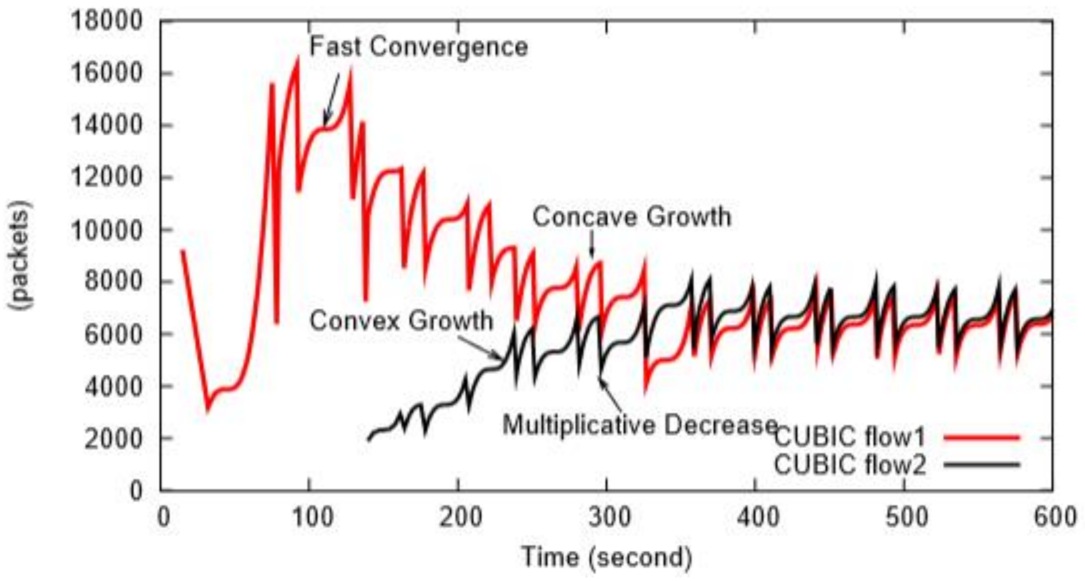
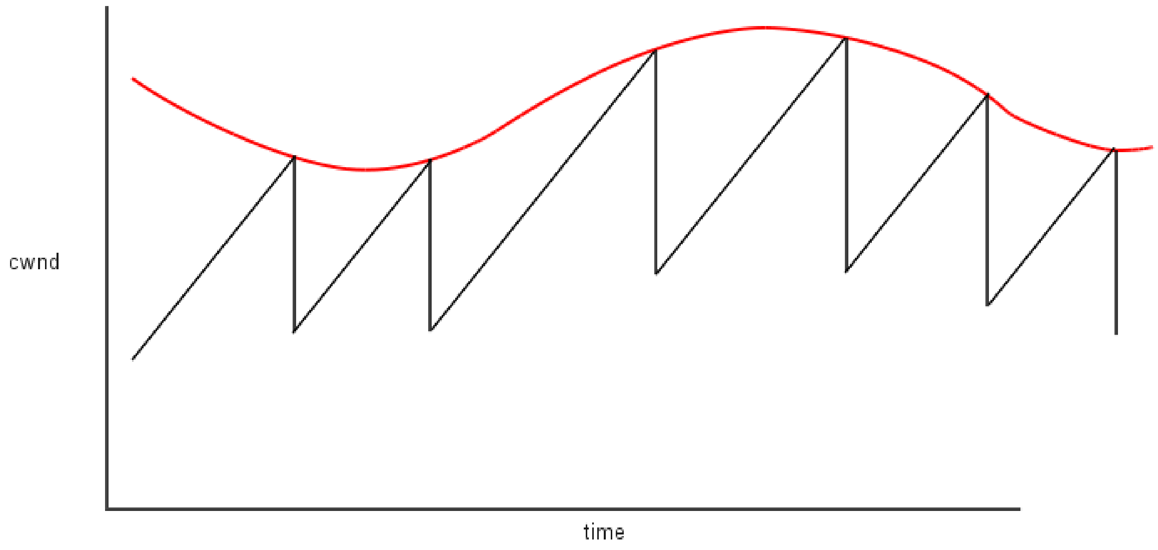
Item #1

The CWND curves for TCP Cubic have plateaus around Wmax which is the window size at the time of the last packet loss event. Equation is W(t) = C(t - K)^3 + Wmax and K = (Wmax * beta/C)^1/3



The CWND curves for TCP Reno have classic "TCP sawtooth" graph, where the peaks are at the points where the slowly rising cwnd crossed above the "network ceiling".



