

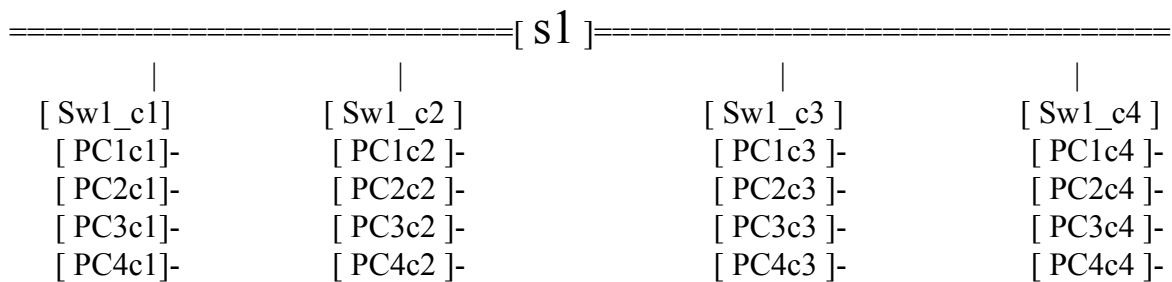
REPORT

COMPUTER NETWORK LAB ASSIGNMENT 1

Submitted by IIT2018187(Abhishek Gupta)

