INDIAN INSTITUTE OF TECHNOLOGY, DELHI

REPORT

Assignment 3

ARNAV TULI | ENTRY NO. 2019CS10424

Course - COL334

Prof. Abhijnan Chakraborty Prof. Huzur Saran

Due October 26, 2021

Contents

Chapter 1

Congestion Control in TCP

I analysed the congestion window (cwnd) size vs time for the following four versions of TCP:

- TCP NewReno
- TCP Highspeed
- TCP Veno
- TCP Vegas

The network configuration (apart from TCP version) was fixed in the experimentation to the following setting:

- **Application Data Rate:** 1 Mbps
- **Packet Size:** 3000 bytes
- **Channel Data Rate:** 8 Mbps
- **Propagation Delay:** 3 ms
- **Error Model:** Rate Error Model
- **Link:** Point-to-Point (TCP source and TCP sink)

The experiment was performed using *First.cc* file (and *plot.py* for plotting), which draws a major portion from *sixth.cc* file (bundled with ns-3.29). The instructions to execute the code are given below:

1. Change directory to ns-allinone-3.29/ns-3.29
2. Copy *First.cc* to *scratch* sub-folder
3. Execute command: `./waf –run "scratch/First –tcpVersion=0"` (TCP NewReno)
4. Execute command: `./waf –run "scratch/First –tcpVersion=1"` (TCP Highspeed)
5. Execute command: `./waf –run "scratch/First –tcpVersion=2"` (TCP Veno)
6. Execute command: `./waf –run "scratch/First –tcpVersion=3"` (TCP Vegas)
7. Plot command: `python plot.py <csv_filename>`

In the next four sections, I present my plots and observations.

1.1 TCP NewReno

Number of packets dropped: 38
Number of packets sent by application: 1209

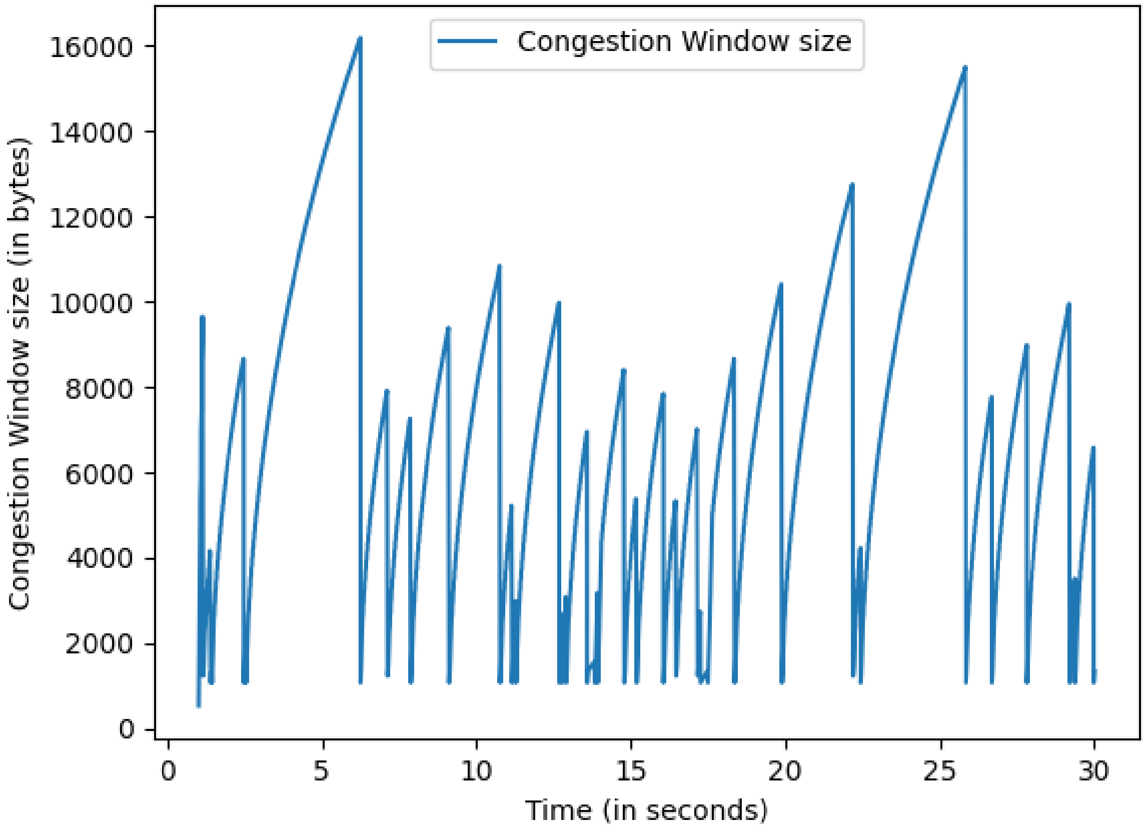

Figure 1.1: Congestion Window (cwnd) size vs time for TCP NewReno

Observations: Initially TCP NewReno follows the **slow start** algorithm as the congestion window size is smaller than default *ssthresh* value (which is 4294967295 bytes in ns-3.29). Whenever a successful ACK is received, the window size is increased by 1 MSS (536 bytes in ns-3.29). This continues till window size = 9648 bytes (18 MSS) at t = 1.12782 seconds, after which it observes a packet drop (triple duplicate ACKs) and the congestion window (and ssthresh) is reduced to a value of 1232 bytes. After this the **congestion avoidance** phase of TCP NewReno starts, where the window size is increased by $\frac{MSS^2}{cwnd}$ bytes for every successful ACK. This leads to the non-linear curves in the graph. Since there are no timeouts, every packet drop is indicated by triple duplicate ACKs, hence, the connection never goes back to *slow-start* phase. Also, the formula used for calculating *ssthresh* (on non-timeout congestion events) is $max(2 * \text{MSS}, 0.5 * \text{bytes in flight})$, which translates to $max(1072, 0.5 * \text{bytes in flight})$. This explains why *ssthresh* and congestion window size never drop below 1072 bytes after the first packet drop in the graph.

1.2 TCP Highspeed

Number of packets dropped: 38
Number of packets sent by application: 1209

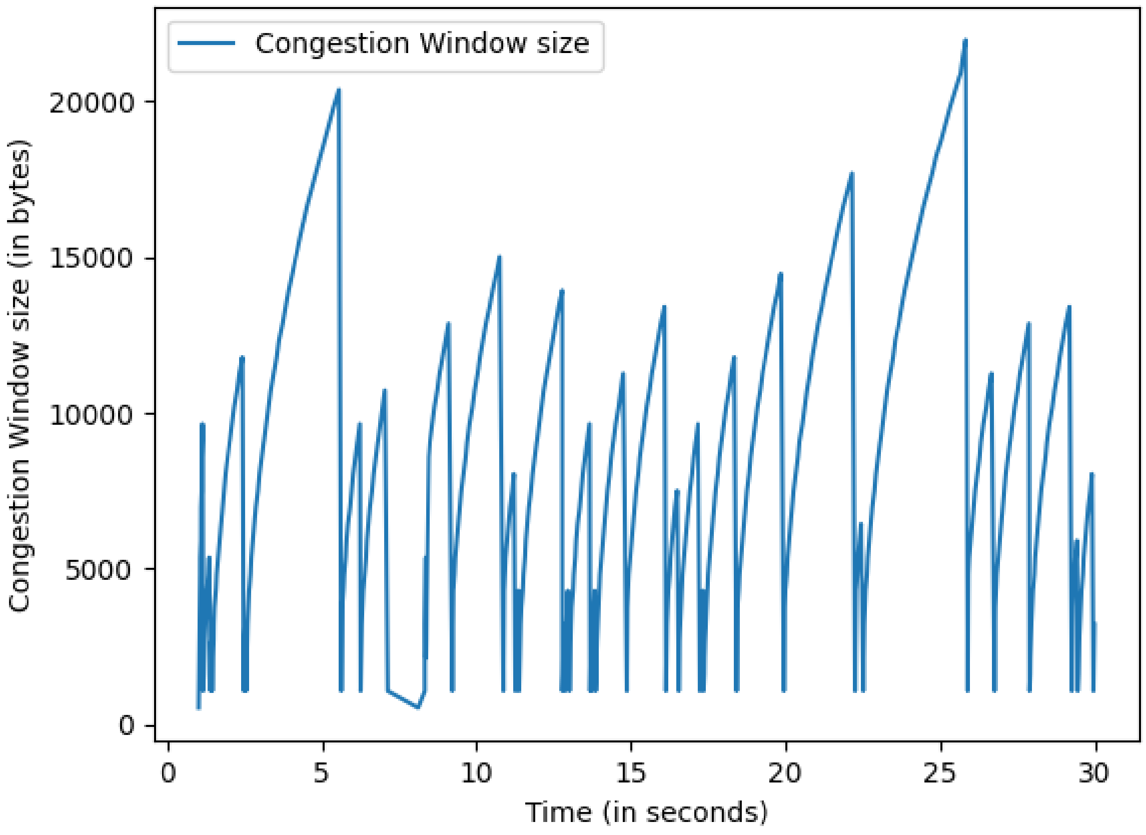

Figure 1.2: Congestion Window (cwnd) size vs time for TCP Highspeed

Observations: The **slow start** phase of TCP HighSpeed is the same as that of TCP NewReno, and the congestion window size is increased by 1 MSS (536 bytes) for every successful ACK. At t = 1.12782, the connection experiences a packet drop (triple duplicate ACKs), and HighSpeed shifts to its **congestion avoidance** phase. In this phase, rather doing an increase of $\frac{MSS^2}{cwnd}$ bytes for every successful ACK, HighSpeed does an increment of $a(\frac{cwnd}{MSS})\frac{MSS^2}{cwnd}$ bytes per successful ACK. Here 'a' is a function implemented as a lookup table in the protocol specifications. For small values of $\frac{cwnd}{MSS}$, the function value is simply '1' and HighSpeed behaves just like NewReno, giving similar shaped non-linear curves. However, when $\frac{cwnd}{MSS} > 38$, which is the case at around 25.7294 seconds, the function value becomes '2' and the increment per ACK increases. This leads to a change in the curvature, which can be observed in the graph (around t=25s). Also, there is a single timeout event that occurs at t = 8.10441 seconds, and the protocol shifts back to **slow start** phase, till the next packet drop. For completeness, the formula used for calculating *ssthresh* (on non-timeout congestion events) is $max(2 * \text{MSS}, cwnd * (1 - b(\frac{cwnd}{MSS})))$, which translates to $max(1072, cwnd * (1 - b(\frac{cwnd}{MSS})))$ ('b' is again a function implemented as a lookup table in the protocol specifications). This explains why *ssthresh* and congestion window size never drop below 1072 bytes on non-timeout packet drops.