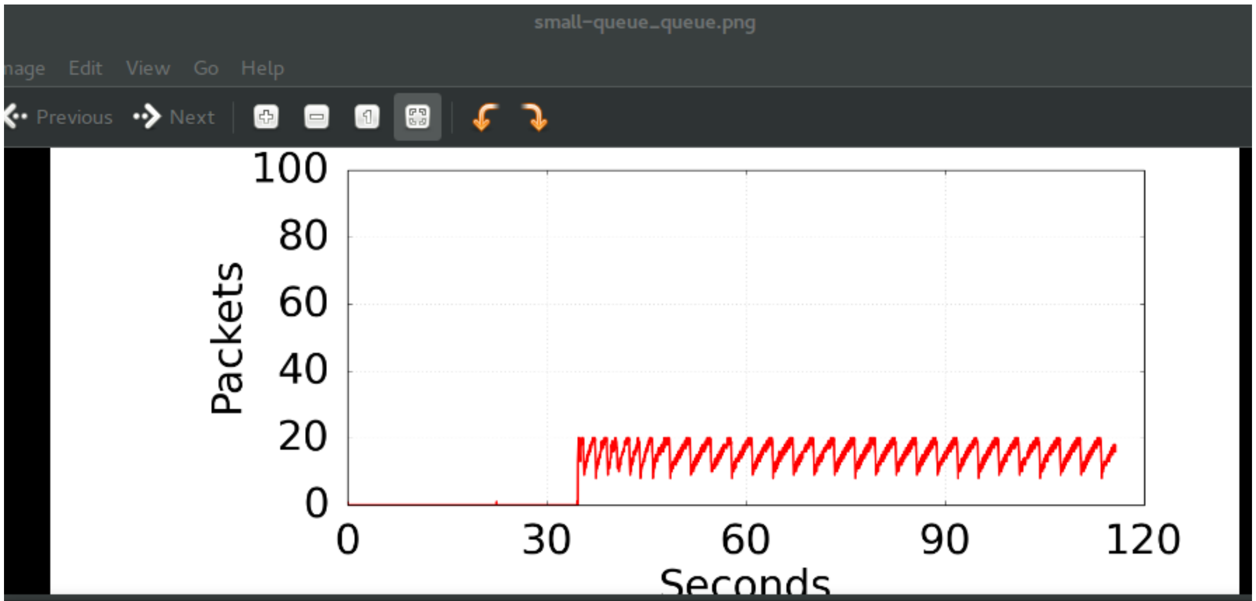
Item #2

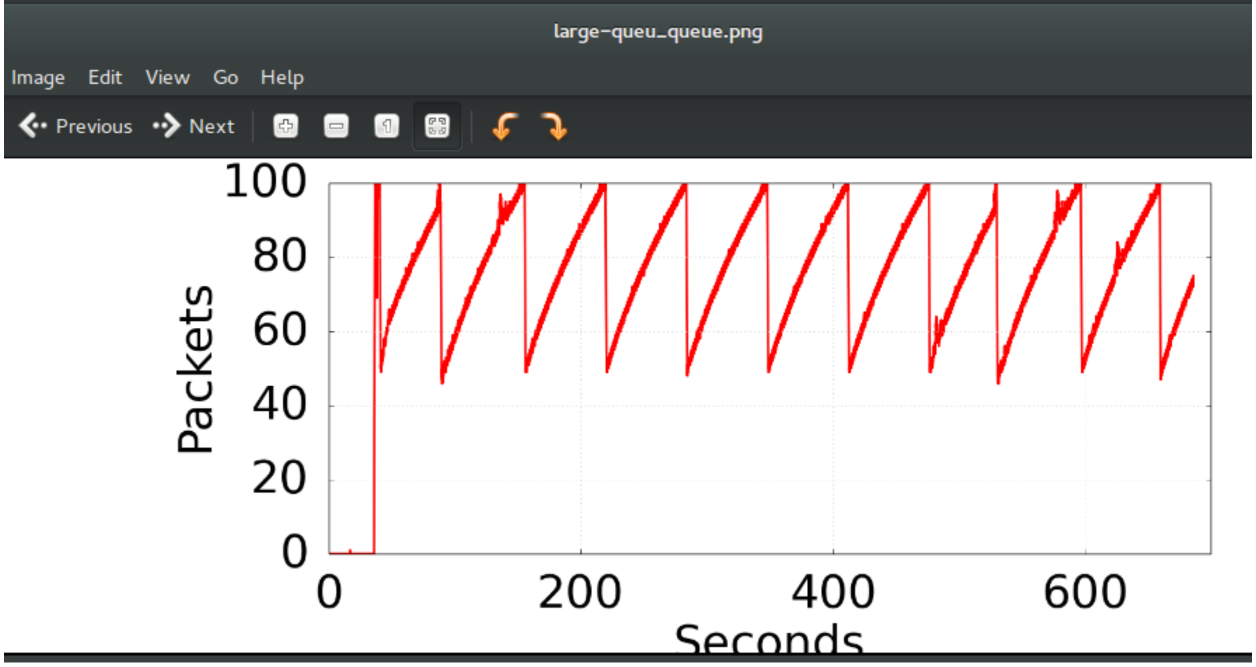
TCP Cubic is based on W(t) = C(t - K)^3 + Wmax and K = (Wmax * beta/C)^1/3, W is a portion of Wmax initially when the network starts up (basically non-zero value). Wmax can be an arbitrary value to begin with and it will be converged to the good and appropriate value based from TCP Cubic algorithm in the end. The curves have plateaus around Wmax, which is the window size at the time of the last packet loss event. It has a concave growth until the window size becomes Wmax or no packet is dropped. Once it reaches Wmax, it continues to grow into a convex profile due to the equation for W(t). The exponential growth continues till there is a packet loss. If bandwidth per flow increases, the graph will include both concave and convex curve with Wmax at the top. If bandwidth per flow decreases, only a part of the concave curve can show up and the plateau may not reach around Wmax.

