

Flow Aware Network (QoS) based SDN Controller (POX)

Apoorva GM 01666754

Harish Prakash 01669175

University of Massachusetts Lowell

Abstract: The paper explains about the design of a SDN controller (Pox) with implementation of Flow Aware Network approach. In this project, we have designed SDN controller which controls the flow with respect to the bandwidth. Flow Aware Network (FAN) means that traffic management is based on user-defined flows. The definition of a flow in Flow-Aware Networking comes from [8]: “By flow we mean a flight of datagrams, localized in time and space and having the same unique identifier.” Here we have designed a topology with four hosts and two switches. To provide a service at a reasonable level, under the terms of congestion, some priorities and discriminations must be imposed. The aforementioned architectures propose the use of a reservation protocol and a packet marking scheme, respectively; however, these solutions require proper inter-domain agreements, complex router implementations, and, most of all, end user compliance. Besides IntServ and DiffServ, many other QoS architectures have been proposed for IP networks. Based on the priorities set for each path to the switch the bandwidth will be decided. An efficient and robust QoS architecture for IP networks requires that the user– network interface remains the same as today, no signaling protocol or packet marking is introduced, and no new user–operator or operator–operator agreements are signed. These constraints are very strict, yet they have been met.

Keywords: SDN controller, Flow Aware Network, Pox

I. Introduction

Let’s go through the basics of the flow aware network before going through the design implementation.

The success of the Internet lies in its simplicity; however, this comes at a cost of only best effort non-differentiated service. For years, institutions such as the IETF have been trying to introduce a QoS architecture to the current IP network. Unfortunately, the proposed QoS models, i.e., IntServ [1] and DiffServ [2,3], are not suitable for the Internet as a whole. To provide a service at a

reasonable level, under the terms of congestion, some priorities and discriminations must be imposed. The aforementioned architectures propose the use of a reservation protocol and a packet marking scheme, respectively; however, these solutions require proper inter-domain agreements, complex router implementations, and, most of all, end user compliance. Besides IntServ and DiffServ, many other QoS architectures have been proposed for IP networks. An efficient and robust QoS architecture for IP networks requires that the user– network interface remains the same as today, no signaling protocol or packet marking is introduced, and no new user–operator or operator–operator agreements are signed. These constraints are very strict, yet they have been met. This chapter introduces a novel approach to achieving QoS guarantees in the Internet—Flow-Aware Networking, or FAN for short. The description of FAN starts with which shows why the new QoS architecture is needed.

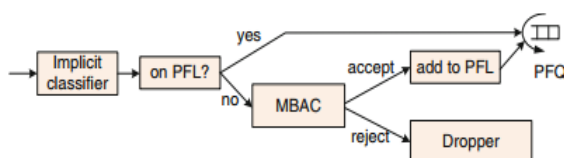
II. Quality of Service

IETF introduced two ideas on how to assure QoS. Chronologically, the first was Integrated Services. IntServ has many advantages, such as real (as opposed to statistical) assurances, easy control in nodes, use of a reservation protocol, and the option to create various traffic profiles. However, there are certain disadvantages, which make IntServ unsuitable for larger networks. These include maintaining information about all flows in every node and demanding that end users explicitly define required transmission parameters. These pros and cons make IntServ a good solution for dealing with a small network, where all the end users are known, traffic is mostly defined, and every router in the network can be easily configured by a single network operator. To overcome the scalability issue, IETF introduced a new idea—the Differentiated Services. At the cost of certain constraints, DiffServ avoids the problems that eventually halted the development of its predecessor. That is why in DiffServ the assurances are statistical and the admission control blocks are only placed at the borders of each DiffServ domain. Moreover, the inner nodes do not keep the flow information, which suits it better to

larger networks; however, the scalability issue is not completely overcome. Still, all routers in a domain must be pre-configured so that the per-hop behavior matches the actual classes of service which are provided inside the domain. Although DiffServ is more flexible and scalable than its predecessor, it still has features which make it unsuited for extra-large networks like the Internet. It can be said that IntServ and DiffServ represent a trade-off between fine service granularity and scalability. Over the years, many attempts to alleviate this relationship have been proposed, including combined use of IntServ and DiffServ, or new and better congestion control mechanisms cooperating with the service isolation provided by DiffServ. However, no solution has attracted enough attention to be widely implemented and used

