NETWORKS LAB 5.2

REPORT

S Mohana Prasad
CS12B025

We will first understand each of these protocols and that would drive us towards understanding the plots that we have got.

QUESTION 1 a

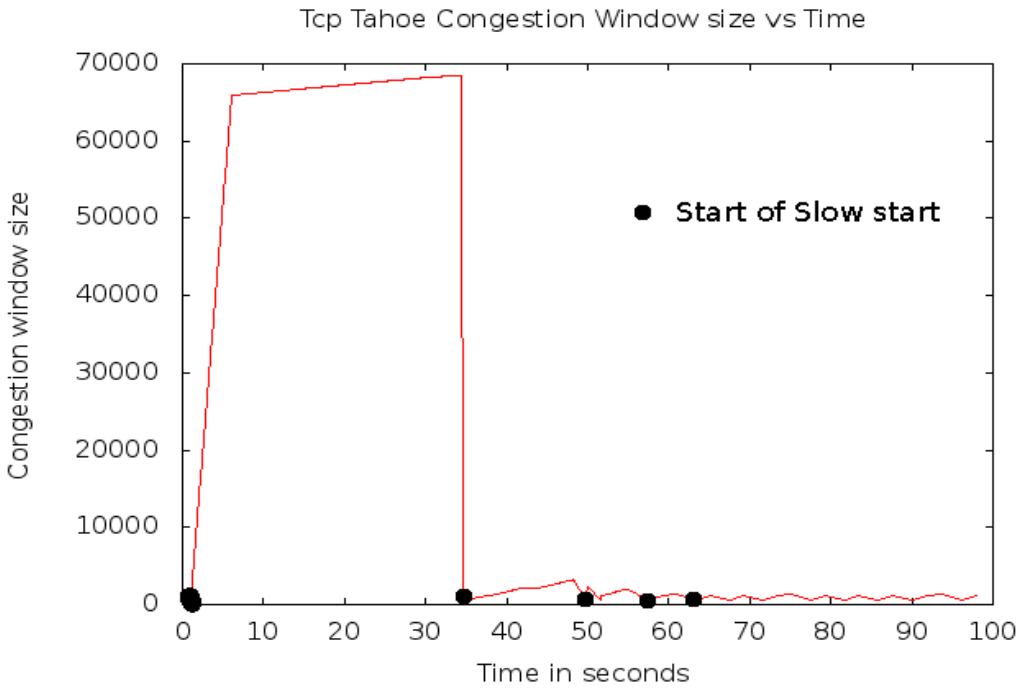
For the first part, the bottle neck link had a capacity of 500kbps and the TCP protocol was sending at 200kbps. The UDP application was sending at half and full capacity of the bottle neck link, as mentioned in the question.

TCP Tahoe:

It starts with a window of 1 MSS. For every packet acknowledged, the congestion window increases by 1 MSS so that the congestion window effectively doubles for every round-trip time (RTT). When the congestion window exceeds the ssthresh threshold, the algorithm enters a new state, called congestion avoidance. For congestion avoidance Tahoe uses ‘Additive Increase Multiplicative Decrease’. A packet loss is taken as a sign of congestion and Tahoe saves the half of the current window as a threshold. value. It then set CWD to one
and starts slow start until it reaches the threshold value. After that it increments linearly until it encounters a packet loss. Thus it increase it window slowly as it approaches the bandwidth capacity. The important thing is that Tahoe detects packet losses by timeouts. In many implementations we have coarse grain timeouts that occasionally check for time outs.

The problem with Tahoe is that it takes a complete timeout interval to detect a packet loss and in fact, in most implementations it takes even longer because of the coarse grain timeout. Also since it doesn’t send immediate ACK’s, it sends cumulative acknowledgements, there fore it follows a ‘go back n ‘ approach. Thus every time a packet is lost it waits for a timeout and the pipeline is emptied. This offers a major cost in high band-width delay product links.

