.
Create an iperf QoS-UDP stream from the server to the client	Data flows over the longer path.	The controller is able to select a non-optimal path that will still guarantee QoS.

Call admission control states that preventing the admission on of a new QoS customer on the QoS network inorder to avoid degradation of services of the already serving customers. This test required us to do additional configuration on the controller end.Due to time constraint, we were unable to pass this test case.

B. Testing Procedures (System)

1. General:

The general format of all tests is the same. Each Test is distinguished by a topology and configuration, as well as special cases. Each Test contains three Sets, distinguished by w hether the Test is running a non-QoS stream, a QoS stream, or both. Each Set is then Run 3 times for accuracy. Deviations from this procedure occur only in the noted instances.

In all tests, packet capture software (tshark/wireshark) should be run at the destination node. The utility dropwatch will also be installed on the switches, if possible, to monitor where and when packets are lost.

Quality is defined in the tests below as a combination of throughput, packet drop, and signal-to-noise ratio of the streamed signal. The data being streamed will be a 30 second clip of either audio or video.

Action	Expected Observations	Conclusion	Status
Slices are entered into GENI according to the topology and brought online.	Slices are now accessible to SSH.	The simulated network is online.	DONE
Appropriate software is installed and OpenFlow is configured.	Applications should be functioning. OpenFlow should log pings as being routed.	The network is prepared for the test.	DONE
Appropriate streaming servers and clients are started, and prepared to begin streaming.	The programs should be running.	The streams are ready to begin.	DONE
Test Specific configurations are executed here.			
Traffic Generators	Tshark instances	The network is now	DONE

are brought online.	should log background packets.	simulating its load.	
Test is executed here.			

2. General Topology Tests

These tests should be run on the General Topology. The tests are meant to replicate the diversity of possible situations that occur on the wider internet. Test 1 is designed to compare QoS traffic to non-QoS traffic in a fully loaded network. Test 2 is designed to all for comparison within both QoS and non-Qos traffic as the network starts out with little traffic.

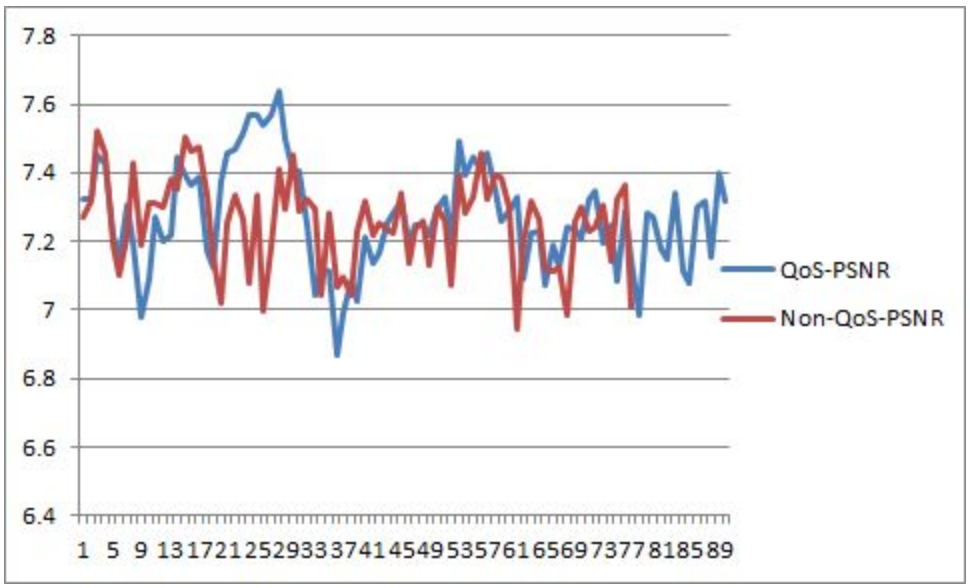
Test 1	Actions	Expected observations	Conclusion
	Test Specific configurations		
	Traffic Generators should be configured to load the network completely. The exact value should be ⅔ of the link capacity, as some of the traffic will need to take a two-hop path to reach its destination		
	Test Execution		
t=0	Activate streaming application(s).	Packets begin arriving at destination.	The stream is being forwarded properly.
t=0-30	Stream transmits data.	QoS traffic transmits clearly with low drop rate and high throughput. Non-QoS is of lesser quality, with higher drop and more noise on the signal.	The QoS allows for clear transmission across a busy network, were normal traffic suffer degradation.
t=30	Streaming ceases.	Packets finish arriving. Data has been gathered.	Recorded comparison can validate QoS improvement.

