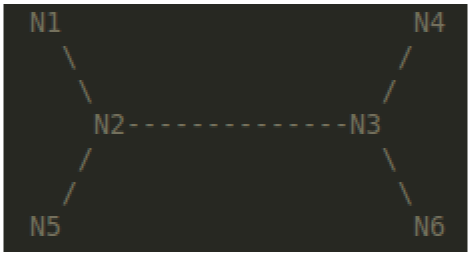
CS 276 HW 1 Report

Wei Dai
Fall 2014

Part 1 - Fairness Between TCP Variants

This portion of the experiment consists of an investigation into the fairness between TCP variations. Tahoe/Tahoe, Reno/Reno, NewReno/Reno, Vegas/Vegas, and NewReno/Vegas variants are investigated. A six node NS-2 network topology with 10Mb links is established as follows below:

A CBR is at N2, with a corresponding sink at N3. N1 shares a TCP stream with N4, as does N5 and N6. In this fashion, packet loss rate and bandwidth consumed by the three flows were measured, and then compared with the bandwidth of the CBR flow from N2 to N3. This allowed the variation of the CBR flow in order to judge how the other nodes' bandwidths were affected.



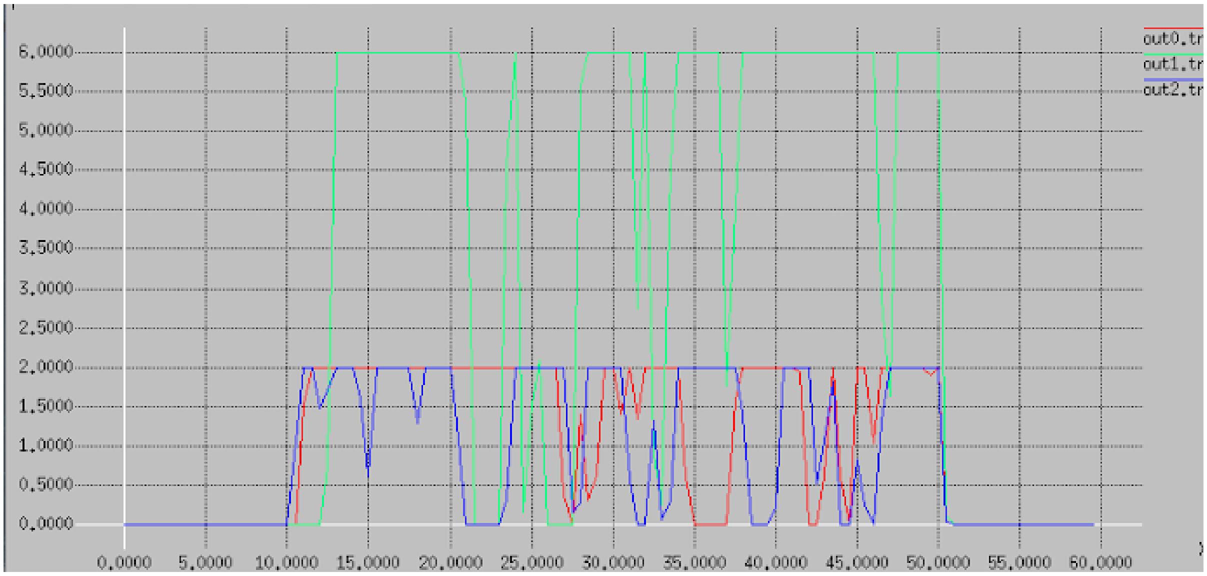


Figure 1 - Bandwidth usage in Mb, where out0.tr is N1-N4 [2Mb], out1.tr is N2-N3 [6Mb], out2.tr is N5-N6 [2Mb]

Shown above is a graph of bandwidth usage, where the CBR source is allocated 6Mb of bandwidth, and the other two TCP streams 2Mb, respectively. The behavior is as expected, since the link between N2 and N3 is 10Mb, typically each stream is allowed to operate at maximum capacity, right before congestion occurs.

On the next page in Figure 2, is an example of the network with some congestion, where the CBR is raised to 9Mb, with the other two sources remaining consistent. Congestion is immediately noticeable when compared to the previous figure, as the link remains capped at 10Mb.