All these features can be clearly seen in the plot. We find that there is initially a slow start (exponential). After 30 seconds when a congestion occurs, it directly sets cwnd to 1 MSS and starts all over again. Subsequently also, we see similar behaviour where once a timeout is hit, it goes back to slow start. We also find that, even though at 30th second, we started congestion, its only about the 32nd second when our TCP reacts, this is attributed to the coarse grain timeout checks.



TCP Reno:

Reno retains the basic principle of Tahoe, such as slow starts and the coarse grain re-transmit timer. Reno retains the basic principle of Tahoe, such as slow starts and the coarse grain re-transmit timer. Another modification that RENO makes is in that after a packet loss, it does not reduce the congestion window to 1. Since this empties the pipe. It does fast re-transmit.

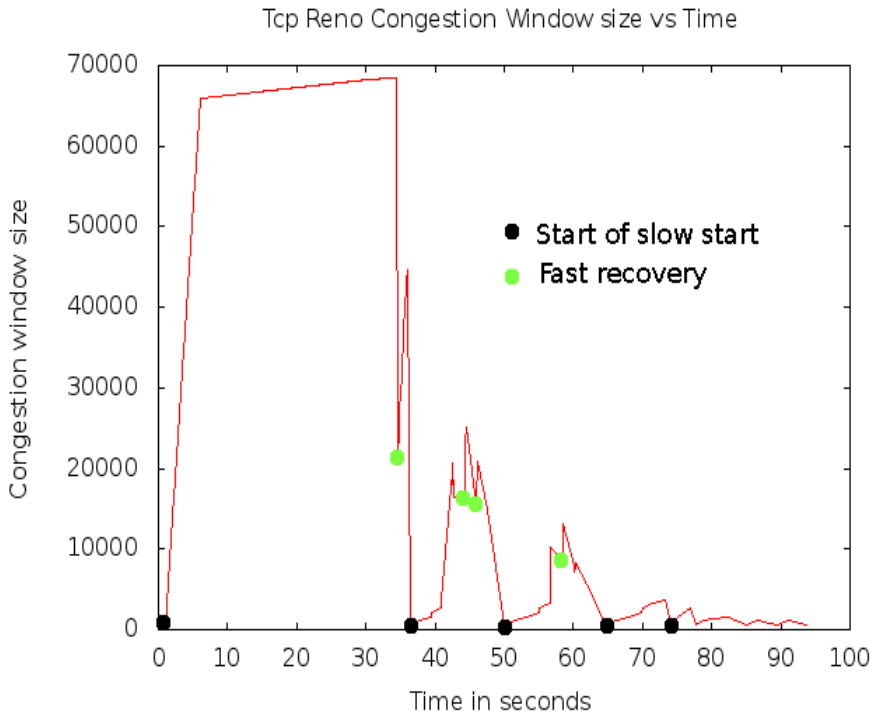
- Each time we receive 3 duplicate ACK's we take that to mean that the segment was lost and we re-transmit the segment immediately and enter 'Fast Recovery'
- Set SSthresh to half the current window size and also set CWD to the same value.
- For each duplicate ACK receive increase CWD by one. If the increase CWD is greater than the amount of data in the pipe then transmit a new segment else wait. Whenever we receive a fresh ACK we reduce the CWND to SSthresh.

Thus we don't empty the pipe, we just reduce the flow. We continue with congestion avoidance phase of Tahoe after that. If an ACK times out, slow start is used as it is with Tahoe.

In short, Tahoe only uses a timeout for detecting congestion, while Reno uses timeout and Fast-Retransmit. Tahoe sets the congestion window to 1 after packet loss, while Reno sets it to half of the latest congestion window.

Reno perform well when the packet losses are small. But when we have multiple packet losses in one window then its performance falls and becomes equal to that of Tahoe. The reason is that it can only detect a single packet losses. If there is multiple packet drop then the first info about the packet loss comes when we receive the duplicate ACK's. But the information about the second packet which was lost will come only after the ACK for the retransmitted first segment reaches the sender after one RTT. if the widow is very small when the loss occurs

then we would never receive enough duplicate acknowledgements for a fast retransmit and we would have to wait for a coarse grained timeout. Thus it cannot effectively detect multiple packet losses.