III. Flow Aware Network (FAN)

Flow-Aware Networking is a new direction for QoS assurance in IP networks. The original idea was initially introduced by J. Roberts et al. in [4,5] and then presented as a complete system in 2004 [6,7]. Their intention was to design a novel QoS architecture that would be possible to use in networks of all sizes, including the global IP network—the Internet. In [8], the belief that adequate performance can be assured much more simply than in classical QoS architectures and more reliably than in over-provisioned best effort networks is expressed. The goal of FAN is to enhance the current IP network by improving its performance under heavy congestion. To achieve this, certain traffic management mechanisms to control link sharing are proposed, namely measurement-based admission control [8] and fair scheduling with priorities [6,9]. The former is used to keep the flow rates sufficiently high to provide a minimal level of performance for each flow in case of overload. The latter realizes the fair sharing of link bandwidth while ensuring negligible packet latency for flows emitting at lower rates. All the new functionalities are performed by a unique router, named the Cross-Protect router. This device alone is responsible for providing admission control and fair queuing.



<https://google.cs.neu.edu>

All incoming packets are first classified into flows. The flow identification process is implicit and its goal is not to divide flows into different classes, but only to create an instance on which the service differentiation will be performed. Then, all the flows that are currently in progress are present on the Protected Flow List (PFL) and are forwarded unconditionally, whereas all new flows are subject to admission control. The admission control in FAN is measurement based (MBAC) which implies that the accept/reject decisions are based only on the current link congestion status. If a new flow is accepted, it is put onto the PFL list and then all successive packets from this flow are forwarded without checking the status of the outgoing link by MBAC. In FAN, admission control and service differentiation are implicit. In classic explicit service differentiation architectures, user requirements are signaled to the network and the nodes perform differentiation actions based on the information received. For example, the provision of better quality is a consequence of the explicit identification of a certain transmission. On the contrary, implicit service differentiation performs differentiation actions based on traffic characteristics and network measurements. In FAN, there is no need for a priori traffic specification, as well as no class of service distinction. Both streaming and elastic flows achieve the necessary QoS without any mutual detrimental effect. Nevertheless, streaming and elastic flows are implicitly identified inside the FAN network. This classification, however, is based solely on the current flow peak rate. All flows emitting at lower rates than the current fair rate are referred to as streaming flows, and the packets from those flows are prioritized. The remaining flows are referred to as elastic flows. FAN is intended to be suited even to the whole Internet. This is due to a number of constraints that have been imposed by the designers. First of all, nodes do not need to exchange any information among themselves; they do not even need any explicit information about the flows. All information is gathered through local measurements. Flow-Aware Networking is based on the XP mechanism, which enhances the current IP router functionality. One of the most important aspects of FAN is that it is only an enhancement of the currently existing IP network. Both networks can easily coexist. Moreover, the advantages of FAN can be seen even if not all nodes are FAN based. This means that it is possible (and advisable) to improve the network gradually by replacing

nodes, starting with those that are attached to the most heavily congested links.

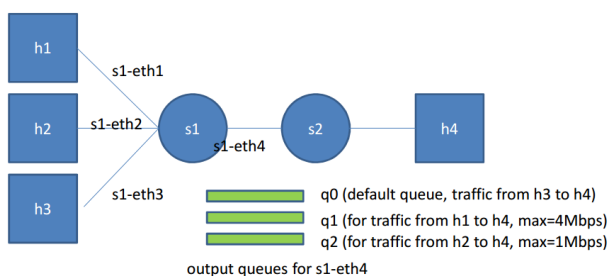
IV. FAN Approach

FAN is flow oriented. This means that traffic management is based on user-defined flows. The definition of a flow in Flow-Aware Networking comes from [8]: “By flow we mean a flight of datagrams, localized in time and space and having the same unique identifier.” The datagrams are localized in space as they are observed at a certain interface (e.g., on a router) and in time, as they must be spaced by no more than a certain interval, which is usually a few seconds. The space localization means that a typical flow has many instances, one at every interface on its path.

