Comparing TCP Reno and TCP Cubic

Updates: I am replacing renocubic.py with renocubic2.py and randomtelnet.py with udprandomtelnet.py. There are explanations below.

Your second Mininet project is to compare the throughput of TCP Reno and TCP Cubic, for various combinations of bandwidth and queue size.

Start with the file renocubic2.py. Verify the following initial bandwidth settings:

BottleneckBW=40
fastbw = 200

You will also need to set DELAY and QUEUE as needed. I've eliminated the duplicate settings, so this should be clearer.

You will need a somewhat larger blockcount; I recommend 100,000. This will be entered into the sender1.py/sender2.py files, or else on the command line. It might be easiest to make two separate sender.py files, *eg* sender1.py and sender2.py. If you use the command line, the full command line for sender.py for the TCP Reno connection is

```
python3 sender.py 100000 10.0.4.10 5430 reno
```

The cubic connection will use port 5431 and "cubic". (In theory, you really should try reversing the ports to confirm that makes no difference. But having the two senders use separate hosts (h1 and h2) is likely more important, and that's harder to arrange.)

To receive data, use dualreceive2.py on h4. This is a modified version of dualreceive.py so that you can give it a blockcount so it knows when to stop, and then calculates the ratio for you. Start it like this, where the number on the command line is the same as the blockcount used by the two sender programs:

```
python3 dualreceive2.py 100000
```

When dualreceive2.py is done, it will close the connections. This may lead to a "ConnectionResetError" on the sending side. Ignore it.

The new version of renocubic2.py does a much more accurate job of setting the queue size at r. The outbound interface at r, r-eth4, needs to have delay, bandwidth and queue set. But the delay and queue settings interact, as the netem qdisc implements delay by storing packets in the queue.

One approach is to have a second router, r2, just past r: r--r2-----h4, and move the delay to r2. This is the "optimal" arrangement, as there are apparently some other interactions between netem delay and queue settings. But the fix can largely be achieved with just one routing node, r, and having a three-layer queuing discipline:

root: netem with delay only (qdisc 1:)
middle: htb with bandwidth only (qdisc 10:, class is 10:1)
leaf: netem with queue only (qdisc 20:)

Outbound packets arrive and go to the leaf netem. They can be withdrawn only at the rate as specified by the htb layer, so if too many arrive, packets are dropped. This is exactly the behavior we want.

After packets are withdrawn from the leaf queue by the htb layer, consistent with the htb rate specification, they then enter the "long queue" created by the root netem, which implements delay. So, first queue, then rate, then delay.

One more thing

A bare-bones competition is highly subject to **TCP phase effects**. So you need to introduce some randomizing traffic. This can be done using udprandomtelnet.py. You should verify the following variable values: BottleneckBW=40 (40 mbit/sec) and density=0.02 (2%) (since two connections are using the same h1--r link). The version of udprandomtelnet.py linked to here has these values preset.

I've switched to UDP because the small TCP packets were running together. With the values above, the mean gap between packets is something like 2.5ms, and TCP tends to combine packets sent within 5-10ms of one another.

The random-telnet solution is still imperfect; there is still a great deal of variability. But it's better than without.

You will run udprandomtelnet.py on h1; the existing parameters should be good though you *will* have to specify the destination address:

```
python3 udprandomtelnet.py 10.0.4.10
```

To receive this traffic, run the following on h4 (5433 is the port number, and we're redirecting output to /dev/null because we're ignoring it)

```
netcat -l -u 5433 > /dev/null
```

I find it easiest to create a separate pair of xterm windows on h1 and h4 for the purposes of running this.

What to do

Get comparison results with DELAY=200 and QUEUE=200, 400, 800 and 1600. Then do this again with DELAY=400, for the same four QUEUE value Because of the variability, you will need to do **at least five runs for each combination**.

queue:	200	400	800	1600
DELAY=200				
DELAY=400				

For each of the eight combinations, **submit:**

- the Reno-to-Cubic bandwidth ratio of your runs (reported directly for you by dualreceive2.py if you add the blockcount parameter)
- the average bandwidth ratio
- the **standard deviation** (enter your results in a spreadsheet, and let that calculate it for you (it might be the STDEV() function). Have the spreadsheet calculate AVERAGE() for you too!)

Note the bandwidth x delay product here for DELAY=200 (one way) is 5 MBps x 0.4 sec = 2MB, which is ~1300 packets.

You should also **submit a conclusion**: does Reno do better with larger queues, or not? Does Reno do better with the larger DELAY?

Finally, tell me the **bottleneck-link utilization** for at least one trial in each of the eight combinations. The bottleneck link is 5 MB/sec. Note the finishing time, and the total transferred up until then. Divide to get the throughput. How does it compare to 5 MB/sec? Packet headers run an extra 54 bytes, which is about 3.6%, and there's 2% taken up by the randomtelnet traffic. So you really have an effective throughput of around 4.72 MB/sec.