TCP Reno also begins with an arbitrary value for cwnd and increase until packet is dropped. its cwnd will set to W * beta. If bandwidth per flow increases, the sawtooth may reach high spikes than previous peaks for high Wmax. If bandwidth per flow decreases, the peaks may be lower than previous peaks.

Item #3

Smaller queue performs better than larger queue in terms of the graph below. Smaller queue has less RTT and latency than large queue in the comparison diagram below. That is because the packets sent from the upper link, quickly filled up the larger queue and no signal of packet loss received in the larger queue.





Item #4

I saw Cubic results demonstrate to begin with a linear increase till the first packet loss occurred while I expected to see a start with a concave and convex profile and in the large queue the curve first reach Wmax and then drops to convex region till packet loss occurs and then reach the tips again while it still remains in concave and convex profile. This can be explained by the below algorithm from the paper:

The paper also mentions about fast convergence, in which if Wmax evaluates to be less than Wlast_max, it implies network conditions have changed and further less bandwidth is available, hence Wmax is further reduced. This explains why there is alternatively a concave-convex graph along with concave graph.

Item #5

The prediction goes well with the graphs match up with the actual results in both small and large queue.

In both small and large queue, compared by TCP Reno, CUBIC is denser in cubic shape and have plateaus around Wmax which is the window size at the time of the last packet loss event. If network bandwidth per flow remains constant, then only concave region of the cubic growth function would appear along with the plateau near Wmax. If bandwidth per flow increases, the graph will include both concave and convex curve with Wmax at the tipping point. If bandwidth per flow decreases, only a part of the concave curve will be seen and the plateau may not be reached around Wmax.

Item #6

Switching to long lived flow has a bad effect on RTT and latency, the download time grew from 1.0s to 5.7s. When packets are queued at long lived flow, it takes a long time for the previous packets to be cleared while single flow has less packets in queue resulting in less number of packets require less time to get cleared at drop rate. Also another factor, long lived flow adjusts the decrease factor by a function of RTTs which is engineered to estimate the queue size in the network path of the current flow, which is an overhead to RTT and latency.

Single flow statistics:

```
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=20.9 ms

64 bytes from 10.0.0.2: icmp_seq=3 ttl=64 time=20.7 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=20.7 ms

64 bytes from 10.0.0.2: icmp_seq=6 ttl=64 time=20.9 ms

64 bytes from 10.0.0.2: icmp_seq=7 ttl=64 time=20.6 ms

64 bytes from 10.0.0.2: icmp_seq=8 ttl=64 time=20.7 ms

64 bytes from 10.0.0.2: icmp_seq=9 ttl=64 time=20.5 ms

64 bytes from 10.0.0.2: icmp_seq=10 ttl=64 time=20.7 ms

--- 10.0.0.2 ping statistics ---

10 packets transmitted, 10 received, 0% packet loss, time 9012ms

rtt min/avg/max/mdev = 20.506/20.700/20.936/0.235 ms

Connecting to 10.0.0.1:80... connected.

HTTP request sent, awaiting response... 200 OK
```