All flows in FAN are divided into either streaming or elastic, and hence two classes of service. This distinction is implicit, which means that the system categorizes the flows based on their current behavior. There is no need for a priori traffic specification as the classification is based solely on the current flow peak rate. All flows emitting at lower rates than the current fair rate are referred to as streaming flows, and packets from those flows are prioritized. The remaining flows are referred to as elastic flows. The association of a flow with a certain class is not permanent. If a flow, initially classified as streaming, surpasses the current fair rate value, it is degraded to the elastic flow category. Analogously, a flow is promoted to streaming if at any time it emits at a lower rate than the current fair rate. Note that both factors, i.e., flow bitrate and current fair rate, can change.

V. Implementation of FAN on SDN Controller(POX)

A. Design



The design consists of four hosts and two switches. Hosts H1 H2 H3 each has one path to reach the switch S1-eth1, S1-eth2, S1-eth3. Switch S1 and S2 are connected. Each of the paths is given priorities and given queue numbers such as q0 q1 q2. q0 is the

path from h1 to h4. q1 is the path from h1 to h4 with 4Mbps bandwidth. q2 is the path from h3 to h4 with 1Mbps bandwidth.

Depending on the sensors connected to the patient and the priorities set to them the data will be sent over the channel with respect to the bandwidths set with the path and the Queue.

The code for the topology and the controller are as follows.

```

SDN_tutorial_VM_64bit - VMware Workstation 12 Player (Non-commercial use on
Player
Applications: [Tutorials | SDN Hub - Moz... Pox - File Manager
1 from mininet.topo import Topo
2 from mininet.nodelib import LinuxBridge
3 from mininet.cli import CLI
4 from mininet.net import Mininet
5 from mininet.node import CPULimitedHost, Host, Node
6 from mininet.node import OVSKernelSwitch, UserSwitch
7 from mininet.node import Controller, RemoteController, OVSController
8
9 class MyTopo( Topo ):
10     "Simple topology example."
11     def __init__( self ):
12         "Create custom topo."
13         # Initialize topology
14         Topo.__init__( self )
15         # Add hosts and switches
16         h1 = self.addHost( 'h1' )
17         h2 = self.addHost( 'h2' )
18         h3 = self.addHost( 'h3' )
19         h4 = self.addHost( 'h4' )
20         s1 = self.addSwitch( 's1' )
21         s2 = self.addSwitch( 's2' )
22         # Add links
23         self.addLink( h1, s1 )
24         self.addLink( h2, s1 )
25         self.addLink( h3, s1 )
26         self.addLink( s1, s2 )
27         self.addLink( s2, h4 )
28 topos = { 'mytopo': ( lambda: MyTopo( ) ) }
29

