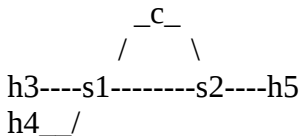


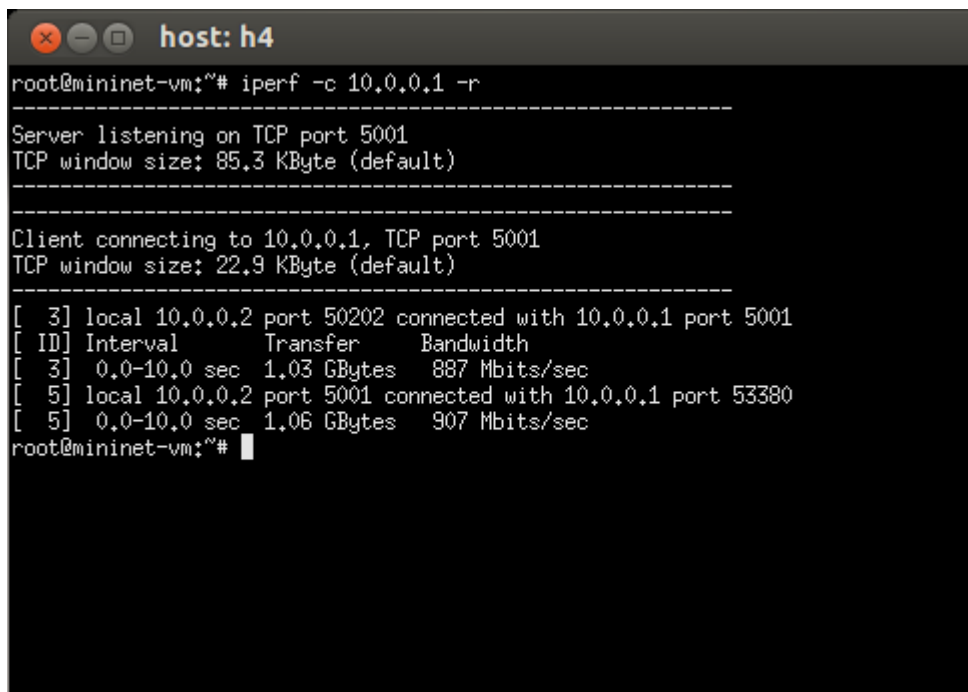
QoS on OpenFlow 1.0 with OVS 1.4.3 and POX inside Mininet

Note to reader: if you want to skip past the learning steps with flawed results and onto valuable results, skip to 4

For reference purposes, we will define a 2 switch, 3 host topology. In this arrangement h5 might be a server and h3/h4 might be competing clients, or h3 and h4 could be service providers, which need to be shaped on s1's central link. We can also note any differences between h3-h5 and the more local h3-h4.



```
sudo mn --custom topo2sw3h.py --topo 2s3h --mac -x -controller=remote
```



```
host: h4
root@mininet-vm:~# iperf -c 10.0.0.1 -r
-----
Server listening on TCP port 5001
TCP window size: 85.3 KByte (default)
-----
Client connecting to 10.0.0.1, TCP port 5001
TCP window size: 22.9 KByte (default)
-----
[ 3] local 10.0.0.2 port 50202 connected with 10.0.0.1 port 5001
[ ID] Interval      Transfer    Bandwidth
[ 3] 0.0-10.0 sec  1.03 GBytes  887 Mbits/sec
[ 5] local 10.0.0.2 port 5001 connected with 10.0.0.1 port 53380
[ 5] 0.0-10.0 sec  1.06 GBytes  907 Mbits/sec
root@mininet-vm:~#
```

Figure 1: 1Gbit/s link tested before applying QoS (-r = round trip; both directions in sequence)

OpenFlow 1.0 is not able to fully configure queues on OVS, and can only set minimum rate queues. OpenFlow 1.2 added support for max rate and experimenter queue properties. The configuration of QoS qdiscs is intended to be external to OpenFlow (i.e. static with the switch), meaning they cannot be dynamically controlled with OpenFlow config messages [0].

