Project 3: Congestion Control & Buffer Bloat

Vincent La
Georgia Tech CS 6250

Before the Experiment

Item #1: In this network, what do you expect the CWND (congestion window) curve for a long lived TCP flow using TCP CUBIC to look like? Contrast it to what you expect the CWND for TCP Reno will look like.

The CWND curve for a long lived TCP flow using TCP CUBIC will have both concave and convex regions as suggested by sections 3.4 and 3.5 in the reading.

The CWND curve for TCP Reno will look like a curve with an exponential slow start phase. However, it will eventually only increase linearly until it reaches W_{max}. Upon reaching W_{max}, we will observe a decrease.

Item #2: Explain why you expect to see this; specifically relate your answers to details in the paper. (We expect you to demonstrate the ability to apply what you have already learned from the paper in your answer to this question, and that you've furthermore thought about what you read in order to make a prediction. So justifying your prediction is important. — It's okay if your prediction ends up being wrong as long as it was based on an understanding and analysis of the paper!)

According to the reading CUBIC: A New TCP-Friendly High-Speed TCP Variant (Equation 1 and Equation 2) we know that:

- Window Growth Function of CUBIC: $W(t) = C(t - K)^3 + W_{max}$

Where:

- C is a CUBIC parameter,
- t is the elapsed time from the last window reduction,
- and K is the time period that the above function takes to increase W to Wmax when there is no further loss event

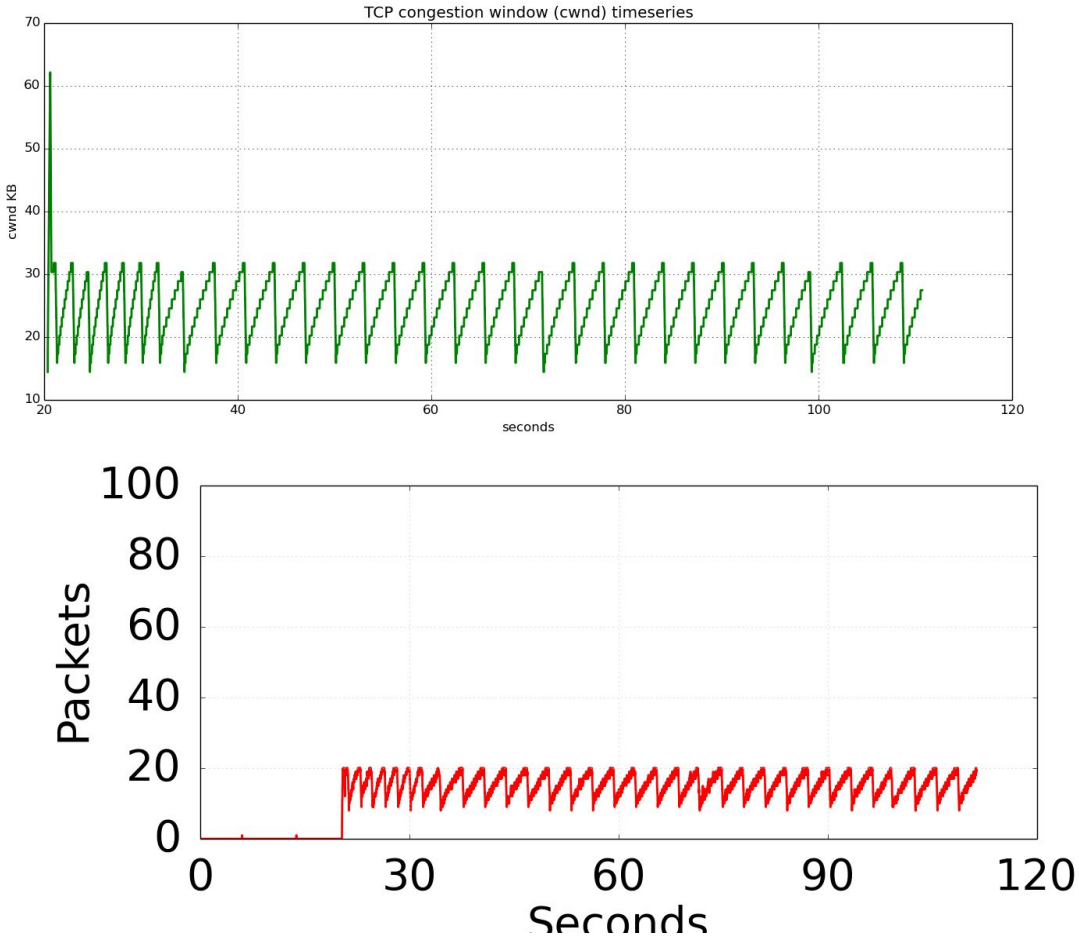
We know that K is calculated by using the following equation: $K = (W_{max}\beta/C)^{1/3}$

Putting this all together, for the CWND curve for a long lived TCP flow using TCP CUBIC, we know that after a loss event W_{max} is set to be where the loss event occurred. In the congestion avoidance phase, it grows in a *concave* fashion until W_{max} is reached.
However, upon reaching W_{max}, we reach the convex phase.

