Item #1:

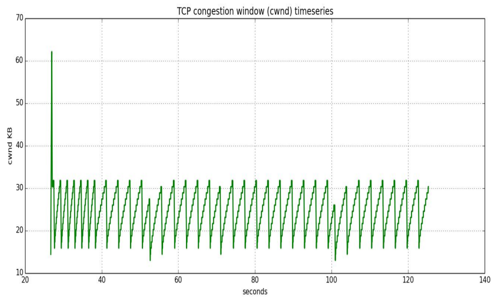
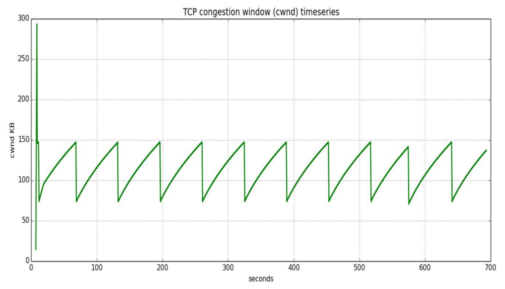
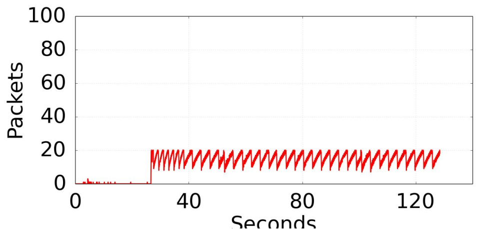
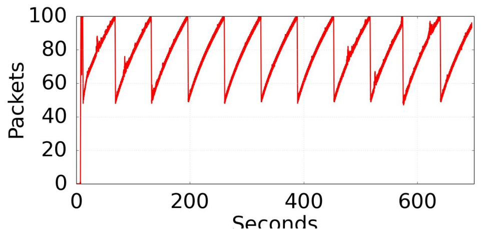
Both graphs will experience a period of slow start in the form of exponential growth and then the cyclical movement between congestion window increase and decrease.
The difference is that for TCP Reco, there will be linear increase and once the max congestion window is reached, the window is reset to 0.5 * max congestion window where packet loss happened (AIMD). But for CUBIC, it will be the cubic curve where starts with concave increase and once the last time max window is reached, max probing starts in the form of convex increase. After a packet loss, the window size is reset to (1-beta) * packet loss window size, where beta are usually set to < 0.5 to enable slower convergence.

Item #2:

According to Part 3.4 to Part 3.6 in the paper, the curve has a concave region at steady state and then a convex region for max probing, and then experience a multiplicative decrease by reducing the window size by a factor of beta. So in the form of the curve, CUBIC would show cyclic cubic ones each going through the 3 parts of:

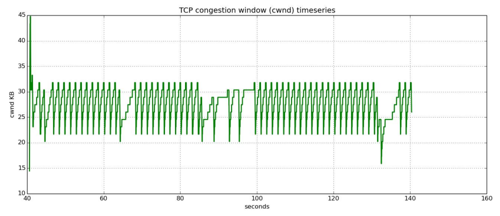
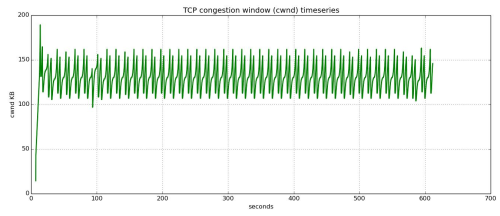
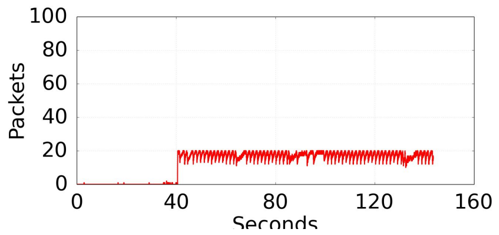
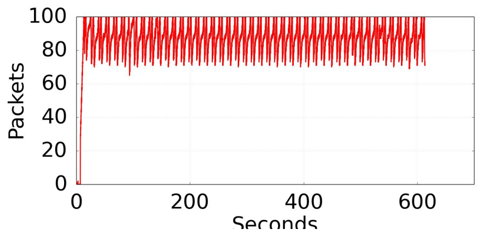
1. Concave region when cwnd <= Wmax
2. Convex region when cwnd > Wmax
3. Multiplicative decrease when cwnd value finally leads to a packet loss.