Logs:

Non-QoS video received at Client

QoS video received at Client

Peak Signal to Noise Ratio(PSNR) of the videos received at client

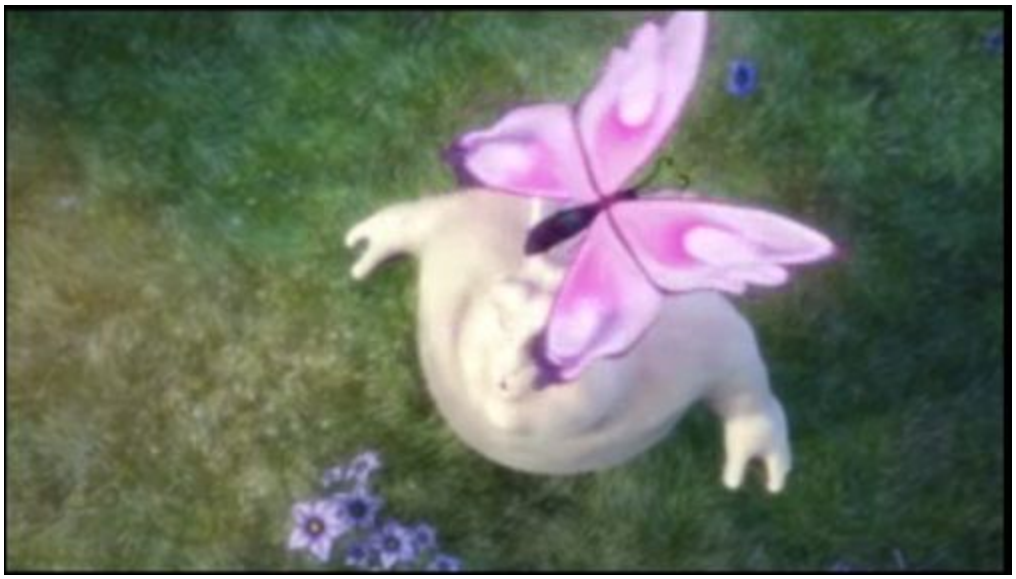


Test 2	Actions	Expected observations	Conclusion
	Test Specific configurations		
	Traffic Generators should be configured to load the network slightly. ⅓ of link capacity should be sufficient. It must also have configured burst traffic to happen on 5 second intervals, for 5 seconds each. These burst should be set to overwhelm (3/3) link capacity.		
	Test Execution		
t=0	Activate streaming application(s).	Packets begin arriving at destination.	The stream is being forwarded as expected.
t=0-5	Stream transmits data.	All streams should arrive normally with negligible packet drop.	The network has sufficient space for all traffic.
t=5-10	Burst begins.	QoS traffic continues transmitting clearly. Non-QoS traffic harshly drops packets and signal	QoS can maintain data flow over an overloaded network.

