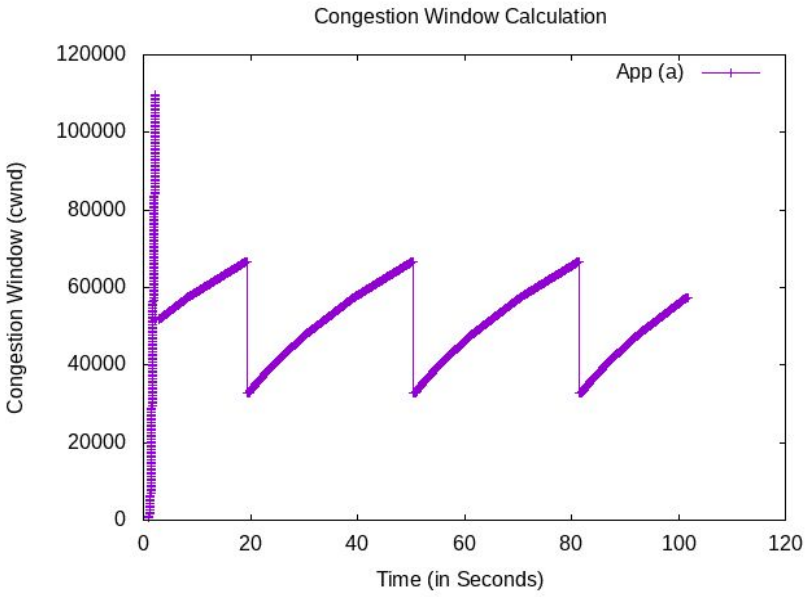
CS341 Practice #3

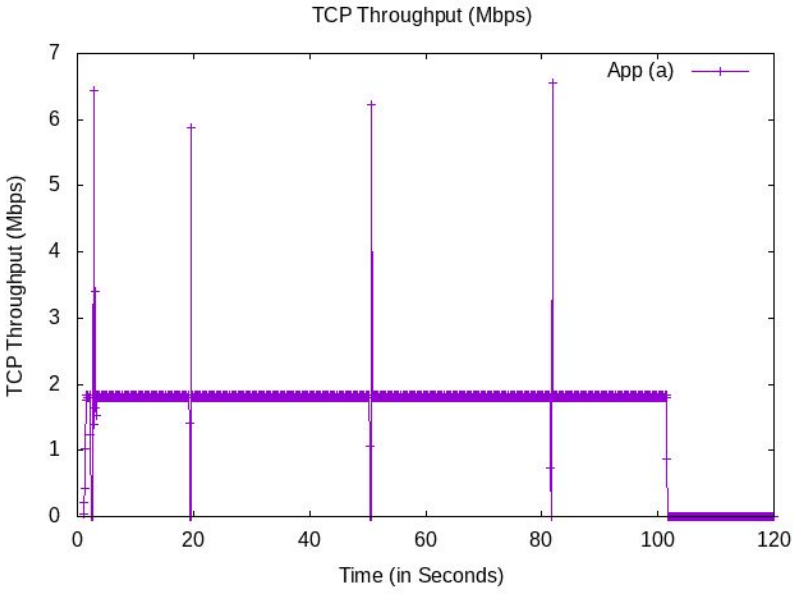
20170481 Yuseung Lee

[Task 1]