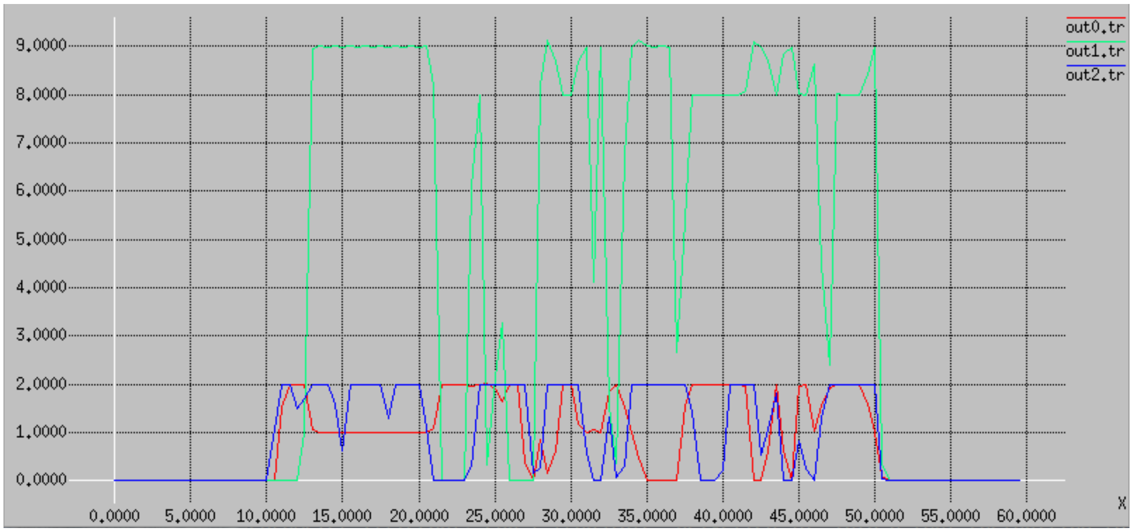


Figure 2 - Bandwidth usage in Mb, where out0.tr is N1-N4 [2Mb], out1.tr is N2-N3 [9Mb], out2.tr is N5-N6 [2Mb]

Lastly, below in Figure 3 is an example illustrating what happens when the network is totally congested, with all 3 nodes vying for bandwidth at 15Mb rates.

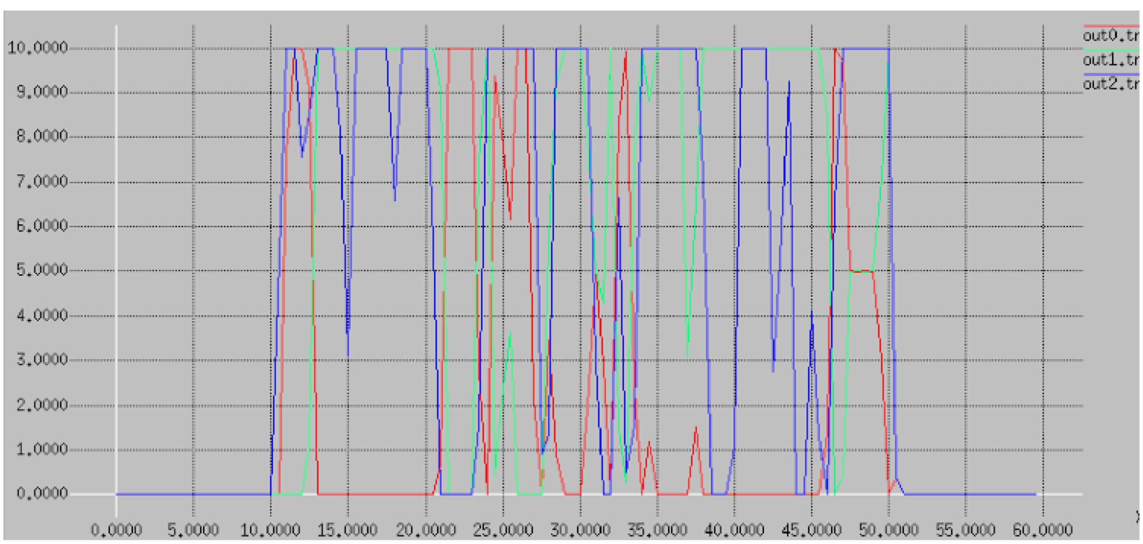


Figure 3 - Bandwidth usage in Mb, where out0.tr is N1-N4 [15Mb], out1.tr is N2-N3 [15Mb], out2.tr is N5-N6 [15Mb]

While the figures do not present anything dramatically significant, it is of note to mention that already there are traces of unfairness presenting itself, with the CBR bandwidth hog in Figures 1 and 2 largely dominating the other two streams. Despite the fact it is using more bandwidth than the other two combined, it still takes a sort of priority in the way it slows down the other two connections in order to maintain its relatively higher bandwidth. That is not to say it does not become slowed down at times, because it does also suffer from the global network congestion.

