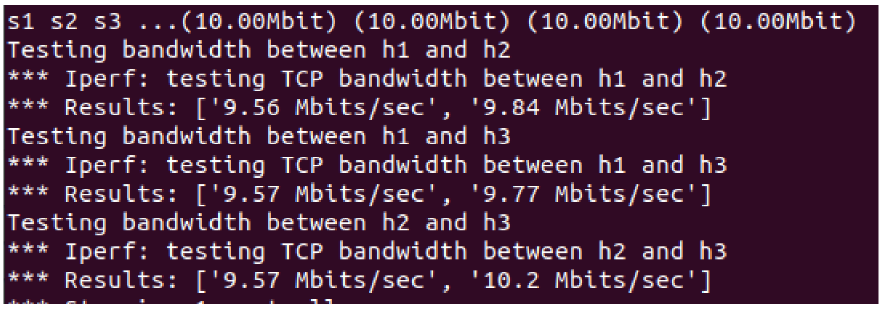
CS339 lab2

邬心远

519021910604

Q1

By using `net.iperf((h1, h2), l4Type='TCP')`, TCP throughput between h1 and h2 can be tested. The code is implenmented in `q1.py`. And the result came like:

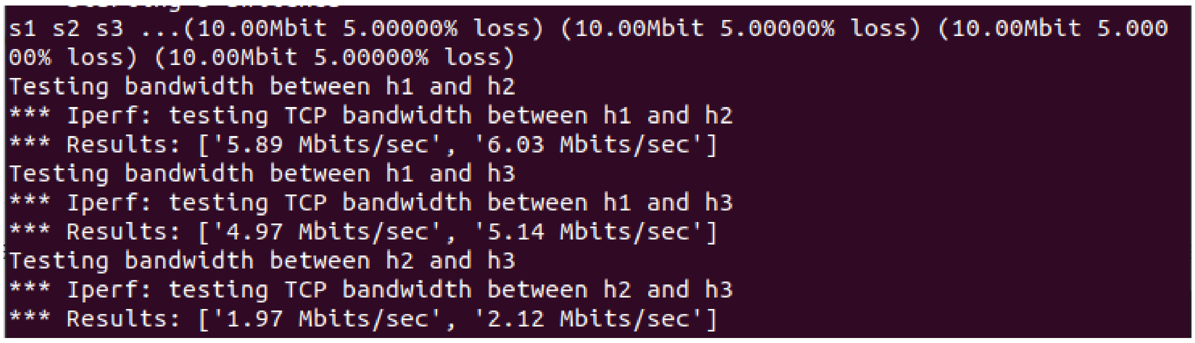


```
s1 s2 s3 ...(10.00Mbit) (10.00Mbit) (10.00Mbit) (10.00Mbit)
Testing bandwidth between h1 and h2
*** Iperf: testing TCP bandwidth between h1 and h2
*** Results: ['9.56 Mbits/sec', '9.84 Mbits/sec']
Testing bandwidth between h1 and h3
*** Iperf: testing TCP bandwidth between h1 and h3
*** Results: ['9.57 Mbits/sec', '9.77 Mbits/sec']
Testing bandwidth between h2 and h3
*** Iperf: testing TCP bandwidth between h2 and h3
*** Results: ['9.57 Mbits/sec', '10.2 Mbits/sec']
```

All the throughput is close to the bandwidth we set in program.

Q2

By specialize the `loss` parameter when using `addlink` function, we can build a network with certain transmission loss. The code for thos part is included in `q2.py`. The iperf testing result is shown in the following picture:

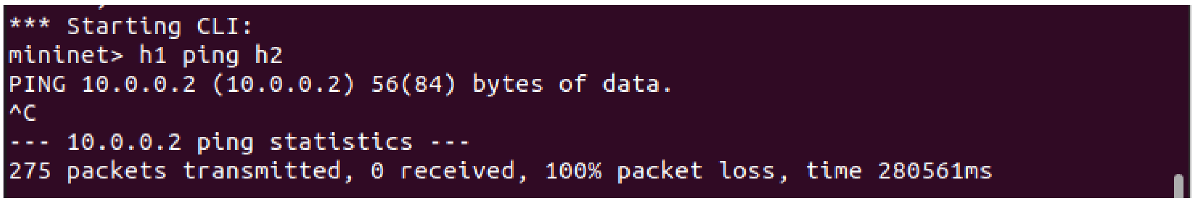


```
s1 s2 s3 ...(10.00Mbit 5.00000% loss) (10.00Mbit 5.00000% loss) (10.00Mbit 5.000
00% loss) (10.00Mbit 5.00000% loss)
Testing bandwidth between h1 and h2
*** Iperf: testing TCP bandwidth between h1 and h2
*** Results: ['5.89 Mbits/sec', '6.03 Mbits/sec']
Testing bandwidth between h1 and h3
*** Iperf: testing TCP bandwidth between h1 and h3
*** Results: ['4.97 Mbits/sec', '5.14 Mbits/sec']
Testing bandwidth between h2 and h3
*** Iperf: testing TCP bandwidth between h2 and h3
*** Results: ['1.97 Mbits/sec', '2.12 Mbits/sec']
```

The throughput now falls dramatically, far greater than the loss rate.

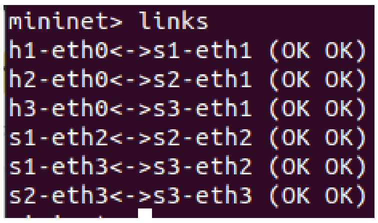
Q3

After adding another link between h1 and h2, the whole network is congested, h2 can't receive any of the packets h1 sends.



```
*** Starting CLI:
mininet> h1 ping h2
PING 10.0.0.2 (10.0.0.2) 56(84) bytes of data.
^C
--- 10.0.0.2 ping statistics ---
275 packets transmitted, 0 received, 100% packet loss, time 280561ms
```

And by using the command `links`, we can figure out how the network is like.



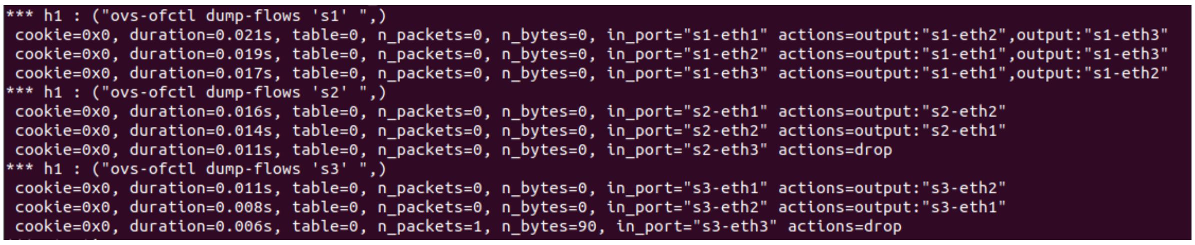
```
mininet> links
h1-eth0<->s1-eth1 (OK OK)
h2-eth0<->s2-eth1 (OK OK)
h3-eth0<->s3-eth1 (OK OK)
s1-eth2<->s2-eth2 (OK OK)
s1-eth3<->s3-eth2 (OK OK)
s2-eth3<->s3-eth3 (OK OK)
```

To avoid the network being congested, we just need to break the circle between swithers, which can be realized by breaking the last link in the belowing picture. We can use command `ovs-ofctl` to customize the network as we like. The following code can change the network into the one in **Q2**. This part is also implemented in `q3_2.py`.

```
> ovs-ofctl add-flow 's1' 'in_port=1 actions=output:2,3'
> ovs-ofctl add-flow 's1' 'in_port=2 actions=output:1,3'
> ovs-ofctl add-flow 's1' 'in_port=3 actions=output:1,2'
> ovs-ofctl add-flow 's2' 'in_port=1 actions=output:2'
> ovs-ofctl add-flow 's2' 'in_port=2 actions=output:1'
> ovs-ofctl add-flow 's2' 'in_port=3 actions=drop'
> ovs-ofctl add-flow 's3' 'in_port=1 actions=output:2'
> ovs-ofctl add-flow 's3' 'in_port=2 actions=output:1'
> ovs-ofctl add-flow 's3' 'in_port=3 actions=drop'
```