(1.1) Scenario 1: App (a) is running

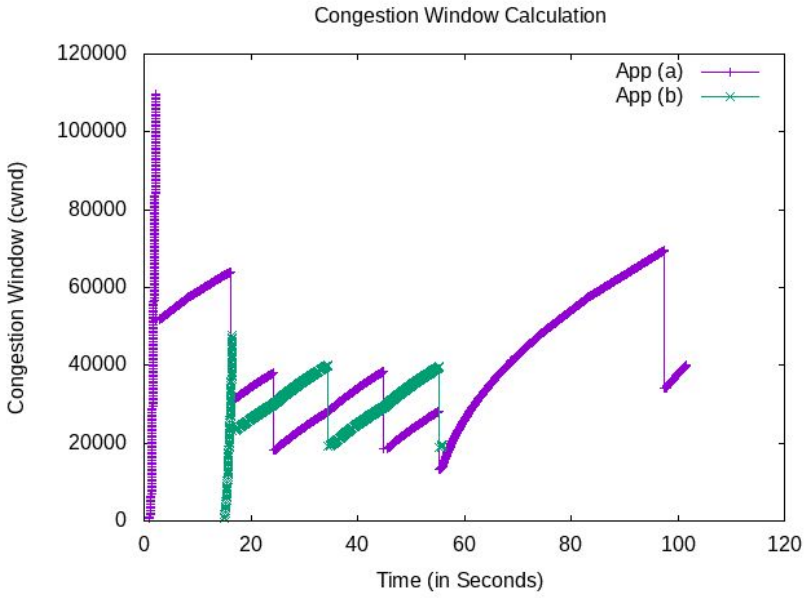


<Congestion Window Calculation>

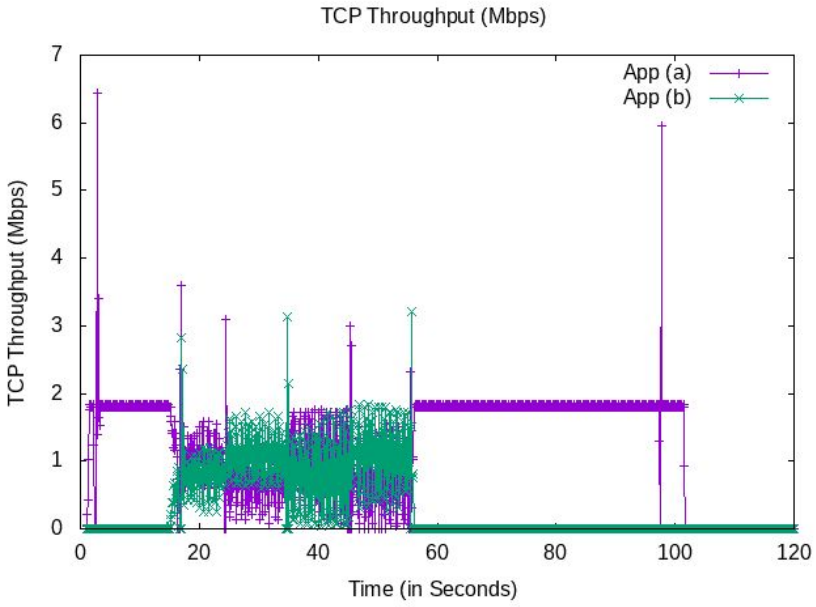


<TCP Throughput>

(1.2) Scenario 2: App (a), (b) are running.