In the default Tahoe case where the total allocated bandwidths totaled to 10Mb, at the brink of congestion, there was no packet loss. This is an intuitive result as exactly the right amount of bandwidth was being used. However, when the CBR rate from N2 to N3 is increased marginally from 6Mb to 8Mb, which pushes the bandwidth over the 10Mb link cap, we begin experiencing packet loss, as demonstrated in Figure 4, below.

Name	ID	Gen. Packets	Gen. Bytes	T. Lost Packets	T. Lost Bytes
tcp	14	1967	2 Mb	14	14 Kb
ack	14	1953	76 Kb	6	240 bytes
cbr	23	8138	8 Mb	297	290 Kb
tcp	56	1965	2 Mb	14	14 Kb
ack	56	1951	76 Kb	6	240 bytes

Figure 4 - Graph of packet loss [CBR 8Mb]. ID represents node relation, e.g. 14 is N1 to N4

Gradually increasing the bandwidth taken by the CBR source, from 8Mb rates to 12Mb, more packets were dropped. However, the majority of the dropped packets were from the CBR source. This is a natural conclusion, as most of the packets sent through the channel were by the CBR source. This can be seen in the figure below.

Name	ID	Gen. Packets	Gen. Bytes	T. Lost Packets	T. Lost Bytes
tcp	14	90	90 Kb	29	29 Kb
ack	14	69	3 Kb	0	0 bytes
cbr	23	12788	12 Mb	1222	1 Mb
tcp	56	49	49 Kb	14	14 Kb
ack	56	37	1 Kb	0	0 bytes

Figure 5 - Graph of packet loss [CBR 12Mb]. ID represents node relation, e.g. 14 is N1 to N4

Moving along, next the other variants of TCP were examined at the same congestion levels with a CBR source of 9Mb, with the other TCP streams operating at a rate of 2Mb. First up, is the Reno/Reno scheme.

Name	ID	Gen. Packets	Gen. Bytes	T. Lost Packets	T. Lost Bytes
tcp	14	712	722 Kb	32	33 Kb
ack	14	693	27 Kb	0	0 bytes
cbr	23	9300	9 Mb	92	90 Kb
tcp	56	1881	2 Mb	15	15 Kb
ack	56	1870	73 Kb	9	360 bytes

Figure 6 - Reno/Reno [CBR 9Mb]

As expected, in Figure 6, the Reno/Reno configuration improves drastically upon the original Tahoe set up. This is probably due to the fact that it is optimized for when there is only one packet dropped, however there is still a significant amount of packet loss due to situations where there may be multiple packets dropped from a data window.

The next experiment is with the NewReno/Reno set up. We would expect some sort of performance gain over Reno/Reno, and there is a marginal decrease in packets dropped, as expected and shown below in Figure 7.

Name	ID	Gen. Packets	Gen. Bytes	T. Lost Packets	T. Lost Bytes
tcp	14	1520	2 Mb	18	18 Kb
ack	14	1510	59 Kb	17	680 bytes
cbr	23	9300	9 Mb	82	80 Kb
tcp	56	1060	1 Mb	28	28 Kb
ack	56	1043	41 Kb	2	80 bytes

Figure 7 - NewReno/Reno [CBR 9Mb]

Where Reno and NewReno detect congestion after it has happened via packet drops, Vegas attempts to detect congestion before it happens based upon increasing RTTs. As such, we would expect to have fewer dropped packages, as this algorithm is meant to decrease packet loss and stop congestion before it happens. In Figure 8, the results are shown as expected.

Name	ID	Gen. Packets	Gen. Bytes	T. Lost Packets	T. Lost Bytes
tcp	14	1331	1 Mb	13	13 Kb
ack	14	1318	51 Kb	3	120 bytes
cbr	23	9300	9 Mb	3	3 Kb
tcp	56	1151	1 Mb	1	1000 bytes
ack	56	1150	45 Kb	14	560 bytes

Figure 8 - Vegas/Vegas [CBR 9Mb]

There is quite a dramatic decrease in packet loss for Vegas/Vegas compared to all the other variants reviewed in this experiment. As such, it follows that a mixture of NewReno/Vegas would not be as effective as a full Vegas configuration in terms of packet loss, or perhaps, lack thereof. The NewReno/Vegas results are displayed below in Figure 9.