1.3 TCP Veno

Number of packets dropped: 38
Number of packets sent by application: 1209

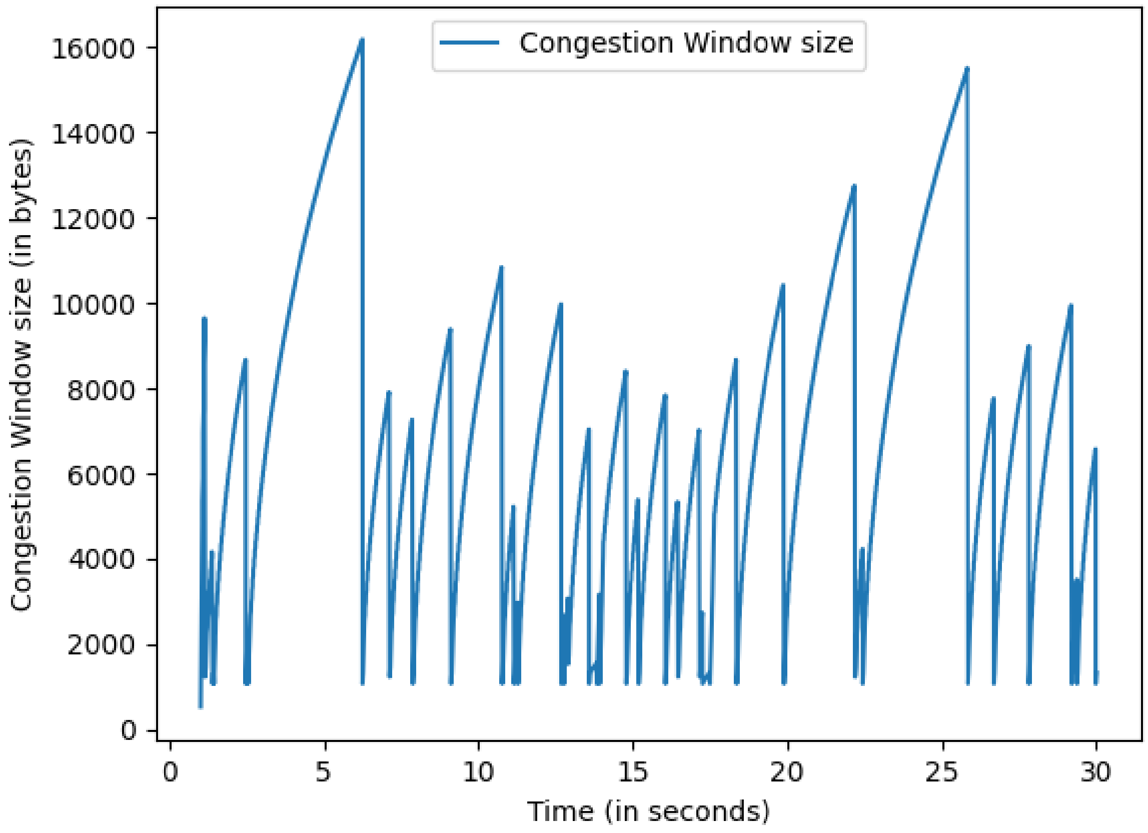

Figure 1.3: Congestion Window (cwnd) size vs time for TCP Veno

Observations: TCP Veno uses delay-based information to determine how to increase its congestion window. Initially, TCP Veno follows the same **slow start** phase like TCP NewReno till t = 1.12782 after which it experiences a packet loss (triple duplicate ACKs) and reduces its congestion window (and *ssthresh*) to 1232 bytes. The protocol then shifts to its **congestion avoidance** phase, which in low congestion environments is exactly the same as in TCP NewReno. TCP Veno tries to determine the number of backlog packets using the formula $Diff * BaseRTT$, where $Diff = \frac{cwnd}{BaseRTT} - \frac{cwnd}{RTT}$. If $Diff * BaseRTT < \beta$ (β is protocol parameter), then TCP

Veno increments the congestion window by $\frac{MSS^2}{cwnd}$ bytes for every successful ACK (like NewReno). Otherwise, the congestion window is incremented by $\frac{MSS^2}{cwnd}$ bytes for every *other* successful ACK (non-consecutive). In our situation, involving only two nodes, the latter condition is not met and the protocol behaves like TCP NewReno. This is also observed from the graph, in the form of similar non-linear curves. One way in which TCP Veno is different from NewReno even in low-congestion settings is the way in which *ssthresh* is set. The formula used for calculating *ssthresh* (on non-timeout congestion events) is $max(2 * \text{MSS}, 0.8 * \text{bytes in flight})$, which translates to $max(1072, 0.8 * \text{bytes in flight})$ (0.8 instead of 0.5 (NewReno)). Due to this, the congestion window size on packet drop can be different, and from the data, this difference can be observed at t = 12.9118 seconds, where congestion window size is 1542 bytes for TCP Veno and 1072 bytes for TCP NewReno. Similar to before, this formula also explains why *ssthresh* and congestion window size never drop below 1072 bytes on non-timeout packet drops.

1.4 TCP Vegas

Number of packets dropped: 39
Number of packets sent by application: 1209

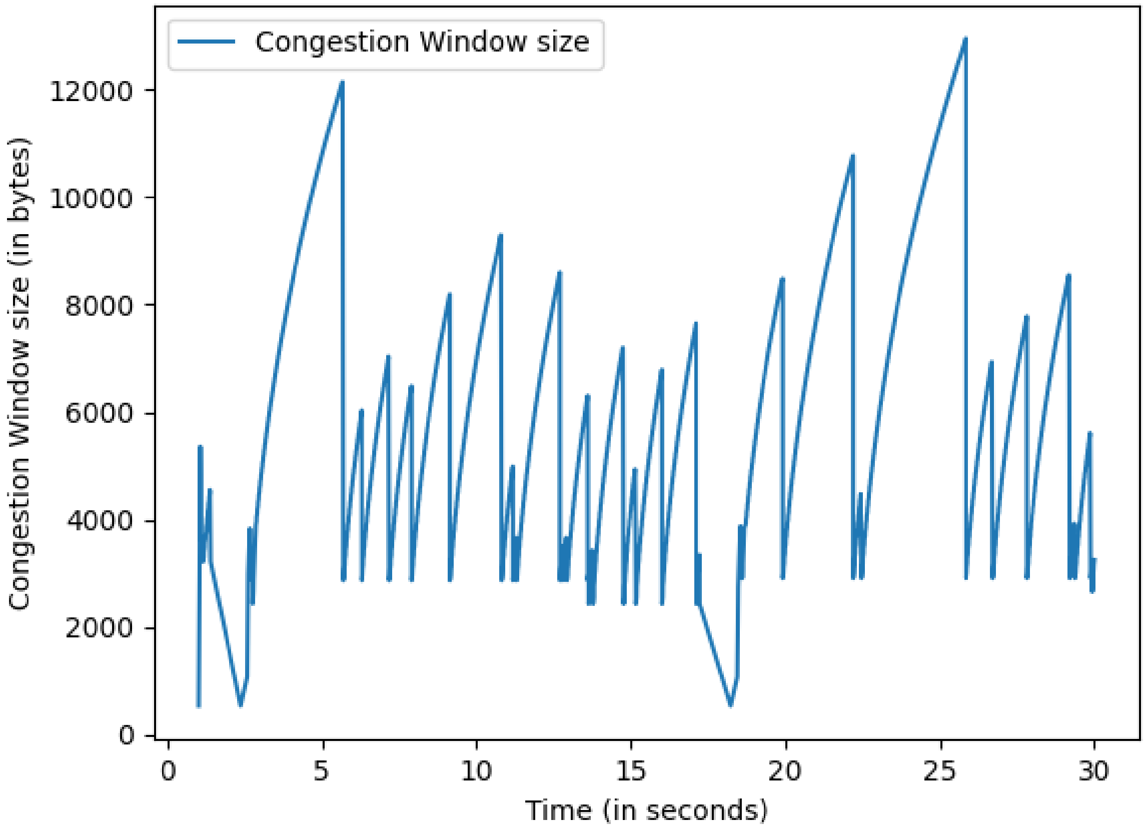

Figure 1.4: Congestion Window (cwnd) size vs time for TCP Vegas

Observations: TCP Vegas is a pure delay-based congestion control protocol, and tries to avoid packet drops by reducing its congestion window before actually detecting a packet drop. It uses the same mechanism as TCP Veno to determine backlog packets (using $Diff * BaseRTT$). It also uses three parameters γ, α and β and tries to keep $Diff * BaseRTT$ in between α and β. Initially, Vegas starts with the standard **slow start** phase and increments congestion window by 1 MSS for every successful ACK. Observe that while in slow-start phase at t = 1.07864, the network does not experience any packet drop, but Vegas still reduces its window size to 3752 bytes from 5360 bytes, because $Diff * BaseRTT > \gamma$. At the same time it also sets the new *ssthresh* to be $3752 - MSS = 3216$ bytes. This is one major way in which Vegas is different from other variants. Hence, when the network actually experiences packet drop for the first time at t = 1.15065 seconds, the congestion window reduces to 3216 bytes rather than 1232 bytes as in previous sections. After this point, Vegas follows standard **congestion avoidance** and switches between congestion avoidance phase and slow start phase (on timeout), just like any other TCP protocol. The main difference in TCP Vegas is that it reduces the congestion window if $Diff * BaseRTT > \beta$ (in congestion avoidance) and if $Diff * BaseRTT > \gamma$ (in slow start), which is indicated as going *too fast* in Vegas jargon. At the same time, if the bandwidth is underutilized (going *too slow*) or $Diff * BaseRTT < \alpha$, the congestion window is increased. Other facts that I can observe from the graph is that there are two packet timeouts, which happen at t = 2.36901 and t = 18.2307 seconds. In this case, the congestion window is reset to 1 MSS (536 bytes), and the **slow start** phase is initiated (while cwnd < ssthresh). When $cwnd \geq ssthresh$, the protocol changes its phase to the standard **congestion avoidance** phase, subject to α and β. One last major observation is that how Vegas handles it *ssthresh* value. It uses two formulae. One is $ssthresh = max(ssthresh, 0.75 * cwnd)$, which is used whenever (not necessarily on congestion events) Vegas has enough RTT samples. Second one is $ssthresh = max(min(ssthresh, cwnd - MSS), 2 * MSS)$. This one is used when non-timeout congestion events occur. The way in which its effect can be observed is by noticing that even on packet drops (due to triple duplicate ACKs), the congestion window is mostly being reduces to a value above 2000 bytes, whereas in previous TCP versions, the window was being reduced to around 1072 bytes mostly.

Packet Drops and other observations:

- The number of packet drops for each protocol were almost similar (38, 38, 38, 39). Since, we are fixing the application and channel data rate and using the same error model, with the same erorr rate, we can expect the packet drops to be almost the same, as all the protocols implement the standard TCP version with some changes: using weights in HighSpeed, using Delay based estimates in Veno and Vegas. Veno and HighSpeed were observed to behave like NewReno for the majority of time (as far as algorithms used are concerned). Hence, it is not unexpected that they have identical packet drops in the current situation. Since, Vegas was the most different protocol from these (pure delay based), it somehow had 1 more packet drop than others (39). This can possibly be due to the fact that the minimum congestion window size is higher in Vegas, and hence, the node is able to send more packets, thus, resulting in more packet drops. (RateErrorModel is stochastic and uses a probabilty to drop a received packet. Hence, more packets sent/received imply (on average) more packet drops)

- TCP Veno and TCP NewReno have identical maximum congestion window size (16179 bytes), whereas TCP HighSpeed has the highest maximum congestion window size (21976 bytes), among the four versions, which is expected as it is designed for high-capacity channels. On the other hand, TCP Vegas has smallest maximum congestion window size (12955 bytes), as it is the most cautious of them all.

Chapter 2

Effect of Data Rate on Congestion Window

I analysed the congestion window (cwnd) size vs time for different values of application and channel data rate. The network configuration (apart from data rates) was fixed in the experimentation to the following setting:

- **TCP version:** NewReno
- **Packet Size:** 3000 bytes
- **Propagation Delay:** 3 ms
- **Error Model:** Rate Error Model
- **Link:** Point-to-Point (TCP source and TCP sink)

The experiment was performed using *Second.cc* file (and *plot.py* for plotting), which draws a major portion from *sixth.cc* file (bundled with ns-3.29). The instructions to execute the code are given below:

1. Change directory to ns-allinone-3.29/ns-3.29
2. Copy *Second.cc* to *scratch* sub-folder
3. Execute command: `./waf -run "scratch/Second -appDR=a -channelDR=c"`
   (Here 'a' and 'c' are application and channel data rates respectively (in Mbps))
4. Plot command: `python plot.py <csv_filename>`

In the next section, I present my plots and observations.