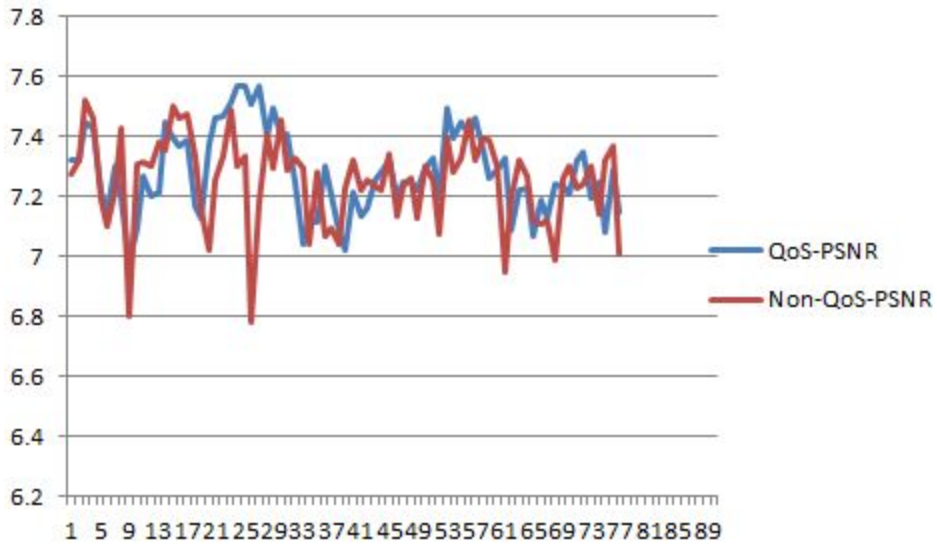
		degrades.	
t=10-30	Repeat the last two steps.		
t=30	Streaming ceases.	Packets finish arriving. Data has been gathered.	Recorded comparison can validate QoS improvement.

Non-QoS video received at Client

QoS video received at Client



Peak Signal to Noise Ratio(PSNR) of the videos received at client



3. Direct Topology Tests

These tests should be run on the Direct Topology. The tests are meant to gather data in a less random environment. Test 1 is designed to allow statistics to be gathered as load gradually increases on a link. Test 2 is designed to have a direct comparison between QoS and non-QoS traffic that are running precisely in parallel.

Test 1	Actions	Expected observations	Conclusion
	Test Specific configurations		
	Traffic generators should be configured to increase load over the S1-S2 link after being triggered until the generator is producing enough traffic to fill the link. The gradient should take 10 seconds. It should then maintain that volume for another 10 seconds.		
	Test Execution		
t=0	Activate streaming application(s).	Packets begin arriving at destination.	The stream is being properly forwarded.
t=5	Begin cross-traffic.	Non-QoS quality begins dropping packets and the signal, QOS quality maintains.	The QoS is allowing important traffic to flow despite significant obstruction.
t=25	cross-traffic ends.	Quality recovers for non-QoS, QoS should continue unaffected.	The link has returned to an under-utilized state.
	End of Run		

Non-QoS video received at Client

QoS video received at Client

Test 2	Actions	Expected observations	Conclusion
	Test Specific configurations		
	Traffic generators should be configured to immediately increase load over the S1-S2 link after being triggered. This traffic should be sufficient to fill the link. It should then maintain that volume for another 10 seconds.		
	Test Execution		

t=0	Activate streaming application(s).	Packets begin arriving at destination.	The stream is being properly forwarded.
t=10	Begin cross-traffic.	Non-QoS begins dropping packets and the signal, QOS quality maintains.	The QoS is allowing important traffic to flow despite significant obstruction.
t=20	Cross-traffic ends.	Quality recovers.	The link has returned to an under-utilized state.
	End of Run		

Non-QoS video received at Client

QoS video received at Client

C. Demo Procedures

Demo-ID	Scenario	Expected Observation	Conclusion
D-1	Launch the General Topology and activate the Controller. Start pinging between Traffic Generators, Servers, and Client.	Pings arrive at their destination.	General Topology slice is online and functional.
D-2	Connect to the client with an X-enabled session and launch the audio visualizer.	Visualizer is visible on demo computer.	Client is ready to stream.
D-3	Launch the Cross-Traffic Generator to generate 1 Gbps each. Begin streaming the audio file from QoS and non-QoS servers to the client.	The two visualizes should be distinguishably different. The visualization of non-QoS traffic should have missing signal as shown by the bars of the	QoS traffic is passing through the congested network unscathed while non-QoS traffic suffers degradation.

