LAB 1 REPORT

Name: Uday Kiran Ravuri

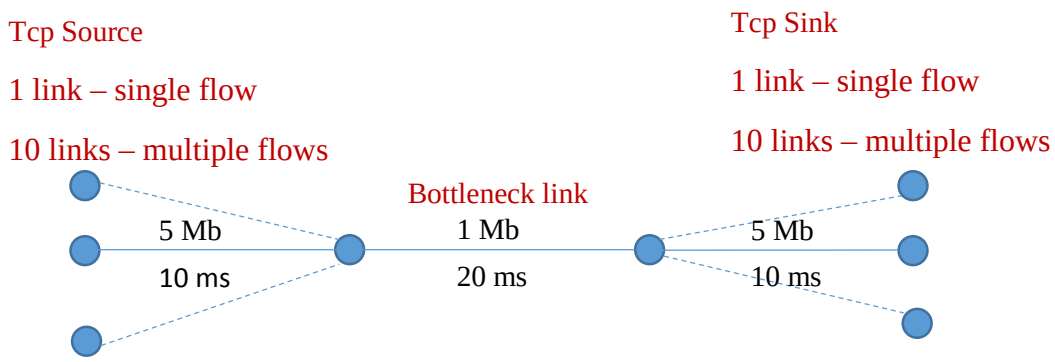
GT ID: 903145978

This project uses the ns-3 simulator to measure and compare the goodput (Received bytes/Simulation time) of a single (and multiple) flow(s) from a TCP source to a TCP Sink through a bottleneck link. The parameters which are varied to analyse performance are:

1) Queue limit (on bottleneck link)
2) Segment sizes
3) Window sizes

Two variants of TCP are used here: TcpTahoe and TcpReno

The network topology is as shown below:



This configuration can be described as a 'dumbbell', in which the nodes in the middle represent the routers (form the bridge of the dumbbell) and the nodes on either side of the routers are the leaf nodes. The ones on the left act as Tcp Source (on which BulkSendApplication is installed) and the ones on the right act as Tcp Sink (on which PacketSinkApplication is installed). The data rate of the devices and delays on the links are modelled as shown in the figure.

The bottleneck link implements a DropTailQueue with varying queue limits.

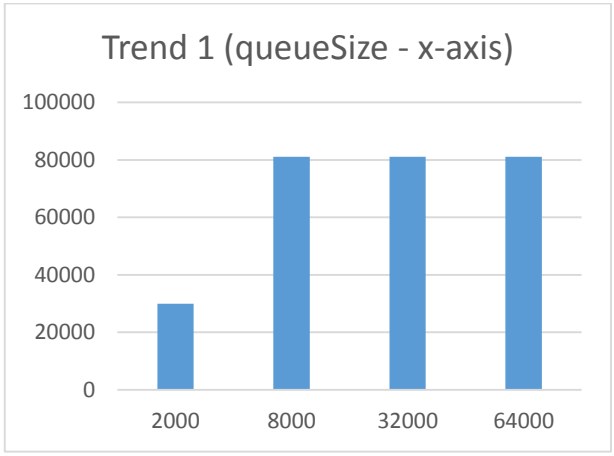
The experimental parameters in this project are:

1) MaxBytes (for BulkSendApplication) → 100,000,000
2) Number of Tcp flows → Single, 10 simultaneous
3) Maximum receiver advertised window size → 2000, 8000, 32000, 64000 bytes
4) Queue limits → 2000, 8000, 32000, 64000 bytes
5) Tcp segment size → 128, 256, 512 bytes
6) Tcp protocol → Tcp Tahoe, Tcp Reno

In multiple flow case, the individual flows are started at random times in the range [0.0, 0.1] with a random seed 11223344.