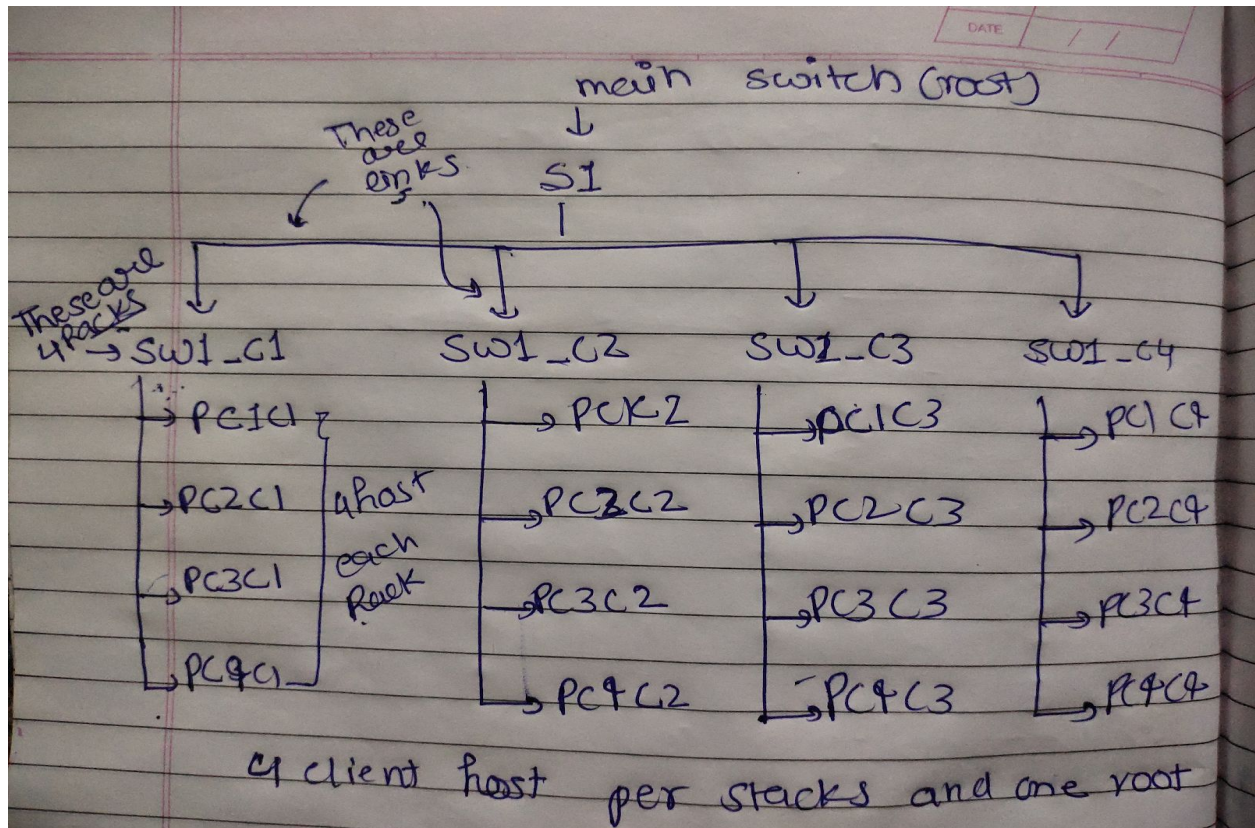
Question 1

1. Build a Data Center Topology as shown below using Mininet CLI (DatacenterBasicTopo). In this topology, we have four brackets, each with four forces and one top-of-rack (ToR) switch. These ToR switches are connected to a central root switch.



Ans:

Flow chart:



Any host can reach any other host because each host is linked by its racks switch which is further linked with the root switch just like a tree. As we know in tree we can reach one node to another by a simple path..just like that.

Suppose pc4c1 wants to reach pc4c4. The path of that reachability

Pc4c1 -> connects sw1_c1 -> connects root switch s1 -> connects switch sw1_c4 -> pc4c4..

Explaining Function:

1. def build(self):