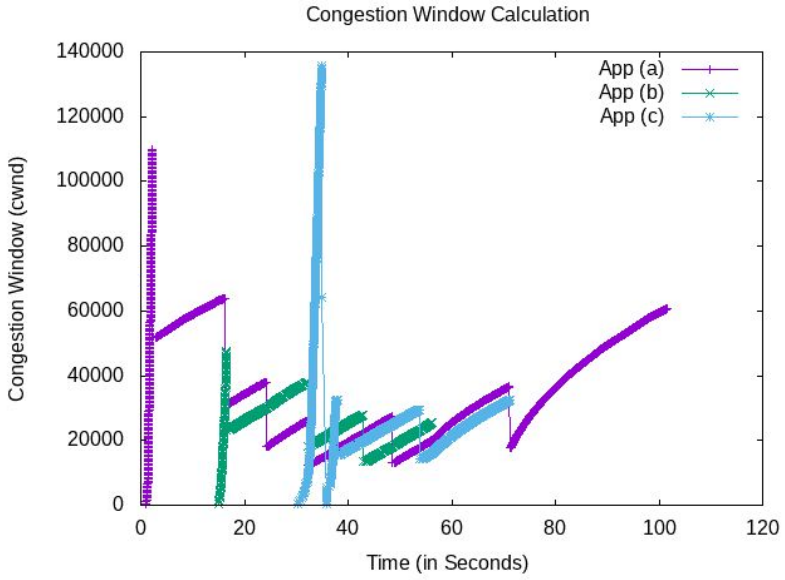
<Congestion Window Calculation>



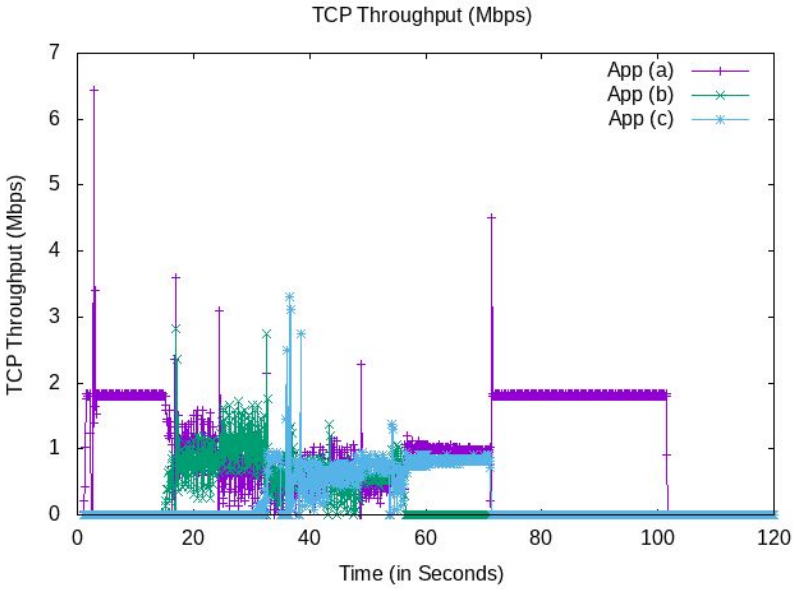
<TCP Throughput>

Fast recovery (which halves the cwnd value) occurs more frequently in Scenario 2 than it does in Scenario 1. This is because App (b) starts running at 15s so App (a) and (b) need to share the same link for Node 1 to Node 0. In the TCP Throughput graph, we can check that when the throughput for App (a) increases then the throughput for App (b) decreases, and the opposite is also true. This is the result of congestion control to prevent the amount of packets traversing to Node 0 from exceeding 2 Mbps. After the packets from App (b) are all sent, the throughput for App (a) becomes the same as Scenario 1 and cwnd keeps increasing since it is now the only application sending packets to Node 0.

(1.3) Scenario 3: App (a), (b), (c) are running.