2.1 Effect of Channel Data Rate

In this part, I fixed the application data rate to 2 Mbps and varied the channel data rate in the list [2, 4, 10, 20, 50]. The plots for each configuration is given below:

Channel Data Rate: 2 Mbps
Number of packets dropped: 62
Number of packets sent by application: 2417

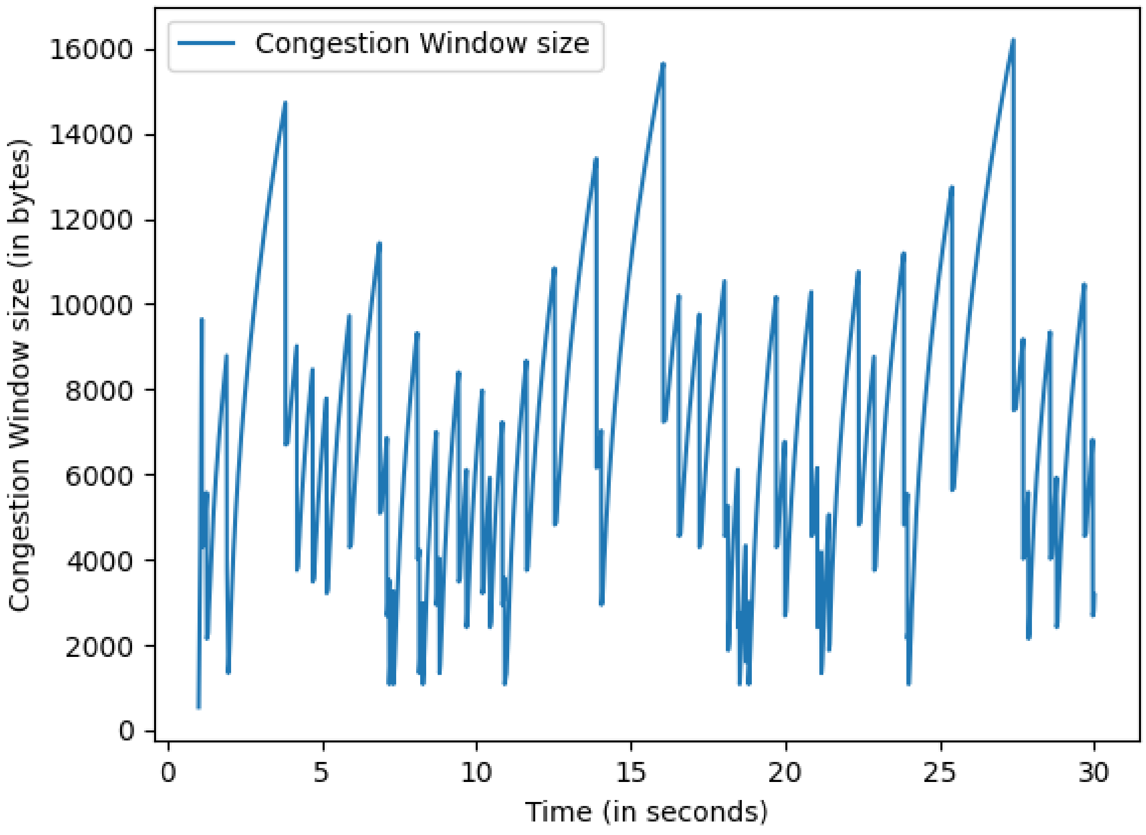

Figure 2.1: Congestion Window (cwnd) size vs time for ChannelDR = 2 Mbps

Channel Data Rate: 4 Mbps
Number of packets dropped: 72
Number of packets sent by application: 2417

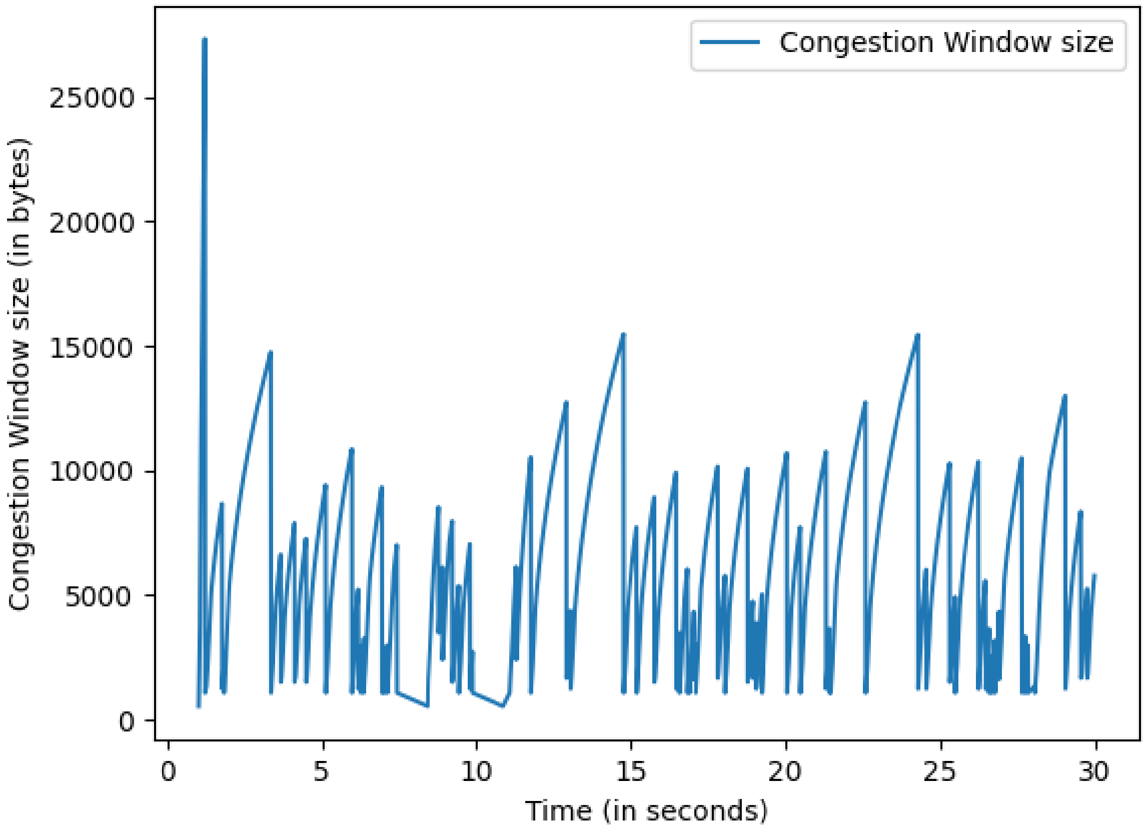

Figure 2.2: Congestion Window (cwnd) size vs time for ChannelDR = 4 Mbps

Channel Data Rate: 10 Mbps
Number of packets dropped: 73
Number of packets sent by application: 2417

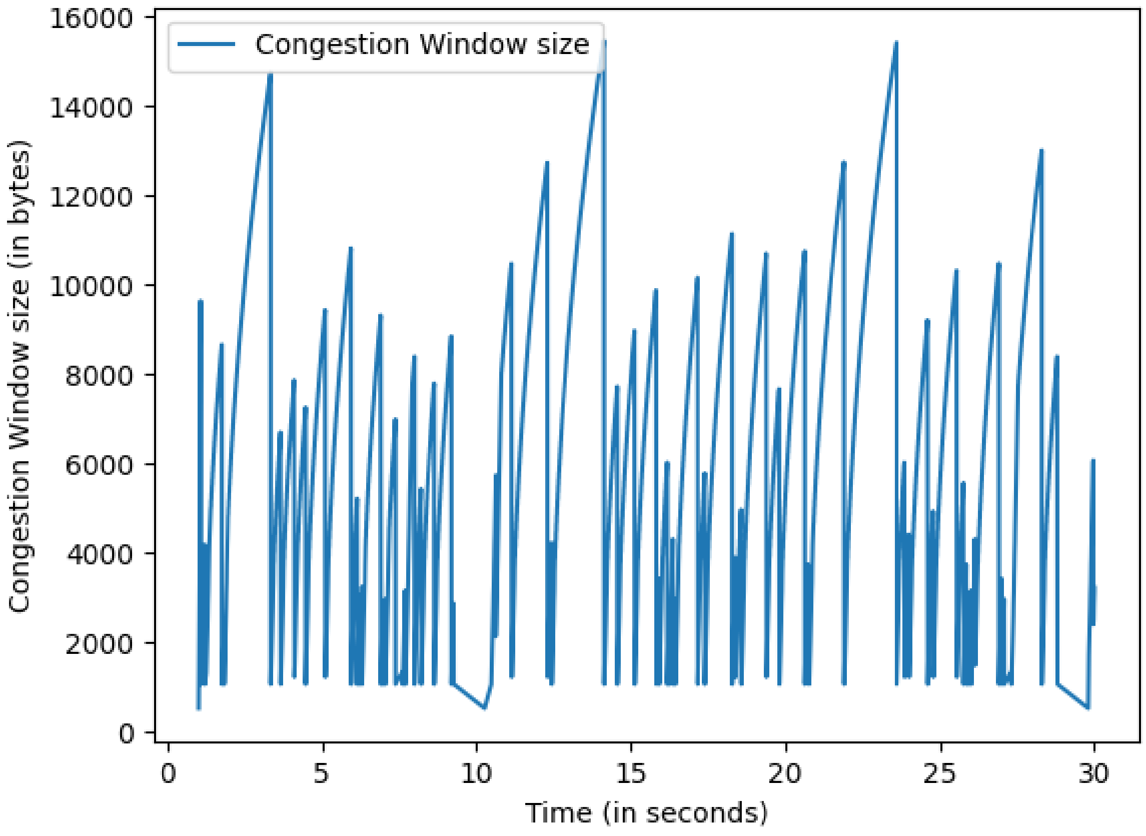

Figure 2.3: Congestion Window (cwnd) size vs time for ChannelDR = 10 Mbps

Channel Data Rate: 20 Mbps
Number of packets dropped: 74
Number of packets sent by application: 2417

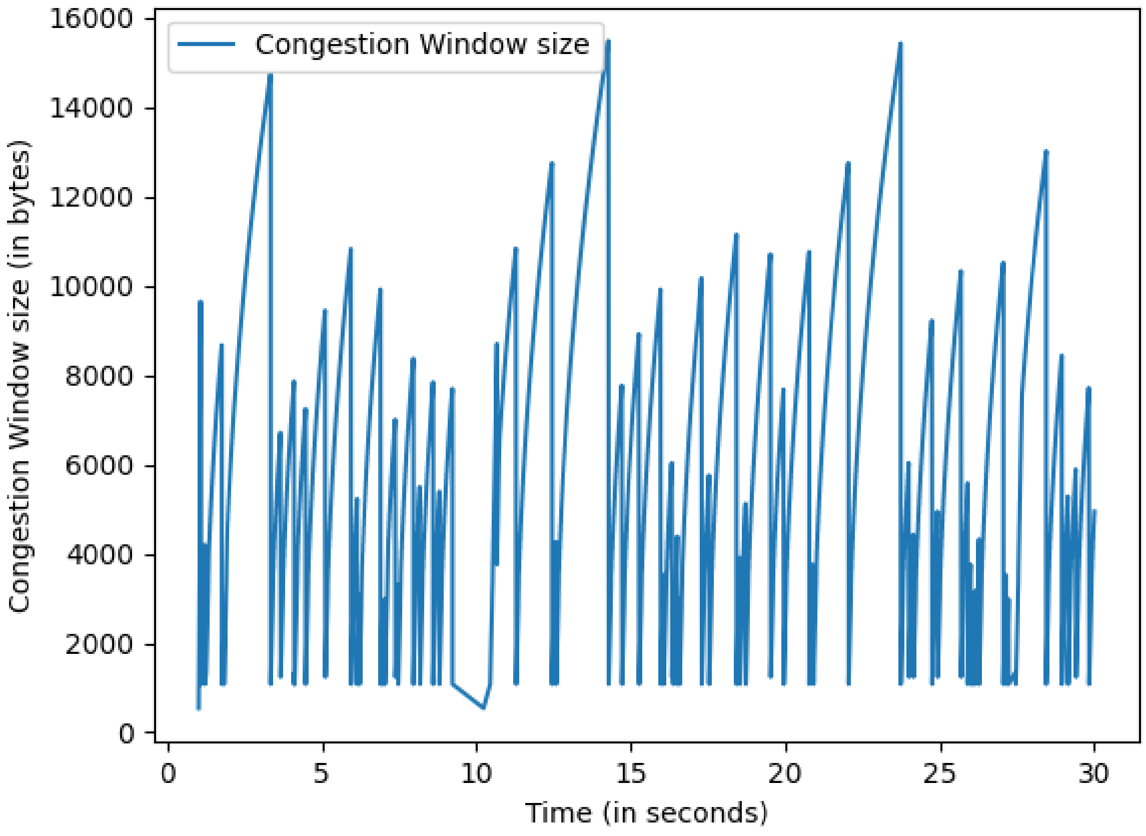

Figure 2.4: Congestion Window (cwnd) size vs time for ChannelDR = 20 Mbps

Channel Data Rate: 50 Mbps
Number of packets dropped: 75
Number of packets sent by application: 2417