This function create root switch and link to 4 another (ToR) switch and calling Another function makeRakes to add hosts with Tors.

2. def makeRakes(self, loc):

This function creates and links hosts to their corresponding ToRs.

Used libraries:

We used all libraries from mininet and open vSwitch

These libraries help to create switches and make connections.

Algorithm :

In this question we don't need any special algorithm to make that basic topologies. We just use a loop to create multiple switches and multiple links.

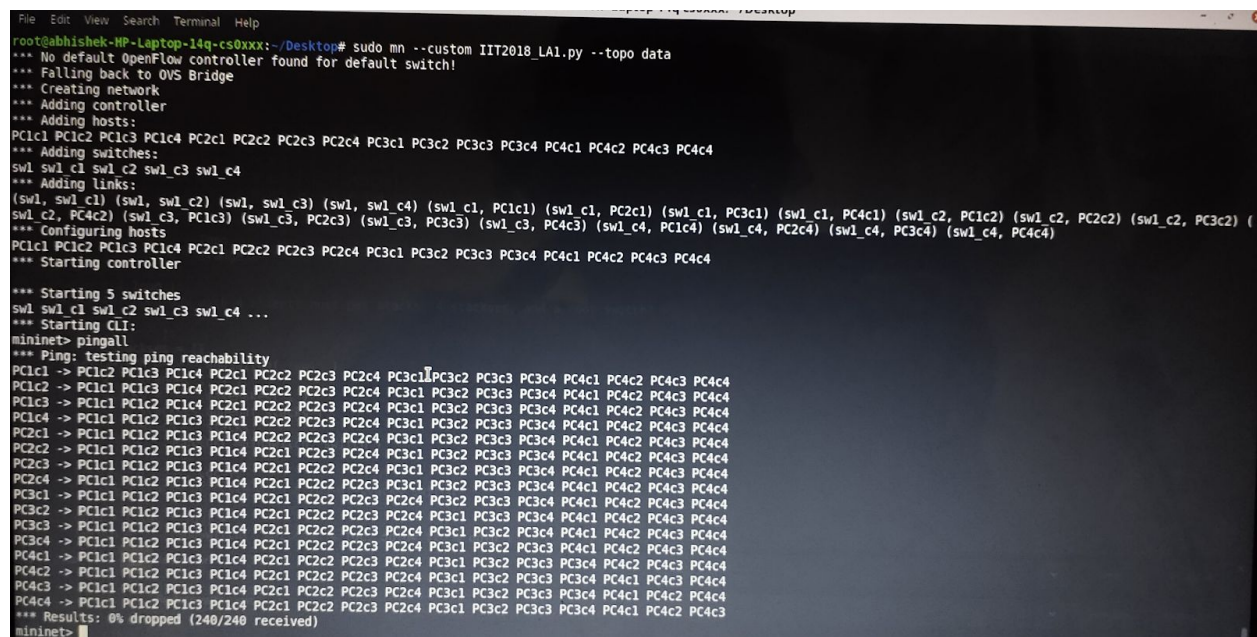
```
for i in xrange( 1, 4 ):
    stacks = self.makeRakes( i )
    self.stackses.append( stacks )
    for switch in stacks:
        self.addLink( mainSwitch, switch )
```