		visualizer being low or absent as compared to the QoS visualizer.	
D-4	Once the audio has completed streaming, shut down the Cross-Traffic Generators. Prepare the video to be streamed and saved. Wait for latent Iperf connections to terminate.	ps aux | grep iperf should show no remaining active iperf processes.	The system has returned to the idle state and is ready for the next demo.
D-5	Launch the Cross-Traffic Generator to generate 1 Gbps each. Begin streaming the video file from QoS and non-QoS servers to the client, where it is saved.	Two new video files are present on the client.	The videos have been streamed and saved successfully.
D-6	Shut down the Cross-Traffic Generators, and scp the saved video files to the demo computer.	ps aux | grep iperf should show no remaining active iperf processes, and the two video files should be present on the demo computer.	The system has returned to the idle state, and the videos are ready to be displayed.
D-7	Play the two videos side-by-side on the demo computer.	The QoS video should be near unharmed. The non-QoS video should be pixelated and possibly be missing whole frames.	QoS traffic is passing through the congested network unscathed while non-QoS traffic suffers degradation.

VIII. Self-study Plan and Observations

a. Description of Base Case-

The base case scenario of our system is to study the treatment of multimedia based flows in an OpenFlow system without support for QoS. The system uses Dijkstra's algorithm (described above) as the default. As part of our project, we implemented LARAC as well. This change does not see much focus in our main project, but is still worth consideration. The interaction of these algorithms with the OpenFlow controller and the OpenVswitch and their combined impact on the local performance of the network cannot be ignored.

b. Characteristics to Observe-

i. Video quality improvement

Video streamed with QoS support shows better quality than that streamed without QoS support with cross-traffic in the network.

ii. Frame loss improvement

The video streamed with QoS support shows considerably less frame loss except the times when new flows are pushed to the switches. At those times we find drop of a frame or two

iii. Jitter Improvement, if any

Similar to frame loss improvement, there was jitter only when changing paths in QoS but very less otherwise.

For the last two characteristics, we are actually observing over some little changes to make to the code to improve over them further, specifically strategic pushing of flows to the switches so as to improve loss of packet due to changed flow.

c. Range of Scenarios to Observe-

i. Difference in paths calculated by Dijkstra and LARAC

Dijkstra's algorithm calculates paths based on least hop-count. The LARAC calculates paths minimizing total delay and link utilization.

Below are the routes calculated dynamically using LARAC

```
<Response [200]>
[0, 1, 2, 3]
<Response [200]>
[0, 1, 4, 3]
<Response [200]>
[0, 5, 4, 3]
<Response [200]>
```