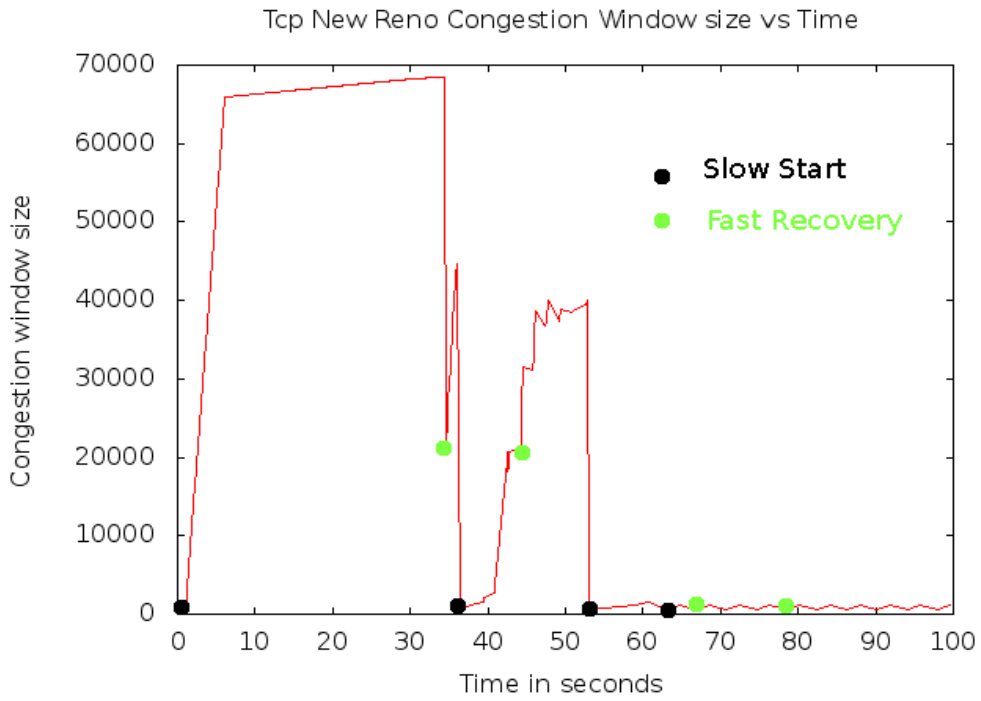
We find that, Reno doesn't wait for coarse grain timeout for re transmission, soon after 30 seconds there is a drop in the window size to half the value. This shows that 3 duplicate packets have been got. From the fast recovery point, we find that it shows a spike and comes back to zero.
His means there was a timeout that occurred (too much congestion and subsequent packets didn't reach as well). This leads to a slow start followed by a spike and again a fast recovery. We find that it doesn't come back to zero many times as seen in Tahoe and hence gives a better throughput.

TCP NewReno:

New-Reno also enters into fast-retransmit when it receives multiple duplicate packets, however it differs from RENO in that it doesn’t exit fast-recovery until all the data which was out standing at the time it entered fast recovery is acknowledged. Thus it overcomes the problem faced by Reno of reducing the CWD multiples times. It exits Fast recovery when all the data in the window is acknowledged.