Item #3:

Small Queue	Large Queue
Iperf 	iperf 
Queue 	Queue 

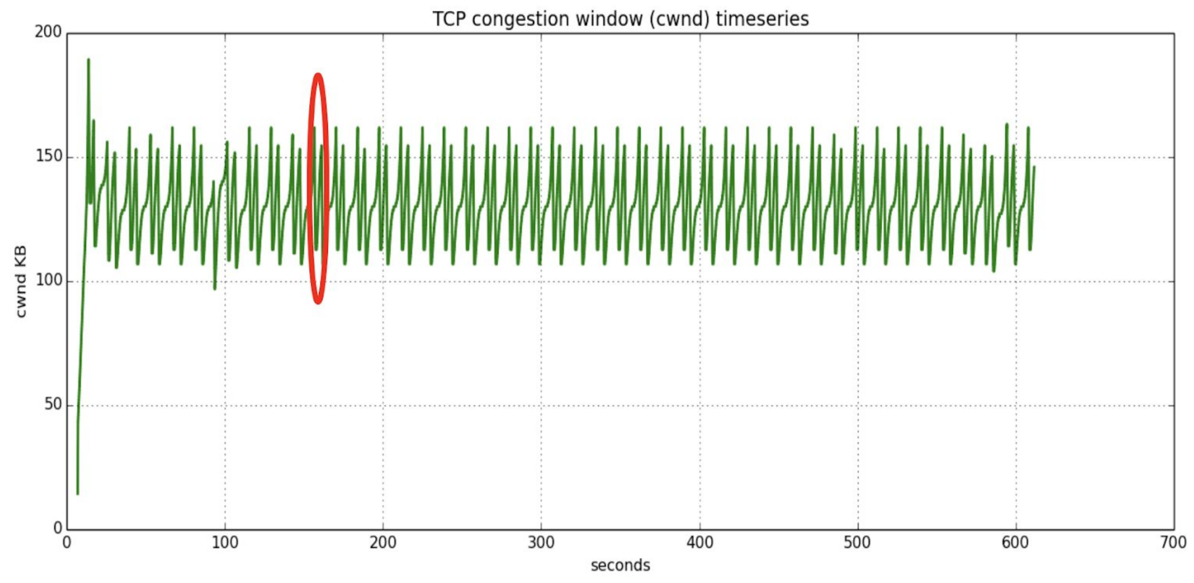
The small queue congestion window ranges from 15 KB to 30 KB. Buffer packet occupancy ranges from 10 to 20.
The large queue congestion window ranges from 75 KB to 150 KB. Buffer packet occupancy ranges from 50 to 100.
Time for large queue sender to increase its congestion window to max = 50 * large queue RTT = 75 seconds
Time for small queue sender to increase its congestion window to max = 15 * small queue RTT = 3 seconds
=> RTT for the large queue is longer and it meets the expectation that the bigger the buffer size the higher the latency. This is because the larger the buffer size, more packets are accumulated in the buffer when the buffer is full and packets need to wait longer to be sent out.

Item #4:

Small Queue	Large Queue
Iperf 	iperf 
Queue 	Queue 

I can see that for the small queue, the window curve starts to show the trend of concave then convex but not quite cubic or smooth, probably a reason of the low sampling rate. And for large queue, the curve shows the cubic trend.

Unexpected result:
For the large queue, the cubic trend is only shown every other time. There exists this kind of cyclic pair: a full cubic curve (by full cubic I mean the concave + convex curve pattern) and a part cubic curve (by part cubic I mean the part where it only shows the concave part before reaching Wmax). One of the pair is circled in red as below.



According to the paper: In part 3.7 of the paper it says: *In fast convergence, at a loss event, if the current value of Wmax is less than the last value of it, Wlast max, this indicates that the saturation point experienced by this flow is getting reduced because of the change in available bandwidth. Then we allow this flow to release more bandwidth by reducing Wmax further.* The reason is that every other time we see Wlast_max < cwnd and fast convergence is triggered. The Wmax to be used for getting the next cubic curve is even further reduced. It doesn't happen for the small queue because the small queue doesn't have enough buffer and the cwnd wouldn't grow as fast as the large queue at its peak cwnd.

Item #5:

The prediction in Item #1 matched most of the results other than the fast convergence covered in Item #4. The small queue size better matched my predictions. This is because I only considered it theoretical and didn't think about how external environment would influence the network. The existence of a router connecting the server and the end houst and the size of the buffer also have impact on the cwnd adjustment process.