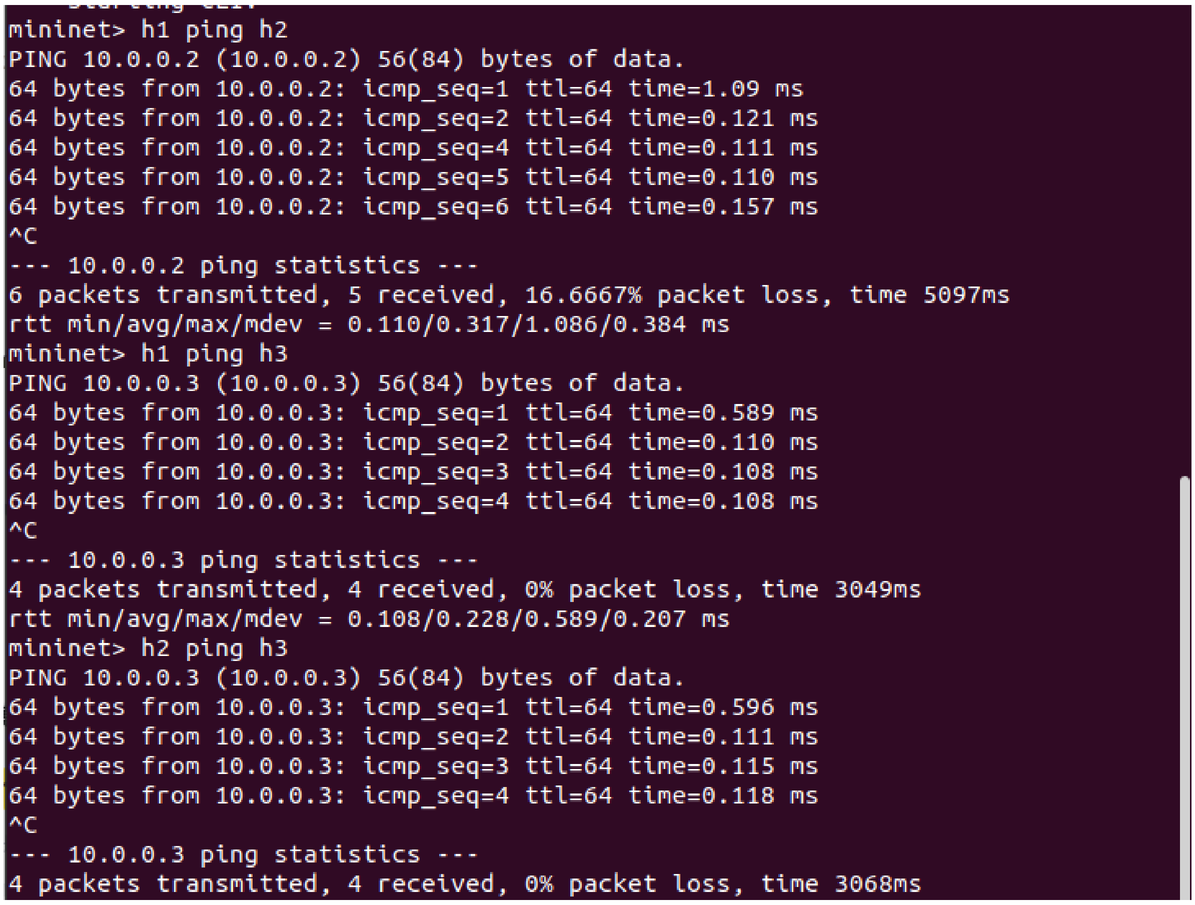
After this, we can use the command `ovs-ofctl dump-flows s1/s2/s3` to show the new flow rules.

