November 4, 2022

1 Running the Code and reproducing results

Plotted using matplotlib and numpy. Please check for libraries before running scripts.

a) Copy the files "run.sh" and "gg.py" into the ns-3.29 folder

b) Copy the files "Part1_334.cc", "part2.cc", "part3.cc" in the ns-3.29/scratch/ folder

c) Open the terminal in the ns3-29 folder

d) Run "chmod +x run.sh" to make the script executable

e) Run "./run.sh"

f) 4 new Folders are created, "part1" has the data files and graph images for part1, and similarly for part2a, part2b, part3.The pcap files show up in the ns-3.29 folder itself.

2 Part 1 - Congestion Control Protocols

2.1 Tables and Data

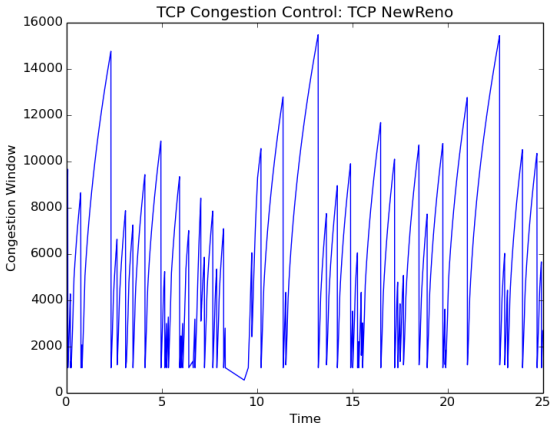
The functions that count the number of packets dropped and the maximum Congestion Window size have been implemented in "gg.py". The results obtained from the terminal, and tabulated are shown below:-

Congestion Protocol	Maximum Window size	Number of Packets Dropped
NewReno	15462	60
Vegas	11252	64
Veno	15462	61
Westwood	15471	62

```
gman@shadow:~/Downloads/ns-allinone-3.29/ns-3.29$ python gg.py
TCP Congestion Control: TCP NewReno
Number of Packets Dropped =  60
Maximum Congestion Window Size =  15462
/home/gman/Downloads/ns-allinone-3.29/ns-3.29/gg.py:62: Matplot
ning: The close_event function was deprecated in Matplotlib 3.6
ved two minor releases later. Use callbacks.process('close_even
.)) instead.
  plt.close()
TCP Congestion Control: TCP Vegas
Number of Packets Dropped =  64
Maximum Congestion Window Size =  11252
TCP Congestion Control: TCP Veno
Number of Packets Dropped =  61
Maximum Congestion Window Size =  15462
TCP Congestion Control: TCP WestWood
Number of Packets Dropped =  62
Maximum Congestion Window Size =  15471
```

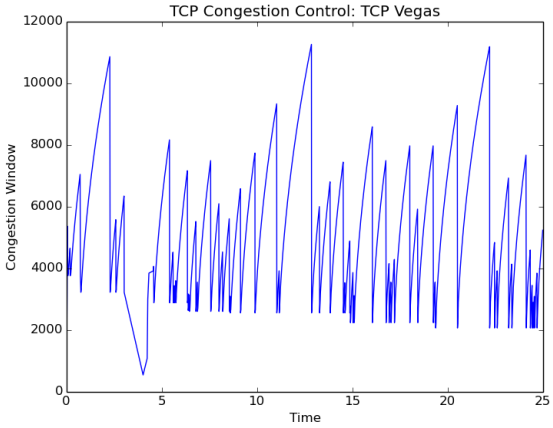
2.2 Graphs and Protocols

2.2.1 TCP NewReno



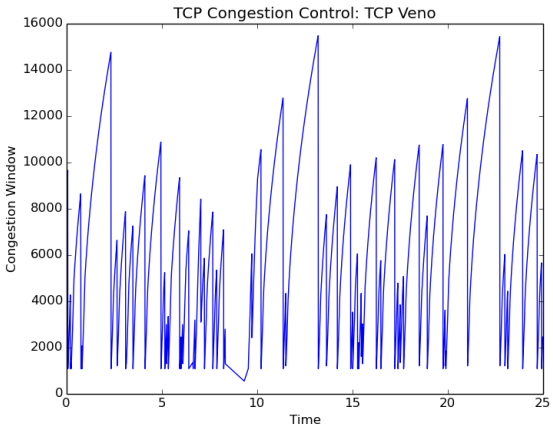
This is an improvement of the standard reno protocol, which suffered from the inability to distinguish between packet drops of the same cwind, and consequently unnecessary drops. By introducing a ”partial ack” to help flag packet drops of belonging to the same window or another, we only drop multiple times if the drops belong to different cwindows. This substantially increases throughput.