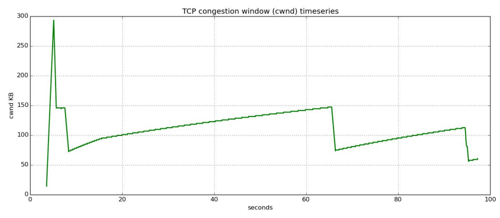
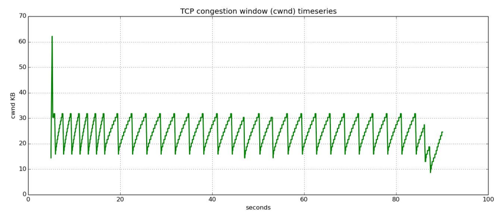
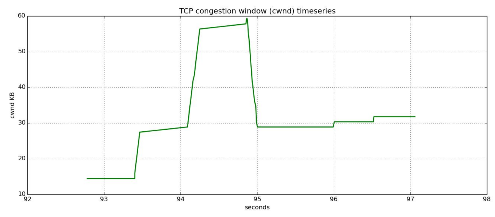
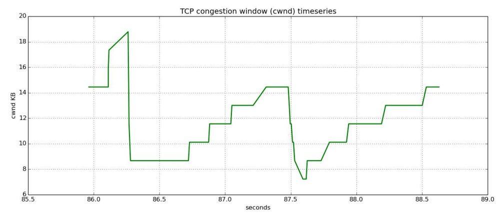
Item #6:

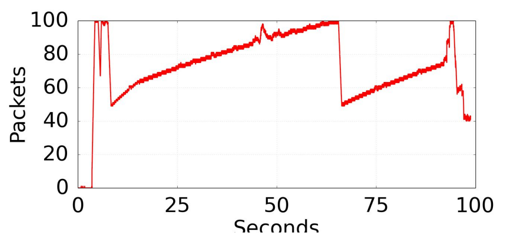
Before the existence of a long lived flow the avg RTT is 20.38 ms. Takes 1 second to download the page.
The recorded throughput tends to stable at 1.41 Mbits/sec, which is pretty close to that of h2's link rate.
After the long lived flow started: avg RTT is 726.007 ms. Takes 7.8 seconds to download.
With the long lived flow running in the background, the router buffer is full with the packets to be sent from the streaming video sender.

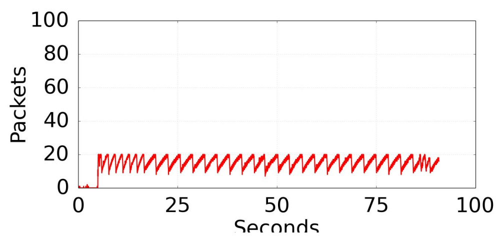
To evaluate RTT, we let h1 ping h2, which tracks the between that h1 sending a packet to h2 and getting the response. As the many packets in the buffer are queued already given h1 keep sending at a higher rate before packet loss (compared to the case without the flow, the buffer has few packets queued), the ping packet needs to wait before it can be sent and the same thing for the case the response being sent from h2 to h1. That's why the RTT increased. And for the download time is the same thing. It's like h2 sends HTTP GET request and wait for the response from h1. Download time also increased due to latency increase.

Item #7:

Small queue performs better. It took 3.7 seconds for the larger queue and 2.0 seconds for the small queue to download can be seen from the console output and wget graphs.
<u>Reason</u>: The small queue occupancy, as can be seen from the graph, always fills up quickly and the sender knows to slow down the sending rate. The large queue however, can buffer many packets so the sender's sending rate can go really high before it realized the buffer is full through a packet loss. As a result, for the large queue, there are more packets in the buffer, packets added to the queue (which can just be the web page download HTTP response packets) need to wait until all the packets already in queue to be cleared out of the buffer and it takes a longer time, which means higher latency.

Experiment 1 (Large queue)	Experiment 2 (Small queue)
iperf 	iperf 
wget 	wget 
queue	queue





Item #8:

When using two queues, the long lived flow still increased the download time. But not as much as in using 1 queue for both flows. When using two queues the RTT stays the same in the presence of the long lived flow.

At the beginning, the network RTT is about 20ms. And the download took 1 second. After starting the long live flow, the RTT is still about 20ms and the download time is 2 seconds.