Name	ID	Gen. Packets	Gen. Bytes	T. Lost Packets	T. Lost Bytes
tcp	14	1772	2 Mb	15	15 Kb
ack	14	1763	69 Kb	14	560 bytes
cbr	23	9300	9 Mb	12	12 Kb
tcp	56	700	684 Kb	6	6 Kb
ack	56	694	27 Kb	7	280 bytes

Figure 9 - NewReno/Vegas [CBR 9Mb]

From the investigations into Tahoe/Tahoe, Reno/Reno, NewReno/Reno, Vegas/Vegas, and NewReno/Vegas, it can be said that for the particular simulation conducted, Vegas/Vegas was most effective in terms of reducing packet loss.

When taking away one TCP stream and leaving a single flow of each TCP variant (Reno, NewReno, Vegas), the results were less dramatic. Vegas still appeared to be most effective once congestion was ramped up. Since the stream between N1 and N4 was removed, the stream from N5 to N6 was increased to a rate of 5Mb, and the CBR rate was ramped up steadily until 9Mb, from which good congestion results were finally observed.

Name	ID	Gen. Packets	Gen. Bytes	T. Lost Packets	T. Lost Bytes
cbr	23	16088	15 Mb	14	14 Kb
tcp	56	1746	2 Mb	54	55 Kb
ack	56	1708	67 Kb	11	440 bytes

Figure 10 - Reno @ 5Mb [CBR 9Mb]

Name	ID	Gen. Packets	Gen. Bytes	T. Lost Packets	T. Lost Bytes
cbr	23	16088	15 Mb	15	15 Kb
tcp	56	1890	2 Mb	52	53 Kb
ack	56	1854	72 Kb	13	520 bytes

Figure 11 - NewReno @ 5Mb [CBR 9Mb]

Name	ID	Gen. Packets	Gen. Bytes	T. Lost Packets	T. Lost Bytes
cbr	23	16088	15 Mb	15	15 Kb
tcp	56	1880	2 Mb	11	11 Kb
ack	56	1869	73 Kb	4	160 bytes

Figure 11 - Vegas @ 5Mb [CBR 9Mb]

While Figures 10 and 11 depict Reno and NewReno as having similar results with the single stream, again, Vegas is the standout performer in this category under congestion.

Part 2 - Influence of Queuing

This portion of the experiment is mainly dependant upon comparisons between DropTail and Random Early Drop (RED) queuing algorithms. Using the same topology as earlier, the TCP flow was initialized first and stabilized, with the UDP and CBR flows beginning afterwards. This was tested with both TCP Reno and SACK variants.

In the case of DropTail, when the network became congested, after the queue became full the rest of the traffic is dropped. This results in a large cap of packet loss after the buffer threshold. The throughput when the network is not congested, is resultingly pretty good.

Displayed in Figure 2.1 below, it appears that DropTail drops a number of packets in both Reno and SACK.

Name	ID	Gen. Packets	Gen. Bytes	T. Lost Packets	T. Lost Bytes
tcp	14	802	814 Kb	37	38 Kb
ack	14	786	31 Kb	9	360 bytes
cbr	23	8500	8 Mb	425	415 Kb
cbr	56	4101	2 Mb	114	56 Kb

Figure 2.1 - DropTail w/ Reno

For RED, packets are dropped from flows before the congestion cap is reached so that it is able to provide a more overall stable performance. RED also seemingly is more fair than the DropTail variant due to relegating some queue sizes in order to drop packets earlier, with the larger bandwidth use having a proportional drop rate. As seen in Figure 2.2, below, there is significantly fewer packets dropped with RED. As a significant trade-off though, is the throughput. As we can see a much larger about of packets were generated in Figure 2.1.

Name	ID	Gen. Packets	Gen. Bytes	T. Lost Packets	T. Lost Bytes
tcp	14	324	328 Kb	6	6 Kb
ack	14	320	13 Kb	0	0 bytes
cbr	23	8500	8 Mb	18	18 Kb
cbr	56	4101	2 Mb	8	4 Kb

Figure 2.2 - RED w/ Reno

Next, the topology is shifted to have a 500 byte packet UDP flow, and a 1000 byte packet TCP flow. Both Reno TCP and SACK are used again, but this time with a middle link of only 1.5Mbps. A comparison between Reno and SACK prove that the results are similar, but again, the difference between DropTail and RED are significant.