Part 1: TCP Reno iperf

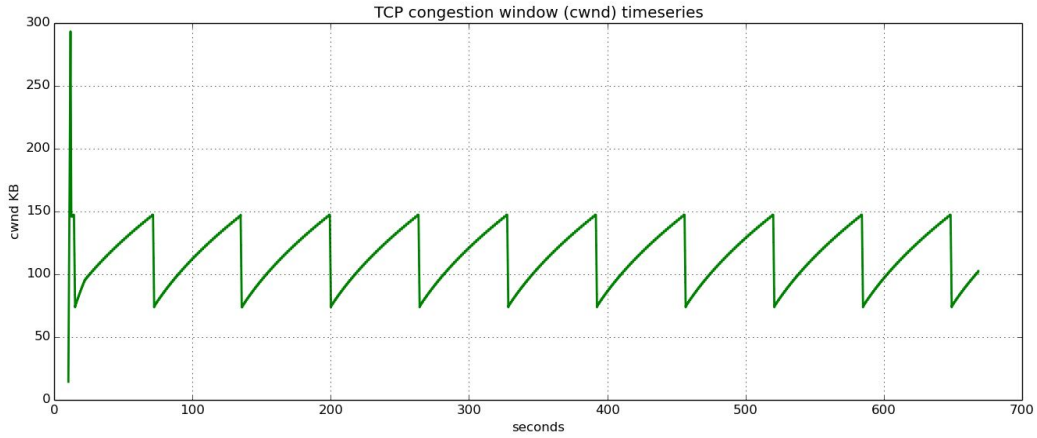
Item #3: With respect to queue occupancy, how are the graphs different? What do you think causes these differences to occur? (Make sure to include the graphs with your answer).

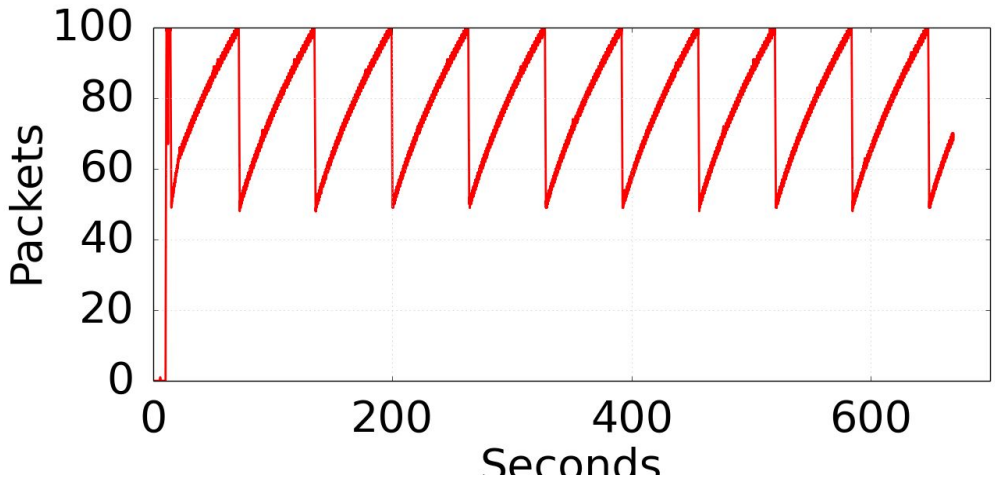
First, let's look at small-queue_tcp_cwnd_iperf.png and small-queue_queue.png:



For this small queue graph, notice a higher frequency of the congestion window moving up and down rapidly.

However, looking at corresponding images for the large-queue graphs:





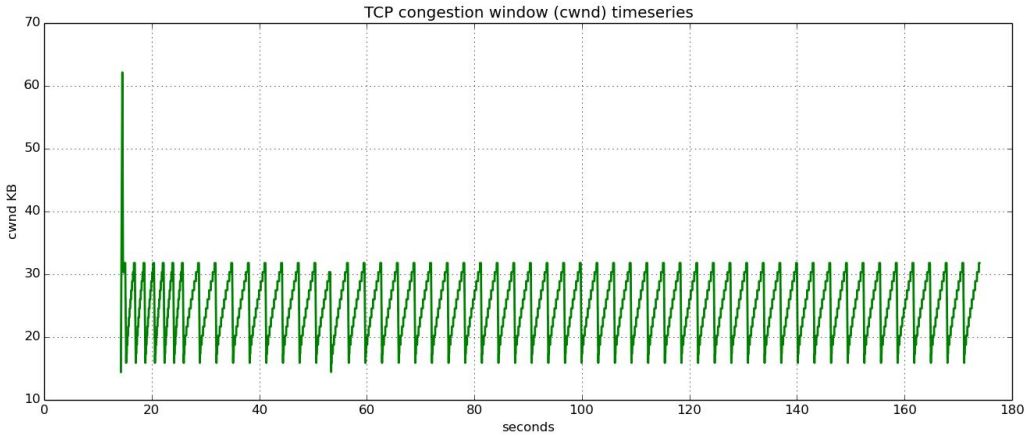
We see that the congestion window grows at a much slower pace.