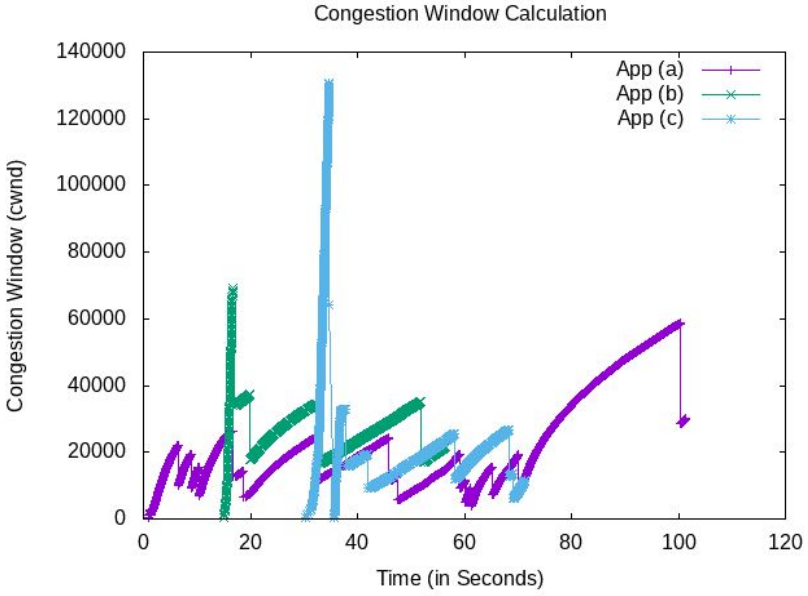
<Congestion Window Calculation>



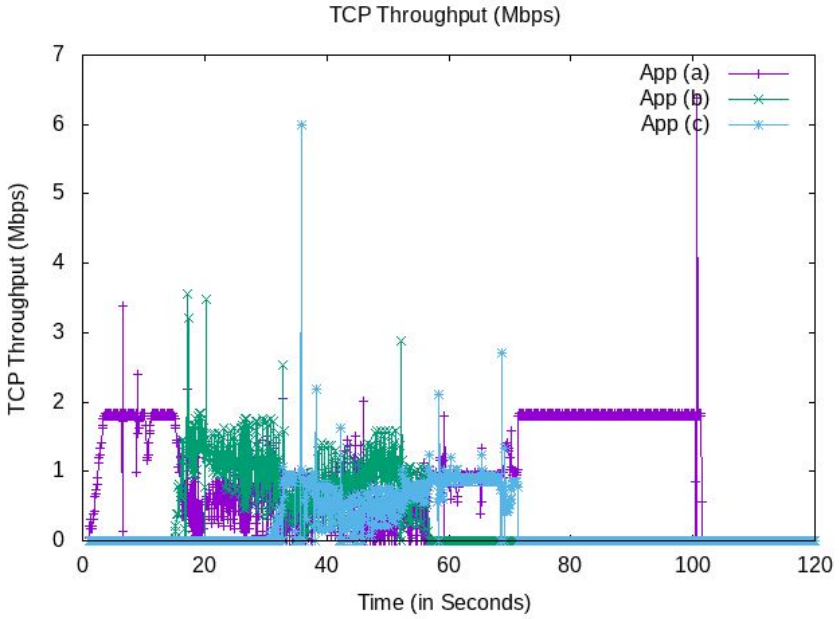
<TCP Throughput>

Before 30 sec, the throughput and cwnd values are the same as Scenario 2. At 30 sec App (c) starts running so there are now 3 applications sharing the link from Node 1 to Node 0. We can see that in App (c) a timeout occurs and thus the cwnd value is initialized to 1 MSS and restarts slow start. The TCP Throughput graph shows that due to congestion control, the sum of packets traversing to Node 0 doesn't exceed 2 Mbps. At 55 sec App (b) has sent all its packets and therefore the throughput for App (a) and (c) is increased because now only 2 applications share the link to Node 0. After App (c) has sent all its packets (at about 70 sec) App (a) has the same throughput as it has in Scenario 1.