The result is as expected. Because when using two queues, as described, the scheduler implements fair queueing so that when both queues are busy, each flow will receive half of the bottleneck link rate. For the download time, the long lived flow queue can be expected to be always busy. The bandwidth available to the download flow (the short flow) is half of that before starting the long flow. That's why the download time doubled. For the RTT, it's measured by time taken for h1 to ping h2, since during each ping the packet size is so small and the h2's link rate doesn't become the bottleneck, also the ping can still have its own queue without having to stay in the buffer so RTT doesn't change.

Item #9:

Repeating Part 3.
With cubic, before starting the long lived flow the average RTT is 20.23ms and the download time is 1s. Throughput doesn't change, also tends to be around 1.4Mbt/s, close to h2 link rate. After starting the long lived flow, the average RTT is 723.03ms and the download time is 8.3s. There **seems to be no major change from TCP Reno before the long lived flow started**, it's expected because the bandwidth doesn't change so RTT and download time doesn't change.

Repeating Part 4.
Experiment 3 download time is 7.6s and experiment 4 download time is 2.6s. Actually After repeating the 4 experiments a few more times, seems that there's a range for the download time. For example, experiment 1: 3.7s ~ 5.5s, Experiment 3: 4.4s ~ 10s.

So **CUBIC algorithm slightly increases the download time compared to TCP Reno** given the same queue size. What is similar between the two algos are the trend: smaller queue size reduces the download time.

TCP Reno actually performed better. **Reason assumption**: TCP Reno actually has lower bandwidth utilization than TCP CUBIC, especially when the flow length (the download flow for example) is much smaller than the time TCP Reno grows its bandwidth to the maximum. So TCP Reno is not as aggressive as TCP CUBIC at the rate of increasing its window size. So the bufferbloat problem caused by the larger sending rate turns out to be alleviated when applying TCP Reno.

Appendix

Item 6 Console output

Before:

Wget:

```
100%[=====================================>] 177,669      175KB/s   in 1.0s

mininet> h1 ping -c 10 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=21.0 ms
64 bytes from 10.0.0.2: icmp_seq=2 ttl=64 time=21.8 ms
64 bytes from 10.0.0.2: icmp_seq=3 ttl=64 time=20.5 ms
64 bytes from 10.0.0.2: icmp_seq=4 ttl=64 time=21.2 ms
64 bytes from 10.0.0.2: icmp_seq=5 ttl=64 time=21.7 ms
64 bytes from 10.0.0.2: icmp_seq=6 ttl=64 time=20.3 ms
64 bytes from 10.0.0.2: icmp_seq=7 ttl=64 time=21.3 ms
64 bytes from 10.0.0.2: icmp_seq=8 ttl=64 time=20.4 ms
64 bytes from 10.0.0.2: icmp_seq=9 ttl=64 time=20.8 ms
64 bytes from 10.0.0.2: icmp_seq=10 ttl=64 time=21.0 ms

--- 10.0.0.2 ping statistics ---
10 packets transmitted, 10 received, 0% packet loss, time 9011ms
rtt min/avg/max/mdev = 20.380/21.050/21.845/0.521 ms
```

After the long lived flow started:

```
mininet> h1 ping -c 10 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=726 ms
64 bytes from 10.0.0.2: icmp_seq=2 ttl=64 time=727 ms
64 bytes from 10.0.0.2: icmp_seq=3 ttl=64 time=738 ms
64 bytes from 10.0.0.2: icmp_seq=4 ttl=64 time=747 ms
64 bytes from 10.0.0.2: icmp_seq=5 ttl=64 time=749 ms
64 bytes from 10.0.0.2: icmp_seq=6 ttl=64 time=743 ms
```

```
64 bytes from 10.0.0.2: icmp_seq=7 ttl=64 time=752 ms
64 bytes from 10.0.0.2: icmp_seq=8 ttl=64 time=755 ms
64 bytes from 10.0.0.2: icmp_seq=9 ttl=64 time=764 ms
64 bytes from 10.0.0.2: icmp_seq=10 ttl=64 time=774 ms

--- 10.0.0.2 ping statistics ---
10 packets transmitted, 10 received, 0% packet loss, time 9000ms
rtt min/avg/max/mdev = 726.007/747.850/774.877/14.560 ms
```

Wget:

```
100%[=======================================>] 177,669     22.3KB/s   in 7.8s
```