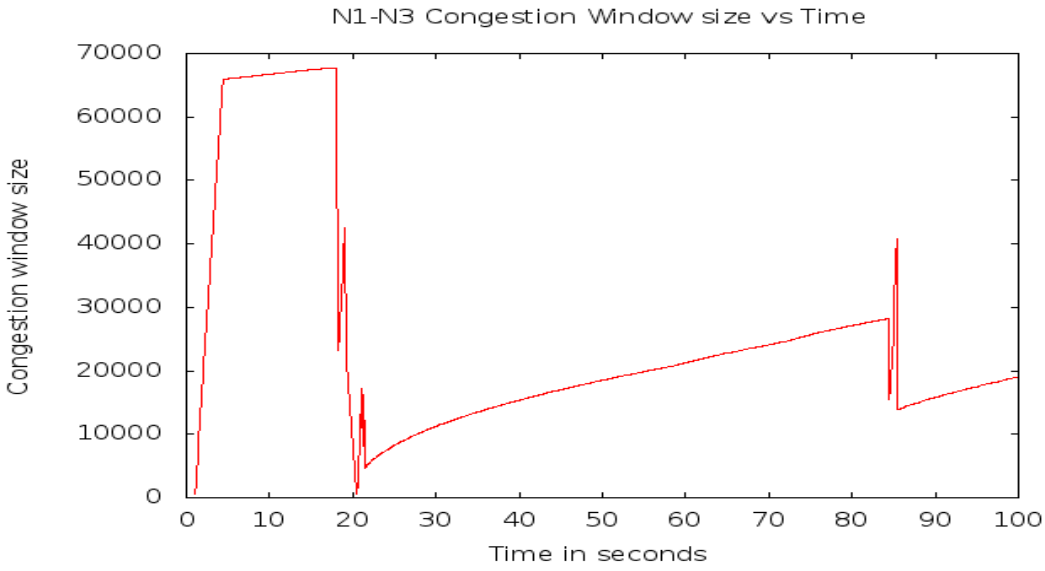
New-Reno suffers from the fact that its take one RTT to detect each packet loss. When the ACK for the first re-transmitted segment is received only then can we deduce which other segment was lost.

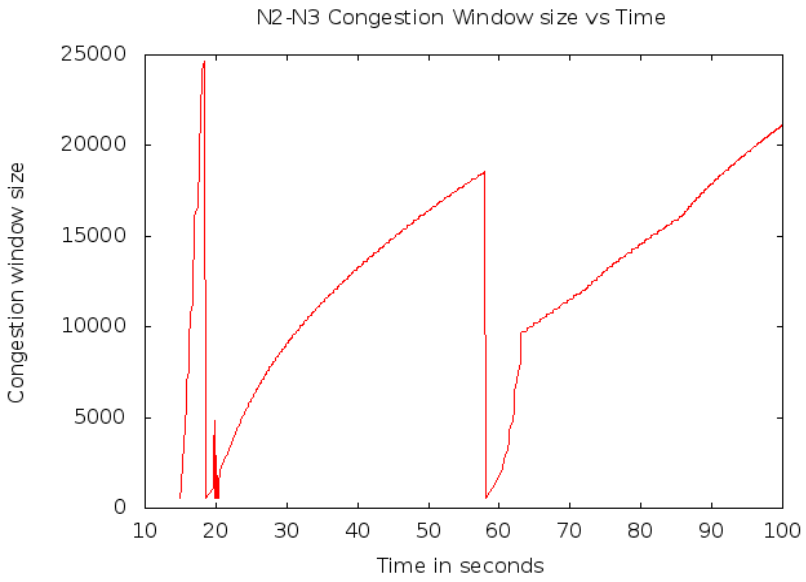
The plot is similar to that of Reno, but we find some differences. We found lots of small small spikes in Reno, while its lesser in New Reno, this is attributed to the fact that New Reno handles multiple packet loss in a window more efficiently. It waits till all the segments in the window are acknowledged and then only enters the recovery mode. This also increases the through put as seen from the experiment.

