ECE 6110- CAD for Computer Networks

Project 2 – Comparison of RED vs DropTail Queuing

Name – Vivian Dsilva gtID 903041189

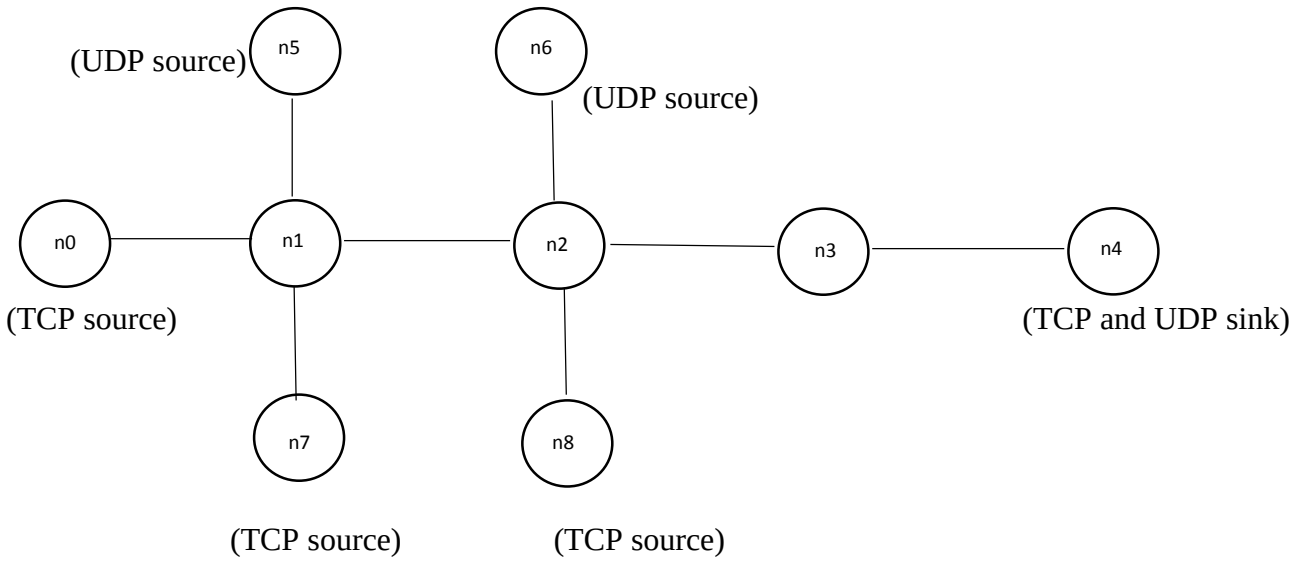
In project 1, we had analyzed TCP throughput with a DropTail Queuing Mechanism.

The paper 'Random Early Detection Gateways for Congestion Avoidance' by Sally Floyd and Van Jacobson [1] suggests a Random Early Detection (RED) gateways for Congestion Avoidance in packet-switched networks. The paper suggests that when the average queue size exceeds a preset threshold, the gateway drops or marks each arriving packet with a certain probability, where the exact probability is a function of the average queue size. The paper also suggests that a RED gateway has no bias against bursty traffic and avoids the global synchronization of many connections decreasing their window size at the same time. By providing advance warning of incipient congestion, RED gateways can be useful in avoiding unnecessary packet or cell drops at the gateway. The paper concludes by saying that Random Early Detection gateways are an effective mechanism for congestion avoidance at the gateway, in co-operation with network transport protocols.

The paper 'Tuning RED for Web Traffic' by Mikkel Christiansen, Kevin Jeffay, David Ott and F.Donelson Smith [2] suggests that for web traffic, RED queue management appears to provide no clear advantage over drop-tail FIFO for end-user response times. They claim this after doing simulations of HTTP Responses and Requests on links having both RED and Drop-Tail FIFO and compare the performance of each mechanism.

Thus, these two papers provide two distinct views of RED queuing mechanism for Congestion avoidance. Hence, in project 2, we do a comparative analysis of RED with the traditional DropTail queuing mechanism.

The topology that I used to perform the analysis is shown below:



Thus, there are 3 TCP sources and 2 UDP sources, which makes sure the topology has a mixture of TCP and UDP flows. The TCP and UDP sinks are present in n4 (it has different sink applications for TCP and UDP each) There are 2 bottleneck links in the above topology. They are the links between nodes n1 and n2, n2 and n3. The TCP sources at nodes n0 and n1 and the UDP source at n5 experience two bottlenecks (n1-n2 and n2-n3). The TCP source at node n8 and the UDP source at node n6 experience a single bottleneck link (n2-n3).