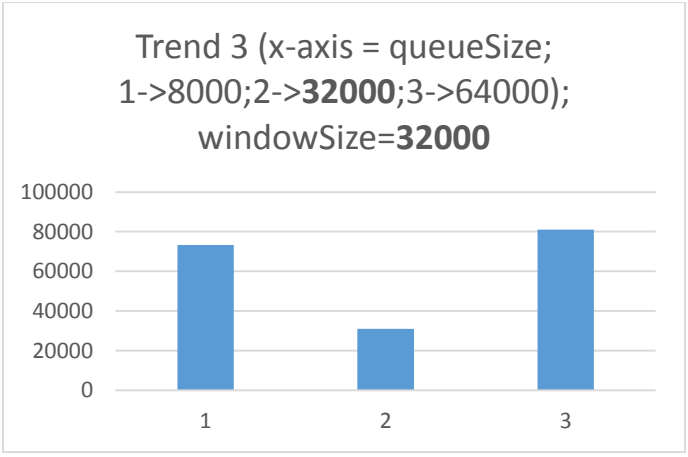
TCP TAHOE SINGLE FLOW

Trend 1: With window and segment sizes kept constant, goodput seems to increase as queue size increases (in many cases, it increases when queue size changes from 2000 to 8000 and then it remains constant for further increase in queue size). We can infer that the amount of data received decreases with less queue size. This parameter is effecting the goodput, even in the single flow case, because our simulations are using the BulkSendApplication which attempts to send 'maxBytes' amount of data as quickly as possible. Greater queue size allows more packets to be sent onto the bottleneck link and hence higher goodput.



Trend 2: With constant queue size and segment size, increasing advertised window size does not appear to have much effect on the goodput. There is improvement as this parameter is changed from 2000 to 8000 bytes but further increase does not affect the goodput. This means that window size of 2000 bytes is restricting the flow of packets in a way that the link is not being utilized to its capacity. Once the receiver advertises a larger window size (8000 bytes and above), performance depends mostly on queue size and segment size.

Trend 3: With low segment sizes (128 and 256 bytes), goodput seems to go down for large values of queue size and window size, especially in cases where both these values match. This trend seems to result from the limited segment size at the transport layer. Low segment size implies that the number of packets sent and accumulated at the queue increases and hence causes this anomaly in goodput. But when the queue size is further increased with other parameters kept the same, then goodput is observed to revert back to its original trend.



Trend 4: When window size is kept constant at 2000 bytes, the queue size does not seem to affect the goodput. However, increasing segment size in this case is causing goodput to decrease. This implies that the number of bytes in the bottleneck link never accumulate beyond 2000 bytes or else increasing queue size would have improved the performance. This trend is quite surprising since the goodput should be increasing with increasing packet size. However, for this topology, increasing packet size coupled with a limited maximum receiver window size is causing less number of packets to be ACKed per RTT and so the congestion window does not increase very quickly (slow-start phase). That is, the number of packets that the sender can transmit per RTT is reduced and hence the congestion window does not grow after few packets. As a result, we see less goodput.

TCP RENO SINGLE FLOW

Tcp Reno has the same kind of performance trends as that of Tcp Tahoe. But some interesting observations are seen when these two flavours are compared with each other:

[In all of the tables in this section, the first row corresponds to Tcp Reno and the second to Tcp Tahoe]

flow	0	windowSize	8000	queueSize	2000	segSize	128	goodput	27189
flow	0	windowSize	8000	queueSize	2000	segSize	128	goodput	29895

In the above case, queue size is a huge limiting factor which results in a large number of dropped packets. Reno is Tahoe + Fast retransmit mode. Hence, Reno will detect the loss of packets (no ACKs received) soon and try to transmit them again. As a result, the network gets congested. So, the goodput decreases slightly.

There are many cases where the underlying reason for the performance difference between Tahoe and Reno is the one discussed above. (Limited queue size).

In contrast, consider the following case:

flow	0	windowSize	64000	queueSize	64000	segSize	128	goodput	59429.1
flow	0	windowSize	64000	queueSize	64000	segSize	128	goodput	33451.4

Here, the queue size is sufficient enough to accumulate the data that Reno could transmit onto the bottleneck link. This results in less packet drops. As fast retransmit mode helps in a positive way, we see a higher goodput when compared to Tahoe with same parameters.

If we use a higher segment size, then number of packets to be received and ACKed is less given the same amount of data transmitted. Hence, queue accumulation is relatively less and results in less packet drops. So, the expected goodput decrease, which resulted due to limited queue size, should not be much. This is confirmed by the simulation results as well, as shown below:

flow	0	windowSize	8000	queueSize	2000	segSize	512	goodput	57264.3
flow	0	windowSize	8000	queueSize	2000	segSize	512	goodput	70923.2

The window size parameter also helps in alleviating the goodput problem of Reno with limited queue size (more window size, Reno's goodput moves closer to that of Tahoe's):

flow	0	windowSize	32000	queueSize	2000	segSize	512	goodput	60356.9
flow	0	windowSize	32000	queueSize	2000	segSize	512	goodput	61387.8

The only two cases where TCP Reno performed better than TCP Tahoe (single flow) is:

flow	0	windowSize	32000	queueSize	32000	segSize	128	goodput	35229.7
flow	0	windowSize	32000	queueSize	32000	segSize	128	goodput	30951.6

flow	0	windowSize	64000	queueSize	64000	segSize	128	goodput	59429.1
flow	0	windowSize	64000	queueSize	64000	segSize	128	goodput	33451.4

We can see above that the window and queue sizes are equal to each other and the segment size is 128 bytes. The fast retransmit feature of TCP Reno clearly helped in these cases.

