Linux Traffic Control: tc

Your first Mininet project is to use the Linux tc command and Linux HTB to set bandwidth and delay on various links.

Start with the file **threehosts.py** The topology is as follows:

```
h1----+
      |
      r ---- 25 Mbps, 50 ms ---- h4
      |
h2----+
```

There are no constraints on any links except the r--h3 link, which is set to a bandwidth of 25 Mbps and a delay of 40 ms one way, 10 ms the other.

Part 1

Use Linux HTB to set bandwidth on the h1--r or h2--r links, so that two flows arriving at h2 have two different bandwidth limits. Then **change** one of the bandwidth limits *while testing is going on*.

Submit the complete list of all **tc** commands you used, including the **tc class change ...** command.

To receive data, use dualreceive.py on h2. To send data, use sender.py. It might be easiest to make two separate sender.py files, *eg* sender0.py and sender1.py. The full commandline for sender.py, sending 2000 blocks to 10.0.3.10 port 5430, is:

```
python3 sender.py 2000  10.0.3.10  5430
```

You can start two flows on h1, from two different ports, or else add a new node, h3, and have one flow start on h1 and the other on h3.

You run htb on r. If we're trying to control traffic through r, we must apply htb to the *downstream* interface, which for h1->h2 traffic is named **r-eth2**. Here are the commands necessary to limit h1->h2 UDP traffic to 1 mbit/sec, and h1->h2 TCP traffic to 10 mbit/sec. When specifying rates, mbit is megabit/sec and mbps is mega**byte**/sec. Also, these units are written adjacent to the numeric value, with no intervening space!

First, set up the htb "root", and a default class (which we eventually do not use, but the point is that traffic goes through the default class until everything else is set up, so traffic is never blocked).

```
tc qdisc del dev r-eth2 root    # delete any previous htb stuff on r-eth2

tc qdisc add dev r-eth2 root handle 1: htb default 10  # class 10 is the default class
```

Now we add two **classes**, one for UDP (class **1**) and one for TCP (class **2**). At this point you can't tell yet what each class is for; that's in the following paragraph. (Usually we would set up a "root" class, right below the root qdisc but above both the classes below, to enable sharing between them. We're not doing that here, though.)

```
tc class add dev r-eth2 parent 1: classid 1:1 htb rate
1mbit  # '1mbit' has no space! The classes are 1:1 and (next line) 1:2

tc class add dev r-eth2 parent 1: classid 1:2 htb rate
10mbit
```

Now we have to create *filters* that assign traffic to one class or another. The flowid of the filter matches the classid above. We'll assign UDP traffic to classid 1:1, and TCP traffic to classid 1:2, although the tc filter command calls these **flowids**. The "parent 1:" identifies the root above with handle 1:. The "u32" refers to the so-called u32 classifier; the name comes from unsigned 32-bit. The 0xff is an 8-bit mask.

```
tc filter add dev r-eth2 protocol ip  parent 1: u32 match
ip protocol 0x11 0xff flowid 1:1    # 0x11 = 17, the UDP
protocol number in the IP header

tc filter add dev r-eth2 protocol ip  parent 1: u32 match
ip protocol 0x6 0xff flowid 1:2        # 0x6 is the TCP
protocol number
```

Now if we start the senders (you'd have to come up with your own UDP sender), we should see UDP traffic getting 1mbit and TCP traffic getting 10mbit. We can also just drop the 1:2 class (and its filter), and have that traffic go to the default. In this case, UDP is limited to 1mbit and TCP is not limited at all.

We can also apply filters to traffic from selected ports or hosts, so different traffic from the same host is treated differently. Here are a couple examples; the first filters by IP *source* address and the second by TCP *destination* port number (16 bits, so we need a 16-bit mask 0xffff):

```
tc filter add dev r-eth2 protocol ip  parent 1: u32 match
ip src 10.0.1.10 flowid 1:1

tc filter add dev r-eth2 protocol ip  parent 1: u32 match
ip dport 5431 0xffff flowid 1:2
```

Finally, we can replace "add" by "change" to update the rules. This makes the most sense for the "tc class" statements that assign rates. (We can also use "del" to delete classes.)

While you're trying things out, it makes most sense to type the tc commands into xterm windows on the respective hosts (or at the mininet> prompt preceded by the hostname). However, after you get things working, it may be easier to move the commands into the threehosts.py file:

```
r.cmd('tc qdisc add dev r-eth2 root handle 1: htb default
10')
```

Commands go in main(), *after* the r = net['r'].

This is a Python string, so you can put numeric values into it with str.format() (eg `'tc qdisc add dev r-eth2 root handle 1: htb default {}'.format(default_class)`)

Part 2

Basically do the same, but now use h4 as the destination, and also set the delay and the queue capacity. *Set a smaller bandwidth on the r--h4 link, and a higher bandwidth on the h1--r link.*

The tricky part here is that there already *is* a queuing discipline attached to r-eth4, so you will have to *change* it rather than *add* it. And you'll have to get the classid m:n numbers correct. Use "tc qdisc show" and "tc class show dev r-eth4" to figure this out. But, basically, here's the hierarchy:

5: the root HTB qdisc
5:1 the HTB class below the root, specifying rate and ceil and burst
10: the **netem** (**net**work **em**ulator) class below HTB that specifies the delay and queue (or "limit")

The queue capacity won't matter for this assignment, but it *will* matter for the next one (TCP Reno vs Cubic).

Verify your delay with the ping command. Note that netem is "classless", so you can't directly set different delays for different traffic classes. (How would you determine which *direction* the delay applies to?)