2.2.2 TCP Vegas



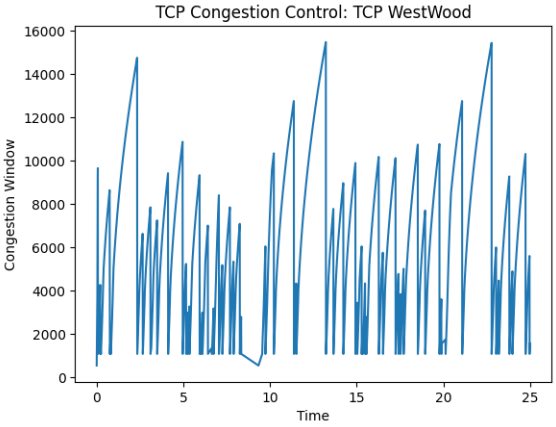
Measures and Dynamically alters congestion window based on the RTTs of the packets. Maintains a buffering mechanism that smoothens the change of RTT as congestion changes. Tries to prevent Packet drops, as opposed to Reno which changes itself according to Packet Drops.

2.2.3 TCP Veno



An innovative blend of VEgas + reNO, this is useful for transmissions with random drop susceptibility (eg. wifi), in which cases a cwindow drop is counter-productive. This protocol hence uses the vegas estimation to estimate whether a drop was random or due to congestion, if former, no action is taken, while the latter is handled by the reno protocol.

2.2.4 TCP Westwood



This protocol continuously estimates, at the TCP sender, the packet rate of the connection by monitoring the ACK reception rate, and accordingly adjusts the cwind.

2.3 Comparisons and Observations

- TCP Vegas's Congestion window does not drop to a constant number unlike the others. This is due to the innovative averaging scheme that determines the number the cngwindow falls down to. This number is usually greater than the standard number used by other protocols, and hence there is more utilisation of the resources in Vegas, since inefficiency due to starting over from scratch every time in other protocols is not present here.

- TCP Reno, Veno and westwood were almost similar amongst themselves. They dropped down to a set value most of the times, and the increasing curve seems to be very sim-

ilar to the curve of TCP Cubic. At around 8 seconds, there is a sudden fall below the usual value these protocols srop down to, and this may be because of a triple ack loss instead of timeout.

- While the recovery from the other drops seems to follow a TCP-cubic like curve only, recovery from the drop at around 8 second is linear (in all the graphs) for a while, before it switches to cubic like behaviour!

2.4 TCP Bbr

Google developed the Bottleneck Bandwidth and Round-trip propagation time (BBR), which is a TCP congestion control algorithm. What sets it apart from other Congestion Control Protocols is not relying just on the on Packet loss for adjusting the congestion window, and the ability to use more of the bandwidth that modern era internet has come to provide, something that was not around when the older protocol were developed. BBR tackles issues like BufferBloat with a ground-up rewrite of congestion control, and it uses latency, instead of lost packets as a primary factor to determine the sending rate. The performance of BBR over other protocols is staggeringly impressive when even moderate packet loss is present. The following table I found at lists the relative performance of these protocols:-

Throughput	Congestion control algorithm (sender)	latency	loss
2.35 Gb/s	Cubic	<1ms	0%
195 Mb/s	Reno	140ms	0%
347 Mb/s	Cubic	140ms	0%
344 Mb/s	Westwood	140ms	0%
340 Mb/s	BBR	140ms	0%
1.13 Mb/s	Reno	140ms	1.5% (sender > receiver)
1.23 Mb/s	Cubic	140ms	1.5% (sender > receiver)
2.46 Mb/s	Westwood	140ms	1.5% (sender > receiver)
160 Mb/s	BBR	140ms	1.5% (sender > receiver)
0.65 Mb/s	Reno	140ms	3% (sender > receiver)
0.78 Mb/s	Cubic	140ms	3% (sender > receiver)
0.97 Mb/s	Westwood	140ms	3% (sender > receiver)
132 Mb/s	BBR	140ms	3% (sender > receiver)