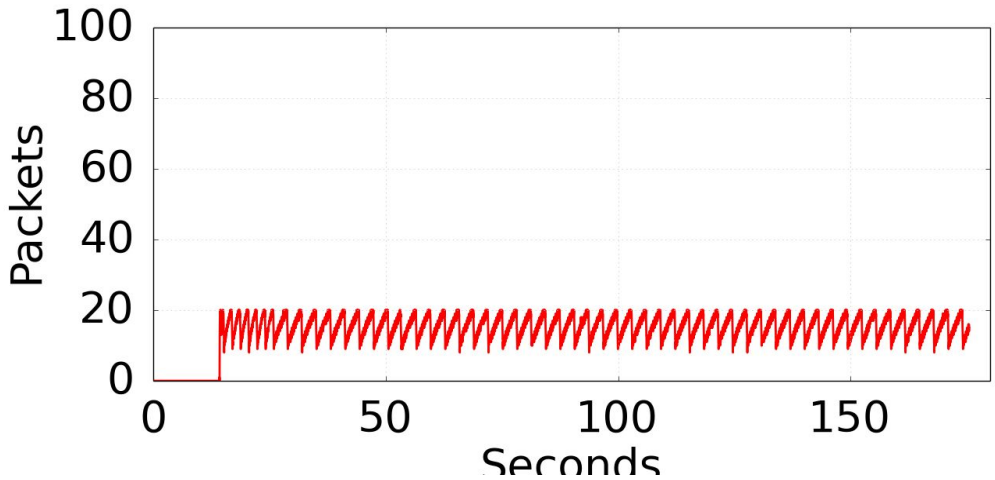
I believe that what causes these differences to occur is that it takes longer time for the packets to be cleared in the larger queue as compared to the smaller queue.

Part 2: TCP CUBIC iperf

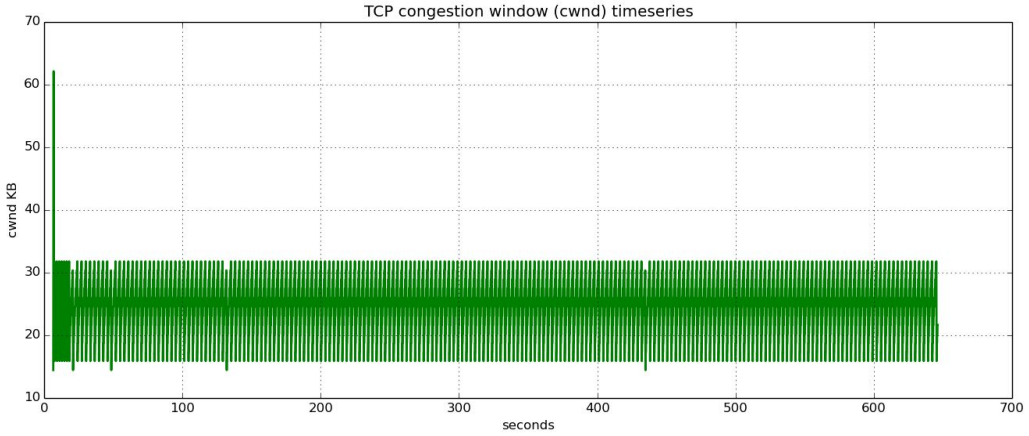
Item #4: What did you actually see in your CUBIC results? What anomalies or unexpected results did you see, and why do you think these behaviors/anomalies occurred? Be sure to reference the paper you read at the beginning of the project to help explain something that you saw in your results. Your hypothesis doesn't necessarily have to be correct, so long as it demonstrates understanding of the paper.

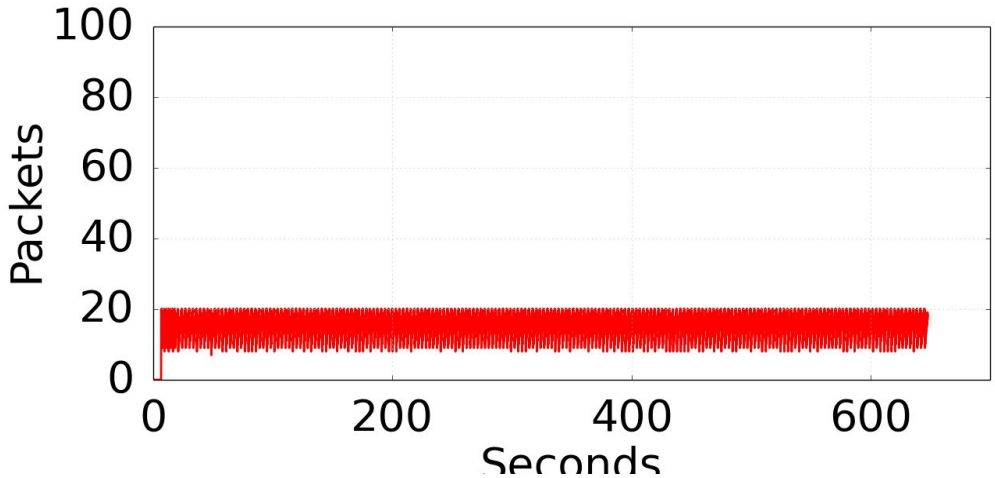
Again, let's look at small-queue-cubic_tcp_cwnd_iperf.png and small-queue-cubic_queue.png:





Furthermore, let's look at the corresponding graphics for the large queue





I expected CWND to grow with both concave and convex regions from the start. However, it started growing linearly till the first packet loss occurred. Furthermore, I expected the CWND curve to be concave only if bandwidth remains constant, but for the large queue I observed something different.

Item #5: Compare your prediction from Item #1 with the congestion window graphs you generated for TCP Reno and TCP CUBIC. How well did your predictions match up with the actual results? If they were different, what do you think caused the differences? Which queue size better matched your predictions, small or large?

My predictions mostly matched with the actual results! The TCP CUBIC exhibited both concave and convex regions. For TCP Reno I saw spiky graph as I predicted.