Normally, we would expect that TCP Reno performs better than Tahoe. But given our topology and the simulations, we are changing only 3 parameters and moreover, we are disabling window scaling option. This has affected Reno's performance drastically.

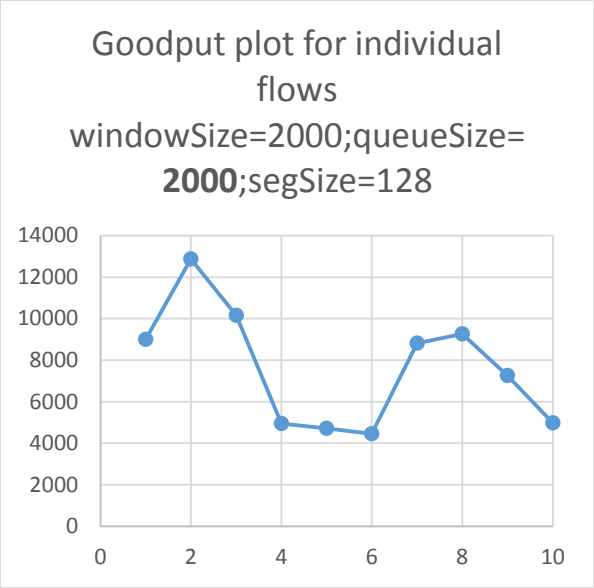
TCP TAHOE/RENO MULTIPLE FLOWS

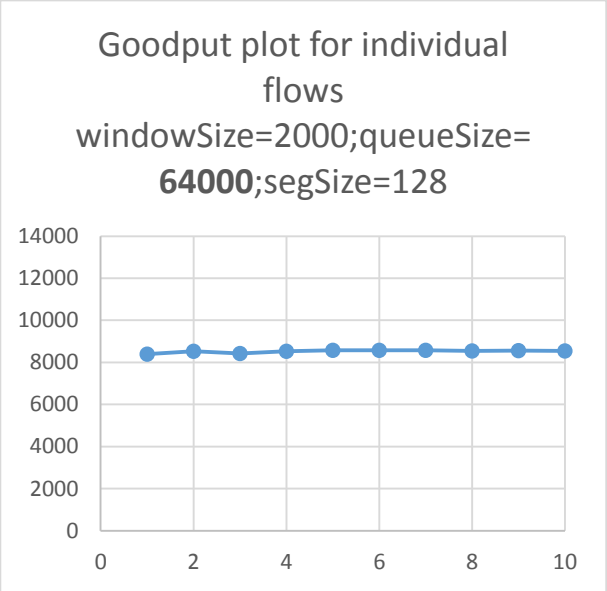
In all our simulations in this section, we have started the individual flows at random times using a uniform random generator with random seed 11223344 and stream set as 6110. The random start times obtained are as shown below:

Flow number	**Start times**
0	0.06655
1	0.00105
2	0.09883
3	0.01597
4	0.05442
5	0.05761
6	0.05719
7	0.02615
8	0.05969
9	0.02653