Name	ID	Gen. Packets	Gen. Bytes	T. Lost Packets	T. Lost Bytes
tcp	14	320	324 Kb	29	29 Kb
ack	14	304	12 Kb	1	40 bytes
cbr	23	1063	1 Mb	140	137 Kb
cbr	56	1026	501 Kb	55	27 Kb

Figure 2.3 - RED w/ Reno

This time with the bandwidth link limit capped at 1.5Mbps, more packets were dropped despite the fact the other streams had lower rates as well. It is of great note however, that this time DropTail is the better performer both under Reno and SACK. As seen in Figure 2.4, the number of packets lost is far fewer than that in Figure 2.3. This is most likely due to the fact that since we have decreased the link down to 1.5Mbps, the DropTail algorithm's buffer is able to handle most of the traffic despite the congestion. When it was at 10Mbps, the rate was simply too high to handle. For a lower level of traffic congestion, the DropTail scheme actually works quite well.

Name	ID	Gen. Packets	Gen. Bytes	T. Lost Packets	T. Lost Bytes
tcp	14	192	194 Kb	20	20 Kb
ack	14	177	7 Kb	2	80 bytes
cbr	23	1063	1 Mb	21	21 Kb
cbr	56	1026	501 Kb	2	1000 bytes

Figure 2.4 - DropTail w/ Reno

The next portion of this experiment involves a topology change. Now three UDP flows with a shared 1.5Mbps link is used. N1 and N7 are sending 1000 byte packets at 1Mbps, where the last flow is sending 500 byte packets at 0.6Mbps. They begin sending at 0.0, 0.1, and 0.2 secs respectively.

```
     N1                        N4
      \                        /
       \                      /
N7------N2-------------N3--------N8
       /                      \
      /                        \
     N5                        N6
```

Using the DropTail queuing algorithm, some results are displayed as follows:

Name	ID	Gen. Packets	Gen. Bytes	T. Lost Packets	T. Lost Bytes
cbr	14	1188	1 Mb	382	373 Kb
cbr	56	1176	1 Mb	202	197 Kb
cbr	78	166	81 Kb	76	37 Kb

Figure 2.5 - Droptail w/ 3 UDP Schema, IDs 14 and 56 @ 1Mb, and 78 @ 0.6Mb

As expected, Connection IDs 14 and 56, running at 1Mbps, performed roughly the same. Some data generated about Connection ID 14 is presented below, as Figure 2.6.

Generated

Packets: 1188
Data: 1 Mb (1188000 bytes)
Ratio: 46.956522% (packets)
48.549244% (bytes)

Dropped

Packets: 382
Data: 373 Kb (382000 bytes)
Ratio: 32.154882% (packets)
32.154882% (bytes)

Packets Delay

Minimum: 0.046 (seconds)
Average: 0.2883525 (seconds)
Maximum: 0.307333 (seconds)

Packets Jitter

Minimum: 0 (seconds)
Average: 0.0015249615 (seconds)
Maximum: 0.008 (seconds)

Throughput Transferred

Minimum: 50 KB/s (51000 B/s)
Average: 79 KB/s (80600.00 B/s)
Maximum: 122 KB/s (125000 B/s)

Throughput Generated

Minimum: 62 KB/s (63000 B/s)
Average: 116 KB/s (118800.00 B/s)
Maximum: 122 KB/s (125000 B/s)

Figure 2.6 - Droptail @ Connection ID 14 [1Mbps]

On the next page, there are more in depth statistics about the lesser UDP stream 78 in Figure 2.7, running at 0.6Mbps and sending 500, rather than 1000 byte packets. It is clear that this Droptail scheme seems to be favoring the more heavyweight connections, as they generate more throughput, yet drop a smaller ratio of generated packets. The only thing the lesser of the three streams has an advantage in is a very slightly lower average packet delay time, clocking in at ~0.2203 seconds vs. ~0.2884 seconds.