Item 7 Console output

Larger queue took 3.7 seconds:

```
100%[=======================================>] 177,669     47.5KB/s   in 3.7s
```

Small queue took 2 seconds:

```
100%[=======================================>] 177,669     87.3KB/s   in 2.0s
```

Item 8 Console output

Before:

```
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=20.5 ms
64 bytes from 10.0.0.2: icmp_seq=2 ttl=64 time=21.1 ms
64 bytes from 10.0.0.2: icmp_seq=3 ttl=64 time=20.2 ms
64 bytes from 10.0.0.2: icmp_seq=4 ttl=64 time=20.3 ms
64 bytes from 10.0.0.2: icmp_seq=5 ttl=64 time=20.7 ms
64 bytes from 10.0.0.2: icmp_seq=6 ttl=64 time=21.9 ms
64 bytes from 10.0.0.2: icmp_seq=7 ttl=64 time=22.0 ms
64 bytes from 10.0.0.2: icmp_seq=8 ttl=64 time=20.3 ms
64 bytes from 10.0.0.2: icmp_seq=9 ttl=64 time=20.8 ms
64 bytes from 10.0.0.2: icmp_seq=10 ttl=64 time=20.2 ms
--- 10.0.0.2 ping statistics ---
10 packets transmitted, 10 received, 0% packet loss, time 9014ms
rtt min/avg/max/mdev = 20.264/20.862/22.064/0.641 ms
```

Wget:

```
100%[=======================================>] 177,669      175KB/s   in 1.0s
```

After:

```
mininet> h1 ping -c 10 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=20.3 ms
```

```
64 bytes from 10.0.0.2: icmp_seq=2 ttl=64 time=20.6 ms
64 bytes from 10.0.0.2: icmp_seq=3 ttl=64 time=21.0 ms
64 bytes from 10.0.0.2: icmp_seq=4 ttl=64 time=20.6 ms
64 bytes from 10.0.0.2: icmp_seq=5 ttl=64 time=21.2 ms
64 bytes from 10.0.0.2: icmp_seq=6 ttl=64 time=20.4 ms
64 bytes from 10.0.0.2: icmp_seq=7 ttl=64 time=21.1 ms
64 bytes from 10.0.0.2: icmp_seq=8 ttl=64 time=20.4 ms
64 bytes from 10.0.0.2: icmp_seq=9 ttl=64 time=20.3 ms
64 bytes from 10.0.0.2: icmp_seq=10 ttl=64 time=20.3 ms
--- 10.0.0.2 ping statistics ---
10 packets transmitted, 10 received, 0% packet loss, time 9013ms
rtt min/avg/max/mdev = 20.331/20.663/21.264/0.376 ms
Wget:
100%[=======================================>] 177,669     87.3KB/s   in 2.0s
```

Item 9 Console output

Repeating part 3:
Before:

```
mininet> h2 wget http://10.0.0.1
--2018-10-27 03:46:42--  http://10.0.0.1/
Connecting to 10.0.0.1:80... connected.
HTTP request sent, awaiting response... 200 OK
Length: 177669 (174K) [text/html]
Saving to: ‘index.html’

100%[=======================================>] 177,669      173KB/s   in 1.0s

2018-10-27 03:46:43 (173 KB/s) - ‘index.html’ saved [177669/177669]

mininet> h1 ping -c 10 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=20.6 ms
64 bytes from 10.0.0.2: icmp_seq=2 ttl=64 time=20.3 ms
64 bytes from 10.0.0.2: icmp_seq=3 ttl=64 time=20.4 ms
64 bytes from 10.0.0.2: icmp_seq=4 ttl=64 time=22.0 ms
64 bytes from 10.0.0.2: icmp_seq=5 ttl=64 time=20.3 ms
64 bytes from 10.0.0.2: icmp_seq=6 ttl=64 time=21.1 ms
64 bytes from 10.0.0.2: icmp_seq=7 ttl=64 time=21.3 ms
64 bytes from 10.0.0.2: icmp_seq=8 ttl=64 time=20.2 ms
64 bytes from 10.0.0.2: icmp_seq=9 ttl=64 time=20.3 ms
```

```
64 bytes from 10.0.0.2: icmp_seq=10 ttl=64 time=20.6 ms

--- 10.0.0.2 ping statistics ---
10 packets transmitted, 10 received, 0% packet loss, time 9012ms
rtt min/avg/max/mdev = 20.204/20.742/22.041/0.554 ms
```