Part 3: Resource Contention

Item #6: How does the presence of a long lived flow on the link affect the download time and network latency (RTT)? What factors do you think are at play causing this? Be sure to include the RTT and download times you recorded throughout Part 3.

```
RTT Baseline Measurements
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=21.4 ms
64 bytes from 10.0.0.2: icmp_seq=2 ttl=64 time=20.4 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=21.5 ms
64 bytes from 10.0.0.2: icmp_seq=5 ttl=64 time=20.2 ms
64 bytes from 10.0.0.2: icmp_seq=6 ttl=64 time=20.1 ms
64 bytes from 10.0.0.2: icmp_seq=7 ttl=64 time=20.2 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=20.2 ms
64 bytes from 10.0.0.2: icmp_seq=10 ttl=64 time=20.2 ms
```

Long Flow Network Latency and RTT

```
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=761 ms
64 bytes from 10.0.0.2: icmp_seq=2 ttl=64 time=762 ms
64 bytes from 10.0.0.2: icmp_seq=3 ttl=64 time=773 ms
64 bytes from 10.0.0.2: icmp_seq=4 ttl=64 time=783 ms
64 bytes from 10.0.0.2: icmp_seq=5 ttl=64 time=769 ms
64 bytes from 10.0.0.2: icmp_seq=6 ttl=64 time=778 ms
64 bytes from 10.0.0.2: icmp_seq=7 ttl=64 time=781 ms
64 bytes from 10.0.0.2: icmp_seq=8 ttl=64 time=790 ms
64 bytes from 10.0.0.2: icmp_seq=9 ttl=64 time=800 ms
64 bytes from 10.0.0.2: icmp_seq=10 ttl=64 time=801 ms
```

Finally, we show the time required to download the file:

```
mininet> h2 wget http://10.0.0.1
--2019-06-21 03:53:40--  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.1’

100%[======================================>] 177,669     24.7KB/s
in 7.0s

2019-06-21 03:53:49 (24.7 KB/s) - ‘index.html.1’ saved
[177669/177669]
```

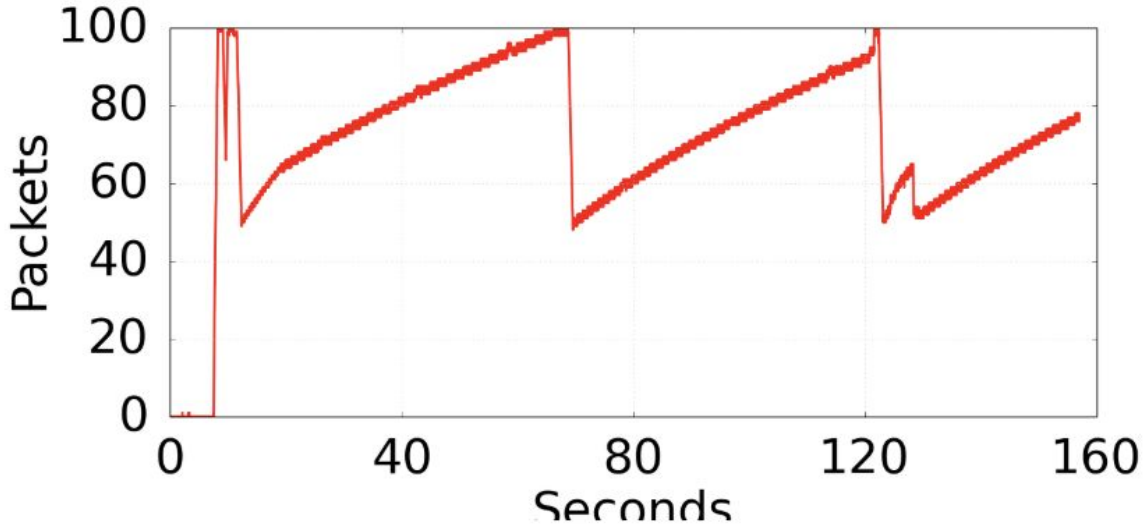
As we can see, long lived flow increased the download time, and network latency (in the baseline we observed ~20ms, afterwards we saw ~800ms). This is because after, there are a lot of packets placed into the queue. These packets from the short flow must be put into the back. Since the bottleneck queue is constant, the packets need to be drained before wget packets. This increases download time and RTT is higher.

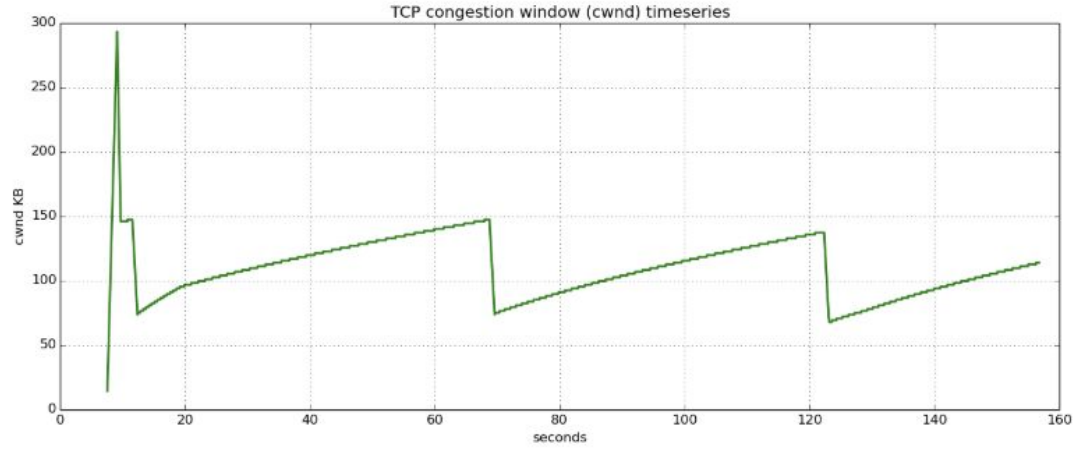
Part 4: Resource Contention (continued)

