For this project you are to install a Mininet virtual machine (see mininet_basics.html), and complete **two** of the experiments below. Neither involves "serious" programming, but you will have to use Python as a configuration language, and maybe be able to do some basic shell scripting.

1. TCP Simultaneous Open

For this project, you are to arrange so that you can view and record TCP simultaneous open. Here is a suitable network configuration:

h1 ---- R ---- h2

The main purpose of the router R is to have a place where you can monitor the traffic, using tcpdump or wireshark.

Start xterms (x-terminals) on each of h1, h2, and R. I will assume that h1's IPv4 address is 10.0.0.10 and h2's is 10.0.1.10; these are the assignments if you run **python routerline.py -N 1**. Then run the following:

- on h2, run **netcat -l 5432**. This creates a TCP listener on port 5432. You can use a different port if you prefer.
- on R, run either **wireshark** or **tcpdump -i r-eth0 tcp**. This listens to the traffic on interface r-eth0, and reports/records all TCP traffic. If you're getting too much, specify **tcp port 5432**. You can also do this in wireshark.
- finally, on h1 run **netcat 10.0.1.10 5432**, and type a few lines of text. You should see the output on h2, and evidence of traffic on R.

But this doesn't get you a simultaneous open. To do that, you'll have to run **netcat -p 5432 *ip-address* 5432** on both h1 and h2 simultaneously. The **-p 5432** specifies the source port; you can also do netcat -p 2000 h2 3000 on h1, and netcat -p 3000 h1 2000 on h2, for example.

To run the two commands simultaneously is impossible, of course. However, you can add some **delay** to the h1--R and R--h2 links, at least 1000 ms worth. You will probably also want to use ssh from, say, R to h1 and h2, to run the commands in quick succession.

To **set the delay**, see the file competition2.py, which unfortunately I did not get pre-installed in the virtual machine (use ssh or scp to transfer, or copy/paste if you are able to get that working, or just open a browser *within* the virtual machine and download it from this page). I've set `DELAY='250ms'` in the file, and, if mininet is running and I type `h1 ping h3`, I get

```
PING 10.0.3.10 (10.0.3.10) 56(84) bytes of data.
64 bytes from 10.0.3.10: icmp_seq=1 ttl=63 time=250 ms
64 bytes from 10.0.3.10: icmp_seq=2 ttl=63 time=250 ms
64 bytes from 10.0.3.10: icmp_seq=3 ttl=63 time=250 ms
```

which is exactly what it is supposed to be.

I did this experiment using a modified version of competition.py, which I named delayline.py. (Specifically, I increased the DELAY variable and made sure r's interfaces were being assigned IP addresses (in my version I also eliminated h2, but that was unnecessary).) The file creates a topology h1 ---- r ---- h3, with a one-way delay on the r---h3 link. The delay applies only in the h3-to-r direction, though, so to arrange a packet crossing you will have to take that into account. But this is good enough, provided you start at h3. Here's the basic idea:

- set the delay high enough
- start the command `netcat -p 5432 10.0.1.10 5432` on h3. This will take the delay time to reach h1.
- As quickly as you can, start the same command on h1, except with IP address 10.0.3.10.

One way to do this quickly is to type both commands, but *don't* press Enter. Then give the h3 window the focus (that is, click on it), and move the mouse over the **h1** window, but don't click yet. Press ENTER, and the h3 command will start, because that window still has the focus. Now click -- on the h1 window -- and press ENTER again to start the previously typed command on h1. In other words, the sequence is ENTER-CLICK-ENTER. That's easy to do quickly; there's no mouse motion in that sequence.

It took me a fair number of tries, but I finally got it to work (almost all my failed attempts were because I typed a port number or IP address wrong). One bright spot: if you get the simultaneous open to work, then typing a string on *either* side will cause it to appear on the other side. I had thought that netcat would simply create the connection (visible in tcpdump) and then lock up.

How much delay do you need? 3000 ms is easy; can you do it in 1000?

Bidirectional delay *is* straightforward to set up, but we don't need it here. All we need is for packets to cross in the wire. So start on h3, and while that SYN packet is making its way to h1, start on h1 before it gets there. Note that if you start on h1, the SYN packet arrives at h3 essentially *immediately*, and there is no hope of arranging a simultaneous open.

There's a problem with competition.py: the commands to set up node r are commented out, *and they use the wrong interfaces*. Here are the commented-out lines:

```
# r.cmd('ifconfig r-eth0 10.0.1.1/24')
# r.cmd('ifconfig r-eth1 10.0.2.1/24')
# r.cmd('ifconfig r-eth2 10.0.3.1/24')
```

But at some point an update to Mininet changed the interface numbering, so instead of eth0,eth1,eth2 it's now eth1,eth2,eth3, and I did not notice this. Change the interface names as follows:

```
r.cmd('ifconfig r-eth1 10.0.1.1/24')
r.cmd('ifconfig r-eth2 10.0.2.1/24')
r.cmd('ifconfig r-eth3 10.0.3.1/24')
```

Bonus points: close the connection on h3 first and then as soon as possible on h1 in order to generate a simultaneous close.