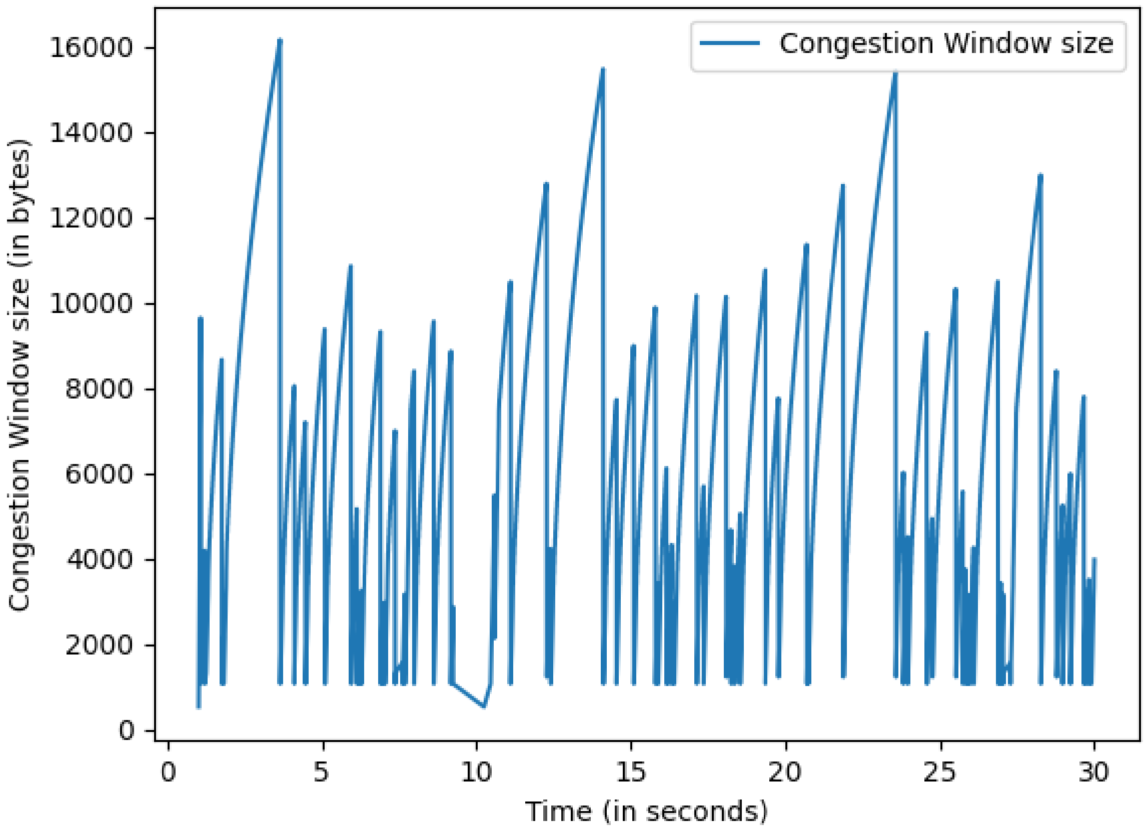

Figure 2.5: Congestion Window (cwnd) size vs time for ChannelDR = 50 Mbps

Trends and Observations:

- First trend is that the number of packets sent by application to the socket is independent of the value of channel data rate. This can be explained as the number of packets sent by the application = $\frac{\text{Time duration (in s)}*\text{App Rate (in bps)}}{\text{Packet size (in bits)}}$. Hence, independence can be easily seen. (as App Rate is fixed to 2 Mbps)
- Second trend is the number of packets that are getting dropped. Now the error model used is the RateErrorModel which is a stochastic model. It uses a probability (or chance) to decide whether or not to drop a received packet. Therefore, we can *expect* more packets to be dropped, if the receiver receives *more* number of packets. So, if I am able to explain why the sender is able to send more packets (or receiver is able to receive more packets), I will be able to justify the increase in number of packet drops. To do this, observe that the RTT for the connection is 6 ms + Transmission Delay at sender (assuming ACK to be small

sized). Since, Transmission Delay = $\frac{\text{MTU (in bits)}}{\text{ChannelRate (in bps)}}$, and MTU is fixed for a given link, it can be seen that as we increase the channel data rate, the transmission delay and hence the RTT will decrease. The decrease can be significant as the transmission delay for naked maximum segment (536 bytes) is 2.144 ms for ChannelRate = 2 Mbps, which is comparable to the propagation delay (=3 ms). Now, the application will send the packet to the socket at every $\frac{3000*8}{\text{AppRate (in bps)}}$ seconds, which is fixed, as the application data rate is fixed (= 2 Mbps). Hence, the bandwidth (channel) utilization will increase with decrease in transmission delay and consequently RTT. This is because, the sender is able to receive ACKs from receiver at a faster rate as RTT decreases. Also, more ACKs imply more updates (increments) to the congestion window size. This in turn allows the sender to send much more data (packets) to the receiver, and hence, getting much more ACKs and so on. Therefore, increase in number of packets that are sent to the receiver with increase in channel data rate is reasonable and can be explained using the above logic. Now, more packets sent/received directly imply more packet drops, as the error model is RateErrorModel (stochastic). Hence, the trend is explained.

- One more interesting thing that can be observed is that the increase in packet drops decreases as we increase the channel rate. For example there was an increase of 10 dropped packets from ChannelRate = 2 Mbps to ChannelRate = 4 Mbps. After that there was only an increase of 1 dropped packet (72 to 73, 73 to 74 and 74 to 75). This can be explained as, the change in Transmission delay becomes smaller as we increase the channel data rate. For ChannelRate $\geq$ 4 Mbps, the delay will decrease by less than 1 ms for packet size = 536 bytes. Hence, we can see that the RTT will converge for large values of data rate (to 6 ms), and the ACKs will be received at almost the same time. Hence, the congestion window updates will become identical, and so will the number of packets sent and packets dropped. This can also be seen from graphs, as the graphs for ChannelRate = 20 Mbps and ChannelRate = 50 are almost identical in shape.
- Third and last trend is the nature of the graph. In all cases, I have used TCP NewReno, hence, they all follow the standard **slow start** and **congestion avoidance** phase, and have similar curves and peak congestion window size (except for 4 Mbps as peak is around 25000 bytes during slow start phase). However, as the channel data rate is increased, the graphs shrink and later converge (see point above). This can be explained from point above. As we are increasing the channel data rate, the number of packets sent is increasing. Hence, the sender is receiving more ACKs per second and is therefore, able to increase its congestion window size faster. Hence, if the sender took 3.6 seconds to reach a congestion window size of 14000 bytes when channel data rate was 2 Mbps, the sender only took 3.1 seconds to reach the same peak for channel rate of 4 Mbps and so on. This difference will converge as we increase the channel data rate (and RTT becomes same), but this observation helps us to reason why the graphs are shrinking and becoming more dense.
- Some observations. No timeout occurs for channel data rate = 2 Mbps. '2' timeouts occur for ChannelRate = 4 Mbps and ChannelRate = 10 Mbps and '1' timeout occurs for ChannelRate = 20 Mbps and ChannelRate = 50 Mbps.

2.2 Effect of Application Data Rate

In this part, I fixed the channel data rate to 6 Mbps and varied the application data rate in the list [0.5, 1, 2, 4, 10]. The plots for each configuration is given below:

Application Data Rate: 0.5 Mbps
Number of packets dropped: 22
Number of packets sent by application: 605

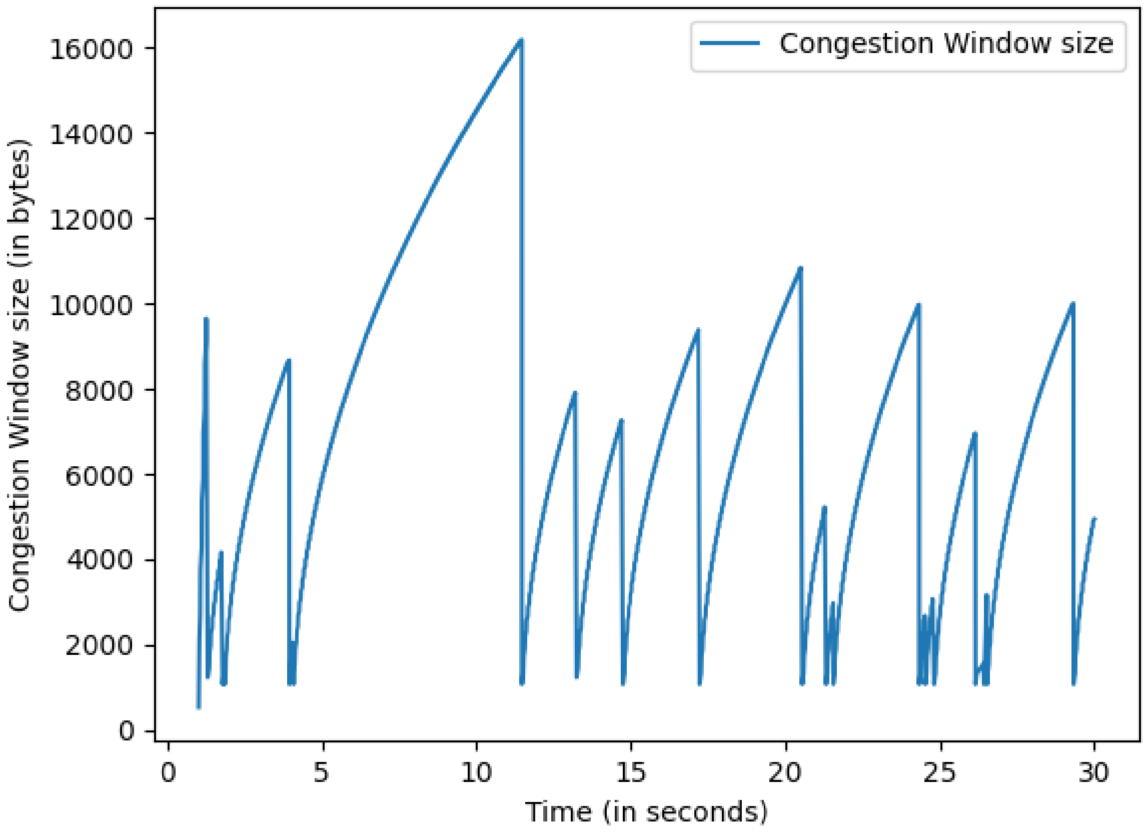

Figure 2.6: Congestion Window (cwnd) size vs time for AppDR = 0.5 Mbps

Application Data Rate: 1 Mbps
Number of packets dropped: 38
Number of packets sent by application: 1209

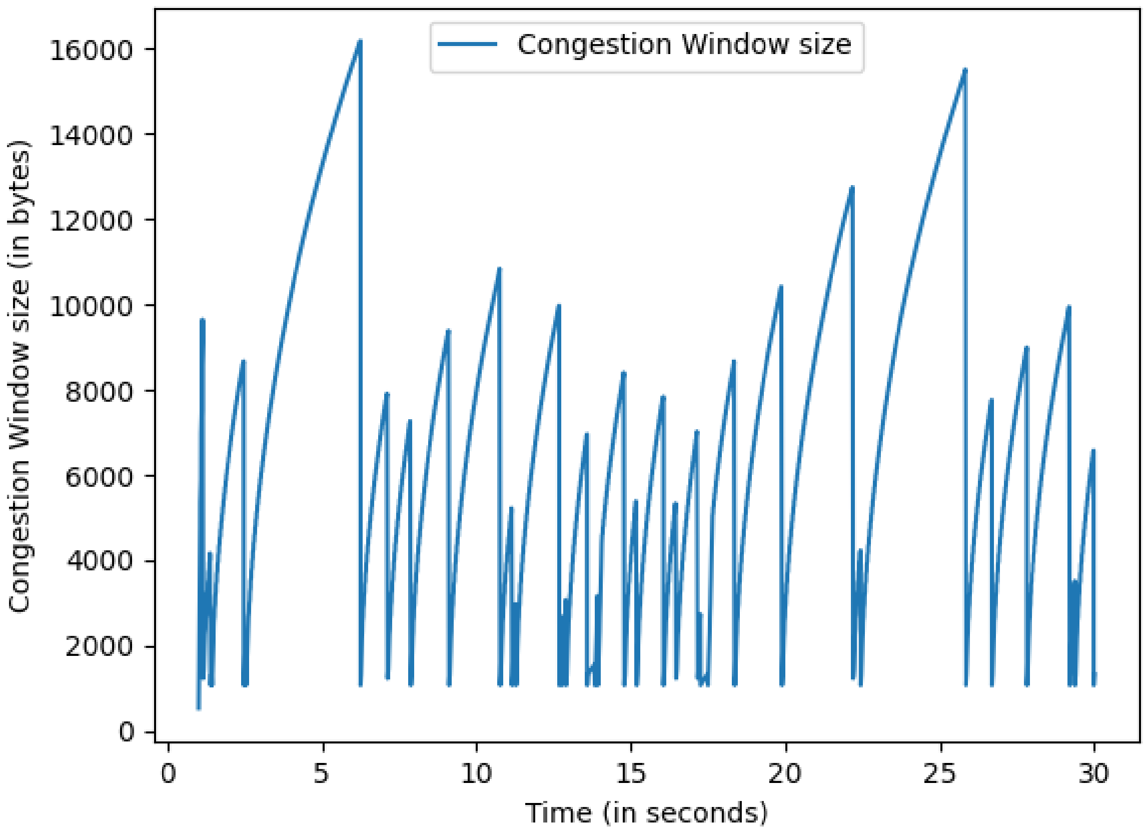

Figure 2.7: Congestion Window (cwnd) size vs time for AppDR = 1 Mbps

Application Data Rate: 2 Mbps
Number of packets dropped: 71
Number of packets sent by application: 2417