Also we use another loop to add hosts to corresponding ToRs

```
switch = self.addSwitch( 'sw1_c%s' % loc, dpid='%x' % dpid )
```

```
for n in xrange( 1, 4 ):
    host = self.addHost( 'PC%s%s' % ( n, loc ) )
    self.addLink( switch, host )
```

Output screen:

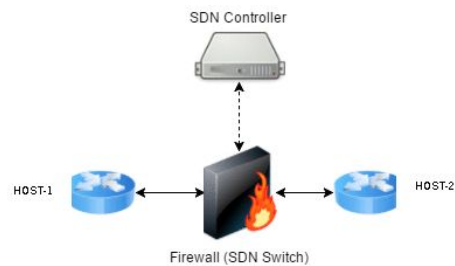


```
File Edit View Search Terminal Help
root@abhishek-HP-Laptop-14g-cs0xxx:~/Desktop# sudo mn --custom IIT2018_LA1.py --topo data
*** No default OpenFlow controller found for default switch!
*** Falling back to OVS Bridge
*** Creating network
*** Adding controller
*** Adding hosts:
PC1c1 PC1c2 PC1c3 PC1c4 PC2c1 PC2c2 PC2c3 PC2c4 PC3c1 PC3c2 PC3c3 PC3c4 PC4c1 PC4c2 PC4c3 PC4c4
*** Adding switches:
sw1 sw1_c1 sw1_c2 sw1_c3 sw1_c4
*** Adding links:
(sw1, sw1_c1) (sw1, sw1_c2) (sw1, sw1_c3) (sw1, sw1_c4) (sw1_c1, PC1c1) (sw1_c1, PC2c1) (sw1_c1, PC3c1) (sw1_c1, PC4c1) (sw1_c2, PC1c2) (sw1_c2, PC2c2) (sw1_c2, PC3c2) (sw1_c2, PC4c2) (sw1_c3, PC1c3) (sw1_c3, PC2c3) (sw1_c3, PC3c3) (sw1_c3, PC4c3) (sw1_c4, PC1c4) (sw1_c4, PC2c4) (sw1_c4, PC3c4) (sw1_c4, PC4c4)
*** Configuring hosts
PC1c1 PC1c2 PC1c3 PC1c4 PC2c1 PC2c2 PC2c3 PC2c4 PC3c1 PC3c2 PC3c3 PC3c4 PC4c1 PC4c2 PC4c3 PC4c4
*** Starting controller
*** Starting 5 switches
sw1 sw1_c1 sw1_c2 sw1_c3 sw1_c4 ...
*** Starting CLI:
mininet> pingall
*** Ping: testing ping reachability
PC1c1 -> PC1c2 PC1c3 PC1c4 PC2c1 PC2c2 PC2c3 PC2c4 PC3c1 PC3c2 PC3c3 PC3c4 PC4c1 PC4c2 PC4c3 PC4c4
PC1c2 -> PC1c1 PC1c3 PC1c4 PC2c1 PC2c2 PC2c3 PC2c4 PC3c1 PC3c2 PC3c3 PC3c4 PC4c1 PC4c2 PC4c3 PC4c4
PC1c3 -> PC1c1 PC1c2 PC1c4 PC2c1 PC2c2 PC2c3 PC2c4 PC3c1 PC3c2 PC3c3 PC3c4 PC4c1 PC4c2 PC4c3 PC4c4
PC1c4 -> PC1c1 PC1c2 PC1c3 PC2c1 PC2c2 PC2c3 PC2c4 PC3c1 PC3c2 PC3c3 PC3c4 PC4c1 PC4c2 PC4c3 PC4c4
PC2c1 -> PC1c1 PC1c2 PC1c3 PC1c4 PC2c2 PC2c3 PC2c4 PC3c1 PC3c2 PC3c3 PC3c4 PC4c1 PC4c2 PC4c3 PC4c4
PC2c2 -> PC1c1 PC1c2 PC1c3 PC1c4 PC2c1 PC2c3 PC2c4 PC3c1 PC3c2 PC3c3 PC3c4 PC4c1 PC4c2 PC4c3 PC4c4
PC2c3 -> PC1c1 PC1c2 PC1c3 PC1c4 PC2c1 PC2c2 PC2c4 PC3c1 PC3c2 PC3c3 PC3c4 PC4c1 PC4c2 PC4c3 PC4c4
PC2c4 -> PC1c1 PC1c2 PC1c3 PC1c4 PC2c1 PC2c2 PC2c3 PC2c4 PC3c1 PC3c2 PC3c3 PC3c4 PC4c1 PC4c2 PC4c3 PC4c4
PC3c1 -> PC1c1 PC1c2 PC1c3 PC1c4 PC2c1 PC2c2 PC2c3 PC2c4 PC3c2 PC3c3 PC3c4 PC4c1 PC4c2 PC4c3 PC4c4
PC3c2 -> PC1c1 PC1c2 PC1c3 PC1c4 PC2c1 PC2c2 PC2c3 PC2c4 PC3c1 PC3c2 PC3c3 PC3c4 PC4c1 PC4c2 PC4c3 PC4c4
PC3c3 -> PC1c1 PC1c2 PC1c3 PC1c4 PC2c1 PC2c2 PC2c3 PC2c4 PC3c1 PC3c2 PC3c3 PC3c4 PC4c1 PC4c2 PC4c3 PC4c4
PC3c4 -> PC1c1 PC1c2 PC1c3 PC1c4 PC2c1 PC2c2 PC2c3 PC2c4 PC3c1 PC3c2 PC3c3 PC3c4 PC4c1 PC4c2 PC4c3 PC4c4
PC4c1 -> PC1c1 PC1c2 PC1c3 PC1c4 PC2c1 PC2c2 PC2c3 PC2c4 PC3c1 PC3c2 PC3c3 PC3c4 PC4c2 PC4c3 PC4c4
PC4c2 -> PC1c1 PC1c2 PC1c3 PC1c4 PC2c1 PC2c2 PC2c3 PC2c4 PC3c1 PC3c2 PC3c3 PC3c4 PC4c1 PC4c2 PC4c3 PC4c4
PC4c3 -> PC1c1 PC1c2 PC1c3 PC1c4 PC2c1 PC2c2 PC2c3 PC2c4 PC3c1 PC3c2 PC3c3 PC3c4 PC4c1 PC4c2 PC4c3 PC4c4
PC4c4 -> PC1c1 PC1c2 PC1c3 PC1c4 PC2c1 PC2c2 PC2c3 PC2c4 PC3c1 PC3c2 PC3c3 PC3c4 PC4c1 PC4c2 PC4c3 PC4c4
*** Results: 0% dropped (240/240 received)
mininet>
```

2. Implement the SDN firewall in the network to obstruct traffic coming its way and filter it according to some *rules*. A general firewall usually protects the system from the internet.