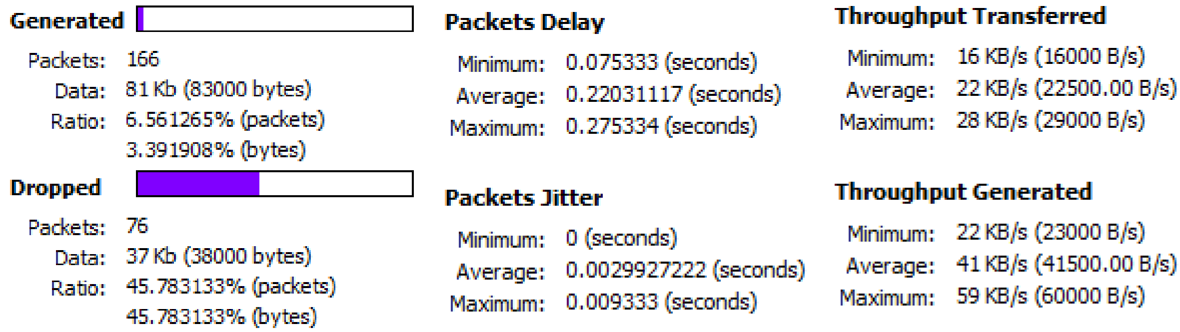


Figure 2.7 - Droptail @ Connection ID 78 [0.6Mbps]

When it comes to RED, shown in Figure 2.8, there appears to be a comparable amount of packet loss on average, but it looks like the distribution may be more fair, with the loss more proportionate to traffic generated. It will take a closer look to make sure.

Name	ID	Gen. Packets	Gen. Bytes	T. Lost Packets	T. Lost Bytes
cbr	14	1188	1 Mb	325	317 Kb
cbr	56	1176	1 Mb	315	308 Kb
cbr	78	166	81 Kb	43	21 Kb

Figure 2.8 - RED w/ 3 UDP Schema, IDs 14 and 56 @ 1Mb, and 78 @ 0.6Mb

Displayed below in Figure 2.9 is more detailed information from the perspective of Connection 78, which is the 1Mbps at 1000 byte packet connection. The statistics are very similar to that of DropTail, but with a margin performance gain in terms of fewer dropped packets.

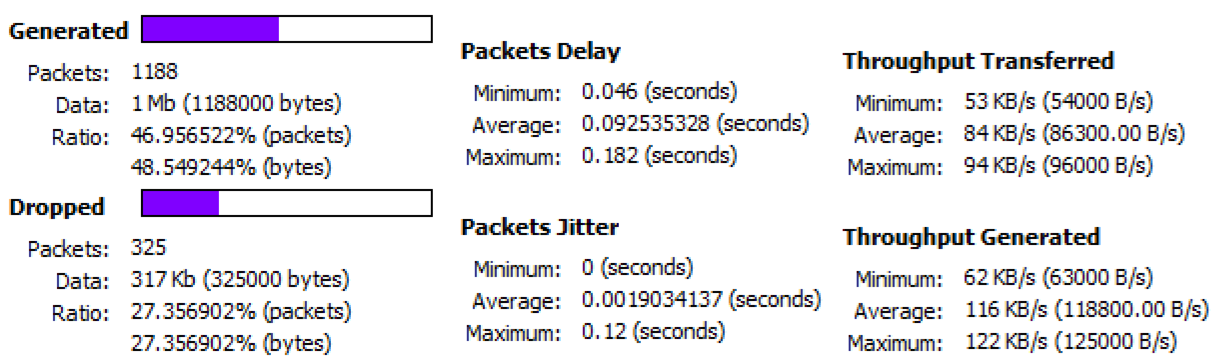


Figure 2.9 - RED @ Connection ID 14 [1Mbps]

Next, Figure 2.10 on the following page proves our hypothesis correct, that RED provides increased fairness among flows, in that packet dropping is made proportional to the amount of bandwidth used per given flow. The drop percentage of Connection 78 for

DropTail was ~46 percent, but while using RED, the packet drop percentage is almost halved to a mere ~26 percent.