The nodes n1, n2 and n3 (which are routers) can behave as either RED or DropTail. Since the performance of RED and DropTail needs to be observed by varying parameters like RTT and Load Size, I have not mentioned the delay and Bandwidth in the above topology. It is given in the command line.

In addition to the Round Trip Time and Load Size, a number of various parameters for RED and DropTail can be varied:

For DropTail:

- ➔ Queue Size
- ➔ Window Size

For RED:

- ➔ minTh (threshold of queue size beyond which every arriving packet is marked with a probability p_b)
- ➔ maxTh (Threshold of queue size beyond which every arriving packet is marked)
- ➔ max_p (Maximum probability of marking a packet – this would be expressed as Linterm = $1/max_p$ in NS3)
- ➔ Queue Length (set to 480 packets , which has been specified in [2] has the most appropriate value of queue length)
- ➔ Packet size (has been set 128 bytes) since I wanted to compare it with the maximum value of the window Size and queue Size of the dropTail queue.
- ➔ wq – This is a weight that controls how fast or how slow the average queue size of the RED queue changes

Results:

(Note: The value of RTT mentioned in all the experiments is just the delay of the links between n0-n1 and n3-n4, since in the code I have only kept the delay of those links variable, the total RTT would be the sum of the delays from the source to the destination and back to the source for all the sources. Also, changes in the UDP source data rate relates to changes in traffic)

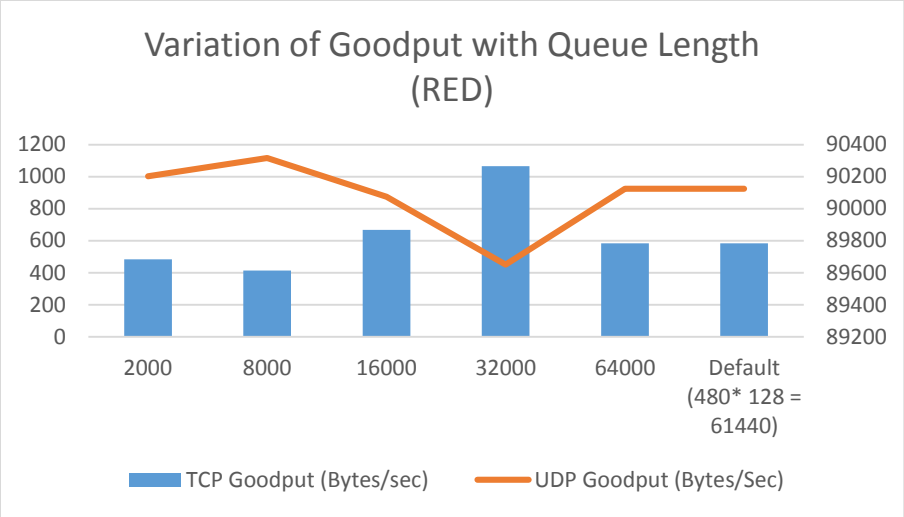
1. **RED vs Droptail Goodput for by Varying Queue Size**

So in this case I compared Droptail vs RED by varying queue size and keeping the other parameters constant.
RTT = 10 ms, wq = 1/128 (since packet size is 128 bytes), LInterm= 50 in the program (actually maxp= 1/50), minTh = 60 packets and maxTh=180 packets (I took this value from the [2]) and UDP source Data rate as 0.8 Mbps

For RED:

Queue length (bytes)	TCP Goodput (Bytes/sec)	UDP Goodput (Bytes/Sec)
2000	482.4	90201.6
8000	413.6	90316.8
16000	666.4	90073.6
32000	1064.8	89651.2
64000	582.4	90124.8
Default (480* 128 = 61440)	582.4	90124.8

I also plotted a graph for the above the table:



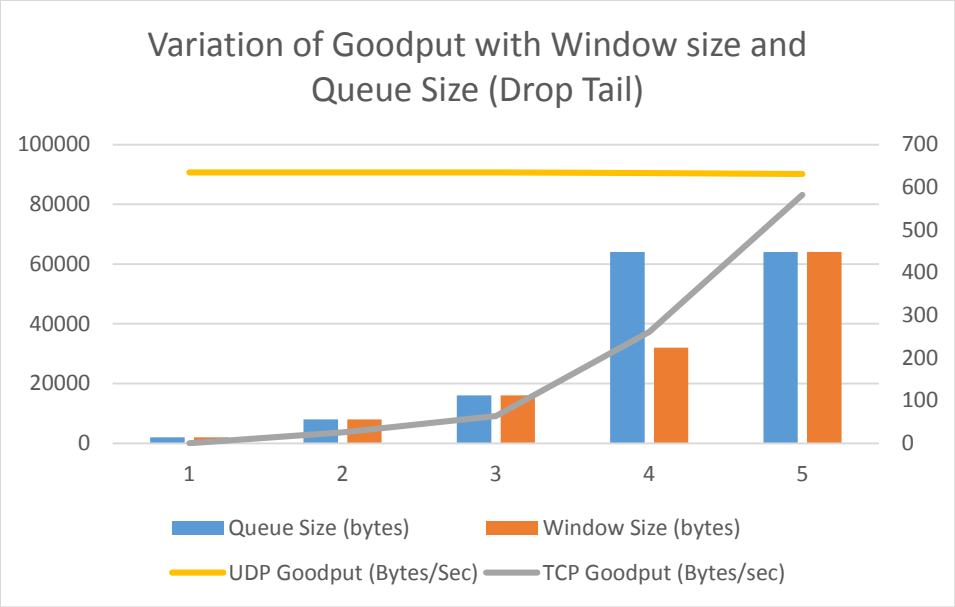
In the graph above, the scale on the left is for TCP goodput while the scale on the right is for UDP goodput. Thus, one can see that on taking into consideration the cumulative goodput (both TCP and UDP), the queue length of 64000 bytes performed the best. Also, the default value performed the same. I chose the default value as 61440 bytes (480*128) because it wouldn't exceed the maximum value of maxTh set by [2]
The reason why 64000 bytes (or 61440 bytes) performed the best was due to the fact that since I set minTh and maxTh as (60,180) packets, there weren't many drops of packets for these values of queues. When the queue length was arbitrarily small (say 2000, or 8000), the packets would have been marked almost immediately, leading to a huge drop in goodput. When the queue length is sufficiently large, the parameters minTh and maxTh come into play thereby decreasing the probability of dropping and increasing the goodput.

For DropTail, I varied the window size and Queue Size and kept constant RTT and UDP Source data rate as the case in the RED

I considered a few combinations of Queue Size and Window Size

Queue Size (bytes)	Window Size (bytes)	TCP Goodput (Bytes/sec)	UDP Goodput (Bytes/Sec)
2000	2000	0	90688
8000	8000	25.6	90675.2
16000	16000	64	90662.4
64000	32000	260.8	90457.6
64000	64000	582.4	90176

I plotted the graph for the table above. It is shown below:



In the graph above, the scale on the left is for the UDP goodput, window size and Queue Size and the graph on the right is for TCP goodput. We can see that the Queue Size and Window Size of 64000 bytes gives a good cumulative goodput (TCP and UDP together) This happens due to the fact that since queue size (i.e. congestion window) is large, more packets can be enqueued. Also, since the advertised receiver window (window Size) is large, more packets can be transmitted from the sources to the sink. Due to this, there will be less dropping of packets and thus goodput increases.

2. Variation of RED parameters

So from the previous section, I observed that window Size and Queue Size of 64000 gave the highest cumulative goodput.

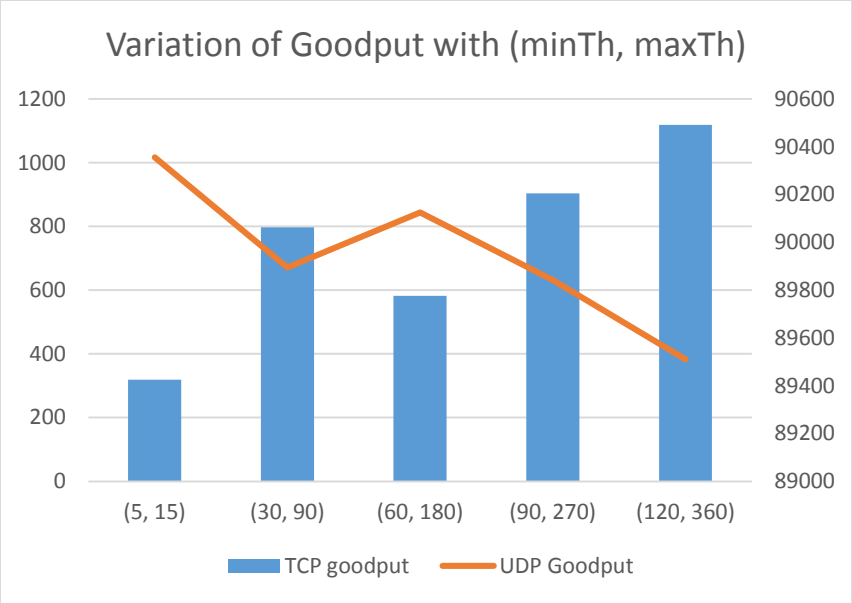
I then tried to tune RED to obtain the maximum Goodput and compare results with the best available goodput for DropTail. During this simulation, the constant values are RTT= 10ms (See: Note before section 1), wq = 1/128 (since packet size is 128 bytes) and qlen set to a value of 64000 bytes since it performed the same as a Queue Length of 64000 bytes. This was the specific case of (minTh, maxTh) as (60,180). I choose the default value as 61440 bytes as it should not exceed the range of maxTh as specified in [2]