```

Code for the controller is as follows

```

1 from pox.core import core
2 import pox.openflow.libopenflow_01 as of
3 from pox.lib.util import dpid_to_str, str_to_dpid
4 import time
5 log = core.getLogger()
6 s1_dpid=0
7 s2_dpid=0
8 def _handle_ConnectionUp (event):
9     global s1_dpid, s2_dpid
10    print "ConnectionUp: ",
11    dpid_to_str(event.connection.dpid)
12    #remember the connection dpid for switch
13    for m in event.connection.features.ports:
14        if m.name == "s1-eth1":
15            s1_dpid = event.connection.dpid
16            print "s1_dpid=", s1_dpid
17        elif m.name == "s2-eth1":
18            s2_dpid = event.connection.dpid
19            print "s2_dpid=", s2_dpid
20 def _handle_PacketIn (event):
21    global s1_dpid, s2_dpid
22    # print "PacketIn: ", dpid_to_str(event.connection.dpid)
23    if event.connection.dpid==s1_dpid:
24        msg = of.ofp_flow_mod()
25        msg.priority = 1
26        msg.idle_timeout = 0
27        msg.hard_timeout = 0
28        msg.match.dl_type = 0x0806
29        msg.actions.append(of.ofp_action_output(port = of.OFPP_ALL))
30        event.connection.send(msg)
31        msg = of.ofp_flow_mod()
32        msg.priority = 100
33        msg.idle_timeout = 0
34        msg.hard_timeout = 0
35        msg.match.dl_type = 0x0800
36        msg.match.nw_src = "10.0.0.1"
37        msg.match.nw_dst = "10.0.0.4"
38        msg.actions.append(of.ofp_action_enqueue(port = 4, queue_id=1))
39        event.connection.send(msg)
40        msg = of.ofp_flow_mod()
41        msg.priority = 100
42        msg.idle_timeout = 0
43        msg.hard_timeout = 0
44        msg.match.dl_type = 0x0800
45        msg.match.nw_src = "10.0.0.2"
46        msg.match.nw_dst = "10.0.0.4"
47        msg.actions.append(of.ofp_action_enqueue(port = 4, queue_id=2))
48        event.connection.send(msg)
49        msg = of.ofp_flow_mod()
50        msg.priority = 10
51        msg.idle_timeout = 0
52        msg.hard_timeout = 0
53        msg.match.dl_type = 0x0800
54        msg.match.nw_dst = "10.0.0.1"
55        msg.actions.append(of.ofp_action_output(port = 1))
56        event.connection.send(msg)
57        msg = of.ofp_flow_mod()
58        msg.priority = 10
59        msg.idle_timeout = 0
60        msg.hard_timeout = 0
61        msg.match.dl_type = 0x0800
62        msg.match.nw_dst = "10.0.0.2"
63        msg.actions.append(of.ofp_action_output(port = 2))
64        event.connection.send(msg)
65        msg = of.ofp_flow_mod()
66        msg.priority = 10
67        msg.idle_timeout = 0
68        msg.hard_timeout = 0
69        msg.match.dl_type = 0x0800
70        msg.match.nw_dst = "10.0.0.3"
71        msg.actions.append(of.ofp_action_output(port = 3))
72        event.connection.send(msg)
73        msg = of.ofp_flow_mod()
74        msg.priority = 10
75        msg.idle_timeout = 0
76        msg.hard_timeout = 0
77        msg.match.dl_type = 0x0800
78        msg.match.nw_dst = "10.0.0.4"
79        msg.actions.append(of.ofp_action_output(port = 4))
80        event.connection.send(msg)
81    elif event.connection.dpid==s2_dpid:
82        msg = of.ofp_flow_mod()
83        msg.priority = 1
84        msg.idle_timeout = 0
85        msg.hard_timeout = 0
86        msg.match.in_port = 1
87        msg.actions.append(of.ofp_action_output(port = 2))
88        event.connection.send(msg)
89        msg = of.ofp_flow_mod()
90        msg.priority = 1
91        msg.idle_timeout = 0
92        msg.hard_timeout = 0

```

```

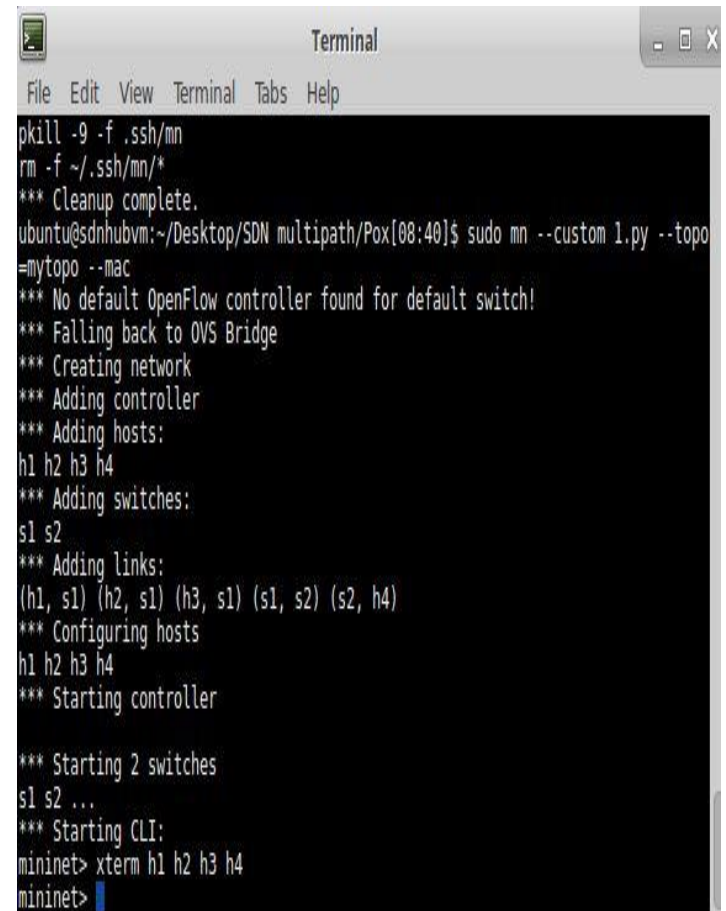
92    msg.hard_timeout = 0
93    msg.match.in_port = 2
94    msg.actions.append(of.ofp_action_output(port = 1))
95    event.connection.send(msg)
96    def launch():
97        core.openflow.add_listener_by_name("ConnectionUp", _handle_ConnectionUp)
98        core.openflow.add_listener_by_name("PacketIn", _handle_PacketIn)