(1.4) Scenario 4: Add Error Rate Model



<Congestion Window Calculation>

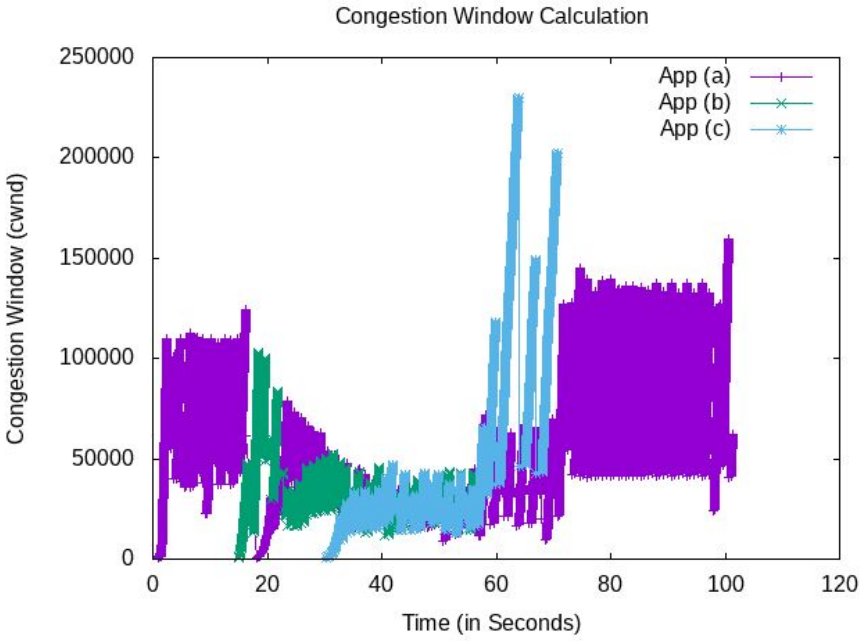


<TCP Throughput>

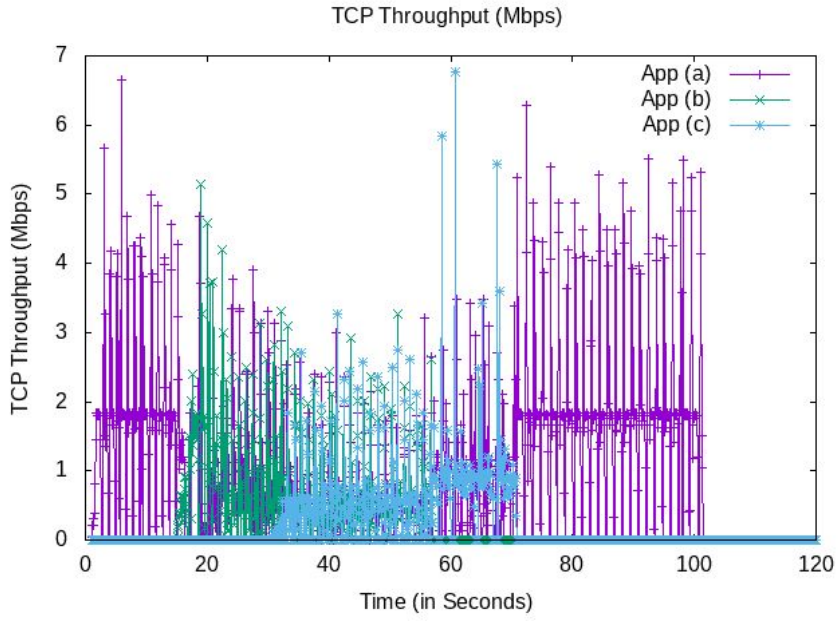
Compared to Scenario 3, fast recovery occurs more frequently in Scenario 4 and therefore cwnd values are kept at a lower size. This is because when each application detects a packet loss at the link to Node 0, TCP Reno implements the fast recovery by halving the cwnd value. One interesting point is that although cwnd is lower than the previous case and there are packet losses, the time that each application stops sending packets is almost the same as Scenario 3 (a: 100 sec, b: 55 sec, c: 70 sec). This is because the simulation code doesn't contain any code for packet retransmission, so this implementation is actually similar to UDP since delivery is not guaranteed.

[Task 2]