After:

```
mininet> h1 ping -c 10 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=735 ms
64 bytes from 10.0.0.2: icmp_seq=2 ttl=64 time=736 ms
64 bytes from 10.0.0.2: icmp_seq=3 ttl=64 time=777 ms
64 bytes from 10.0.0.2: icmp_seq=4 ttl=64 time=813 ms
64 bytes from 10.0.0.2: icmp_seq=5 ttl=64 time=612 ms
64 bytes from 10.0.0.2: icmp_seq=6 ttl=64 time=652 ms
64 bytes from 10.0.0.2: icmp_seq=7 ttl=64 time=735 ms
64 bytes from 10.0.0.2: icmp_seq=8 ttl=64 time=810 ms
64 bytes from 10.0.0.2: icmp_seq=9 ttl=64 time=755 ms
64 bytes from 10.0.0.2: icmp_seq=10 ttl=64 time=599 ms

--- 10.0.0.2 ping statistics ---
10 packets transmitted, 10 received, 0% packet loss, time 9001ms
rtt min/avg/max/mdev = 599.555/723.032/813.510/72.616 ms


mininet> h2 wget http://10.0.0.1
--2018-10-27 03:49:03--  http://10.0.0.1/
Connecting to 10.0.0.1:80... connected.
HTTP request sent, awaiting response... 200 OK
Length: 177669 (174K) [text/html]
Saving to: ‘index.html.3’

100%[=======================================>] 177,669     20.9KB/s   in 8.3s

2018-10-27 03:49:13 (20.9 KB/s) - ‘index.html.3’ saved [177669/177669]
```

Repeating Part 4:

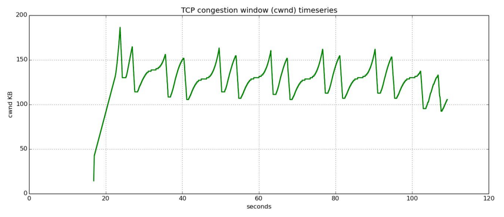
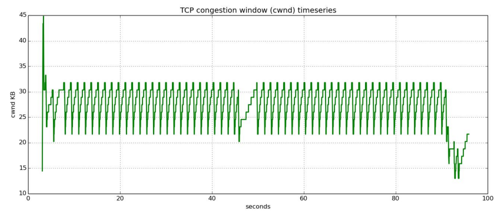
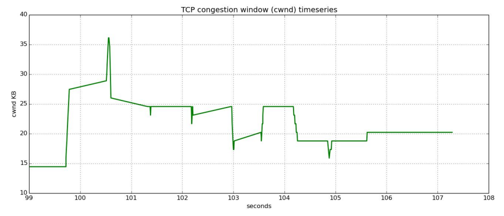
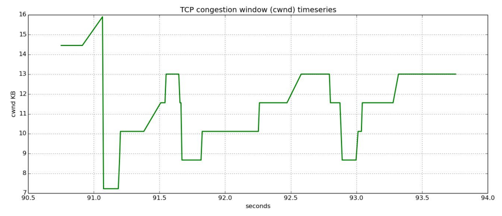
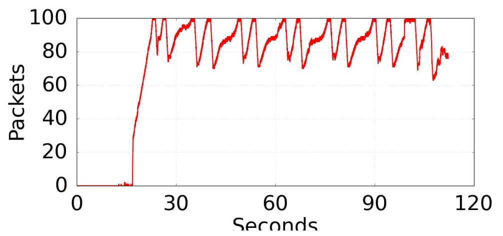
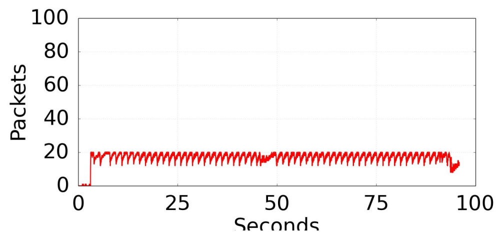
Download time for experiment 3 is 7.6 s (large queue size)

```
100%[=======================================>] 177,669     22.9KB/s   in 7.6s
```

Download time for experiment 4 is 2.6 s (small queue size)

```
100%[=======================================>] 177,669     67.5KB/s   in 2.6s
```

Graphs

Experiment 3 (Large queue)	Experiment 4 (Small queue)
iperf 	iperf 
wget 	wget 
queue 	queue  Packets Seconds