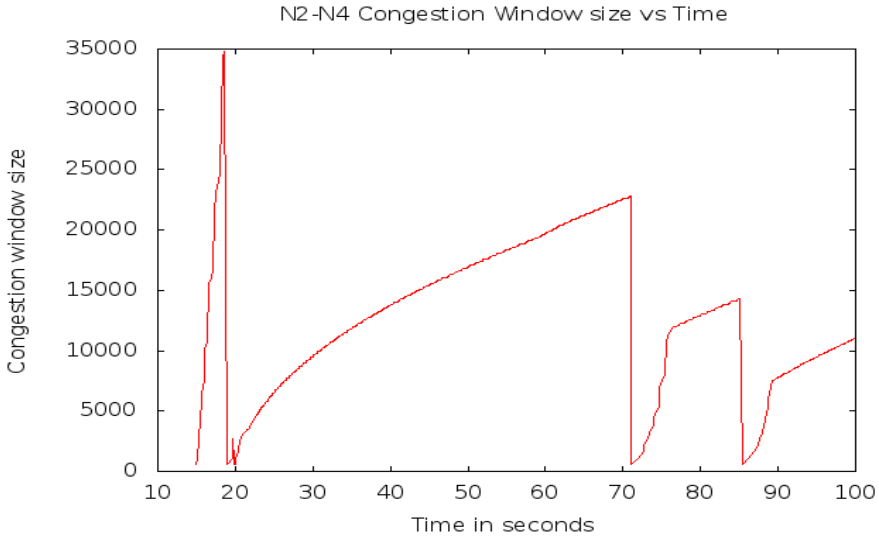


QUESTION 1 b

Lets me first present the graphs. The connections were made as directed.







These three plots help us understand everything thats happening.

Just by observing the number of spikes itself we can guess what protocol each of the links are running. N1-N3 has lots of short spikes indicating it is TCP Reno. N2-N3 has a short spike at 20th second (recovery phase where we increase cwnd as we get duplicate acks) saying its TCP NewReno. N2-N4 is the TCP Tahoe link with no spikes(No fast recovery but only slow start).

N1-N3 link running TCP Reno first had a slow start till the threshold and then it started increasing linearly. As soon as congestion started after 15 second, packets were lost and fast retransmit occurred. It tried to recover (the spike) but it has led to a time out. So, It does a slow start starting from one. A small spike there denotes a recovery phase. Then there was slow increase in the window size and finally another set of 3 duplicate acks are received at around 80 seconds leading to fast retransmit and fast recovery.

TCP NewReno also performs like TCP Reno. We find lesser spikes here as it handles multiple packet loss in a window efficiently. We can also clearly see the slowstart phase and the linear increase phase clearly here. Here New Reno is also performing like Tahoe as there is heavy congestion and we aren't receiving any duplicate acks for fast retransmission.

TCP Tahoe also performs well, it does a slow start everytime there is a timeout. Again the slow start phase and the linear increase phase can be seen clearly.

The throughput got for Reno is far higher than the others, it might be because of the initial 15 seconds when there was no congestion.
Throughput of New reno and Tahoe are comparable but New Reno performs better.

QUESTION 2

My variant of TCP is basically designed to be aggressive. It tries to use more bandwidth and tries to reach the maximum in the least time possible. When many machines running this protocol is put together, it will stabilize to a optimal value.The TCP Reno code was modified and functionalities incorporated.