ii. Delay incorporated while using LARAC and Dijkstra's algorithm

Dijkstra's algorithm is static compared to LARAC which constantly calculates routes based on link utilization since dijkstra's algorithm uses only hop-count as the link cost. Thus the delay introduced by dijkstra is lesser compared to that introduced by LARAC due to constant optimal path computation. From the below logs, it is evident that due to dynamic calculation of path, delay introduced

```
switch     00:00:56:40:08:c1:cd:43
      port id         1     bandwidth_util  0.0   Mbits/sec
      port id         3     bandwidth_util  0.0   Mbits/sec
      port id         2     bandwidth_util  0.0   Mbits/sec
      port id         4     bandwidth_util  0.0   Mbits/sec
      port id         local      bandwidth_util  0.0   Mbits/sec
switch     00:00:1e:96:5d:c9:b0:49
      port id         1     bandwidth_util  0.0   Mbits/sec
      port id         3     bandwidth_util  0.0   Mbits/sec
      port id         2     bandwidth_util  0.0   Mbits/sec
      port id         local      bandwidth_util  0.0   Mbits/sec
switch     00:00:da:f5:a7:c4:3a:49
      port id         1     bandwidth_util  0.0   Mbits/sec
      port id         3     bandwidth_util  0.0   Mbits/sec
      port id         2     bandwidth_util  0.0   Mbits/sec
      port id         4     bandwidth_util  0.0   Mbits/sec
      port id         local      bandwidth_util  0.0   Mbits/sec
switch     00:00:4a:ff:6e:89:e4:42
      port id         1     bandwidth_util  0.0   Mbits/sec
      port id         3     bandwidth_util  0.0   Mbits/sec
      port id         2     bandwidth_util  0.0   Mbits/sec
      port id         4     bandwidth_util  0.0   Mbits/sec
      port id         local      bandwidth_util  0.0   Mbits/sec
switch     00:00:1a:9b:f5:c7:10:47
      port id         1     bandwidth_util  0.0   Mbits/sec
      port id         3     bandwidth_util  0.0   Mbits/sec
      port id         2     bandwidth_util  0.0   Mbits/sec
      port id         4     bandwidth_util  0.0   Mbits/sec
      port id         local      bandwidth_util  0.0   Mbits/sec
switch     00:00:72:40:b2:ac:47:4b
      port id         1     bandwidth_util  0.0   Mbits/sec
      port id         3     bandwidth_util  0.0   Mbits/sec
      port id         2     bandwidth_util  0.0   Mbits/sec
      port id         4     bandwidth_util  0.0   Mbits/sec
```

```
     port id         local     bandwidth_util  0.0  Mbits/sec
<Response [200]>
[0, 1, 2, 3]
<Response [200]>


switch    00:00:56:40:08:c1:cd:43
     port id         1    bandwidth_util  11.16 Mbits/sec
     port id         3    bandwidth_util  11.16 Mbits/sec
     port id         2    bandwidth_util  0.0  Mbits/sec
     port id         4    bandwidth_util  0.0  Mbits/sec
     port id         local     bandwidth_util  0.0  Mbits/sec
switch    00:00:1e:96:5d:c9:b0:49
     port id         1    bandwidth_util  0.0  Mbits/sec
     port id         3    bandwidth_util  0.0  Mbits/sec
     port id         2    bandwidth_util  0.0  Mbits/sec
     port id         local     bandwidth_util  0.0  Mbits/sec
switch    00:00:4a:ff:6e:89:e4:42
     port id         1    bandwidth_util  0.0  Mbits/sec
     port id         3    bandwidth_util  0.0  Mbits/sec
     port id         2    bandwidth_util  0.0  Mbits/sec
     port id         4    bandwidth_util  0.0  Mbits/sec
     port id         local     bandwidth_util  0.0  Mbits/sec
switch    00:00:da:f5:a7:c4:3a:49
     port id         1    bandwidth_util  11.16     Mbits/sec
     port id         3    bandwidth_util  0.0  Mbits/sec
     port id         2    bandwidth_util  11.16     Mbits/sec
     port id         4    bandwidth_util  0.0  Mbits/sec
     port id         local     bandwidth_util  0.0  Mbits/sec
switch    00:00:1a:9b:f5:c7:10:47
     port id         1    bandwidth_util  0.0  Mbits/sec
     port id         3    bandwidth_util  0.0  Mbits/sec
     port id         2    bandwidth_util  0.0  Mbits/sec
     port id         4    bandwidth_util  0.0  Mbits/sec
     port id         local     bandwidth_util  0.0  Mbits/sec
switch    00:00:72:40:b2:ac:47:4b
     port id         1    bandwidth_util  0.0  Mbits/sec
     port id         3    bandwidth_util  0.0  Mbits/sec
     port id         2    bandwidth_util  0.0  Mbits/sec
     port id         4    bandwidth_util  0.0  Mbits/sec
     port id         local     bandwidth_util  0.0  Mbits/sec
```