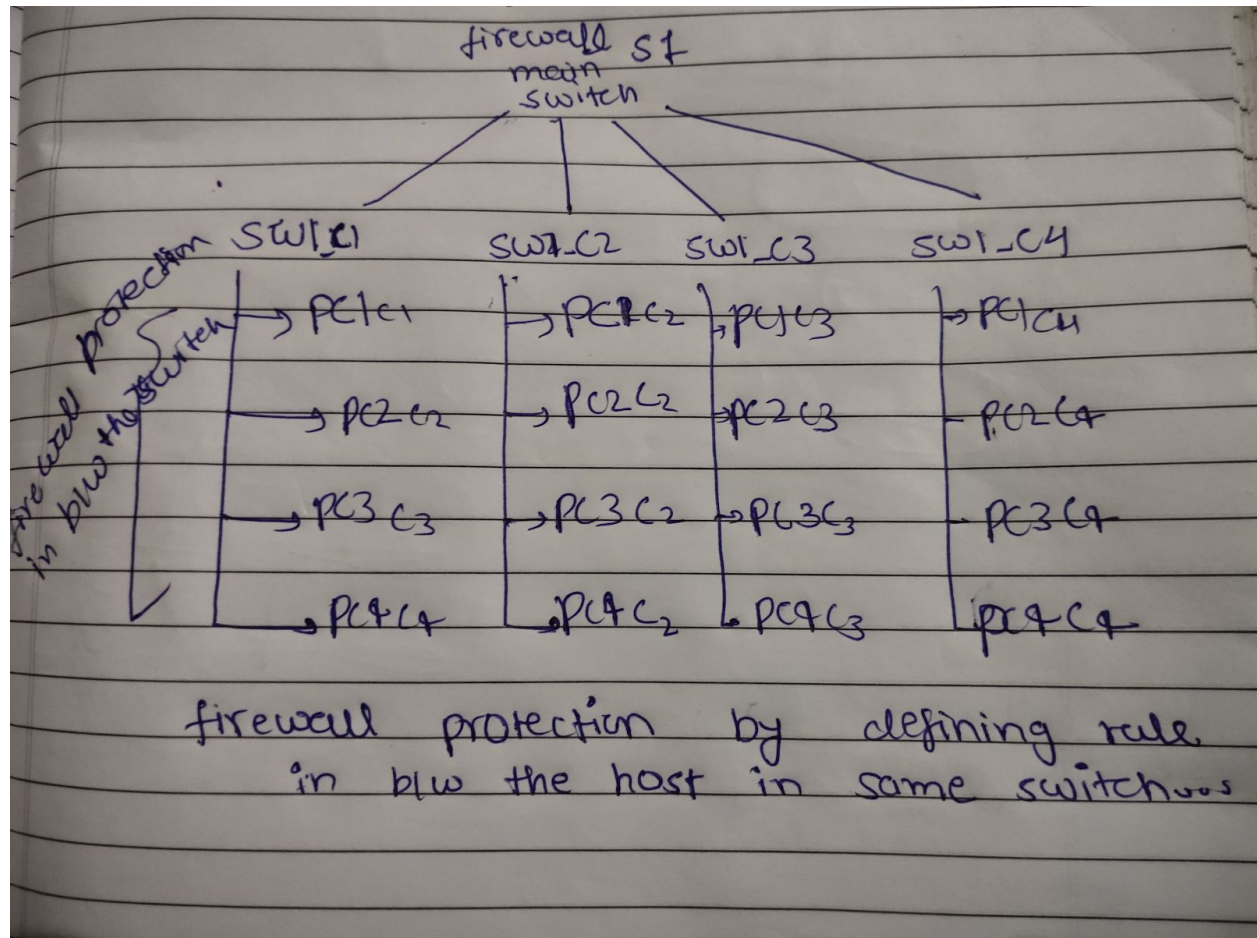
Hint :

[For an SDN based Firewall, an OpenFlow controller is required to filter traffic between hosts according to some rules and accordingly let it pass through or not]. [use POX controller to establish our required *policies* or *rules* and filter traffic between hosts using the switches.]



- PC1c1, PC2c1, PC3c1 and PC4c1 should mutually blocked
- PC1c2, PC2c2, PC3c2 and PC4c2 should mutually blocked
- PC1c3, PC2c3, PC3c3 and PC4c3 should mutually blocked
- PC1c4, PC2c4, PC3c4 and PC4c4 should mutually blocked

Ans:



Explaining function:

We assign mac address to each host to define protocols or rules.