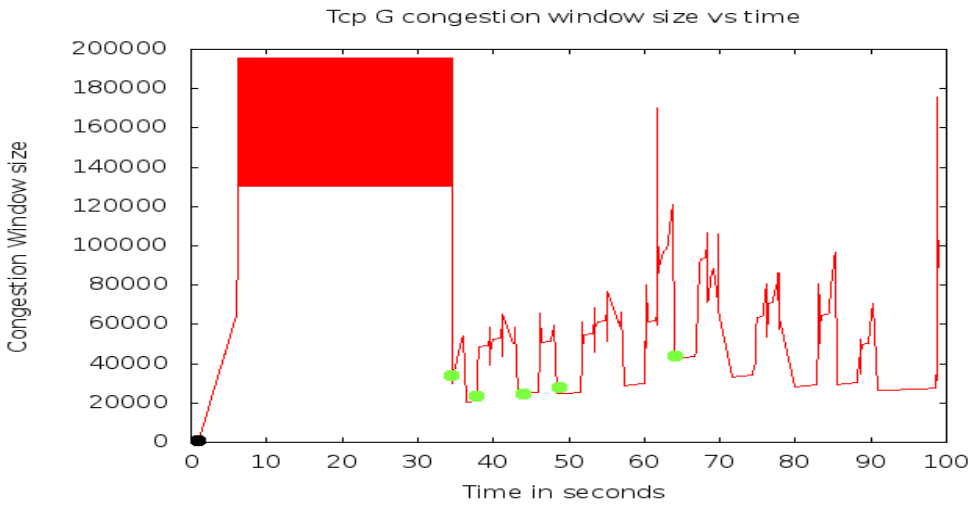
- It starts with a slow start as usual. But , once the threshold is reached, instead of increasing the window linearly, we still increase it exponentially but with a smaller factor.
  - We make n_cwnd=min(RWS,SWS, n_cwnd + n_cwnd/2)
  - Instead of 2 times window we make it 1.5 times.
  - This makes sure we reach the limit really really fast.
- In case of a receiving 3 duplicate acks, instead of making the threshold half, we make it 3/4th. This will work better in scenarios where the congestion doesn't change much. (It increases or decreases only by a little amount). If congestion is not peaky or bursty, this method will be very beneficial.
- In case of reno or New Reno, we do slow start when we hit a time out. But in our algorithm, we make threshold = threshold/2. And we are optimistic that the network has recovered. So, its not a direct drop to 1 but an exponential drop. This is again an optimistic view of the network.

We found that this method gave considerably higher throughput than Reno,New Reno and Tahoe. It gave more than 10% better throughtput.

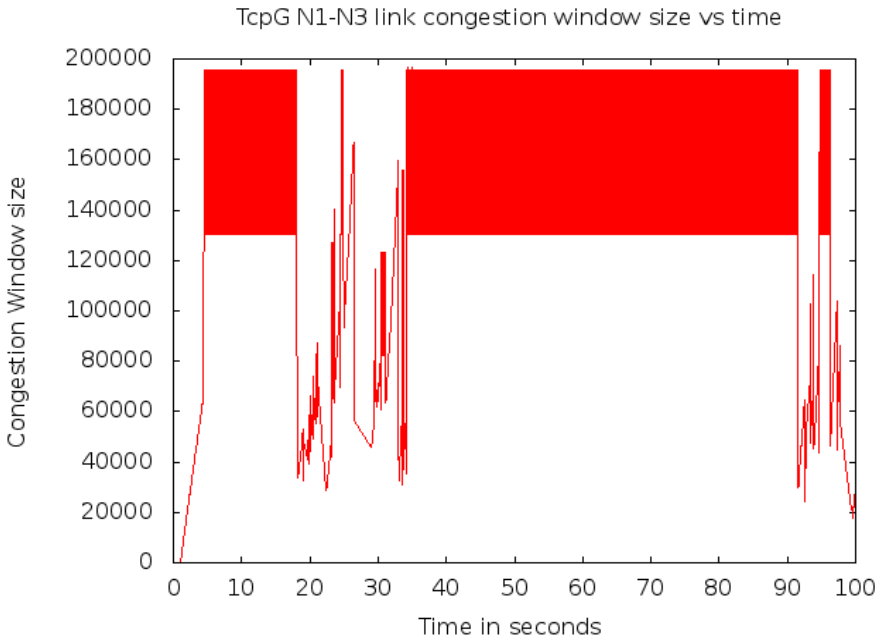
This algorithm instead of increasing and then falling back to 1 like Tahoe, it fluctuates between 2 non zero value.