Alternative method: set up ssh so, from r, you can put ssh commands in a shell script to start netcat on h1 and then netcat on h3. But you may need to do a moderate bit of ssh configuration.

To submit, save the .pcap file from tcpdump (it may have a .pcapng extension; ng for "next generation")

2. TCP Reno vs Cubic

The goal is to compare the total throughput for the two TCP congestion mechanisms. For this example, the relevant Python file is competition.py. TCP Reno and TCP Cubic are both available.

Choose a time interval, *eg* 10 seconds (maybe less for trial runs), and compare the throughput of two connections sharing a link. Do a few runs to get an idea of how consistent your results are. Actually, your time interval will usually be until one connection transfers all of its blocks, so what you're really doing is choosing a value for blockcount large enough that the test runs for 10 seconds (or 30 seconds or whatever).

I recommend comparing Reno to Cubic and also, as a control, Reno to Reno.

It is important to try different RTTs, eg 50 ms, 100 ms and 200 ms (for the basic tests, you can implement delay on the common link; be careful whether it is bidirectional or not). Both connections will usually use the same RTT for any given run, though.

You might also try *different* delays on the two paths. TCP Cubic is supposed to handle this reasonably well for cubic-to-cubic competition. But this is optional.

This one is probably the most straightforward: use competition.py, with dualreceive.py running on h3 and two instances of sender.py on each of h1 and h2 (or both running on h1, which is what I usually do). Note that sender.py takes command-line arguments to control its behavior: the following

```
    python3 sender.py 10000 10.0.3.10 5430 reno
```

sends 10000 blocks to h3 at port 5430 (one of the two used by default in dualreceive.py) using TCP Reno. You can replace 'reno' by 'cubic' to get TCP Cubic.

In the original handout (in the first experiment) I described a file competition2.py, but the differences between it and competition.py are pretty minimal (except that the variable DELAY is set to a different value). It turns out that competition2.py was a file I created *after* distributing the virtual hard disk, so it won't be on your system. Again, all I did was to change DELAY to a new value.

You can also use dualreceive2.py, which calculates the bandwidth ratio for you. This requires a block-count parameter, which should match that of the senders:

python3 dualreceive2.py **100000**

Phase effects

Strictly speaking, the setup here will be influenced by **phase effects**. You are **not** required to adjust for these in this assignment. But the way to address these would be to introduce some random traffic to break up the "phase locks". 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.

This uses 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.

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

```
python3 udprandomtelnet.py 10.0.3.10
```

To receive this traffic, run the following on the destination host, *eg* h3 (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 usually find it easiest to create a separate pair of xterm windows on h1 and h3 for the purposes of running this.

To submit, send me (a) a description of the setup you actually used, (b) some of your output, and (c) a brief discussion of how you determined the relative throughputs of reno and cubic from that output. Note that you'll need to do this for each RTT you used.

3. Bandwidth limits

Set up a topology like this, based on the competition.py file:

```
h1-----+
       |
h2-----R-----h4
       |
h3-----+
```

Run htb on R, and set separate classes for traffic from h1/h2/h3 to h4. Run sender.py on each of h1/h2/h3, and a receiver on h4 (I'm still working on that; dualreceive.py only receives from two and that should be good enough to get started while I work out a more general solution).

Set appropriate bandwidth limits for each of h1/h2/h3, and show that they are obeyed.

Experiment with a large burst for h1, but not h2/h3. What happens?

Can you *change* the bandwidth while traffic is being sent, and see a change in the received bandwidth?

To do this you must set up the Linux bandwidth-throttling system, known as **htb** for **h**ierarchical **t**oken **b**ucket. The command you use is **tc**, for **t**raffic **c**ontrol.

I will for the time being stick with the competition.py configuration with one less host:

```
h1----+
      |
      r ---- h3
      |
h2----+
```

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->h3 and h2->h3 traffic is named **r-eth3**. Here are the commands necessary to limit h1->h3 traffic to 1 mbit/sec, and h2->h3 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-eth3 root     # delete any previous htb stuff on r-eth3

tc qdisc add dev r-eth3 root handle 1: htb default 10  #
class 10 is the default class
```

Now we add two classes, one for h1 (class 1) and one for h2 (class 2). At this point you can't tell yet that class 1 is for h1; that's in the following paragraph.

```
tc class add dev r-eth3 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-eth3 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 traffic *from* h1 (10.0.1.10) to classid 1:1, and traffic from h2 to classid 1:2. 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.

```
tc filter add dev r-eth3 protocol ip  parent 1: u32 match
ip src 10.0.1.10 flowid 1:1

tc filter add dev r-eth3 protocol ip  parent 1: u32 match
ip src 10.0.2.10 flowid 1:2
```

Now if we start the senders, we should see h1 getting 1/10 the bandwidth! Also, each sender should get 1mbit and 10mbit respectively. We can also just drop the 1:2 class (and its filter), and have that traffic go to the default. In this case, h1 is limited to 1mbit and h2 is not limited at all.

We can also apply filters to traffic from selected ports, so different traffic from the same host is treated differently.

We can also 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.)

To submit, send me your own list of tc/htb commands, some output (from, *eg*, dualreceive.py) showing the bandwidth throttling, and a brief discussion.