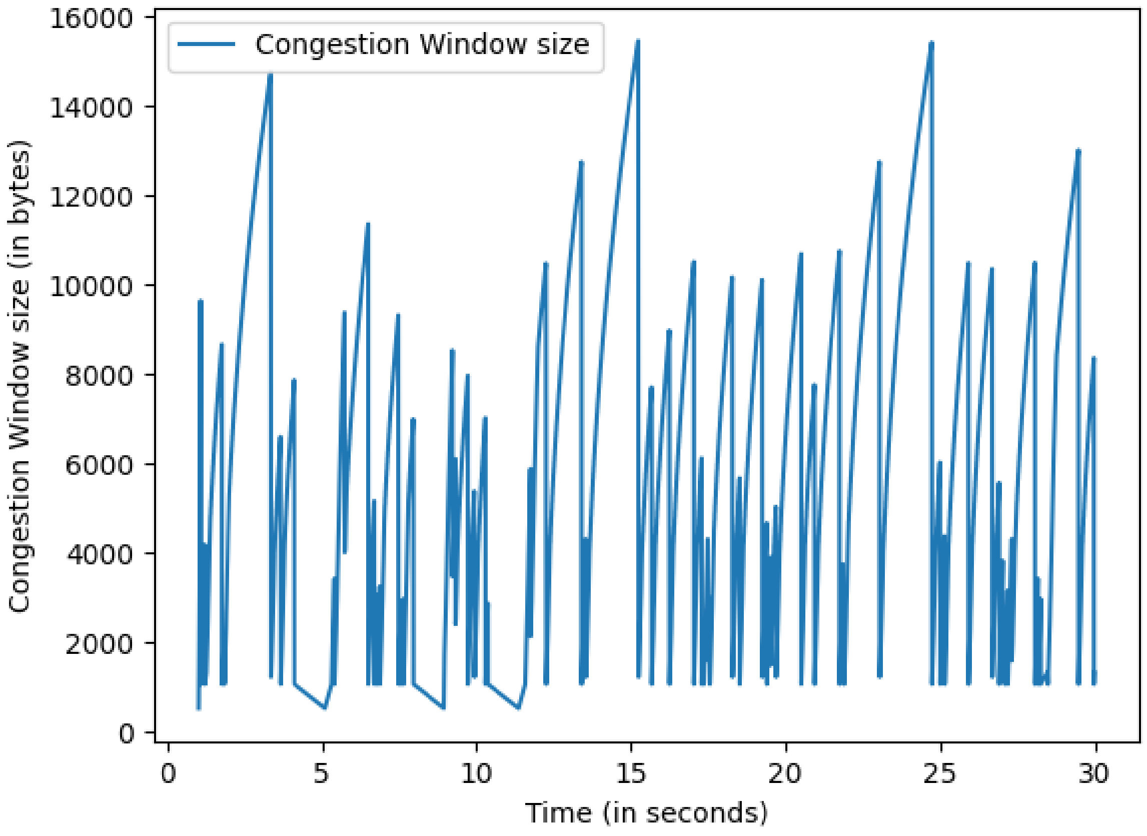

Figure 2.8: Congestion Window (cwnd) size vs time for AppDR = 2 Mbps

Application Data Rate: 4 Mbps
Number of packets dropped: 156
Number of packets sent by application: 4834

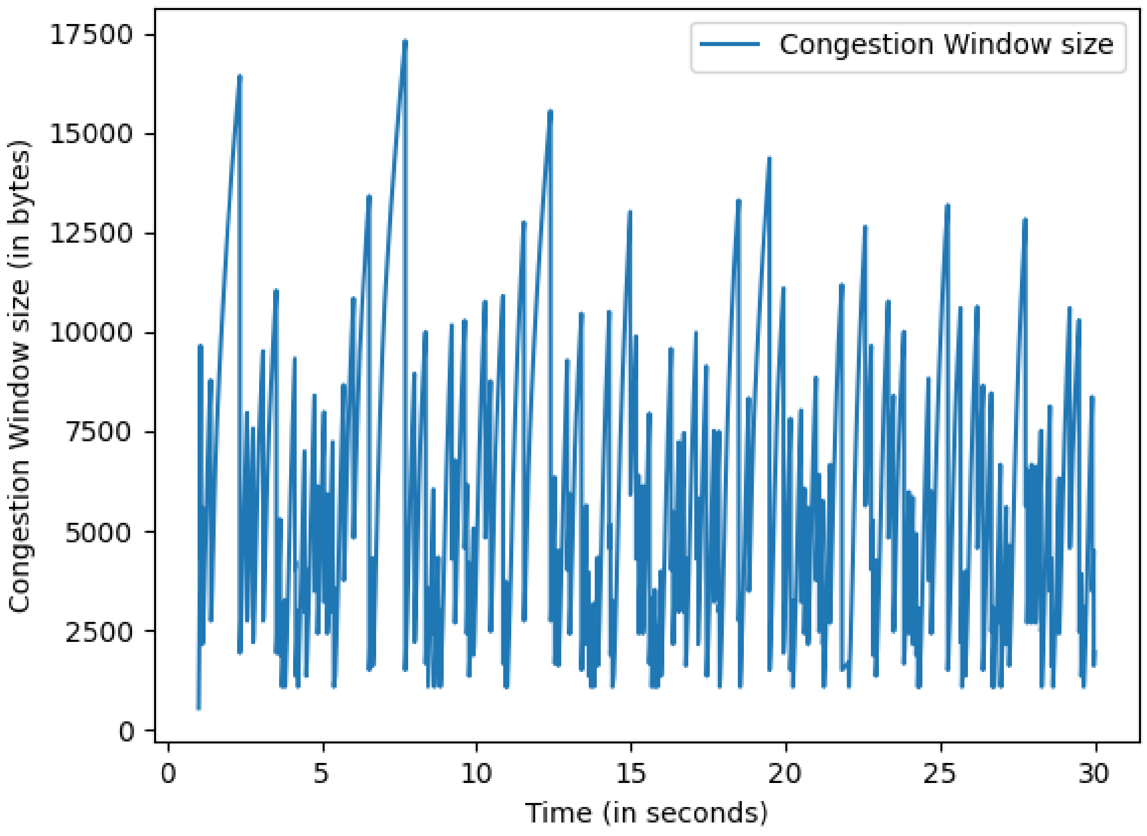

Figure 2.9: Congestion Window (cwnd) size vs time for AppDR = 4 Mbps

Application Data Rate: 10 Mbps
Number of packets dropped: 156
Number of packets sent by application: 12084

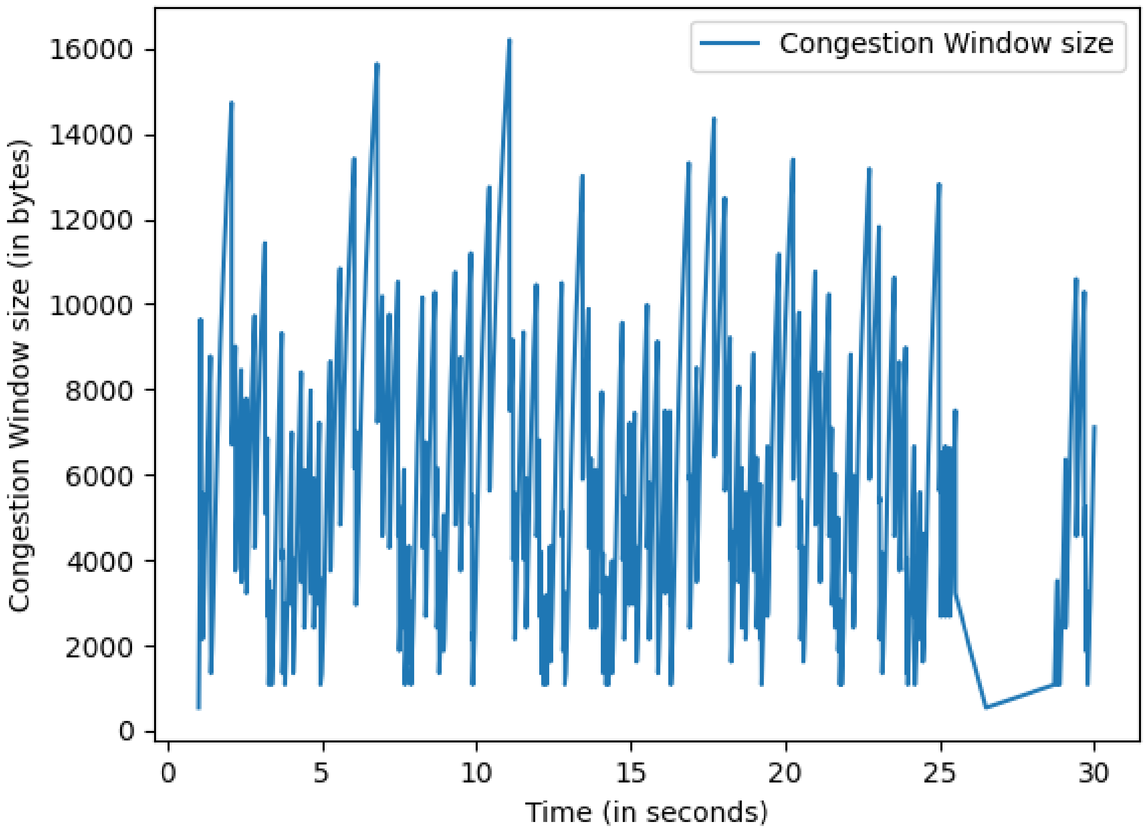

Figure 2.10: Congestion Window (cwnd) size vs time for AppDR = 10 Mbps

Trends and Observations:

- First trend is that as I increase the application data rate, the corresponding number of packets sent by application to the socket increases proportionately. This can be explained as the number of packets sent by the application = $\frac{\text{Time duration (in s)}*\text{App Rate (in bps)}}{\text{Packet size (in bits)}}$. Hence, direct relation can be easily seen.
- Second trend is the number of packets that are getting dropped. Now the error model used is the RateErrorModel which is a stochastic model. It uses a probability to decide whether or not to drop a received packet. Therefore, we can *expect* more packets to be dropped, if the receiver receives more number of packets. So, if I am able to explain why the sender is able to send more packets (or receiver is able to receive more packets), I will be able to justify the increase in number of packet drops. To do this, observe that the RTT for the connection is 6 ms + Transmission Delay at sender (assuming ACK to be small sized). Since,