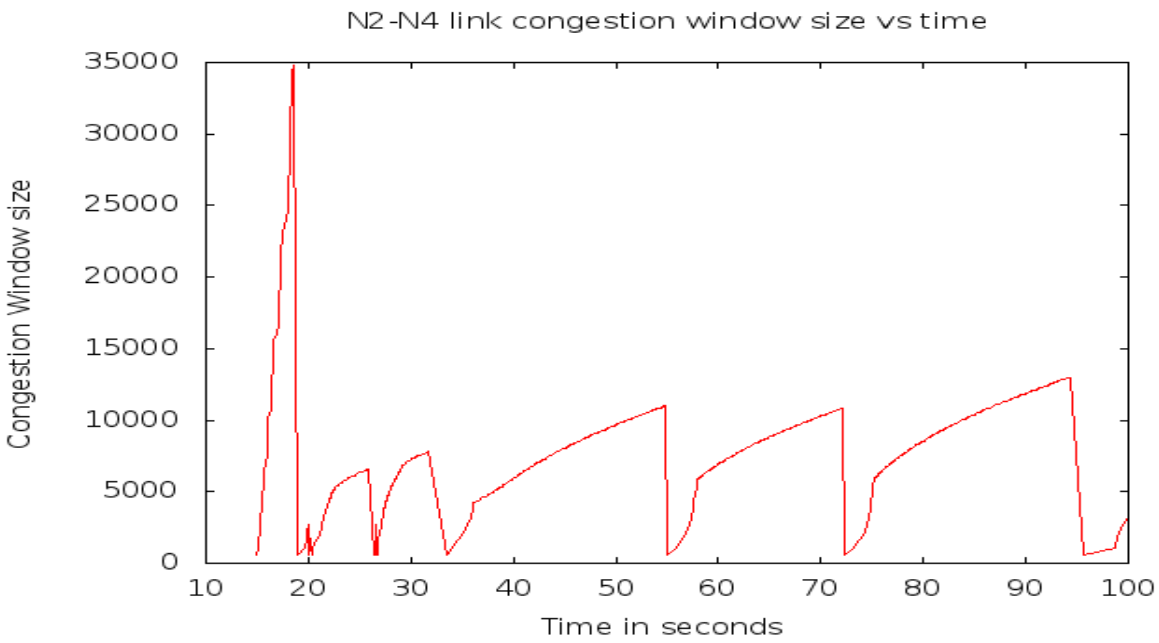
The image clearly shows that there are no other slow start happening other than the start. This also shows that, a slow start everytime was a over kill and that this new Tcp with exponential decrease is a really good way. (In tahoe we did a sudden decrease to 1 and increased exponentially, instead here, we would be decreasing exponentially in case of a time out or 3 duplicated acks).

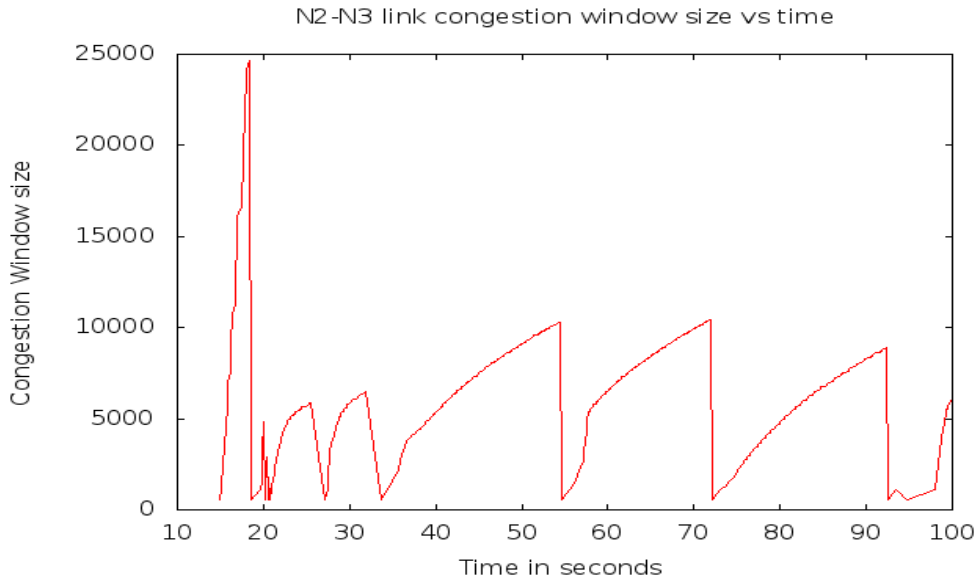
It initially shows an exponential increase and reaches its maximum capacity in no time. So we always increase exponentially. When ever congestion occurs, we reduce the threshold to just ¾ and not ½.