The new rules are:



```
*** h1 : ("ovs-ofctl dump-flows 's1' ",)
 cookie=0x0, duration=0.021s, table=0, n_packets=0, n_bytes=0, in_port="s1-eth1" actions=output:"s1-eth2",output:"s1-eth3"
 cookie=0x0, duration=0.019s, table=0, n_packets=0, n_bytes=0, in_port="s1-eth2" actions=output:"s1-eth1",output:"s1-eth3"
 cookie=0x0, duration=0.017s, table=0, n_packets=0, n_bytes=0, in_port="s1-eth3" actions=output:"s1-eth1",output:"s1-eth2"
*** h1 : ("ovs-ofctl dump-flows 's2' ",)
 cookie=0x0, duration=0.016s, table=0, n_packets=0, n_bytes=0, in_port="s2-eth1" actions=output:"s2-eth2"
 cookie=0x0, duration=0.014s, table=0, n_packets=0, n_bytes=0, in_port="s2-eth2" actions=output:"s2-eth1"
 cookie=0x0, duration=0.011s, table=0, n_packets=0, n_bytes=0, in_port="s2-eth3" actions=drop
*** h1 : ("ovs-ofctl dump-flows 's3' ",)
 cookie=0x0, duration=0.011s, table=0, n_packets=0, n_bytes=0, in_port="s3-eth1" actions=output:"s3-eth2"
 cookie=0x0, duration=0.008s, table=0, n_packets=0, n_bytes=0, in_port="s3-eth2" actions=output:"s3-eth1"
 cookie=0x0, duration=0.006s, table=0, n_packets=1, n_bytes=90, in_port="s3-eth3" actions=drop
```

Now this network can work samely as this in **Q2**, we can also testify it by using `ping` command:



```
mininet> h1 ping h2
PING 10.0.0.2 (10.0.0.2) 56(84) bytes of data.
64 bytes from 10.0.0.2: icmp_seq=1 ttl=64 time=1.09 ms
64 bytes from 10.0.0.2: icmp_seq=2 ttl=64 time=0.121 ms
64 bytes from 10.0.0.2: icmp_seq=4 ttl=64 time=0.111 ms
64 bytes from 10.0.0.2: icmp_seq=5 ttl=64 time=0.110 ms
64 bytes from 10.0.0.2: icmp_seq=6 ttl=64 time=0.157 ms
^C
--- 10.0.0.2 ping statistics ---
6 packets transmitted, 5 received, 16.6667% packet loss, time 5097ms
rtt min/avg/max/mdev = 0.110/0.317/1.086/0.384 ms
mininet> h1 ping h3
PING 10.0.0.3 (10.0.0.3) 56(84) bytes of data.
64 bytes from 10.0.0.3: icmp_seq=1 ttl=64 time=0.589 ms
64 bytes from 10.0.0.3: icmp_seq=2 ttl=64 time=0.110 ms
64 bytes from 10.0.0.3: icmp_seq=3 ttl=64 time=0.108 ms
64 bytes from 10.0.0.3: icmp_seq=4 ttl=64 time=0.108 ms
^C
--- 10.0.0.3 ping statistics ---
4 packets transmitted, 4 received, 0% packet loss, time 3049ms
rtt min/avg/max/mdev = 0.108/0.228/0.589/0.207 ms
mininet> h2 ping h3
PING 10.0.0.3 (10.0.0.3) 56(84) bytes of data.
64 bytes from 10.0.0.3: icmp_seq=1 ttl=64 time=0.596 ms
64 bytes from 10.0.0.3: icmp_seq=2 ttl=64 time=0.111 ms
64 bytes from 10.0.0.3: icmp_seq=3 ttl=64 time=0.115 ms
64 bytes from 10.0.0.3: icmp_seq=4 ttl=64 time=0.118 ms
^C
--- 10.0.0.3 ping statistics ---
4 packets transmitted, 4 received, 0% packet loss, time 3068ms
```