Transmission Delay only depends on packet size and channel data rate, it can be taken to be fixed for our analysis ($\leq$ 2 ms (as MTU is around 1500 bytes)). Now, the application will send the packet to the socket at every $\frac{3000*8}{\text{AppRate (in bps)}}$ seconds, which is 48 ms, 24 ms, 12 ms, 6 ms and 2.4 ms for application data rates of 0.5 Mbps, 1 Mbps, 2 Mbps, 4 Mbps and 10 Mbps respectively. Now , notice that for data rates $\leq$ 4 Mbps, the rate at which the socket receives the data is lower than the rate at which it receives ACK from receiver. Hence, the sender is unable to fully utilize the increased congestion window (on ACK) in case of app data rates of 0.5 Mbps, 1 Mbps and 2 Mbps. Also, this utilization increases as app data rate increases, as the data is received more frequently by the socket. Now, more utilization of congestion window means more data/packets sent through the channel, hence, we can explain the increase in packet drops. Now, one last thing that remains to explain is that why the number of packet drops is the same for data rate = 4 Mbps and 10 Mbps. This is because in both the cases, the socket is receiving application data at a higher rate than at which it receives ACK from receiver. Hence, the congestion window is fully utilized in both the cases, and the excess data is simply stored in the buffer. Therefore, the number of packets sent/received in these two cases are about the same. Hence, number of packets drops are identical (property of RateErrorModel). I tested out this hypothesis by running an application which used 5 Mbps as the data rate. In that case also, the total packet drops were 156. (full utilization)

- Third and last trend is the nature of the graph. In all cases, I have used TCP NewReno, hence, they all follow the standard **slow start** and **congestion avoidance** phase, and have similar curves and peak congestion window size. However, as the application data rate is increased, the graphs shrink. This can be explained from point above. As we are increasing the application data rate, the number of packets sent is increasing. Hence, the sender is receiving more ACKs per second and is therefore, able to increase its congestion window size faster. Hence, if the sender took 10 seconds to reach a congestion window size of 16000 bytes when application data rate was 0.5 Mbps, the sender only took 5 seconds to reach the same peak for app rate of 1 Mbps, and only 2.5 seconds for app rate of 2 Mbps and so on. Hence, this shrinking of graph can be explained using the fact that congestion window size is updated much more frequently for high application data rates.
- Some observations. No timeout occurs for application data rates 0.5 Mbps, 1 Mbps and 4 Mbps. '3' timeouts occur for AppRate = 2 Mbps and '1' timeout occurs for AppRate = 10 Mbps.

Chapter 3

Custom TCP Congestion Control

I implemented a custom variant of TCP by inheriting base class:-TcpNewReno and over-riding the SlowStart and CongestionAvoidance methods with custom updates (as specified in the assignment).
I analysed three configurations of three connections involving three nodes (two TCP sources and one TCP sink). The connection attributes (duration, data rate) and network topology were fixed as per the assignment specification across different configurations.
The experiment was performed using *Third.cc* file (and *plot.py* for plotting), which draws a major portion from *sixth.cc* file (bundled with ns-3.29). Also, TCPNewRenoCSE.cc and TCPNewReno.h (containing custom implementation) inherit code from tcp-congestion-ops.cc and tcp-congestion-ops.h.
The instructions to execute the code are given below:

1. Change directory to ns-allinone-3.29/ns-3.29
2. Copy TCPNewRenoCSE.cc and TCPNewRenoCSE.h to src/internet/model sub-folder
3. Add TCPNewRenoCSE.cc file to 'wscript' present in src/internet sub-folder
4. Copy *Third.cc* to *scratch* sub-folder
5. Execute command: `./waf -run "scratch/Third -config=c"`
   (Here 'c' is the network configuration (1, 2 or 3))
6. Plot command: `python plot.py <csv_filename>`

In the next three sections, I present my plots and observations.

3.1 Configuration 1

In this configuration, all senders use TCP NewReno.
Connection: 1
Number of packets dropped: 31

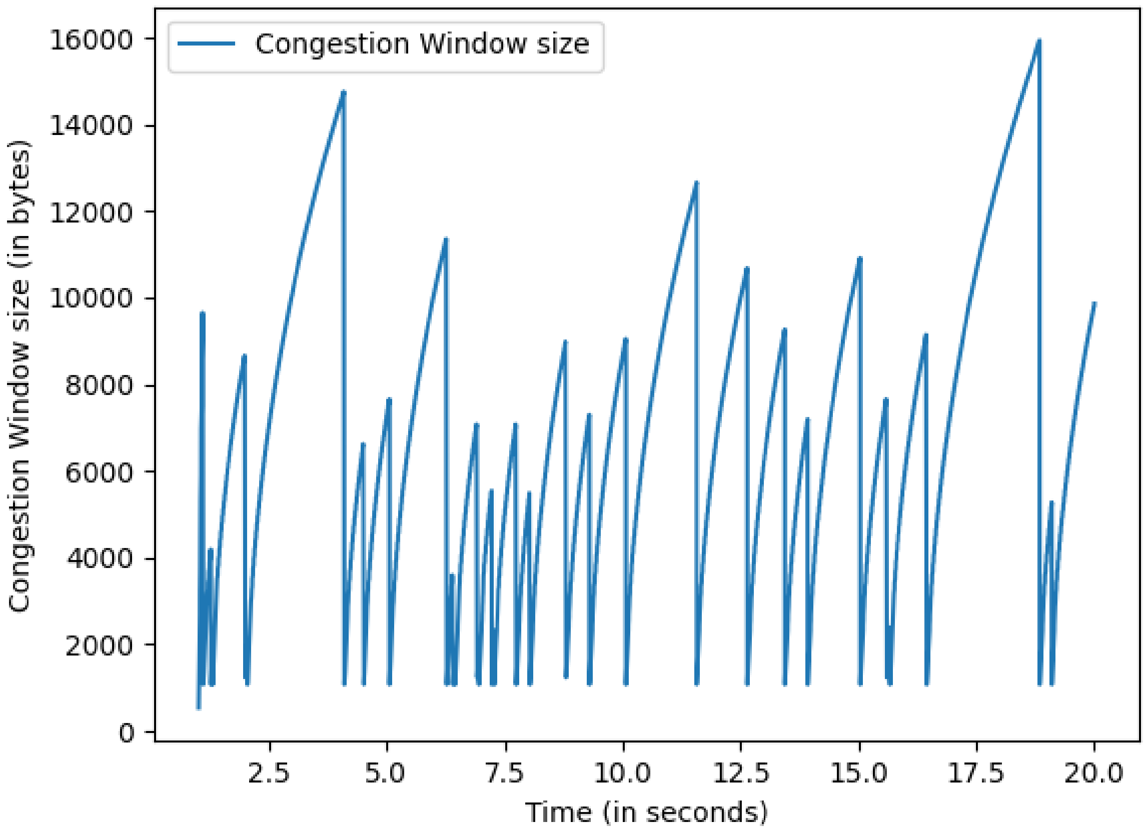

Figure 3.1: Congestion Window (cwnd) size vs time for Connection 1 (TCP NewReno)

Connection: 2
Number of packets dropped: 48

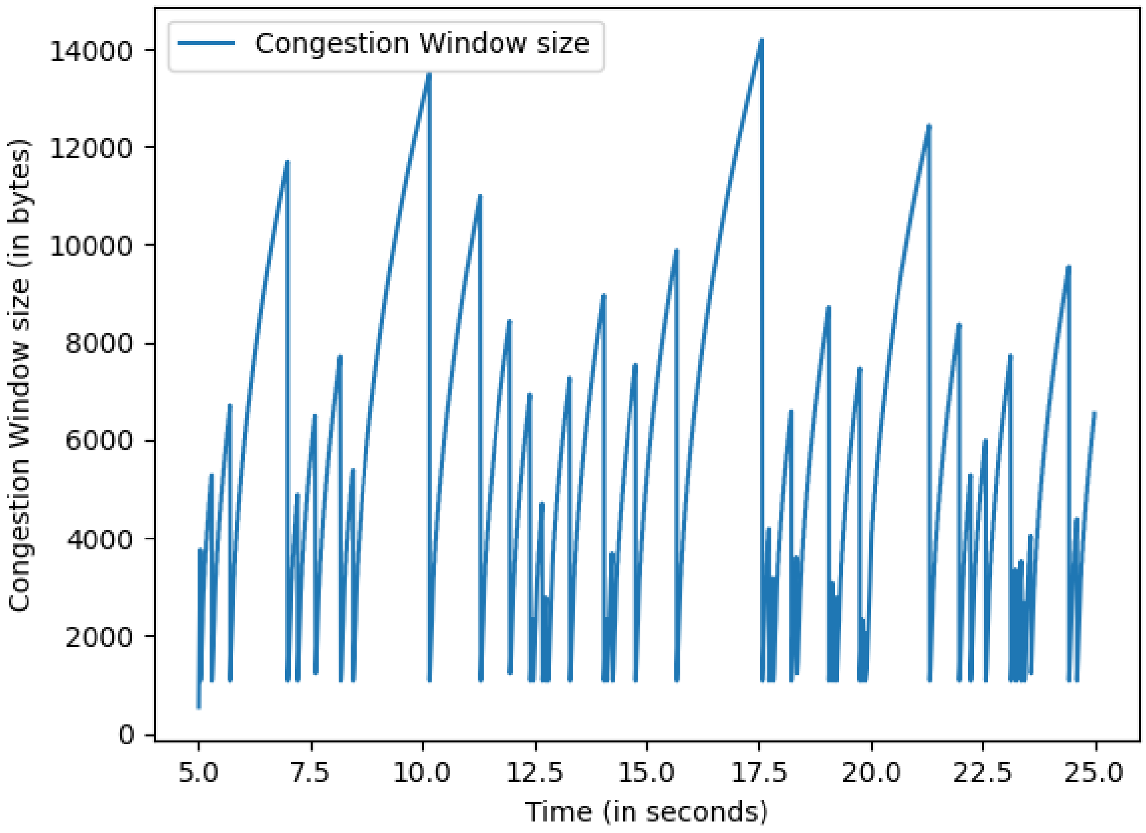

Figure 3.2: Congestion Window (cwnd) size vs time for Connection 2 (TCP NewReno)

Connection: 3
Number of packets dropped: 34

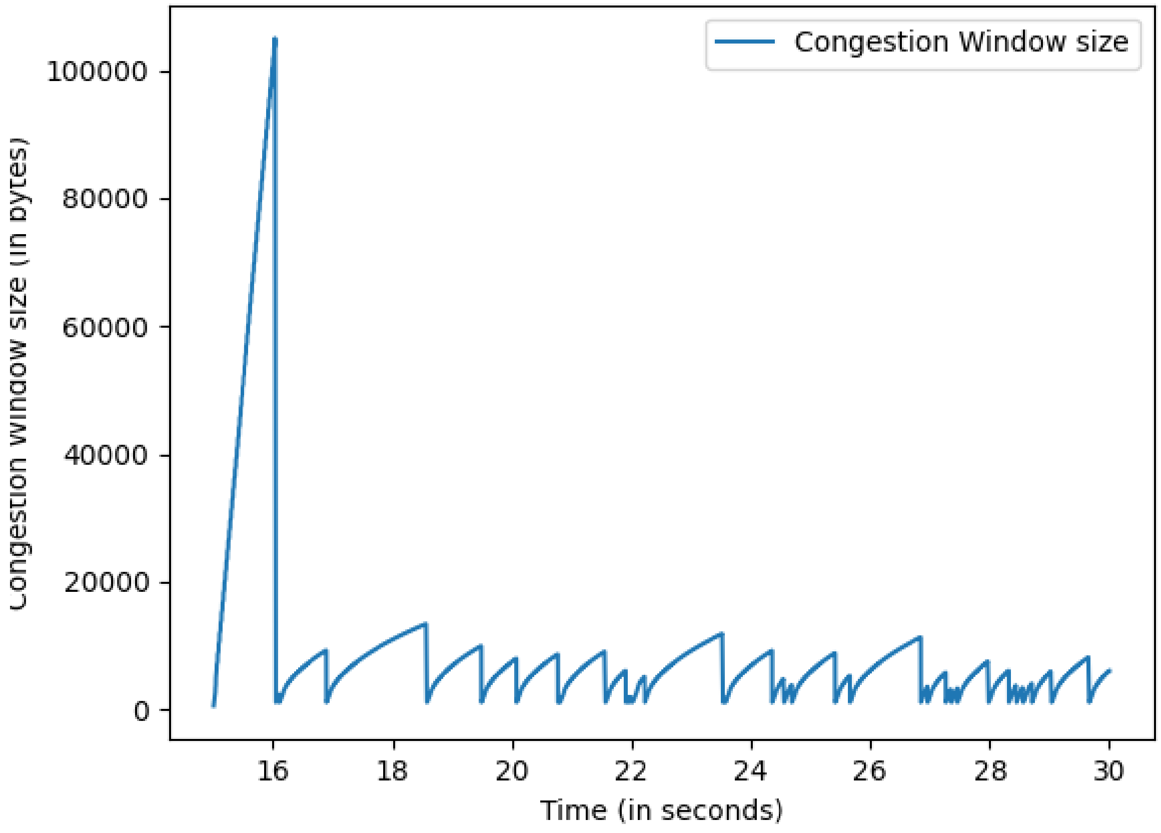

Figure 3.3: Congestion Window (cwnd) size vs time for Connection 3 (TCP NewReno)

Packet Drop Report:

- Number of packets dropped for Connection 1: 31
- Number of packets dropped for Connection 2: 48
- Number of packets dropped for Connection 3: 34

Note: The number of dropped packets for Connection 1 and Connection 2 were determined using the port numbers of the dropped packets.
Note: The error model used is RateErrorModel and the packet errors are generated using probabilistic approach. Average number of packet drops depend on the total packets received by the receiver, and hence, the values (31, 48, 34) are related to the total number of packets being sent for each connection, which depends on App rate, Channel Rate, Packet Size, Protocol used, Propagation Delay and other network factors. Since, everything other than protocol used is fixed, we can directly analyse the performance of our protocol w.r.t dropped packets in next sections.