```

VI. Execution and Results

Sudo mn --custom 1.py --topo=mytopo --mac

This creates a topology with 2 switches with 4 hosts which is defined in 1.py python code.



```

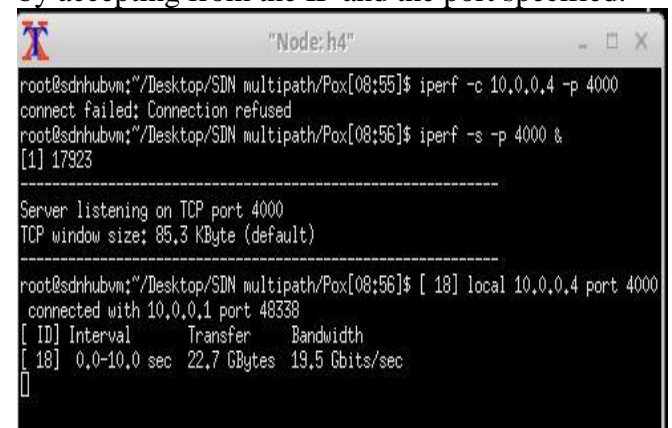
Terminal
File Edit View Terminal Tabs Help

pkill -9 -f .ssh/mn
rm -f ~/.ssh/mn/*
*** Cleanup complete.
ubuntu@sdnhubvm:~/Desktop/SDN multipath/Pox[08:40]$ sudo mn --custom 1.py --topo
=mytopo --mac
*** No default OpenFlow controller found for default switch!
*** Falling back to OVS Bridge
*** Creating network
*** Adding controller
*** Adding hosts:
h1 h2 h3 h4
*** Adding switches:
s1 s2
*** Adding links:
(h1, s1) (h2, s1) (h3, s1) (s1, s2) (s2, h4)
*** Configuring hosts
h1 h2 h3 h4
*** Starting controller

*** Starting 2 switches
s1 s2 ...
*** Starting CLI:
mininet> xterm h1 h2 h3 h4
mininet>

```

Now let's check the iperf from (h4 the destination) by accepting from the IP and the port specified.



```

Node:h4

root@sdnhubvm:~/Desktop/SDN multipath/Pox[08:55]$ iperf -c 10.0.0.4 -p 4000
connect failed: Connection refused
root@sdnhubvm:~/Desktop/SDN multipath/Pox[08:56]$ iperf -s -p 4000 &
[1] 17923

Server listening on TCP port 4000
TCP window size: 85.3 KByte (default)

root@sdnhubvm:~/Desktop/SDN multipath/Pox[08:56]$ [ 18] local 10.0.0.4 port 4000
connected with 10.0.0.1 port 48338
[ ID] Interval      Transfer      Bandwidth
[ 18] 0.0-10.0 sec  22.7 GBytes  19.5 Gbits/sec

```



```

root@sdnhubvm:~/Desktop/SDN multipath/Pox[08:57]$ iperf -s -p 5000 &
[2] 18004

-----
Server listening on TCP port 5000
TCP window size: 85,3 KByte (default)
-----
root@sdnhubvm:~/Desktop/SDN multipath/Pox[08:57]$ [ 18] local 10.0.0.4 port 5000
connected with 10.0.0.2 port 49328
[ ID] Interval      Transfer    Bandwidth
[ 18] 0.0-10.0 sec  22,5 GBytes 19,3 Gbits/sec
[ 18]