2.5 Changes In Files

I edited the ”examples/tutorial/fifth.cc” file to suit my needs. The TCP Congeston Protocol support was provided by addition of the following code :-

```
1  std::string transport_prot = "Newreno";
```

```
cmd.AddValue ("transport_prot", "Transport protocol to use: TcpNewReno, "
              "TcpHybla, TcpHighSpeed, TcpHtcp, TcpVegas, TcpScalable, TcpVeno, "
              "TcpBic, TcpYeah, TcpIllinois, TcpWestwood, TcpWestwoodPlus, TcpLedbat, "
  "TcpLp", transport_prot);
transport_prot = std::string ("ns3::") + transport_prot;

Config::SetDefault("ns3::TcpL4Protocol::SocketType",
TypeIdValue (TypeId::LookupByName (transport_prot)));
```

The Error Model was Incorporated with the following block of code :-

```
Ptr<RateErrorModel> em = CreateObject<RateErrorModel> ();
em->SetAttribute ("ErrorRate", DoubleValue (0.00001));
devices.Get (1)->SetAttribute ("ReceiveErrorModel", PointerValue (em));
```

And the following Parameters were set as per the problem statement

```
pointToPoint.SetDeviceAttribute ("DataRate", StringValue ("5Mbps"));
pointToPoint.SetChannelAttribute ("Delay", StringValue ("3ms"));
Ptr<MyApp> app = CreateObject<MyApp> ();
app->Setup (ns3TcpSocket, sinkAddress, 2000, 1000, DataRate ("2Mbps"));
```

Packet Counting and Maximum Window size was obtained via ”gg.py”, which analysed the log files generated while running the simulations, and counted the number of times a string with the start ”RxDrop” was found/The standard O(n) Maxima finding traversal.

3 Part 2 - Application and Channel Rates

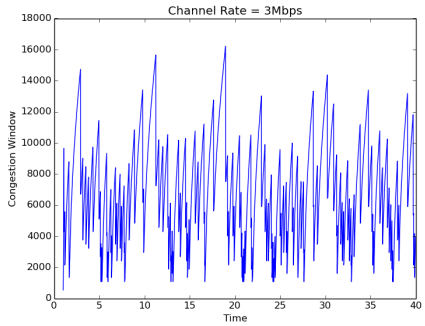
Setting the Application Data Rates and Channel Rates as Parameters that can be set from command line arguments using the following block of code,

```
std::string apprate = "5Mbps";
std::string crate = "4Mbps";
cmd.AddValue ("apprate", "Application Data Rate:", apprate);
cmd.AddValue ("crate", "Channel Data Rate:", crate);
```

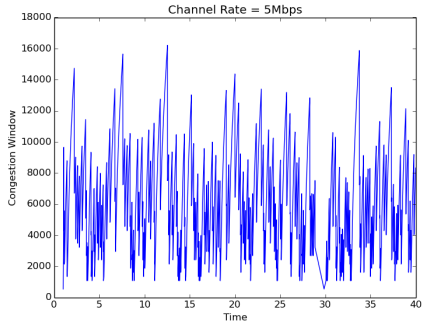
The RateErrorModel, the Packet size, simulation time, channel delay, parameters were set as per the problem statement.

3.1 Varying Channel Rates, Application Rate Constant(=5Mbps)

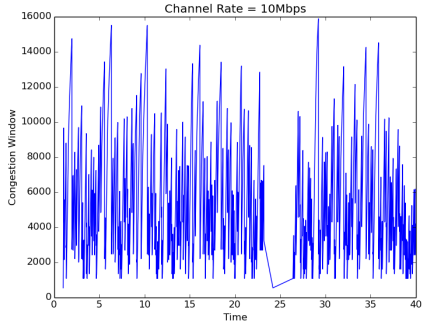
The Command Line argument –crate=yMbps was added to change the values of channel rate, and the Application Data Rate defaulted to 5Mbps.