➔ Varying minTh and maxTh to tune RED

I first tried varying minTh and maxTh to see the changes in goodput.

(minTh, maxTh)	TCP goodput	UDP Goodput
(5, 15)	319.2	90355.2
(30, 90)	796.8	89894.4
(60, 180)	582.4	90124.8
(90, 270)	904	89843.2
(120, 360)	1118.4	89510.4

I plotted a graph for the above tables and got the following:



In the graph above, the scale on the left is for TCP goodput and the scale on the right is for UDP goodput. Now, the variation of the Goodput is obvious. Initially, when (minTh,maxTh) is low (say 5,15), packets will be marked for dropping as soon the average queue length reaches 5 packets. When the queue size reaches 15 packets, the limit has been reached and when the 16th packet arrives, it is dropped. Thus, it leads to an early drop of packets and hence the goodput is less initially. On cumulatively considering both UDP and TCP flows, I found that the values of minTh and maxTh as (90,270). Higher values of (minTh, maxTh) might also improve the goodput, but only upto a certain limit. This is because if (minTh, maxTh) are increased to very high values, it is possible that the queue will start marking packets for dropping very late and this will decrease the goodput.

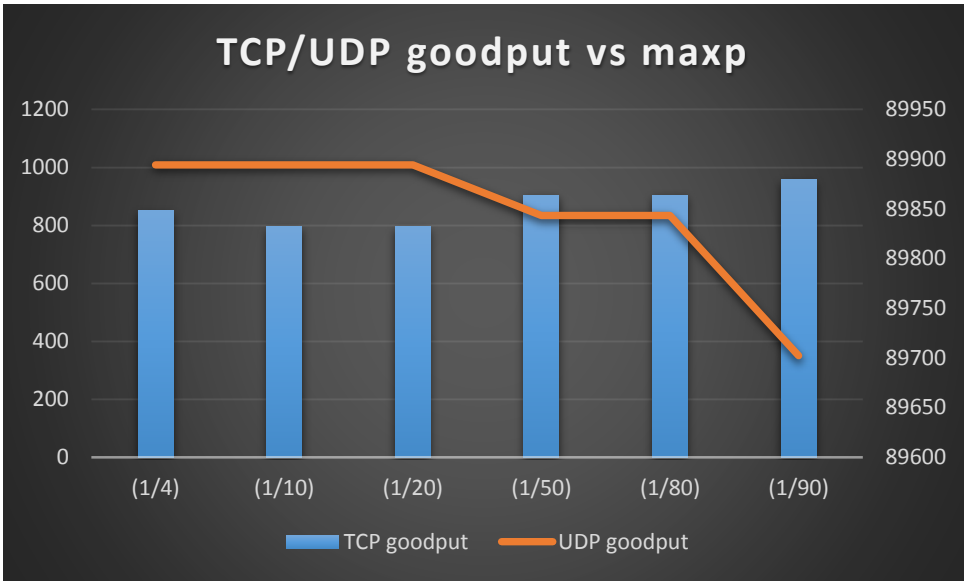
Thus cumulatively the (90,270) configuration was giving the best goodput of 90747.2 bytes/sec for this configuration.

➔ Varying LInterm (i.e. maxP) to tune this existing configuration.

Next, I wanted to see the variation of maxP. According to [2], the values of LInterm is set as 4, 10 and 20 (it is actually max_p = 1/4, 1/10, 1/20), I also used values of 1/50, 1/80 and 1/90

Maxp	TCP goodput	UDP goodput
1/4	850.4	89894.4
1/10	796.8	89894.4
1/20	796.8	89894.4
1/50	904	89843.2
1/80	904	89843.2
1/90	957.6	89702.4

The graph for the table above is as follows:



In the above graph, the scale on the left is for TCP goodput while the graph on the right is for UDP goodput. As seen from the previous experiments, the UDP flows are dominating the network while the TCP flows do not dominate at all. Hence by changing maxp, we can see that UDP flows have started dropping and hence its goodput decreases. Also, the TCP flows which were being subdued now start enjoying a slightly better share in the bandwidth, which can be seen by the increased goodput of TCP flows. I also tried varying the value of maxP from 1/90 to 1/100 and found no change in the goodput of TCP and UDP. Thus, I concluded the best value of maxp would be 1/90. Even though the cumulative goodput is much lesser than maxp being 1/50, maxp being 1/90 allows fairness among the TCP and UDP flows.