(2.1.1) Change *CongestionAvoidance* in tcp-congestion-ops.cc



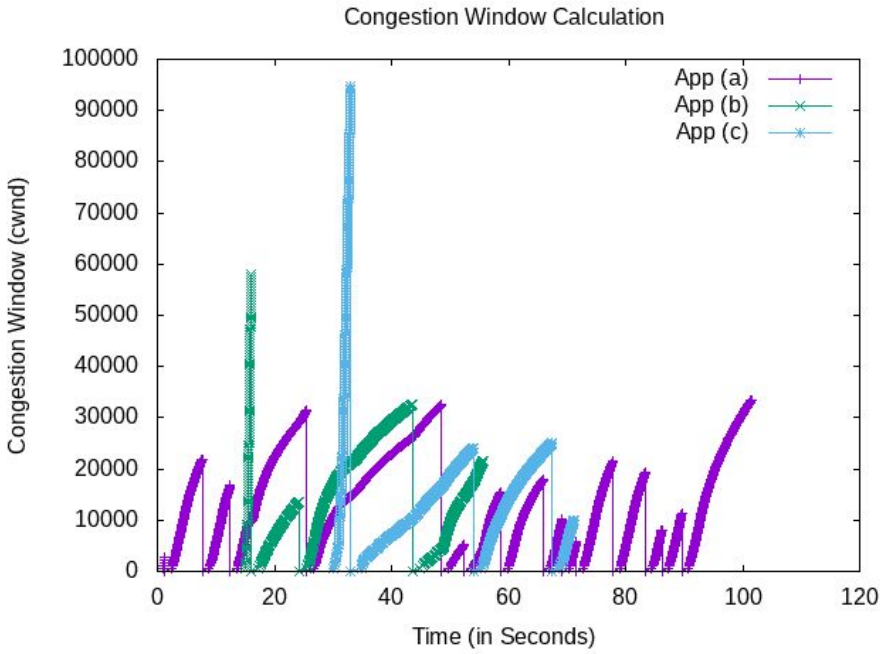
<Congestion Window Calculation>



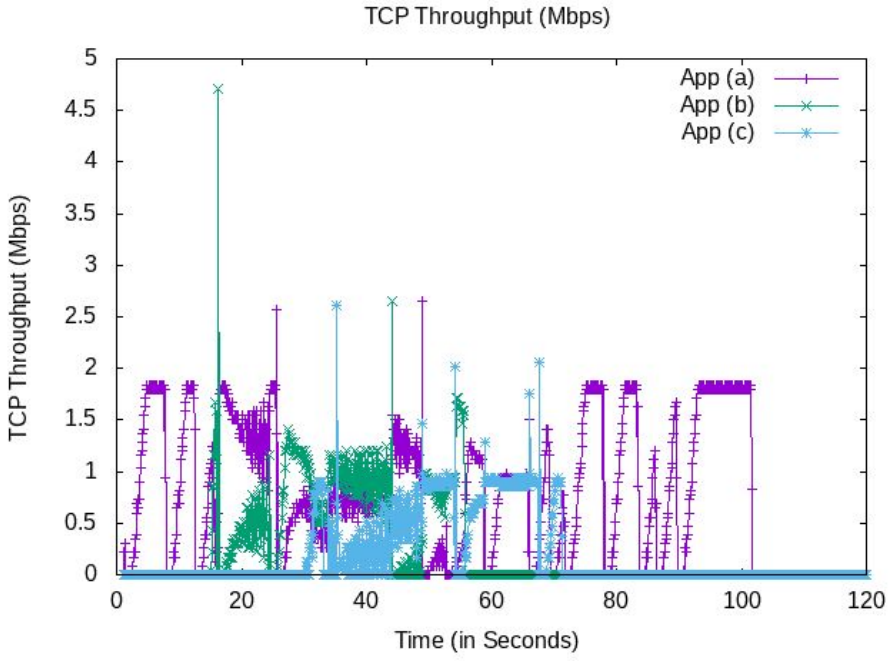
<TCP Throughput>

Normally the cwnd value in Congestion Avoidance is increased linearly by small amounts in order to prevent congestion. But since we changed the adder value in Congestion Avoidance, cwnd now increases by m_segmentSize which is a large value. So in the cwnd graph above we can see that when cwnd is increased by m_segmentSize then congestion immediately occurs and therefore cwnd is decreased by half (fast recovery). This process keeps repeating because every time cwnd is increased by m_segmentSize it exceeds the link capacity. This is why the Congestion Avoidance process is required in congestion control.