Create single QoS with a single Queue (q0) on port s1-eth1 (needs root)

```
ovs-vsctl -- set Port s1-eth1 qos=@newqos -- \
--id=@newqos create QoS type=linux-htb other-config:max-rate=1000000000 queues=0=@q0 -- \
--id=@q0 create Queue other-config:min-rate=4000000 other-config:max-rate=4000000
```

This seems to throttle all egress traffic (going out) on that port:

```
mininet>net
...
s1 lo: s1-eth1:h3-eth0 s1-eth2:h4-eth0 s1-eth3:s2-eth1
...
```

```
^Croot@mininet-vm:~# iperf -c 10.0.0.1 -p 5001 -r
-----
Server listening on TCP port 5001
TCP window size: 85.3 KByte (default)
-----
Client connecting to 10.0.0.1, TCP port 5001
TCP window size: 22.9 KByte (default)
-----
[ 3] local 10.0.0.2 port 50163 connected with 10.0.0.1 port 5001
[ ID] Interval      Transfer      Bandwidth
[ 3]  0.0-10.2 sec  4.88 MBytes  4.00 Mbits/sec
[ 5] local 10.0.0.2 port 5001 connected with 10.0.0.1 port 53341
[ 5]  0.0-10.0 sec  398 MBytes  333 Mbits/sec
root@mininet-vm:~#
```

Figure 2: h3 (10.0.0.1) is server, listening on 5001. h4 (10.0.0.2) is client. client to server is throttled, but not visa versa

The port s1-eth1 is the switch port linked to h3.

Running iperf with h3 server, h4 client:

- h4 → h3 (client to server) throttled to 4Mbit/s
- h3 → h4 (server to client) not throttled

This shows that this method of QoS is egress only i.e. when the switch is forwarding packets towards h3.

A side effect of this appears to be a reduction in the incoming bandwidth on that port, reduced almost by a third.

This is entirely a result of the bottleneck in the virtualised switch processing; later, a reduction in the link bandwidth is shown to relieve this bottleneck.

Selective QoS (TCP port bias)

Instead of applying QoS on queue 0 (default) on port s1-eth1, we will create two queues, one on which we enqueue traffic we want to pass unrestricted (at maximim bandwidth 1Gbit/s) and one on which we enqueue the traffic we want throttled to 4Mbit/s: (needs root)

```
ovs-vsctl -- set Port s1-eth1 qos=@newqos -- \
--id=@newqos create QoS type=linux-htb other-config:max-rate=1000000000 queues=@q0,1=@q1 -- \
--id=@q0 create Queue other-config:min-rate=1000000000 other-config:max-rate=1000000000 -- \
--id=@q1 create Queue other-config:min-rate=4000000 other-config:max-rate=4000000
```

On our POX l2_learning switch (pox/forwarding/l2_learning.py) we will now attempt to throttle traffic with TCP port 666, as this is the port of the devil, and is exclusive to evil communications. All traffic with TCP src/dest 666 outbound on s1-eth1 will be enqueued on queue 1, and all other traffic will be enqueued on the default queue (0).

Python l2_learning.py:

```
if msg.match.tp_src == 666 or msg.match.tp_dst == 666:
```

```

msg.actions.append(of.ofp_action_enqueue(port = port, queue_id=1))
else:
msg.actions.append(of.ofp_action_output(port = port)) ##defaults to q0

```

```

-----
Client connecting to 10.0.0.1, TCP port 5001
TCP window size: 22.9 KByte (default)
-----
[ 3] local 10.0.0.3 port 34568 connected with 10.0.0.1 port 5001
[ ID] Interval      Transfer      Bandwidth
[ 3] 0.0-10.0 sec  497 MBytes   416 Mbits/sec
[ 5] local 10.0.0.3 port 5001 connected with 10.0.0.1 port 41547
[ 5] 0.0-10.0 sec  912 MBytes   764 Mbits/sec
root@mininet-vm:~# iperf -c 10.0.0.1 -p 5001 -r
-----
Server listening on TCP port 5001
TCP window size: 85.3 KByte (default)
-----
Client connecting to 10.0.0.1, TCP port 5001
TCP window size: 22.9 KByte (default)
-----
[ 3] local 10.0.0.3 port 34570 connected with 10.0.0.1 port 5001
[ ID] Interval      Transfer      Bandwidth
[ 3] 0.0-10.0 sec  499 MBytes   418 Mbits/sec
[ 5] local 10.0.0.3 port 5001 connected with 10.0.0.1 port 41549
[ 5] 0.0-10.0 sec  922 MBytes   773 Mbits/sec
root@mininet-vm:~#