Thus, the best parameters that I got for RED that provided a goodput which was much better than the dropTail for RTT = 10 ms and UDP source data rate as 0.8 Mbps is

- (minTh,maxTh) = (90,270)
- maxp = 1/90
- wq = 1/128 (since the packet size is 128 bytes)
- qlen = 64000 bytes

Now, I delved more into what changes can occur in goodput when RTT and Data Rate of the UDP sources change. The change in the data Rate of the UDP sources relate to changes in traffic.

➔ For RED

For these experiments, I kept (minTh,maxTh) = (90,270), maxp = 1/90, wq = 1/128 (since the packet size is 128 bytes), qlen = 64000 bytes.

1. Varying RTT (kept UDP source data rate at 0.8 Mbps) for RED

Please Note that the value of RTT mentioned in all the experiments is just the delay of the links between n0-n1 and n3-n4, since in the code I have only kept the delay of those links variable, the total RTT would be the sum of the delays from the source to the destination and back to the source for all the sources.

The results are tabulated below:

RTT (ms)	TCP goodput (bytes/sec)	UDP goodput (bytes/sec)
10	957.6	89702.4
30	314.4	90073.6
50	421.6	89728
120	263.2	89305.6

Thus, from the table it can be inferred that as RTT increases the cumulative goodput decreases. This happens due to the fact that an increased RTT means that there will more packets in flight. It means that there would be more delay in receiving ACKs and it might be possible that there might be a timeout at the sender and hence the sender might re-transmit, thereby decreasing goodput.

2. Changes to UDP source Data Rate (keeping RTT constant at 10 ms) for RED

This corresponds to changes in traffic in the network
The results are tabulated below:

Data Rate (Mbps)	TCP goodput (bytes/sec)	UDP goodput (bytes/sec)
0.8	957.6	89702.4
1	421.6	90124.8
2	51.2	90688

Thus, we can see that on increasing the UDP traffic, the UDP goodput also increased. The UDP traffic dominates the flows. The TCP flows decrease because it senses more congestion in the network and hence implements the congestion control protocols alongwith RED queuing.

➔ For DropTail (window Size = 64000 bytes and Queue Size= 64000 bytes)

To compare the performance of Droptail with RED for changes with RTT and Data Rate, I performed the following experiments.

1. Varying RTT (kept UDP source data rate at 0.8 Mbps) for Droptail

The results are tabulated below:

RTT (ms)	TCP goodput (bytes/sec)	UDP goodput (bytes/sec)
10	582.4	90176
30	260.8	90265.6
50	260.8	90060.8
120	102.4	89510.4

On comparison with RED, we can see that the Goodput with variation in RTT is much worse.

2. Varying UDP source Data Rate (keeping RTT as 10 ms) for Droptail

Data rate (Mbps)	TCP Goodput (Bytes/sec)	UDP goodput (byes/sec)
0.8	582.4	90176
1	260.8	90470.4
2	51.2	90688

Varying UDP source data rate for droptail did make the Goodput very poor as compared to RED. The final case of 2 Mbps gave the same TCP and UDP Goodput. Thus, RED had no clear advantage over Droptail in this case.

Conclusion:

This in this project, we had to compare RED vs Droptail Queueing. [1] and [2] provide excellent insights on the working of RED and how its performance can be evaluated. [2] was the sole motivation behind the values I used for the experiments to tune RED for better performance. I tried to do all comparisons with a Droptail Queue which was set to a window size and queue size of 64000 bytes. From the above experiments, I could conclude that for the particular condition of Data Rate and RTT that was set, the RED queuing method was slightly better than the droptail queuing method. The goodput obtained by the RED queueing

method was slightly better than the DropTail queueing method with the added advantage that RED queuing has a number of parameters that can be adjusted to obtain a desirable performance. Also, RED can provide fairness between flows by changing maxp. Thus, in the experiments that I performed and strictly under the parameters I set, RED performed slightly better. It is possible that DropTail might perform better for another set of values.

References:
[1] *Random Early Detection Gateways for Congestion Avoidance*, Sally Floyd and Van Jacobson
[2] *Tuning RED for Web Traffic*, Mikkel Christiansen, Kevin Jeffay, David Ott and F.Donelson Smith