And also change from controller to remote controller .

```
net = Mininet( controller=RemoteController )
```

```
PC1c1 = net.addHost( 'PC1c1', ip='10.0.0.1', mac='00:00:00:00:00:01' )
```

Now we will discuss about firewall rules :

For destroying connection between two nodes i made a method `_handle_ConnectionUp`

```
1. def _handle_ConnectionUp (self, event):
    for i in xrange(0,14):
        for j in xrange(i+1,(i/4)*4+4):
            block = of.ofp_match()
            block.dl_src = EthAddr(rule[0])
            block.dl_dst = EthAddr(rule[1])
```

```

flow_mod = of.ofp_flow_mod()
flow_mod.match = block
event.connection.send(flow_mod)

```

```
of.ofp_match()
```

This function destroys the network between source and destination MAC addresses.

```

2. def launch ():
    core.registerNew(SDNFirewall)

```

This function launches the firewall through the controller.

Explaining libraries:

I used one additional Pox library for defining rules and policies of firewalls.

POX is a Python based open source OpenFlow/Software Defined Networking (SDN) Controller. POX is used for faster development and prototyping of new network applications. POX controller comes pre-installed with the mininet virtual machine. Using POX controller you can turn dumb openflow devices into hub, switch, load balancer, firewall devices. The POX controller allows easy way to run OpenFlow/SDN experiments. POX can be passed different parameters according to real or experimental topologies, thus allowing you to run experiments on real hardware, testbeds or in a mininet emulator. In this paper, the first section will contain an introduction about POX, OpenFlow and SDN, then discussion about the relationship between POX and Mininet. Final Sections will be regarding creating and verifying behavior of network applications in POX.

We define some rules for firewall in firewall.py in pox directory. Which runs separately from mininet.

Explaining Algorithm

We don't change the algorithm to add switches and hosts.

We just add a MAC address to each host.

And we also define rules by simple for loop. By defining all pairs which have to be mutually blocked.

The rules consist of a list of pairs of MAC addresses of the hosts to be blocked.

```

def _handle_ConnectionUp (self, event):
    for i in xrange(0,14):                //loop for every source
        for j in xrange(i+1,(i/4)*4+4):    // every source blocked with all destination
            block = of.ofp_match()

```



```

block.dl_src = EthAddr(rule[i]) //source of blocked connection
block.dl_dst = EthAddr(rule[j]) //dest of blocked connection.
flow_mod = of.ofp_flow_mod()
flow_mod.match = block
event.connection.send(flow_mod)