```

Figure 3: Testing with the two queues in place, with a non-throttled port, but is unintentionally throttled since placing the QoS and queue rules on OVS.

```

-----
Client connecting to 10.0.0.1, TCP port 666
TCP window size: 59.4 KByte (default)
-----
[ 3] local 10.0.0.3 port 51582 connected with 10.0.0.1 port 666
[ ID] Interval      Transfer      Bandwidth
[ 3] 0.0-10.2 sec  4.88 MBytes   4.00 Mbits/sec
[ 5] local 10.0.0.3 port 666 connected with 10.0.0.1 port 38445
[ 5] 0.0-10.0 sec  343 MBytes   287 Mbits/sec
root@mininet-vm:~# iperf -c 10.0.0.1 -p 666 -r
-----
Server listening on TCP port 666
TCP window size: 85.3 KByte (default)
-----
Client connecting to 10.0.0.1, TCP port 666
TCP window size: 22.9 KByte (default)
-----
[ 3] local 10.0.0.3 port 51584 connected with 10.0.0.1 port 666
[ ID] Interval      Transfer      Bandwidth
[ 3] 0.0-10.6 sec  5.00 MBytes   3.97 Mbits/sec
[ 5] local 10.0.0.3 port 666 connected with 10.0.0.1 port 38447
[ 5] 0.0-10.0 sec  339 MBytes   284 Mbits/sec
root@mininet-vm:~#

```

Figure 4: Testing QoS discriminating against TCP 666 (throttled to 4Mbit/s.) A large amount of unintentional throttling can also be seen on ingress traffic on that switch port (~900Mbit/s → ~285Mbit/s).

Note that the undesired throttling occurs even when the flow is already present in the switch's flow table, and if the output action for non-666 packets is set to output instead of enqueue on q0 (which effectively happens anyway, as it is the default queue.) This demonstrates that the bottleneck lies with the switch, amplified by its virtualisation.

If we were to restrict the link bandwidth to something small, it could become the bottleneck before the switch operation does, thereby eliminating the unintended bandwidth reduction in the default queue.

Alternative to using `ovs-vsctl`: using OpenFlow's `dpctl` utility (which doesn't seem to work for me on mininet with OpenFlow 1.0 and OVS.)

```
dpctl add-queue tcp:localhost 3 1 6
dpctl add-queue tcp:localhost 3 2 4
```

(also tried `dpctl add-queue unix:/var/run.openvswitch/s1.mgmt # # #`)

Better: tc'd links to simulate lower bandwidths, delay, loss etc.

Same topology as before, but this time `bw=10`, supported by using tc links. This means additional `mn` parameter:

```
sudo mn --custom topo2sw3h.py --topo 2s3h --mac -x --controller=remote --link tc
```

With 10Mbit/s links and no QoS we get an even split between the two queues, very close to 5Mbit/s each (much closer to saturation now that the switch operation bottleneck has been relieved.)

This attempts a 25:70 split between TCP 666 and non-TCP 666 traffic with contention, but full (nearly) bandwidth without contention:

(needs root)

```
ovs-vsctl -- set Port s1-eth1 qos=@newqos -- \
--id=@newqos create QoS type=linux-htb other-config:max-rate=9500000 queues=0=@q0,1=@q1 -- \
--id=@q0 create Queue other-config:min-rate=7000000 other-config:max-rate=9500000 -- \
--id=@q1 create Queue other-config:min-rate=2500000 other-config:max-rate=9500000
```

To test (see Figure 5):

on h3: `iperf -sp 666 & iperf -sp 5001 &`

on h5: `iperf -c 10.0.0.1 -p 666 -r & iperf -c 10.0.0.1 -p 5001 -r &`

Simultaneously runs `iperf` with round trips (both directions) testing TCP 666 against TCP 5001

Note: you will need to `killall iperf` when doing this more than once, as they are backgrounded on the same terminal for a fair start.

```

Node: h5
-----
Server listening on TCP port 5001
TCP window size: 85,3 KByte (default)
-----
Client connecting to 10.0.0.1, TCP port 666
TCP window size: 22,9 KByte (default)
-----
[ 3] local 10.0.0.3 port 39775 connected with 10.0.0.1 port 666
-----
Client connecting to 10.0.0.1, TCP port 5001
TCP window size: 22,9 KByte (default)
-----
[ 5] local 10.0.0.3 port 40495 connected with 10.0.0.1 port 5001
[ ID] Interval      Transfer      Bandwidth
[ 5] 0.0-10,2 sec  8,25 MBytes  6,82 Mbits/sec
[ ID] Interval      Transfer      Bandwidth
[ 3] 0.0-10,3 sec  3,25 MBytes  2,64 Mbits/sec
[ 4] local 10.0.0.3 port 5001 connected with 10.0.0.1 port 34009
[ 5] local 10.0.0.3 port 666 connected with 10.0.0.1 port 53216
[ 4] 0.0-10,7 sec  6,25 MBytes  4,90 Mbits/sec
[ 5] 0.0-10,7 sec  6,12 MBytes  4,80 Mbits/sec

```

Figure 5: Performing traffic shaping test for TCP666 vs. TCP5001; working correctly at almost-saturation.