(2.1.2) Change *EnterRecover, DoRecovery, ExitRecovery*



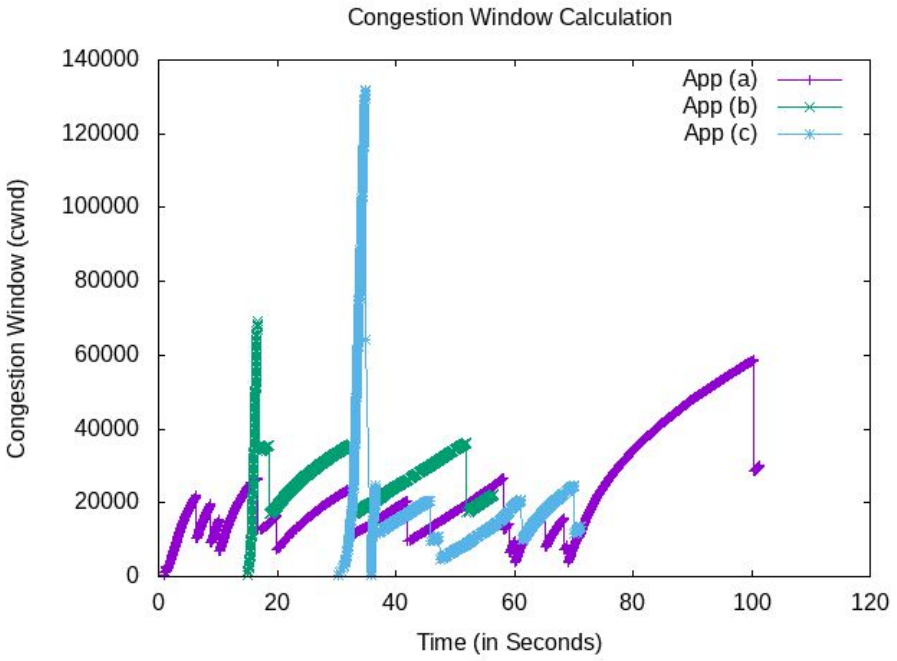
<Congestion Window Calculation>



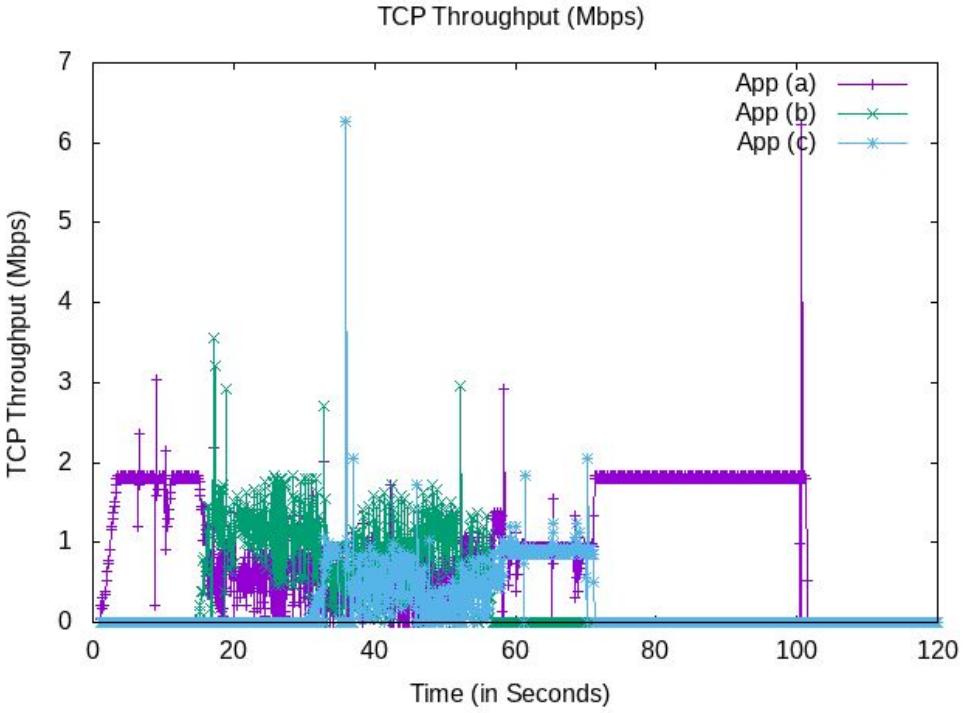
<TCP Throughput>

In this scenario, when packet loss occurs the cwnd is initialized to 1, whereas in Scenario 4 it is halved by fast recovery. This is because the fast recovery code was removed. We can see in the throughput graph that whenever a packet loss occurs the throughput decreases sharply to almost zero and then increases again. This is different from Scenario 4, where it implements fast recovery and the throughput doesn't decrease as much as in this scenario.