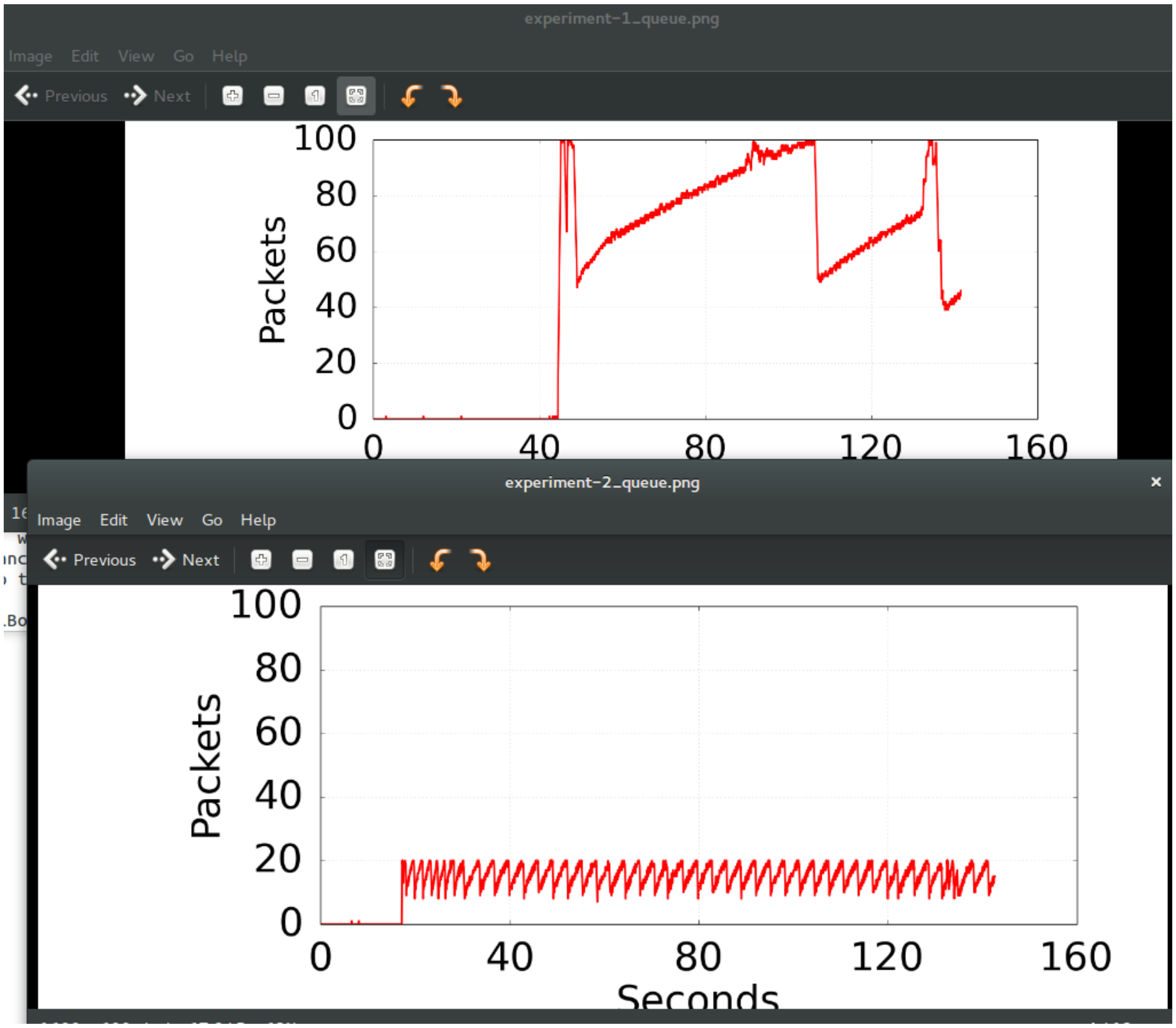
```
Length: 177669 (174K) [text/html]
Saving to: ‘index.html’
100%[======================================>] 177,669      175KB/s   in 1.0s
2019-03-09 22:16:14 (175 KB/s) - ‘index.html’ saved [177669/177669]
```

Long lived flow statistics:

```
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=474 ms
64 bytes from 10.0.0.2: icmp_seq=2 ttl=64 time=482 ms
64 bytes from 10.0.0.2: icmp_seq=3 ttl=64 time=491 ms
64 bytes from 10.0.0.2: icmp_seq=4 ttl=64 time=501 ms
64 bytes from 10.0.0.2: icmp_seq=5 ttl=64 time=510 ms
64 bytes from 10.0.0.2: icmp_seq=6 ttl=64 time=519 ms
64 bytes from 10.0.0.2: icmp_seq=7 ttl=64 time=528 ms
64 bytes from 10.0.0.2: icmp_seq=8 ttl=64 time=539 ms
64 bytes from 10.0.0.2: icmp_seq=9 ttl=64 time=538 ms
64 bytes from 10.0.0.2: icmp_seq=10 ttl=64 time=547 ms
10 packets transmitted, 10 received, 0% packet loss, time 9002ms
rtt min/avg/max/mdev = 474.159/513.370/547.446/24.280 ms
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     30.5KB/s   in 5.7s
2019-03-09 22:19:52 (30.5 KB/s) - ‘index.html.1’ saved [177669/177669]
```