In order to achieve more advanced sharing behaviour where queues can only share with *some* other queues (such as a tiered subscription service with many users, where a user can only share internally i.e. only his own allotted bandwidth) we must take advantage of a hierarchical QoS scheme such as HTB or HFSC. Unfortunately OVS only implements flat versions of these schemes, hence the limited sharing functionality offered from ovs-vsctl. A side effect of using the OVS utility to set QoS (as above) is that the advantages over HTB that HFSC offers [1] are not available, making them similar in effect.

An answer to this problem is the use of Linux's tc instead of OVS utilities. OVS utilities do not use tc internally, but instead use the same underlying mechanisms that tc uses. This is useful knowledge as it means that most functionality (such as showing stats and dumping the current config) from tc is interoperable with the OVS utility[2]. tc offers a much more powerful set of controls compared to those OVS exposes, and can therefore provide additional functionality.

Traffic shaping using only tc with multi-level HTB qdisc (within Mininet)

We will attempt a 2:7 split between discriminated and non-discriminated traffic where both can share up to 90% of the link bandwidth. This demonstrates the technique on a Linux virtual machine, which should work with devices running OpenWrt or similar. This may not be so easy on real Openflow-enabled switch hardware as this assumes usage of a Linux kernel with tc support.

```
tc -s qdisc ls dev s1-eth1
```

Use tc to list the qdisc(s) on s1-eth1.

```
tc qdisc del dev s1-eth1 root
```

Remove current qdisc (reverts to default pfifo-fast classless qdisc) on s1-eth1

```
tc qdisc add dev s1-eth1 root handle 1: htb default 11

tc class add dev s1-eth1 parent 1: classid 1:1 htb rate 9mbit ceil 9mbit
tc class add dev s1-eth1 parent 1:1 classid 1:10 htb rate 7mbit ceil 9mbit
tc class add dev s1-eth1 parent 1:1 classid 1:11 htb rate 2mbit ceil 9mbit
```

The 1st line creates a new HTB qdisc on s1-eth1, making it root with handle 1:.

The following three lines create classes for the root, queue 0 and queue 1 respectively.

These lines are to show how tc would be used to configure HTB on any Linux port, and won't work when used with OVS, as ovs-vsctl must still be used to register queues on the HTB classes. OVS uses the following naming with respect to the above:

1:fff as root instead of *1:1*

1:1 representing queue 0

1:2 representing queue 1

Why the numbering is made inconsistent with the queue numbers is unclear, but we are straying into territory that is outside of what OVS exposes; namely the full hierarchical abilities of HTB. (same goes for HFSC)

Findings so far

The two main ways OpenFlow can implement QoS is either queueing onto ovs queues (can supposedly be configured with OF itself) with QoS on them (egress shaping), or using fifo-fast queueing on the switch and have OF determine the ToS for each packet. Note that this is not shaping but plain prioritisation for interactive traffic. Some combination of the two could be possible. i.e. have OF enqueue a packet onto a certain HTB class (corresponding to a channel where traffic has been reserved) which then has multi-band fifo (such as pfifo-fast) to prioritise the interactive packets over the non-interactive ones. POX and OVS should both support setting ToS / DiffServ octet.

Setting the ToS doesn't seem to work all the time; not sure where the support gap is, though it may be my implementation.

20 second ping test with 10 seconds of iperf at start to see the impact of arbitrary bandwidth usage on RTTs. (note: this lacks control over the protocol, i.e. we are testing ICMP against TCP, which isn't properly controlled for a fair experiment)

```
iperf -c 10.0.0.1 -p 5001 -i 1 | awk '{ print strftime("%T, "), $0; fflush();}'
| tee out1.csv &
ping -c20 10.0.0.1 | awk '{ print strftime("%T, "), $0; fflush();}' | tee
out2.csv
```

TODO: Compare result to that obtained under packet prioritisation i.e. of ICMP messages. In other words: how to falsify ping latency results on your network.

Bibliography

[0] OVS Queue configuration discussion

<http://openvswitch.org/pipermail/discuss/2011-March/004937.html>

HFSC Scheduling with Linux

<http://linux-ip.net/articles/hfsc.en/>

Linux HTB manual

<http://luxik.cdi.cz/~devik/qos/htb/manual/userg.htm>

[1] HFSC - linux traffic shaping's best kept secret (shell script)

<https://gist.github.com/bradoaks/940616>

Helpful tc guide (basic commands)

<http://www.cyberciti.biz/faq/linux-traffic-shaping-using-tc-to-control-http-traffic/>

[2] <http://openvswitch.org/pipermail/discuss/2011-August/005512.html>