[0, 1, 4, 3]
<Response [200]>
switch 00:00:56:40:08:c1:cd:43
port id 1 bandwidth_util 10.09 Mbits/sec
port id 3 bandwidth_util **10.09** Mbits/sec
port id 2 bandwidth_util 0.0 Mbits/sec
port id 4 bandwidth_util 0.0 Mbits/sec
port id local bandwidth_util 0.0 Mbits/sec
switch 00:00:1e:96:5d:c9:b0:49
port id 1 bandwidth_util 0.0 Mbits/sec
port id 3 bandwidth_util 0.0 Mbits/sec
port id 2 bandwidth_util 0.0 Mbits/sec
port id local bandwidth_util 0.0 Mbits/sec
switch 00:00:4a:ff:6e:89:e4:42
port id 1 bandwidth_util 0.0 Mbits/sec
port id 3 bandwidth_util 0.0 Mbits/sec
port id 2 bandwidth_util 0.0 Mbits/sec
port id 4 bandwidth_util 0.0 Mbits/sec
port id local bandwidth_util 0.0 Mbits/sec
switch 00:00:da:f5:a7:c4:3a:49
port id 1 bandwidth_util 10.14 Mbits/sec
port id 3 bandwidth_util 10.0 Mbits/sec
port id 2 bandwidth_util 20.14 Mbits/sec
port id 4 bandwidth_util 0.0 Mbits/sec
port id local bandwidth_util 0.0 Mbits/sec
switch 00:00:1a:9b:f5:c7:10:47
port id 1 bandwidth_util 0.0 Mbits/sec
port id 3 bandwidth_util 0.0 Mbits/sec
port id 2 bandwidth_util 9.97 Mbits/sec
port id 4 bandwidth_util **9.97 Mbits/sec**
port id local bandwidth_util 0.0 Mbits/sec
switch 00:00:72:40:b2:ac:47:4b
port id 1 bandwidth_util 0.0 Mbits/sec
port id 3 bandwidth_util 0.0 Mbits/sec
port id 2 bandwidth_util 0.0 Mbits/sec
port id 4 bandwidth_util 0.0 Mbits/sec
port id local bandwidth_util 0.0 Mbits/sec
[0, 5, 4, 3]
<Response [200]>
switch 00:00:56:40:08:c1:cd:43
port id 1 bandwidth_util 10.1 Mbits/sec

```
     port id         3    bandwidth_util  10.1 Mbits/sec
     port id         2    bandwidth_util  0.0  Mbits/sec
     port id         4    bandwidth_util  0.0  Mbits/sec
     port id         local     bandwidth_util  0.0  Mbits/sec
switch    00:00:1e:96:5d:c9:b0:49
     port id         1    bandwidth_util  0.0  Mbits/sec
     port id         3    bandwidth_util  0.0  Mbits/sec
     port id         2    bandwidth_util  0.0  Mbits/sec
     port id         local     bandwidth_util  0.0  Mbits/sec
switch    00:00:4a:ff:6e:89:e4:42
     port id         1    bandwidth_util  10.48     Mbits/sec
     port id         3    bandwidth_util  0.0  Mbits/sec
     port id         2    bandwidth_util  10.48     Mbits/sec
     port id         4    bandwidth_util  0.0  Mbits/sec
     port id         local     bandwidth_util  0.0  Mbits/sec
switch    00:00:da:f5:a7:c4:3a:49
     port id         1    bandwidth_util  10.09     Mbits/sec
     port id         3    bandwidth_util  10.42     Mbits/sec
     port id         2    bandwidth_util  20.51     Mbits/sec
     port id         4    bandwidth_util  0.0  Mbits/sec
     port id         local     bandwidth_util  0.0  Mbits/sec
switch    00:00:1a:9b:f5:c7:10:47
     port id         1    bandwidth_util  0.0  Mbits/sec
     port id         3    bandwidth_util  10.47     Mbits/sec
     port id         2    bandwidth_util  10.43     Mbits/sec
     port id         4    bandwidth_util  20.9 Mbits/sec
     port id         local     bandwidth_util  0.0  Mbits/sec
switch    00:00:72:40:b2:ac:47:4b
     port id         1    bandwidth_util  0.0  Mbits/sec
     port id         3    bandwidth_util  0.0  Mbits/sec
     port id         2    bandwidth_util  0.0  Mbits/sec
     port id         4    bandwidth_util  0.0  Mbits/sec
     port id         local     bandwidth_util  0.0  Mbits/sec
[0, 5, 2, 3]
<Response [200]>
```