Item #7

A smaller queue download speed is much faster than a larger queue. From the below diagram comparison between large queue and small queue, we can find out that larger queue delays the buffer flow and fill up the buffers quickly without seeing packet loss resulting in extra delay in network latency while small queue with smaller buffer reduces the buffers and traffic come in access link never exceeds the upper link provided, shaping traffic before the buffer filled up, thus improve the download time.



Item #8

The download time and RTT is better at long lived flow this time compared to Item #6, the result is expected since long lived flow maintain a separate queue for each flow and then put iperf and wget traffic into different queues thus no more stuck behind of the packets in the buffer queue and buffer is not filled up since there are 2 queues, the link rate received are cut in half. However the download time from long lived flow with traffic control, though is improved, still not as good as download time from single flow.

Short flow:

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=26.8 ms
64 bytes from 10.0.0.2: icmp_seq=2 ttl=64 time=26.0 ms
64 bytes from 10.0.0.2: icmp_seq=3 ttl=64 time=21.3 ms
64 bytes from 10.0.0.2: icmp_seq=4 ttl=64 time=20.2 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=20.7 ms
64 bytes from 10.0.0.2: icmp_seq=7 ttl=64 time=21.4 ms
64 bytes from 10.0.0.2: icmp_seq=8 ttl=64 time=20.5 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=20.6 ms

--- 10.0.0.2 ping statistics ---
10 packets transmitted, 10 received, 0% packet loss, time 9015ms

Length: 177669 (174K) [text/html]
Saving to: ‘index.html’

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

2019-03-10 01:39:27 (175 KB/s) - ‘index.html’ saved [177669/177669]
```

Long lived flow:

```
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=25.3 ms
64 bytes from 10.0.0.2: icmp_seq=2 ttl=64 time=26.8 ms
64 bytes from 10.0.0.2: icmp_seq=3 ttl=64 time=21.2 ms
64 bytes from 10.0.0.2: icmp_seq=4 ttl=64 time=20.8 ms
64 bytes from 10.0.0.2: icmp_seq=5 ttl=64 time=20.5 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=20.9 ms

64 bytes from 10.0.0.2: icmp_seq=8 ttl=64 time=20.9 ms

64 bytes from 10.0.0.2: icmp_seq=9 ttl=64 time=20.6 ms

64 bytes from 10.0.0.2: icmp_seq=10 ttl=64 time=20.4 ms


--- 10.0.0.2 ping statistics ---

10 packets transmitted, 10 received, 0% packet loss, time 9013ms

rtt min/avg/max/mdev = 20.179/21.820/26.884/2.206 ms

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


2019-03-10 01:51:09 (87.3 KB/s) - ‘index.html.1’ saved [177669/177669]
```

Item #9

No, the change in congestion control algorithm did not make significant differences in the results.

This was not the results that I expected, since I ran the experiment twice, however, the Cubic algorithm did not perform better, sometimes worse than the TCP Reno. For smaller RTTs, the basic TCP Cubic strategy runs the risk of being at a competitive disadvantage compared to TCP Reno. For this reason, TCP Cubic makes a TCP-Friendly adjustment in the window-size calculation: on each arriving ACK, cwnd is set to the maximum of W(t) and the window size that TCP Reno would compute. The TCP Reno calculation can be based on an actual count of incoming ACKs, or be based on the formula $(1-\beta)\times$Wmax + $\alpha\times$t/RTT. Note that this adjustment is only “half-friendly”, which guarantees that TCP Cubic will not choose a window size smaller than TCP Reno’s, but places no restraints on the choice of a larger window size. A consequence of the TCP-Friendly adjustment is that, on networks with modest bandwidth×delay products, TCP Cubic behaves exactly like TCP Reno.

Another part of the reason may due to early works in this area analyzed the simplest cases, very little noise in the measurements. Under these simplifications they were able to develop algorithms that behaved well in theory. The problems that a congestion algorithm tries to solve are very hard. The fact that in actual practice there will be many unrelated flows makes the problem even harder in large RTTS and larger bandwidths.