```

Output screen:

```

*** Configuring hosts
PC1c1 PC1c2 PC1c3 PC1c4 PC2c1 PC2c2 PC2c3 PC2c4 PC3c1 PC3c2 PC3c3 PC3c4 PC4c1 PC4c2 PC4c3 PC4c4
*** Starting controller

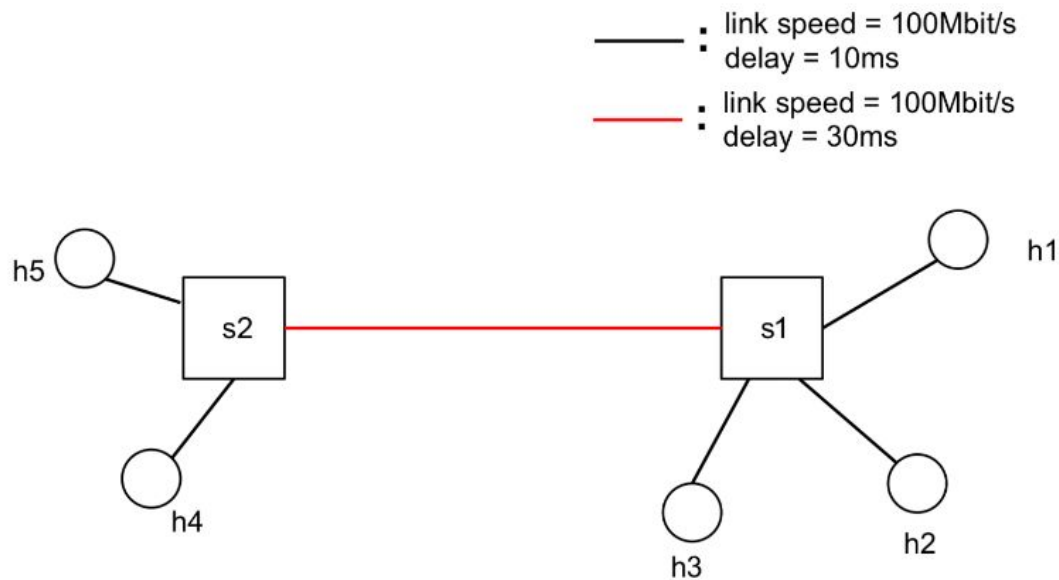
*** Starting 5 switches
sw1 sw1_c1 sw1_c2 sw1_c3 sw1_c4 ...
*** Starting CLI:
mininet> pingall
*** Ping: testing ping reachability
PC1c1 -> PC1c2 PC1c3 PC1c4 PC2c1 PC2c2 PC2c3 PC2c4 PC3c1 PC3c2 PC3c3 PC3c4 PC4c1 PC4c2 PC4c3 PC4c4
PC1c2 -> PC1c1 PC1c3 PC1c4 PC2c1 PC2c2 PC2c3 PC2c4 PC3c1 PC3c2 PC3c3 PC3c4 PC4c1 PC4c2 PC4c3 PC4c4
PC1c3 -> PC1c1 PC1c2 PC1c4 PC2c1 PC2c2 PC2c3 PC2c4 PC3c1 PC3c2 PC3c3 PC3c4 PC4c1 PC4c2 PC4c3 PC4c4
PC1c4 -> PC1c1 PC1c2 PC1c3 PC2c1 PC2c2 PC2c3 PC2c4 PC3c1 PC3c2 PC3c3 PC3c4 PC4c1 PC4c2 PC4c3 PC4c4
PC2c1 -> PC1c2 PC1c3 PC1c4 PC2c2 PC2c3 PC2c4 PC3c1 PC3c2 PC3c3 PC3c4 PC4c1 PC4c2 PC4c3 PC4c4
PC2c2 -> PC1c1 PC1c3 PC1c4 PC2c1 PC2c3 PC2c4 PC3c1 PC3c2 PC3c3 PC3c4 PC4c1 PC4c2 PC4c3 PC4c4
PC2c3 -> PC1c1 PC1c2 PC1c4 PC2c1 PC2c2 PC2c4 PC3c1 PC3c2 PC3c3 PC3c4 PC4c1 PC4c2 PC4c3 PC4c4
PC2c4 -> PC1c1 PC1c2 PC1c3 PC2c1 PC2c2 PC2c3 PC3c1 PC3c2 PC3c3 PC3c4 PC4c1 PC4c2 PC4c3 PC4c4
PC3c1 -> PC1c2 PC1c3 PC1c4 PC2c1 PC2c2 PC2c3 PC2c4 PC3c2 PC3c3 PC3c4 PC4c1 PC4c2 PC4c3 PC4c4
PC3c2 -> PC1c1 PC1c3 PC1c4 PC2c1 PC2c2 PC2c3 PC2c4 PC3c1 PC3c3 PC3c4 PC4c1 PC4c2 PC4c3 PC4c4
PC3c3 -> PC1c1 PC1c2 PC1c4 PC2c1 PC2c2 PC2c3 PC2c4 PC3c1 PC3c2 PC3c4 PC4c1 PC4c2 PC4c3 PC4c4
PC3c4 -> PC1c1 PC1c2 PC1c3 PC2c1 PC2c2 PC2c3 PC2c4 PC3c1 PC3c2 PC3c3 PC3c4 PC4c1 PC4c2 PC4c3 PC4c4
PC4c1 -> PC1c2 PC1c3 PC1c4 PC2c1 PC2c2 PC2c3 PC2c4 PC3c1 PC3c2 PC3c3 PC3c4 PC4c2 PC4c3 PC4c4
PC4c2 -> PC1c1 PC1c3 PC1c4 PC2c1 PC2c2 PC2c3 PC2c4 PC3c1 PC3c2 PC3c3 PC3c4 PC4c1 PC4c3 PC4c4
PC4c3 -> PC1c1 PC1c2 PC1c4 PC2c1 PC2c2 PC2c3 PC2c4 PC3c1 PC3c2 PC3c3 PC3c4 PC4c1 PC4c2 PC4c4
PC4c4 -> PC1c1 PC1c2 PC1c3 PC2c1 PC2c2 PC2c3 PC2c4 PC3c1 PC3c2 PC3c3 PC3c4 PC4c1 PC4c2 PC4c3
*** Results: 19% dropped (192/240 received)
mininet>