As expected TcpG performs aggressively and gives 50% better throughput that TCP Reno in the second part of the experiment. The through puts of TCP New Reno and Tahoe were decreased by around 30% and 40% respectively.

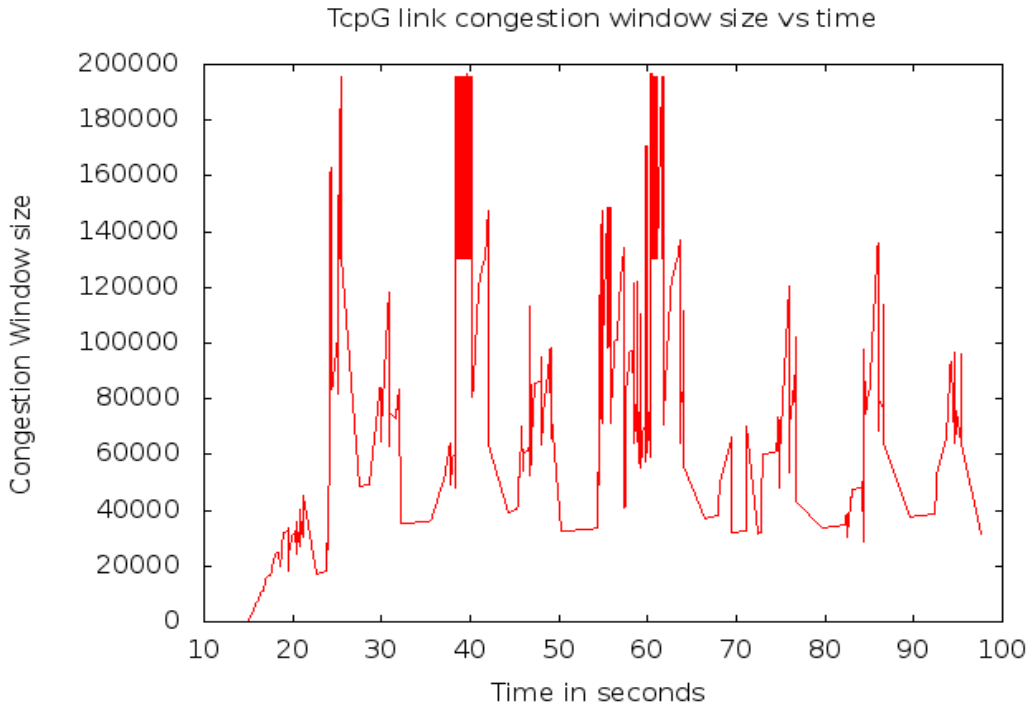






The real test comes when all the three applications running are our Tcp-G.

The following is the graph.



When all the applications were chosen as Tcp-G, It performed better than the cases when all the three cases were Tcp Tahoe or Tcp Reno. This protocol also gave a uniform distribution of resources across the different applications.

References:

- Sample code 2 and 1
- https://www.nsnam.org/docs/release/3.12/models/html/tcp.html - for learning how to bind different tcp variants to different nodes
- http://evanfarrer.blogspot.in/2011/07/using-gnuplot-to-make-simple-line.html for gnu plot
- Wikipedia
- http://inst.eecs.berkeley.edu/~ee122/fa05/projects/Project2/SACKRENEVEGAS.pdf