```

```

root@sdnhubvm:~/Desktop/SDN multipath/Pox[08:58]$ [ 18] local 10.0.0.4 port 6000
connected with 10.0.0.3 port 52622
[ ID] Interval      Transfer    Bandwidth
[ 18] 0.0-10.0 sec  24,5 GBytes 21,0 Gbits/sec
[ 18]

```

Run the command iperf from h1 to the destination H4 with the destination port number.

```

root@sdnhubvm:~/Desktop/SDN multipath/Pox[08:55]$ iperf -c 10.0.0.4 -p 4000

Client connecting to 10.0.0.4, TCP port 4000
TCP window size: 1020 KByte (default)
-----
[ 17] local 10.0.0.1 port 48338 connected with 10.0.0.4 port 4000
[ ID] Interval      Transfer    Bandwidth
[ 17] 0.0-10.0 sec  22,7 GBytes 19,5 Gbits/sec
root@sdnhubvm:~/Desktop/SDN multipath/Pox[08:56]$

```

Run the command iperf from h2 to the destination H4 with the destination port number.

```

root@sdnhubvm:~/Desktop/SDN multipath/Pox[08:55]$ iperf -c 10.0.0.4 -p 5000

Client connecting to 10.0.0.4, TCP port 5000
TCP window size: 85,3 KByte (default)
-----
[ 17] local 10.0.0.2 port 49328 connected with 10.0.0.4 port 5000
[ ID] Interval      Transfer    Bandwidth
[ 17] 0.0-10.0 sec  22,5 GBytes 19,3 Gbits/sec
root@sdnhubvm:~/Desktop/SDN multipath/Pox[08:57]$

```

Run the command iperf from h3 to the destination H4 with the destination port number.

```

root@sdnhubvm:~/Desktop/SDN multipath/Pox[08:55]$ iperf -c 10.0.0.4 -p 6000

Client connecting to 10.0.0.4, TCP port 6000
TCP window size: 680 KByte (default)
-----
[ 17] local 10.0.0.3 port 52622 connected with 10.0.0.4 port 6000
[ ID] Interval      Transfer    Bandwidth
[ 17] 0.0-10.0 sec  24,5 GBytes 21,1 Gbits/sec
root@sdnhubvm:~/Desktop/SDN multipath/Pox[08:58]$

```

In the new terminal run the controller
./pox.py log.level controller

```

ubuntu@sdnhubvm:~/pox[08:55] (eel)$ ./pox.py log.level controller
POX 0.5.0 (eel) / Copyright 2011-2014 James McCauley, et al.
INFO:core:POX 0.5.0 (eel) is up.

```

Now in the new terminal run the ovs-vsctl command.

The command below shown in the screenshot with the s1-eth4 and qos queue will be specified in the command with the bandwidth will be created in the command prompt.

```

ubuntu@sdnhubvm:~[20:51]$ sudo ovs-vsctl -- set Port s1-eth4 qos=@newqos -- --id
=@newqos create QoS type=linux-htb other-config:max-rate=1000000000 queues=0=@q0
,1=@q1,2=@q2 -- --id=@q0 create Queue other-config:min-rate=100000000 other-conf
ig:max-rate=1000000000 -- --id=@q1 create Queue other-config:min-rate=400000000
other-config:max-rate=4000000000 -- --id=@q2 create Queue other-config:min-rate=
100000000 other-config:max-rate=1000000000
[sudo] password for ubuntu:
5d2b5467-41fe-45a2-9bfc-19b0f1f47d0d
403df4eb-fea3-44a8-b111-009b081a539a
8cfa8431-0841-46e2-a2dd-c981a1baeddf
e2615887-3cb5-4dbb-948d-ea50ba6fd074

```

Post starting the controller run the iperf commands from destination h4 and source h1 h2 h3.