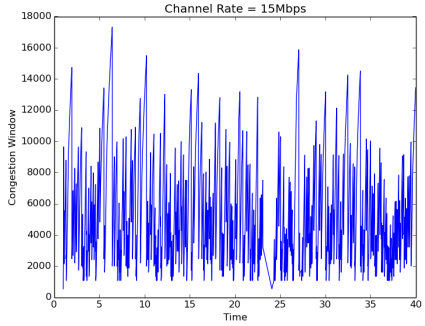
Channel Rate = 3Mbps, Application Data rate = 5Mbps



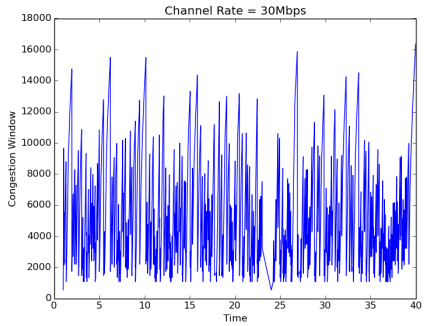
Channel Rate = 5Mbps, Application Data rate = 5Mbps



Channel Rate = 10Mbps, Application Data rate = 5Mbps



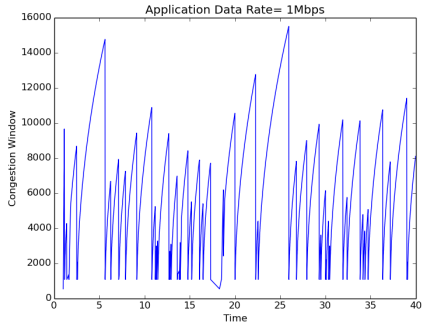
Channel Rate = 15Mbps, Application Data rate = 5Mbps



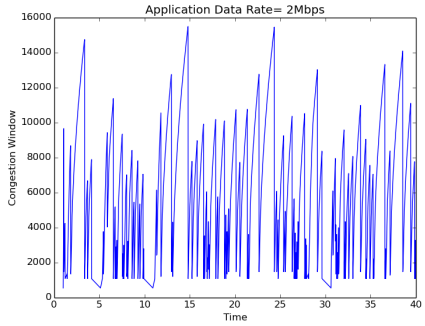
Channel Rate = 30Mbps, Application Data rate = 5Mbps

3.2 Varying Application Rates, Channel Rate Constant(=4Mbps)

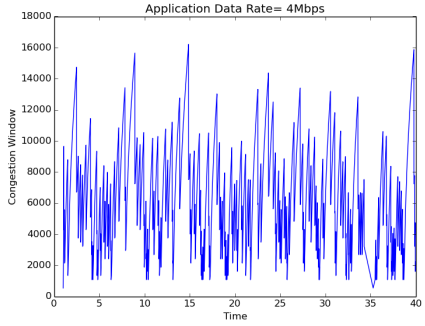
The Command Line argument –apprate=yMbps was added to change the values of Application Data Rate, and the Channel Rate defaulted to 4Mbps.



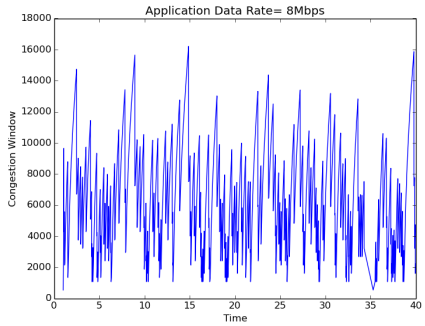
Application Data rate = 1Mbps, Channel Rate = 4Mbps



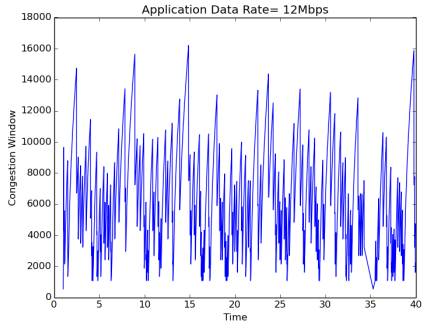
Application Data rate = 2Mbps, Channel Rate = 4Mbps



Application Data rate = 4Mbps, Channel Rate = 4Mbps



Application Data rate = 8Mbps, Channel Rate = 4Mbps



Application Data rate = 12Mbps, Channel Rate = 4Mbps

3.3 Observations & Explanations

- Q1) Increasing the channel rate(as long as it is lower than the application rate) means increasing the data that reaches the buffers, and hence results in an increased congestion and more frequent drops. After the channel rate surpassed the application rate, almost similar graphs were obtained, since the channel can now accommodate the Application and any extra difference is irrelevant.