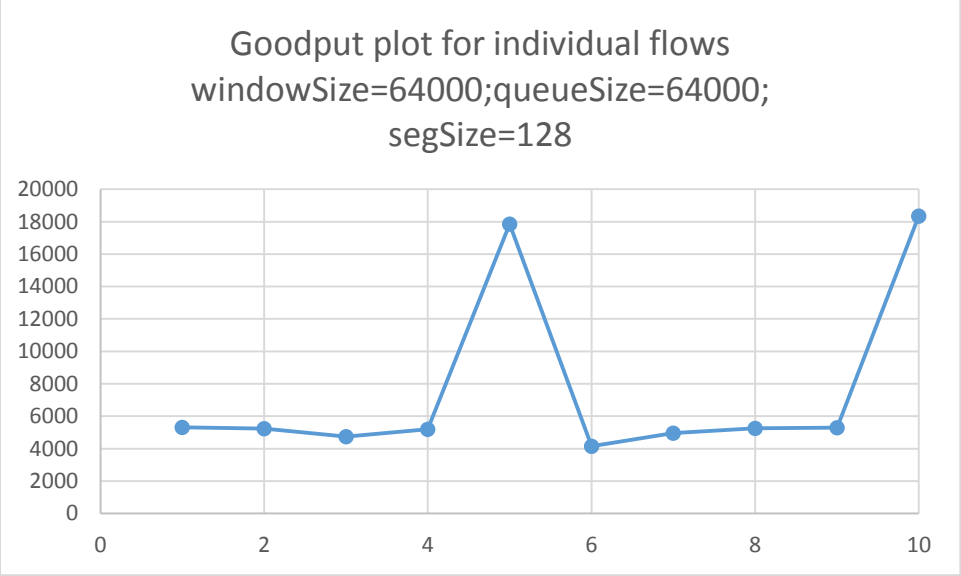
In many multiple flow simulations, TCP Tahoe performed better than TCP Reno in the overall sense (total goodput = sum of all flows' goodputs). This is for the same reasons as explained in the previous discussion on single flow simulations.

Fairness problem (discussion on TCP Tahoe):

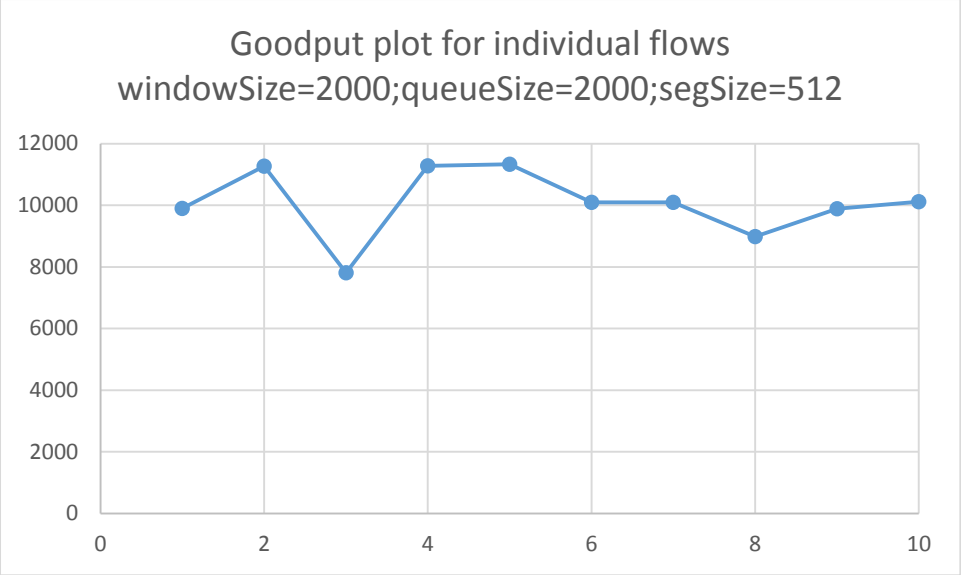




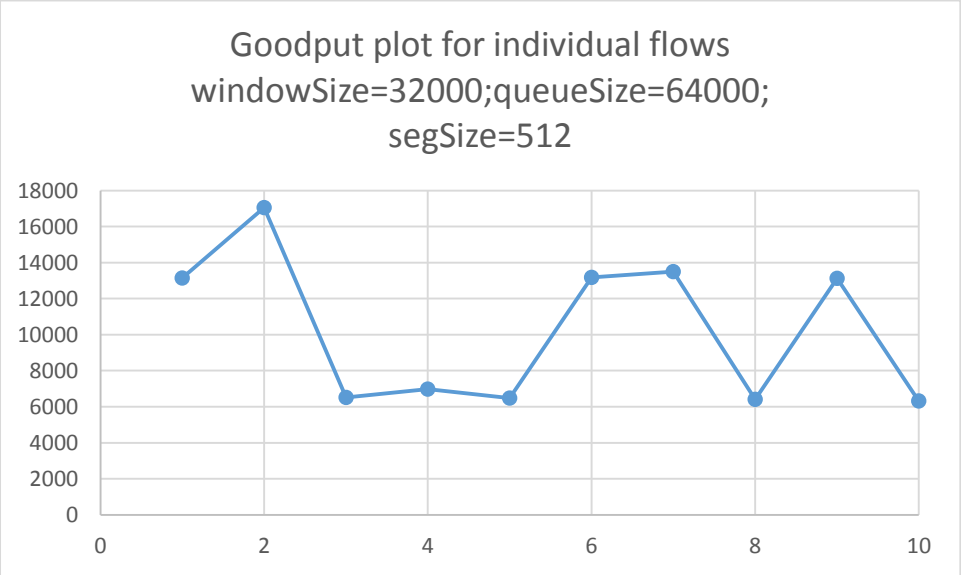
From the above figures, we observe that queue size greatly affects the fairness among individual flows. As we know, the flows start at random times. It is possible that one flow could be in the congestion avoidance phase while another could be in the slow-start phase. At this point, if the queue gets full, for example, the packets would be dropped and hence congestion occurs. When this happens, the individual flows end up setting widely varying 'ssthresh' values. This clearly affects the number of bytes transmitted by each flow onto the links. If the queue size is sufficient enough, then packets from all flows get through to the receiver albeit them being in different phases of congestion control.