On h1 terminal **Iperf -c 10.0.0.4 -p 4000**

On h2 terminal **Iperf -c 10.0.0.4 -p 5000**

On h3 terminal **Iperf -c 10.0.0.4 -p 6000**

On h4 terminal **iperf -s -p 4000 &**
Iperf -s -p 5000 &
Iperf -s -p 6000 &

The outputs of the above commands are as follows.

```

Node: h2
root@mininet-ws:/njlabs# iperf -c 10.0.0.4 -p 5000
Client connecting to 10.0.0.4, TCP port 5000
TCP window size: 85.3 KByte (default)
[ 4] local 10.0.0.2 port 47584 connected with 10.0.0.4 port 5000
[ ID] Interval      Transfer      Bandwidth
[ 4] 0.0-11.3 sec  1.25 MBytes  935 Kbits/sec
root@mininet-ws:/njlabs#

Node: h1
root@mininet-ws:/njlabs# iperf -c 10.0.0.4 -p 4000
Client connecting to 10.0.0.4, TCP port 4000
TCP window size: 85.3 KByte (default)
[ 4] local 10.0.0.1 port 40678 connected with 10.0.0.4 port 4000
[ ID] Interval      Transfer      Bandwidth
[ 4] 0.0-10.3 sec  4.12 MBytes  3.36 Mbits/sec
root@mininet-ws:/njlabs#

Node: h3
root@mininet-ws:/njlabs# iperf -c 10.0.0.4 -p 6000
Client connecting to 10.0.0.4, TCP port 6000
TCP window size: 85.3 KByte (default)
[ 4] local 10.0.0.3 port 56165 connected with 10.0.0.4 port 6000
[ ID] Interval      Transfer      Bandwidth
[ 4] 0.0-10.0 sec  181 MBytes  151 Mbits/sec
root@mininet-ws:/njlabs#

```

Here the bandwidth is split according to the priorities set for the queue. This is the final output for the implemented flow aware based SDN controller (pox).

VII. Future Implementations

This controller is now implemented on a small topology. In future, we can implement same thing on the bigger topology with more number of queues and the more number of priorities which has bigger challenges of creating and giving queue priorities.

VIII. References

- [1] R. Braden, D. Clark, and S. Shenker, "Integrated Services in the Internet Architecture an Overview," IETF RFC 1633, June 1994.
- [2] S. Blake, D. Black, M. Carlson, E. Davies, Z. Wang, and W. Weiss, "An Architecture for Differentiated Services," IETF RFC 2475, December 1998.
- [3] K. Nichols, S. Blake, F. Baker, and D. Black, "Definition of the Differentiated Services Field (DS Field) in the IPv4 and IPv6 Headers," IETF RFC 2474, December 1998.
- [4] T. Bonald, S. Oueslati-Boulahia, and J. Roberts, "IP traffic and QoS control," in Proc. World

Telecommunications Congress, WTC 2002, Paris, France, September 2002.

- [5] J. Roberts and S. Oueslati, "Quality of Service by Flow Aware Networking," Philosophical Transactions of The Royal Society of London, vol. 358, pp. 2197–2207, 2000.
- [6] Kortebi, S. Oueslati, and J. W. Roberts, "Cross-protect: implicit service differentiation and admission control," in Proc. High Performance Switching and Routing, HPSR 2004, Phoenix, AZ, USA, 2004, pp. 56–60.
- [7] J. Roberts, "Internet Traffic, QoS and Pricing," in Proc. the IEEE, vol. 92, September 2004, pp. 1389–1399.
- [8] <http://csie.nqu.edu.tw/smallko>
- [9] S. Oueslati and J. Roberts, "A new direction for quality of service: Flow-aware networking," in Proc. 1st Conference on Next Generation Internet Networks - Traffic Engineering, NGI 2005, Rome, Italy, 2005.
- [10] Kortebi, S. Oueslati, and J. Roberts, "Implicit Service Differentiation using Deficit Round Robin," in Proc. 19th International Teletraffic Congress, ITC 2005, Beijing, China, August/September 2005.
- [11] http://users.ecs.soton.ac.uk/drn/ofertie/opennflow_qos_mininet.pdf

Link to GitHub: <https://github.com/pharish11/SDN123>