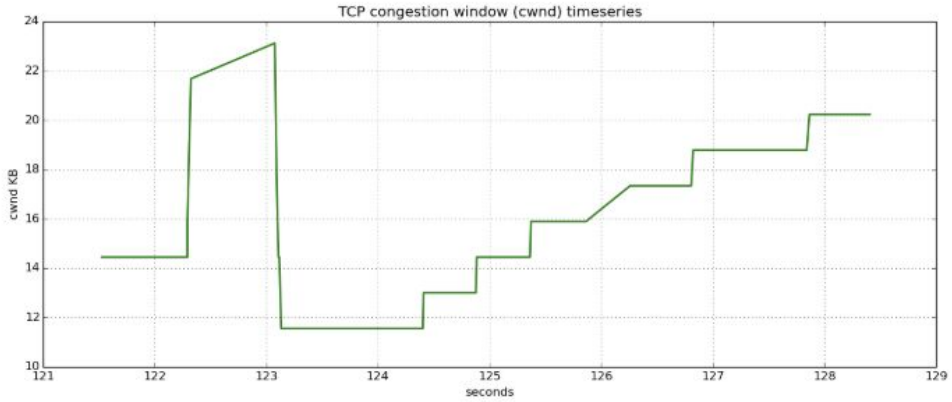
Item #7: How does the performance of the download differ with a smaller queue versus a larger queue? Why does reducing the queue size reduce the download time for wget? Be sure to include any graphs that support your reasoning.

Experiment 1 (Small Queue)

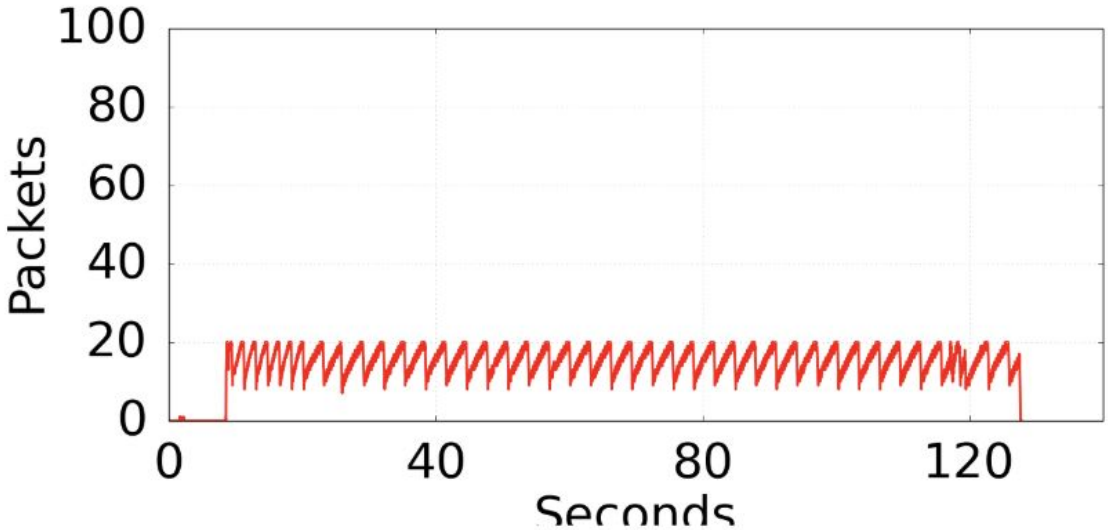
Let’s first take a look at the graphs for Experiment 1:

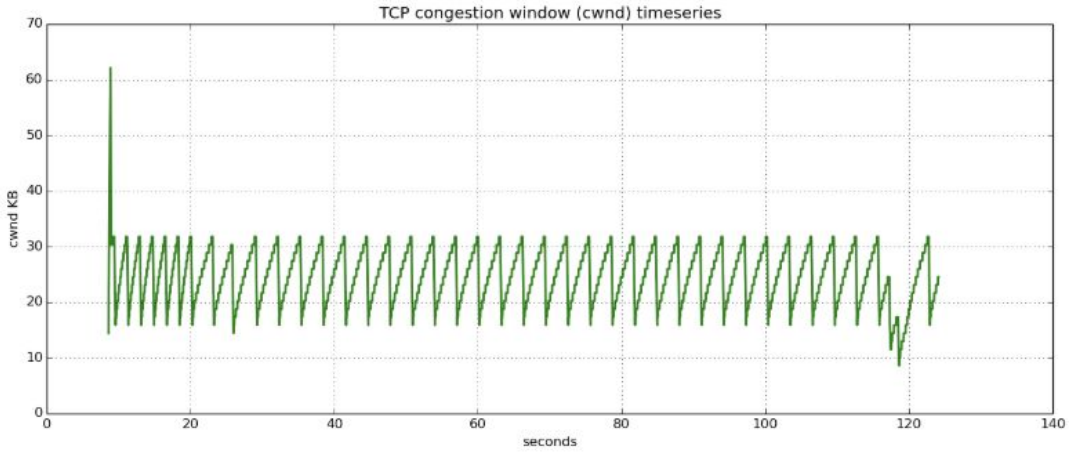

Packets
100
80
60
40
20
0
0
40
80
120
160
Seconds

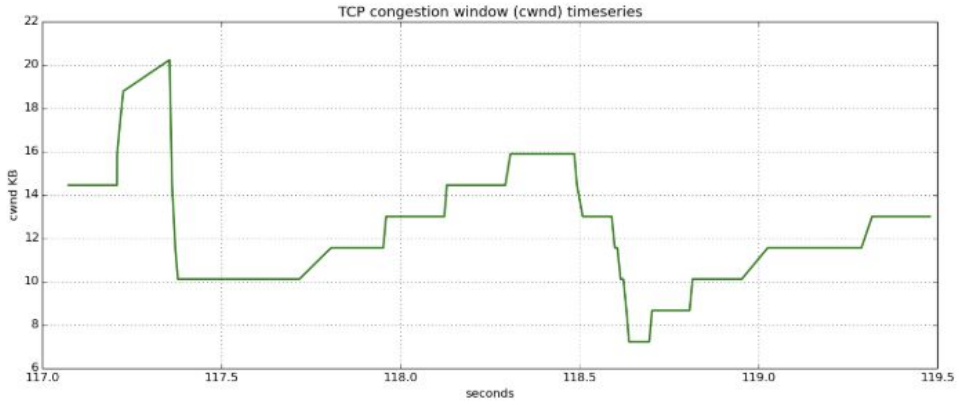

TCP congestion window (cwnd) timeseries
cwnd KB
300
250
200
150
100
50
0
0
20
40
60
80
100
120
140
160
seconds


TCP congestion window (cwnd) timeseries
cwnd KB
24
22
20
18
16
14
12
10
121
122
123
124
125
126
127
128
129
seconds

Experiment 2 (Large Queue):







Based on these graphs, we see that the Experiment 1 (small queue) performed better than large queue. Let's first look at the wget graph for the large queue:

There are delays; the queue is filled with iperf packets. However, for the small queue wget graph, cwnd to increase is much shorter time. The large queue graph there seems to be potentially weird behavior that may related to buffer bloat. Experiment 1 small queue download time was much lower. Reducing queue time reduces download because the rate at which queue is cleared is independent. When packets of the webpage are queued at the end of the larger queue, it takes longer for previous packets to be cleared.

Part 5: Using Traffic Control to Prevent Buffer Bloat

Item #8: How does the presence of a long lived flow on the link affect the download time and network latency (RTT) when using two queues? Were the results as you expected? Be sure to include the RTT and download times you recorded throughout Part 5.

First, we show some output. The first shown here is the network latency (average RTT) before starting the long lived flow:

```
mininet> h1 ping -c 10 h2
nohup: appending output to 'nohup.out'
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=20.4 ms
64 bytes from 10.0.0.2: icmp_seq=3 ttl=64 time=20.1 ms
64 bytes from 10.0.0.2: icmp_seq=4 ttl=64 time=21.7 ms
64 bytes from 10.0.0.2: icmp_seq=5 ttl=64 time=20.8 ms
64 bytes from 10.0.0.2: icmp_seq=6 ttl=64 time=22.1 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=20.4 ms
64 bytes from 10.0.0.2: icmp_seq=10 ttl=64 time=20.1 ms

--- 10.0.0.2 ping statistics ---
10 packets transmitted, 10 received, 0% packet loss, time 9010ms
rtt min/avg/max/mdev = 20.177/20.819/22.138/0.671 ms
```

Next, we record the time required in seconds to download file before the long lived flow:

```
mininet> h2 wget http://10.0.0.1
--2019-06-21 04:18:59--  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      175KB/s
in 1.0s

2019-06-21 04:19:00 (175 KB/s) - 'index.html' saved [177669/177669]
```

Now, we start the long lived flow. Next we record the network latency (average RTT) after starting the long lived flow:

```
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=22.7 ms
64 bytes from 10.0.0.2: icmp_seq=2 ttl=64 time=22.6 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=20.4 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=20.3 ms
64 bytes from 10.0.0.2: icmp_seq=7 ttl=64 time=20.2 ms
64 bytes from 10.0.0.2: icmp_seq=8 ttl=64 time=20.1 ms
64 bytes from 10.0.0.2: icmp_seq=9 ttl=64 time=20.7 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 9011ms
rtt min/avg/max/mdev = 20.199/20.825/22.743/0.964 ms
```

Next, we record the time required to download the file in the presence of the long lived flow:

```
mininet> h2 wget http://10.0.0.1
--2019-06-21 04:30:57--  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.1'

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

```
2019-06-21 04:30:59 (87.3 KB/s) - 'index.html.1' saved
[177669/177669]
```

Now, let's answer the questions in Item 8. With two queues, there is an improvement in both the network latency (average RTT) and download time. For the baseline, the average RTT is ~20 ms. The download time is about 1 second. After the flow, the average RTT is also still about ~20 sec and the download time for webpage is about 2 seconds. This is as expected. By using two queues with a shared bottleneck drain rate, packets from iperf as well as wget can be processed in parallel.

Part 6: TCP CUBIC

Item #9: Did the change in congestion control algorithm result in any significant differences in the results?

- *If so, which algorithm performed better? What aspect of that particular algorithm do you think caused it to perform better?*
- *If there was no change, was this what you expected? Why or why not?*

First, baseline performance of download speed:

```
mininet> h2 wget http://10.0.0.1
--2019-06-21 04:34:02--  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      175KB/s
in 1.0s

2019-06-21 04:34:03 (175 KB/s) - 'index.html' saved [177669/177669]
```

Next, baseline measurement of network delay:

```
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=21.9 ms
64 bytes from 10.0.0.2: icmp_seq=2 ttl=64 time=22.0 ms
64 bytes from 10.0.0.2: icmp_seq=3 ttl=64 time=21.8 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=20.3 ms
64 bytes from 10.0.0.2: icmp_seq=6 ttl=64 time=20.1 ms
64 bytes from 10.0.0.2: icmp_seq=7 ttl=64 time=20.3 ms
64 bytes from 10.0.0.2: icmp_seq=8 ttl=64 time=22.1 ms
64 bytes from 10.0.0.2: icmp_seq=9 ttl=64 time=21.6 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 9013ms
rtt min/avg/max/mdev = 20.155/21.114/22.136/0.848 ms
```

Next, download speed after long flow:

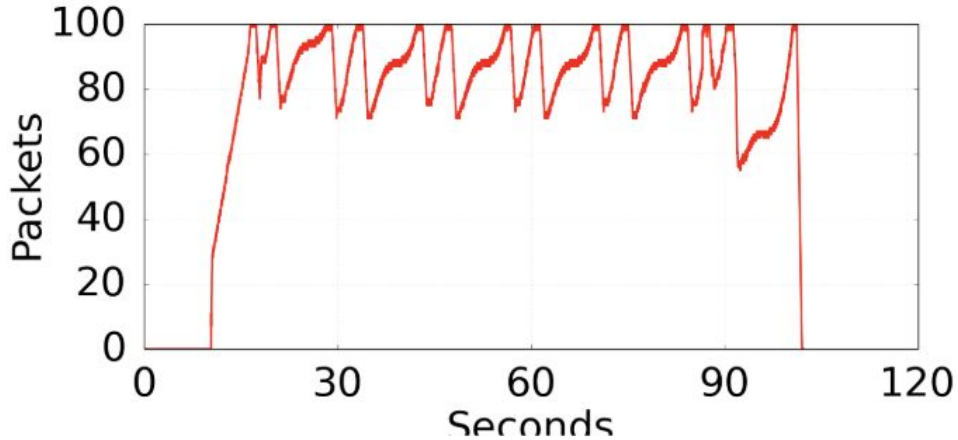
```
mininet> h2 wget http://10.0.0.1
--2019-06-21 04:36:51--  http://10.0.0.1/
Connecting to 10.0.0.1:80... connected.
HTTP request sent, awaiting response... 200 OK
Length: 177669 (174K) [text/html]
Saving to: 'index.html.1'

100%[======================================>] 177,669     25.3KB/s
in 6.8s

2019-06-21 04:36:59 (25.3 KB/s) - 'index.html.1' saved
[177669/177669]
```

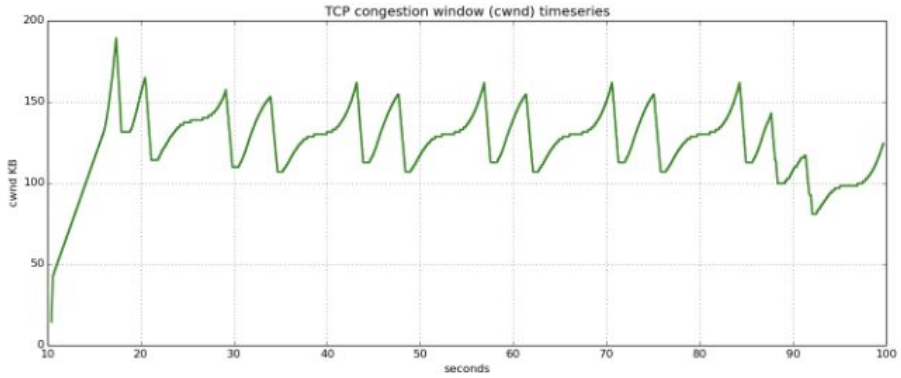
Next, network latency after long flow:

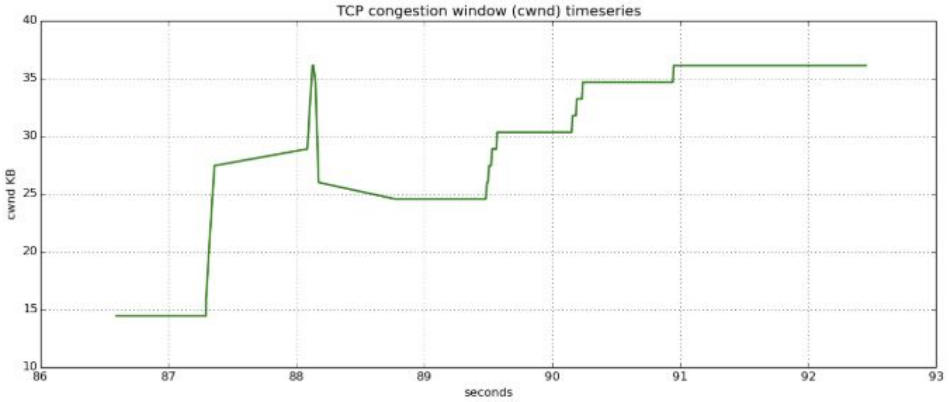
```
mininet> h1 ping -c 10 h2
PING 10.0.0.2 (10.0.0.2) 56(84) bytes of data.
64 bytes from 10.0.0.2: icmp_seq=1 ttl=64 time=798 ms
64 bytes from 10.0.0.2: icmp_seq=2 ttl=64 time=629 ms
64 bytes from 10.0.0.2: icmp_seq=3 ttl=64 time=679 ms
64 bytes from 10.0.0.2: icmp_seq=4 ttl=64 time=778 ms
64 bytes from 10.0.0.2: icmp_seq=6 ttl=64 time=611 ms
64 bytes from 10.0.0.2: icmp_seq=7 ttl=64 time=629 ms
64 bytes from 10.0.0.2: icmp_seq=8 ttl=64 time=671 ms
64 bytes from 10.0.0.2: icmp_seq=9 ttl=64 time=705 ms
64 bytes from 10.0.0.2: icmp_seq=10 ttl=64 time=721 ms