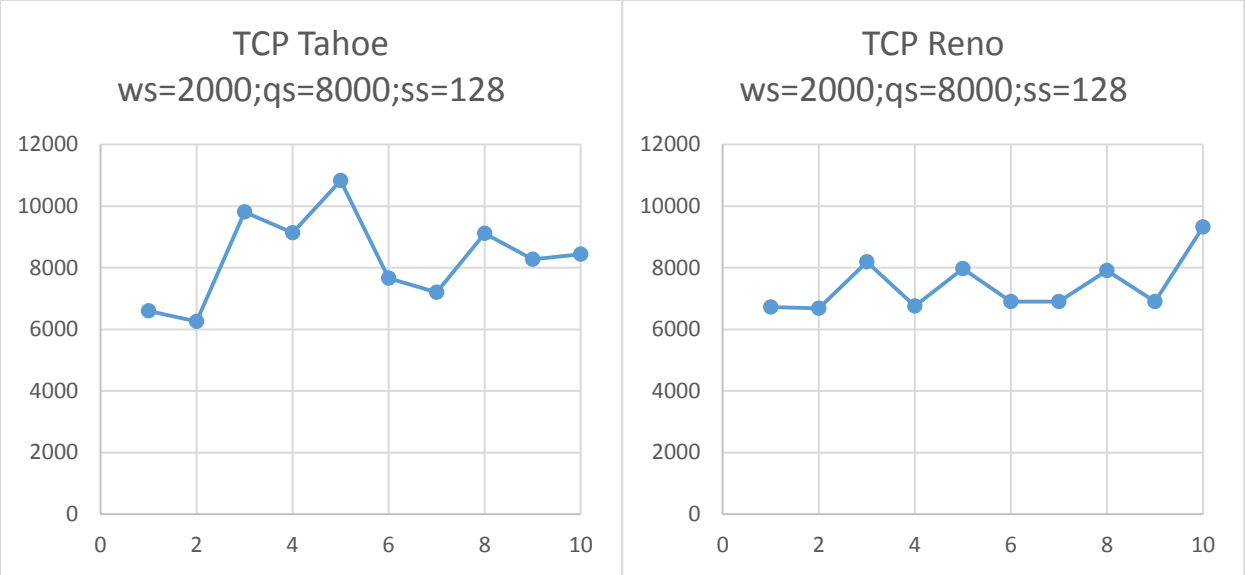
When advertised window sizes are increased, the same fairness pattern as discussed above is seen with the exception that few spikes appear. This indicates that those particular flows have grabbed the link resources most of the time. The rest of the flows, are however fairly allocated (with large queue buffers).



As shown by the above plot, increase in segment size can assure fairness among flows even when window and queue sizes are low. This again is due to the fact that the same amount of data can be transmitted in less number of packets. This implies less packets to be ACKed and less congestion in the network. It also means that the queue buffer can be utilized fairly by all flows. But as the window size increases, the fairness worsens as can be seen in the figure below (high segment and queue size):



Fairness (TCP Tahoe vs TCP Reno):



In many cases, TCP Reno is observed to show the same fairness as Tahoe. However, some exceptions are seen where Reno is fairer. One of them is depicted in the above figures.

To conclude, it seems that we can guarantee fairness among flows by maintaining a large queue (buffer) in the links connecting the sources and sinks, especially bottlenecks. Setting a low window size and large segment size also help in bringing about fairness. For this topology, changing the protocol (Tahoe vs Reno) does not seem to affect the fairness much in case of multiple flows.