3.2 Configuration 2

In this configuration, connection 1 and 2 use TCP NewReno and connection 3 uses TCP NewRenoCSE at source.
Connection: 1
Number of packets dropped: 31

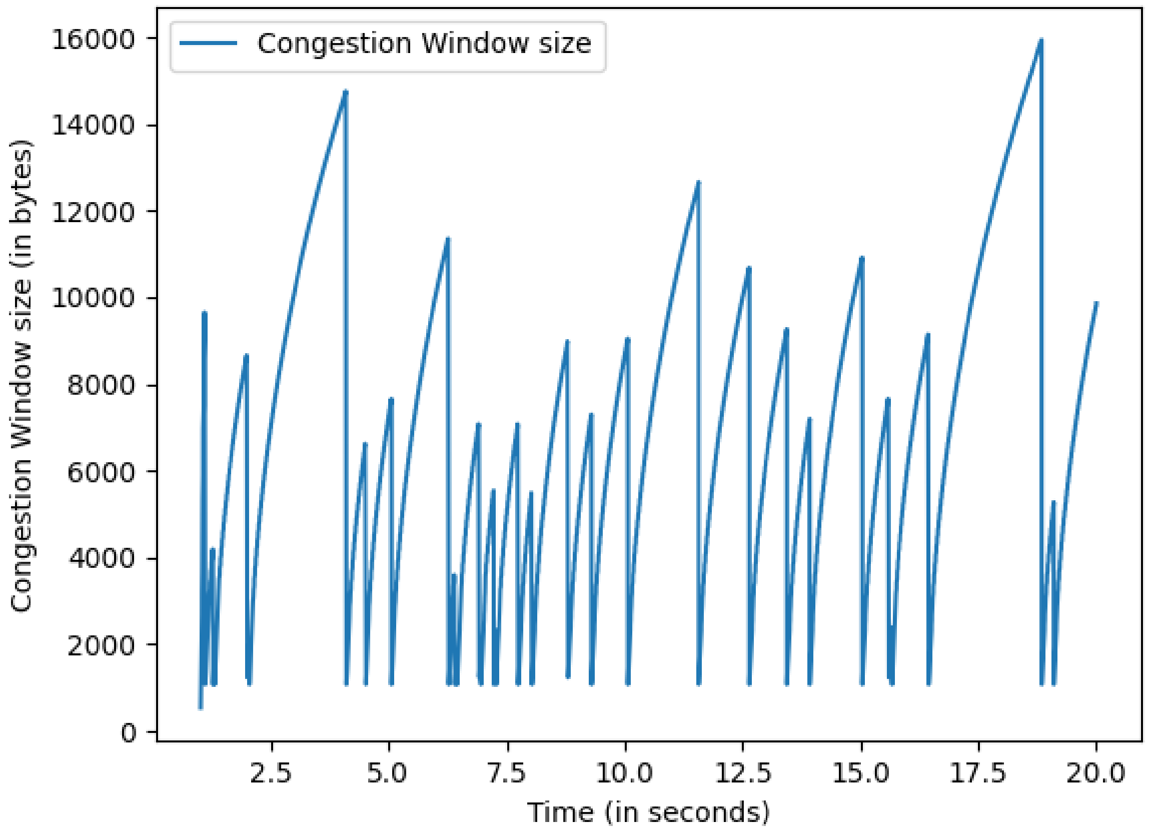

Figure 3.4: Congestion Window (cwnd) size vs time for Connection 1 (TCP NewReno)

Connection: 2
Number of packets dropped: 48

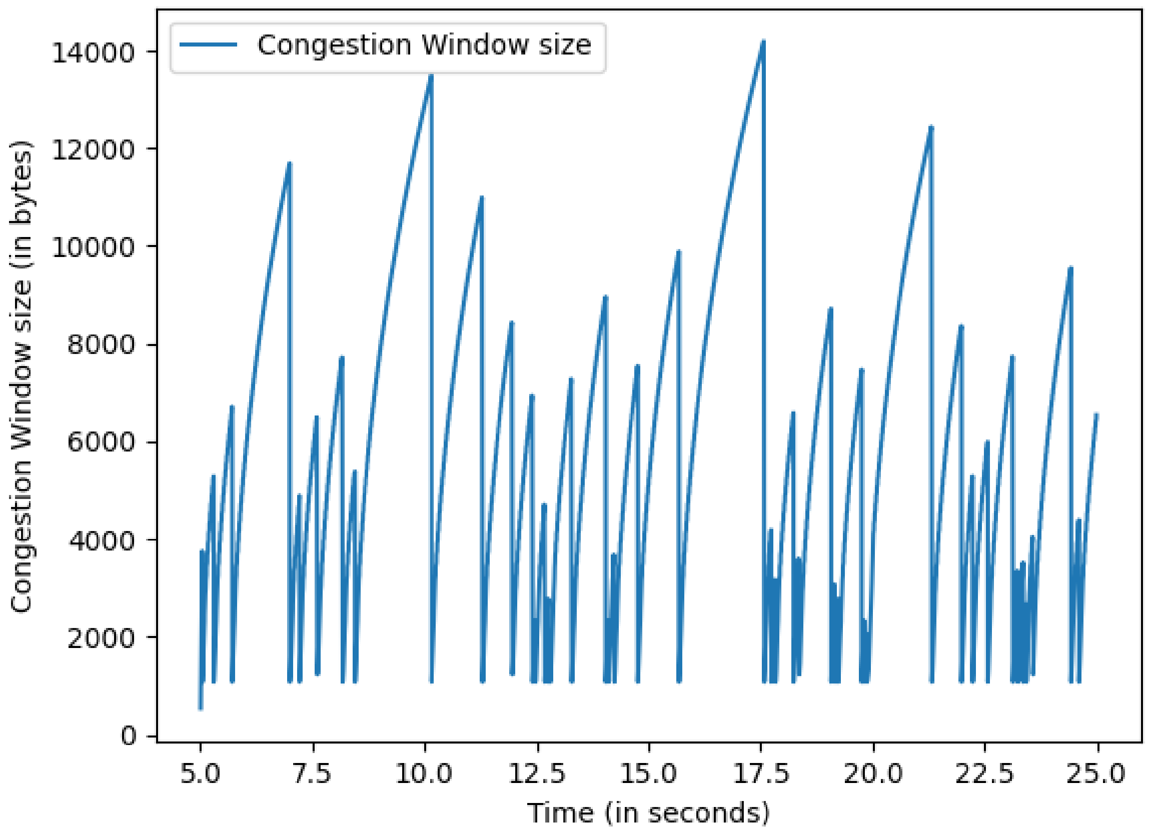

Figure 3.5: Congestion Window (cwnd) size vs time for Connection 2 (TCP NewReno)

Connection: 3
Number of packets dropped: 33

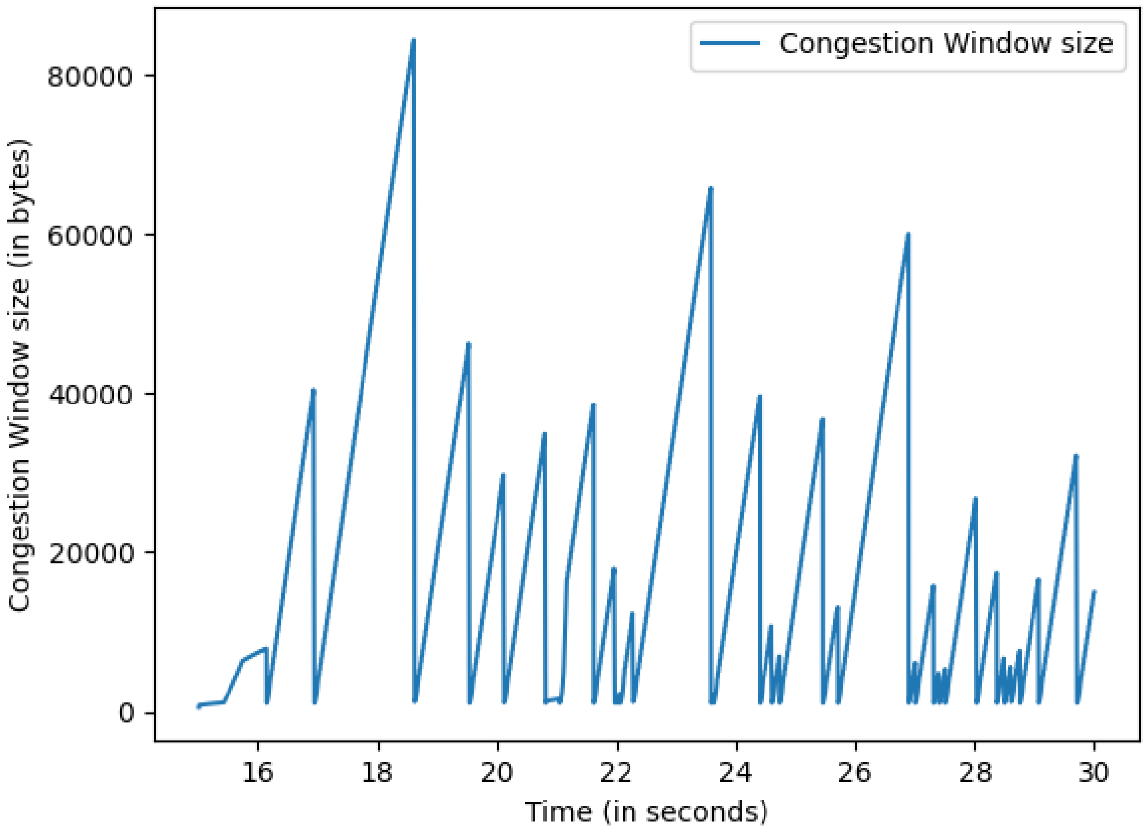

Figure 3.6: Congestion Window (cwnd) size vs time for Connection 3 (TCP NewRenoCSE)

Packet Drop Report:

- Number of packets dropped for Connection 1: 31
- Number of packets dropped for Connection 2: 48
- Number of packets dropped for Connection 3: 33

Note: Since Connection 1 and 2 are still using TCP NewReno, the number of packets sent/dropped is the same, and hence, we get the same value.
Note: Now, Connection 3 is using TCP NewRenoCSE protocol for updating its congestion window. Since, the protocol changed, we can expect the number of dropped packets to change as well. In this case, the change is not by much and the drop count has reduced by 1.