--- 10.0.0.2 ping statistics ---
10 packets transmitted, 9 received, 10% packet loss, time 9009ms
rtt min/avg/max/mdev = 611.741/691.825/798.295/62.050 ms
```

Next, we show the relevant graphs for Experiment 3 and 4

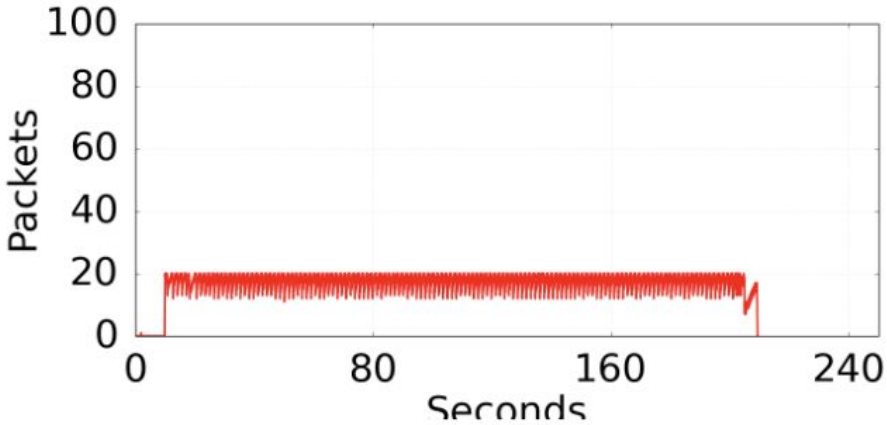
Experiment 3

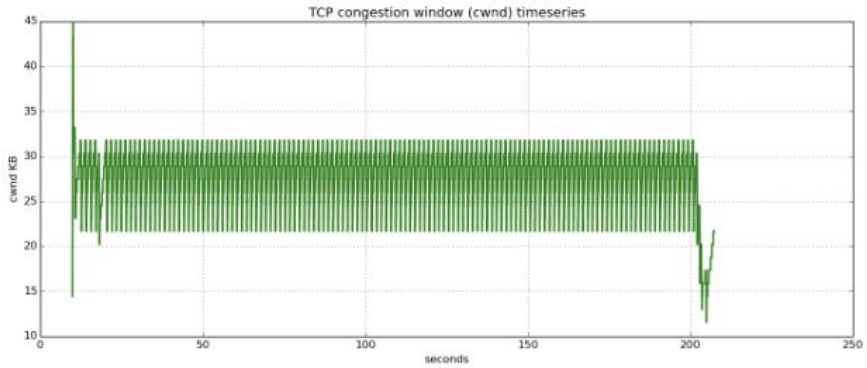


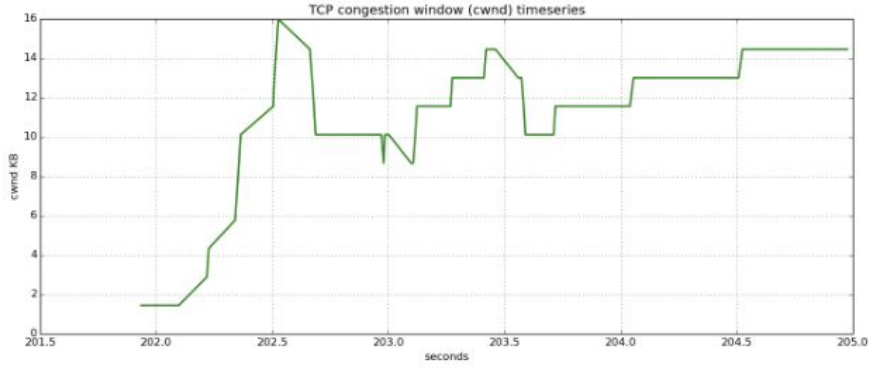




Experiment 4







From the output and the graphs above, we see that the TCP CUBIC algorithm had better performances. This is especially true for the large queues (Experiment 4 vs Experiment 2).

Previously, if we look at the graphs in part 4, there were delays of wget caused by large amount of iperf packets. However, with TCP Cubic there are concave and convex regions and the wget graph has less delays.

TCP CUBIC performs better because it simplifies the BIC-TCP window control by providing a cubic function. CUBIC has its window growth depending only on the time between two consecutive congestion events. Thus independent of RTTs is the window growth which allows CUBIC flows in the same bottleneck to have the same window size.