```

QUESTION 2:

TCP throughput & fairness

In this exercise we will study TCP's throughput and fairness characteristics with Mininet. Consider the topology shown in the figure below.



All links are 100 Mbit/s and have a one-way propagation delay of 10 ms, except for the link between S1 and S2, which has a 30 ms propagation delay. The switches have 1000 packets of buffer at each port, and use PIE active queue management with a target delay of 20 ms.

The goal of this exercise is to learn how the RTT and number of bottleneck links impacts throughput and fairness of TCP flows.

Tasks

Within Mininet, create the topology shown above, and enable PIE with a target delay of 20 ms on the switch interfaces. Then do the following:

Scenario 1:

- Start 10 long lived TCP flows sending data from h2 to h1, and similarly 10 long lived TCP flows from h5 to h1.
- Start back-to-back ping trains from h2 to h1, and h5 to h1. Record the RTTs with ping 10 times a second.
- Measure the following:
 - o The average throughput for h2->h1 and h5->h1 flows. For each group, measure the aggregate throughput of the 10 flows.
 - o The average RTT for h2->h1 and h5->h1 flows.

Scenario 2:

- Now start 10 long lived TCP flows from h4 to h3.

- Repeat the above measurements (average throughput and RTT) for the three groups of flows: h2->h1, h5->h1, and h4->h3.

The starter code sets up the topology and configures PIE. You have to write code to generate the traffic (with Iperf) and do the RTT measurements (with ping).

Starter code

git clone https://github.com/hongzimao/6.829_lab1?files=1
or <https://github.mit.edu/addanki/6.829-lab1.git>

File	Purpose
tcpfairness.py	Creates the topology and sets up PIE.

Answer the following questions below. Remember to keep answers brief.

1. In Scenario 1, which of the following statements is more accurate:
 - o The congestion windows of h2->h1 flows are larger than h5->h1 flows.
 - o The congestion windows of h2->h1 flows are smaller than h5->h1 flows.
 - o The congestion windows of h2->h1 and h5->h1 flows are roughly the same.
2. In Scenario 2, which of the two bottleneck links (S2->S1 or S1->h1) do you expect has the larger drop rate? Briefly explain why

Ans:

Congestion Window (cwnd) is a TCP state variable that limits the amount of data the TCP can send into the network before receiving an ACK. ... Together, the two variables are used to regulate data flow in TCP connections, minimize congestion, and improve network performance

1. Congestion windows will be roughly the same, because of

$CWND = \text{throughput} * RTT$ and TCP's RTT unfairness ratio.

2.

a bottleneck link for a given data flow is a link that is fully utilized (is saturated) and of all the flows sharing this link, the given data flow achieves maximum data rate network-wide.

S1 \rightarrow h1 has the larger drop rate. Use the TCP throughput equation, with Flow A = h5 \rightarrow h1, Flow B = h2 \rightarrow h1, Flow C = h4 \rightarrow h3.

$$p_{ss} = \frac{1}{Tc^2RTTc^2}$$

$$p_{sh} = \frac{1}{Tb^2RTTb^2}$$

We know, $Tc = 100 - x$ and $Tb = 100 - x$ since A and C share a bottleneck link with capacity 100, and the same for A and B. So, $Tc = Tb$. Since $RTTc > RTTb$, we can conclude that p_{sh} is greater.