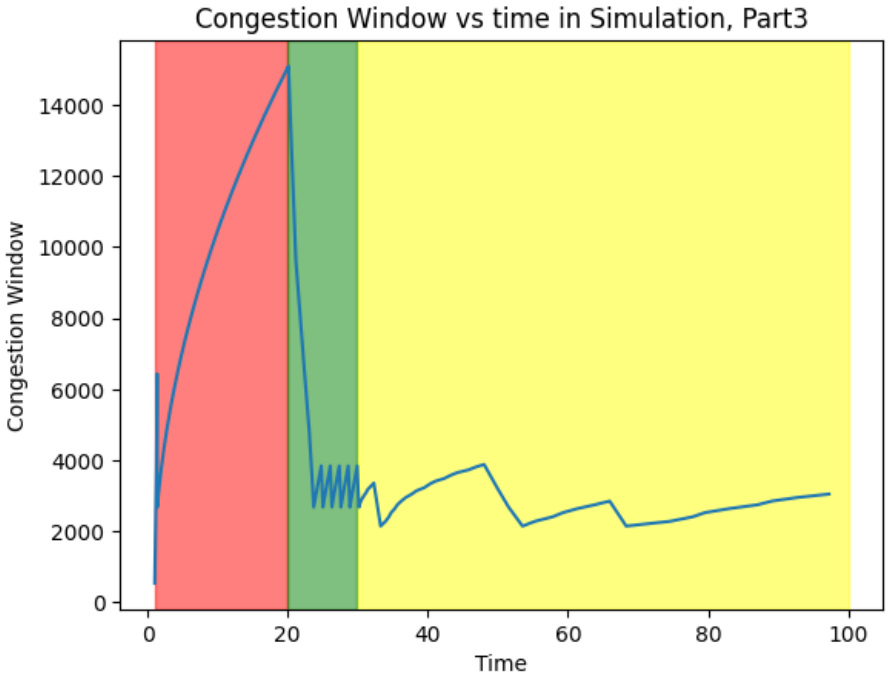
- Q2) Increasing the Application Data rate, while keeping the channel rate constant, increases congestion as expected. For graphs when the Application Rate exceeds or is equal to the Channel rate, Congestion is expected, and is prominent. However, the variation is low since only the bottleneck transmission is being done, and similar graphs are obtained. For application rates lower than the channel rate, the graphs have lower number of drops , since the channel is relatively free, and, varying the application rate does have a significant change in the graph.

- Q3) We observe that If application rate > channel rate and application rate increased, or channel rate > application rate and channel rate increased, no significant variation. This happens because the extra load in the former and extra space in the latter can make no difference to the transmission. In other cases, the closer these values are, the higher the congestion. This is due to one of the being the rate limiter, and the reasons mentioned in the two parts above.

4 Part 3

4.1 Graphs

The Following graph was obtained when the simulation was run



4.2 Effects on cgwind

The UDP transmission was started at 20 seconds, and its' rate was incremented at 30 seconds. The rate stabilises at around 24 seconds, and after 30 seconds, the cgwindow is asymptotic to around 3000. The three regions 0-20, 20-30, 30-100 have been marked to show demarcation.

4.3 Logistics of Part 3

The Topology that was given in the problem statement was built with inspiration from "examples/routing/simple_global_routing.cc". The steps that were followed included

- Creating 5 Nodes.
- Creating 4 node containers for the 4 links, with 2 devices each.
- Setting up the p2p connections and. attributes between the nodes
- Assigning IP addresses to the Network
- Populate the routing Tables
- Create the applications working on nodes 1 and 3.

- Start the TCP connection at 1 till 100 sec
- Start the UDP transmission at 20, change rate at 30 seconds.
- Include tracers for tracing Congestion Window and pcap files.