3.3 Configuration 3

In this configuration, all senders use TCP NewRenoCSE.
Connection: 1
Number of packets dropped: 33

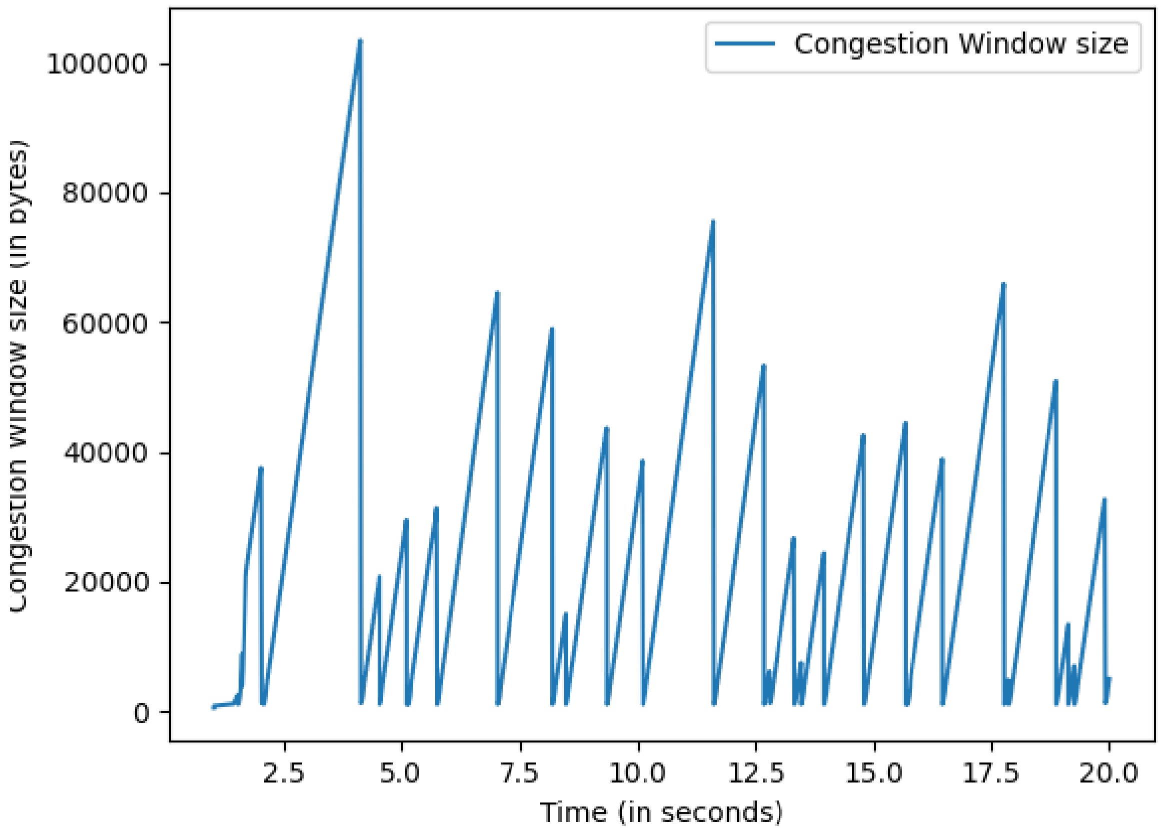

Figure 3.7: Congestion Window (cwnd) size vs time for Connection 1 (TCP NewRenoCSE)

Connection: 2
Number of packets dropped: 44

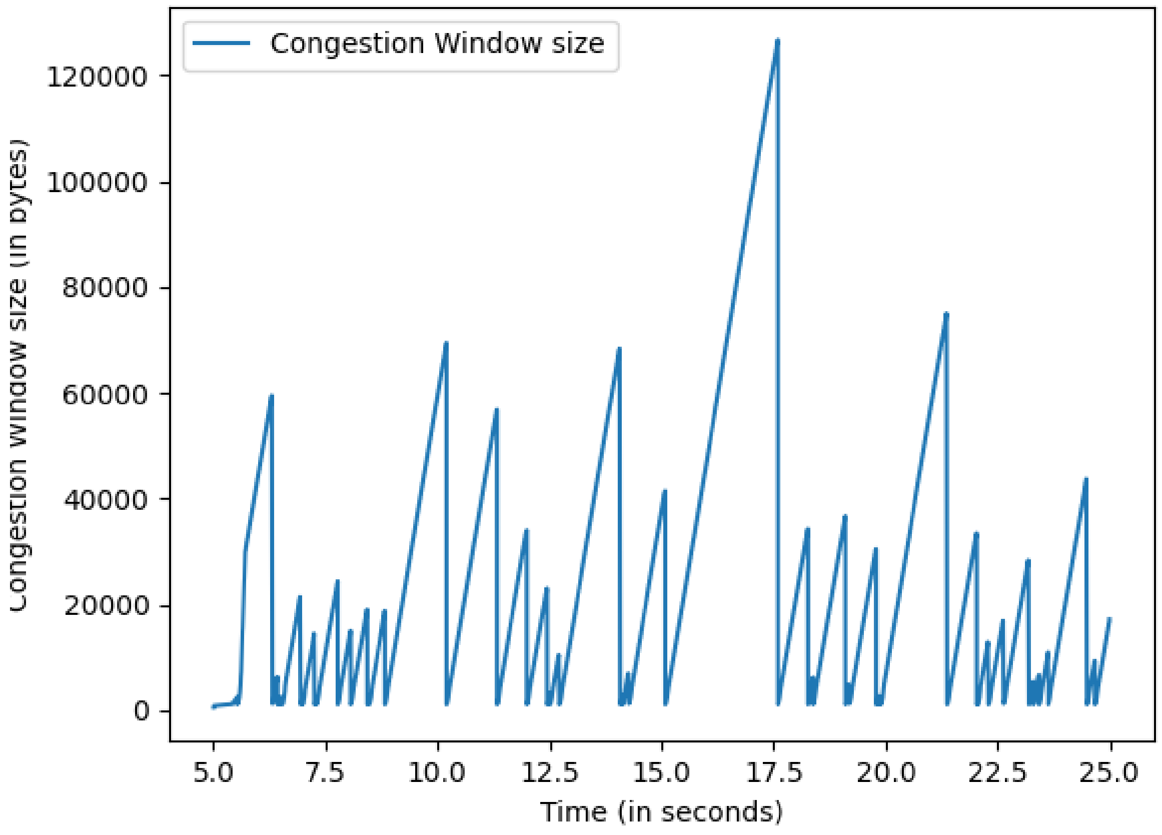

Figure 3.8: Congestion Window (cwnd) size vs time for Connection 2 (TCP NewRenoCSE)

Connection: 3
Number of packets dropped: 33

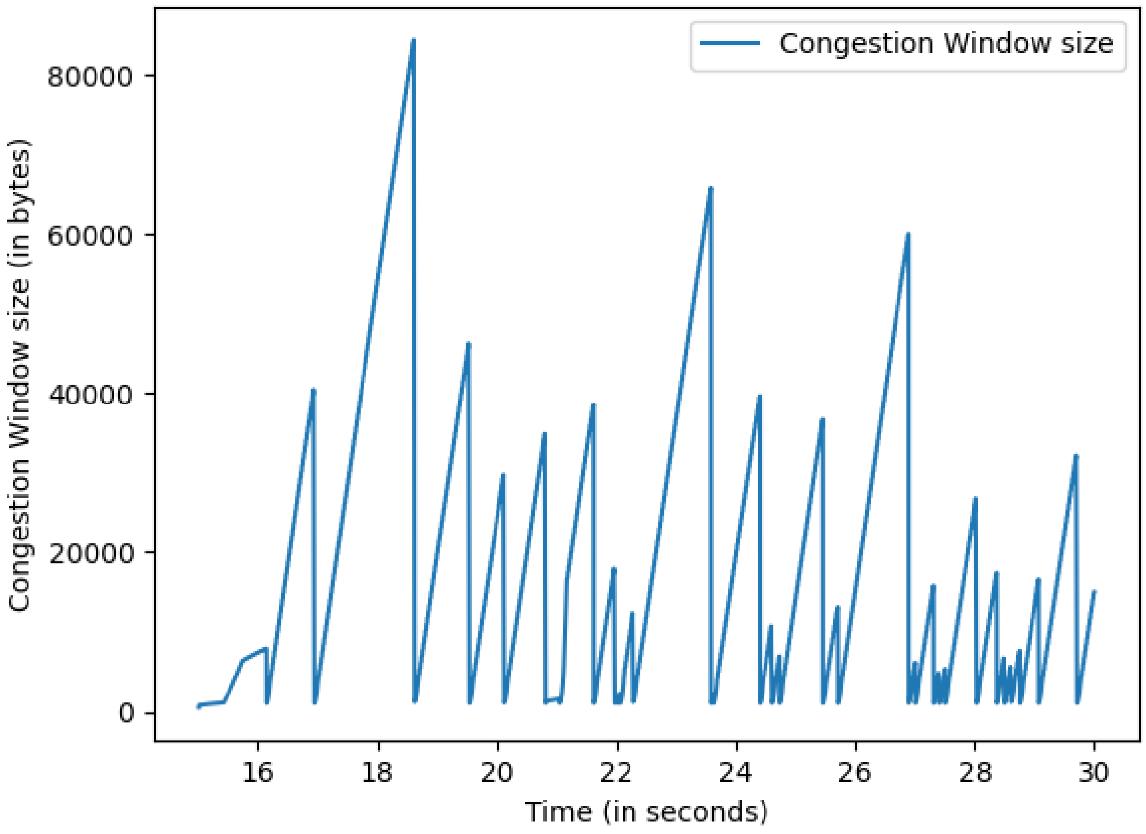

Figure 3.9: Congestion Window (cwnd) size vs time for Connection 3 (TCP NewRenoCSE)

Packet Drop Report:

- Number of packets dropped for Connection 1: 33
- Number of packets dropped for Connection 2: 44
- Number of packets dropped for Connection 3: 33

Note: Since Connection 3 is still using the TCP NewRenoCSE protocol (as in configuration 2), the number of dropped packets do not change and remain the same (33).
Note: Now Connection 1 and 2 are also using TCP NewRenoCSE protocol for updating their congestion window. Since, the protocol changed, we can expect the number of dropped packets to change as well. In this case, the change is not by much. The drop count for Connection 1 increases by 2, whereas the drop count for Connection 2 decreases by 4. Hence, there is an overall decrease of 2 packet drops on Link connecting Node 1 and Node 3.

Trends and Observations:

- Changing the protocol for Connection 3 in configuration 2 does not affect the network consisting of Connection 1 and 2, as the two connecting links are different.
- Similarly, when the protocol for Connection 1 and 2 is changed in configuration 3, it does not affect Connection 3. Basically, using this point and above, I can conclude that the connections 1 and 2, and connection 3 are actually independent of each other, as they use separate link (to different receiving interfaces).
- The congestion avoidance phase varies greatly on a single sender when changing the protocol from TCP NewReno to TCP NewRenoCSE. In TCP NewReno, the congestion window is increased by $\frac{MSS^2}{cwnd}$ bytes for every successful ACK, thus giving non-linear curves in the graph. Whereas, in TCP NewRenoCSE, the congestion window is being increased by $0.5 * MSS$ bytes for every successful ACK, which is an almost linear increase with time. This linearity can also be observed from the graphs of Connection 1 and 2 (configuration 3) and graphs of Connection 3 (configuration 2 and 3).
- Due to this new algrithm, the congestion window is increasing to very large values (1000000 bytes) which is not very good if the application data rate is also large, as then there will be a lot of congestion. But for current scenario, and low application data rate, the effect of TCP NewRenoCSE is not very significant at least when it comes to total number of packet drops.
- The overall packet drops in the network are decreased by 3 when all sources use TCP NewRenoCSE, and the congestion window peaks also increase sharply. Hence, the overall network is impacted by change in protocol but not by much when it comes to packet drops. However, if we compare the congestion window sizes, then the difference is significant due to linear increase in congestion avoidance phase in TCP NewRenoCSE, as compared to non-linear(slow) one in TCP NewReno. Therefore, all senders reach very large congestion window sizes before experiencing a packet loss.
- The slow start phase of TCP NewRenoCSE on the other hand is extremely slow and is just like congestion avoidance phase of TCP NewReno (1.9 instead of 2). Hence, the increase in congestion window is very low towards the beginning of the connection. This is different from TCP NewReno, where the increase in the beginning is linear with number of ACKs received.