(2.2.1) *Veno*

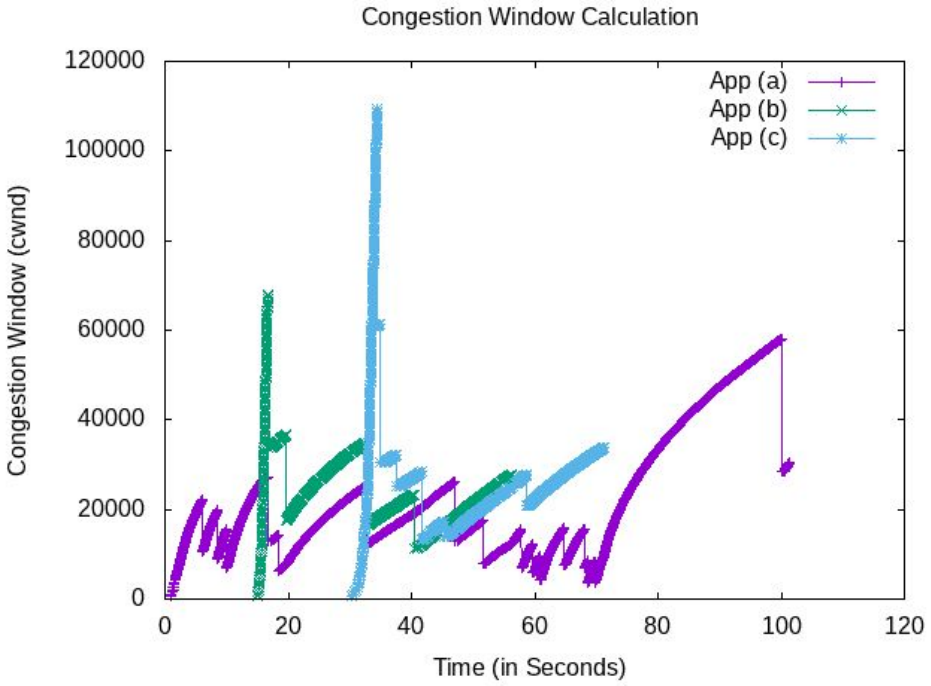


<Congestion Window Calculation>

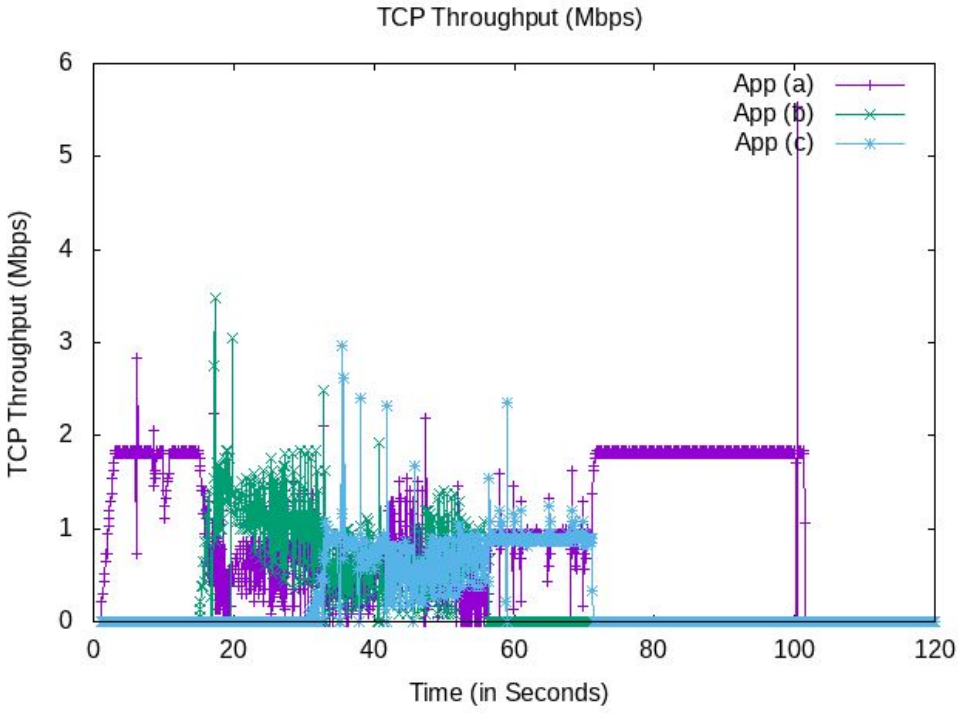


<TCP Throughput>

(2.2.2) ***Yeah***



<Congestion Window Calculation>



<TCP Throughput>

(2.2.3) Implementing *Veno & Yeah*

Total Received Bytes (Bytes)	*NewReno*	*Veno*	*Yeah*
App (a)	14,234,000	14,486,000	14,467,000
App (b)	4,789,000	4,874,000	4,277,000
App (c)	3,322,000	2,984,000	3,723,000

I would prefer using the Yeah algorithm. First, the total received byte by all three applications are as following:

NewReno: 22,345,000 bytes
Veno: 22,344,000 bytes
Yeah: 22,467,000 bytes

We can see that Yeah has the largest number of received bytes, which means that the least amount of congestion occured at the link from Node 1 to Node 0 and thus the number of packet drops were less than the other two algorithms.

Also in the cwnd graph for Yeah we can see that the cwnd value doesn't always decrease to half. Instead it seems to decrease less when there is enough capacity in the link, which means the control control is more efficient than other algorithms in which the cwnd is always decreased in half.