Reflections
Calvin Fernando

1. Learnt about OpenFlow environment and OpenVirtualSwitch
2. Learnt about Network Designing while deciding on the network topology
3. Learnt about iptables in linux
4. Learnt about REST API and the means to interact with the Floodlight Controller from the Controller Application
5. Learnt about implementation of QoS in a network
6. Learnt about Dijkstra's and LARAC
7. Learnt about VideoLAN streaming features
8. Learnt about ExoGENI

Bharat Mukheja-

1. Learnt Openflow implementation for Qos and OpenVSwitch
2. Queues implementation and categorization of traffic
3. Graph maps and path calculation algorithms(Dijkstra, LARAC)
4. ExoGeni
5. REST API application development

Kaustubh Gondhalekar-

1. Learnt about OpenFlow and OpenVirtualSwitch.
2. Learnt about REST API application development.
3. Learnt about Floodlight Controller.
4. Learnt about multithreading in Python.
5. Learnt about Dynamic routing with respect to QoS routes.
6. Learnt about ExoGeni.

Ross Williams

1. Learnt about Internet Traffic patterns
2. Learnt about Python multithreading and task tracking
3. Expanded knowledge on REST API application development
4. Learnt about Dynamic Routing in relation to current traffic
5. Learnt about ExoGeni.

As Expected:

- The traffic targeting and volume selection worked from the start with only minor bugs.
- The statistics collector worked as expected calculating link bandwidth in the network topology.
- Most of the REST APIs for Floodlight worked as expected.

Not As Expected:

- The Static Entry Pusher API didn't work as expected. Particularly pushing flows with IP DSCP field set to 40 failed. Turns out this was actually a bug in the Floodlight code and was promptly fixed by one of the Floodlight developers.
- ExoGeni slices malfunctioned. Spawning and making network topologies to work took a lot of time. A working network topology would fail without any reason the next day which introduced lot of delays in the project.
- Initial tests from the Traffic Generator appeared to show that the maximum bandwidth of each link was around 10Mbps, which was far lower than we expected. Further testing revealed

that the link's true capacity was in the range of 800Mbps. The Traffic Generator was fixed.

References

[1]

1. http://www.projectfloodlight.org/floodlight/
2. http://openvswitch.org/
3. https://floodlight.atlassian.net/wiki/display/floodlightcontroller/User+Documentation
4. https://floodlight.atlassian.net/wiki/pages/viewpage.action?pageId=21856267#app-switcher
5. Lagrange relaxation based method for the QoS routing problem A. Juttner, B. Szviatovski, I. Mecs, Z. Rajko; Published in: INFOCOM 2001. Twentieth Annual Joint Conference of the IEEE Computer and Communications Societies. Proceedings. IEEE
6. OpenQoS: An OpenFlow controller design for multimedia delivery with end-to-end Quality of Service over Software-Defined Networks; H. E. Egilmez, S. T. Dane, K. T. Bagci Published in: Signal & Information Processing Association Annual Summit and Conference (APSIPA ASC), 2012 Asia-Pacific
7. http://openvswitch.org/support/dist-docs/ovs-ofctl.8.txt
8. https://pdfs.semanticscholar.org/146a/8e00a05e32b7c023e228a84c72f48e4017ad.pdf
9. http://ieeexplore.ieee.org/document/6411795/?part=1