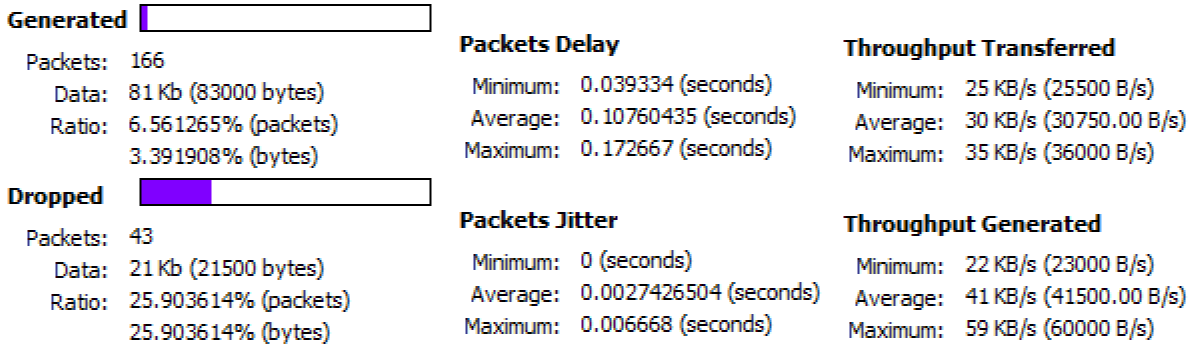


Figure 2.10 - RED @ Connection ID 78 [0.6Mbps]

Additionally, in the grand scheme of things, RED seems to decrease average packet delay for both Connections 14 and 78. There is a significant decrease from ~0.29 to ~0.09, and ~0.22 to ~0.11 seconds, respectively. The delay values are just about cut in half.

As seen from these experiments, some queuing disciplines provide a more fair distribution of bandwidth to each flow. As seen above, RED is far more fair than DropTail. However, in the proper situation, DropTail can be better performance-wise, as seen by the better performance when the common link was decreased to 1.5Mbps.

In general, TCP flows react poorly towards the creation of another CBR flow and back off. This is due to the fact that TCP is directly affected by bandwidth, congestion, buffering, and packet size, let alone the interaction with other flows, such as CBR flows. When congestion arrives due to the CBR flow, packets are lost, and TCP must restart. On the other hand, UDP does not react to packet losses the same way.

Lastly, SACK does not cope well with RED. This is because SACK has a pipe which tries to keep the estimated number of packets outstanding in the path, which can be thrown off with the selective acknowledgement approach.

Appendix

This is a compendium of the code for the project. There are in total more than near 20 separate files to represent each case so in the interest of space I will only attach the two base topologies for parts 1 and 2. The rest of the files may be found a zip attachment.

cs276hw1__p1_2_newreno.tcl

```
# Create a simulator object
set ns [new Simulator]

# Open the nam trace file
set nf [open hw1.nam w]
```

```
$ns namtrace-all $nf

#Open the trace file (before you start the experiment!)
set tf [open p2trace.tr w]
$ns trace-all $tf

#Define a 'finish' procedure
proc finish {} {
	global ns nf
	$ns flush-trace
	#Close the NAM trace file
	close $nf
	#Execute NAM on the trace file
	exec nam hw1.nam &
	exit 0
}

# Agent/TCP - a ``tahoe'' TCP sender
# Agent/TCP/Reno - a ``Reno'' TCP sender
# Agent/TCP/Newreno - Reno with a modification
# Agent/TCP/Sack1 - TCP with selective repeat (follows RFC2018)
# Agent/TCP/Vegas - TCP Vegas
# Agent/TCP/Fack - Reno TCP with ``forward acknowledgment''
# Agent/TCP/Linux - a TCP sender with SACK support that runs TCP congestion control modules from Linux
kernel
# The one-way TCP receiving agents currently supported are:
# Agent/TCPSink - TCP sink with one ACK per packet
# Agent/TCPSink/DelAck - TCP sink with configurable delay per ACK
# Agent/TCPSink/Sack1 - selective ACK sink (follows RFC2018)
# Agent/TCPSink/Sack1/DelAck - Sack1 with DelAck
# The two-way experimental sender currently supports only a Reno form of TCP:
# Agent/TCP/FullTcp

# Insert your own code for topology creation
# and agent definitions, etc. here
################################################################################
######
#            N1            N4
#             \           /
#              \         /
#               N2--------------N3
#              /         \
#             /           \
#            N5            N6

$ns color 23 Blue
$ns color 14 Green
$ns color 56 Red
```

```
# Create six nodes
set n1 [$ns node]
set n2 [$ns node]
set n3 [$ns node]
set n4 [$ns node]
set n5 [$ns node]
set n6 [$ns node]

# Create a duplex link between the nodes
$ns duplex-link $n1 $n2 10Mb 10ms DropTail
$ns duplex-link $n2 $n5 10Mb 10ms DropTail
$ns duplex-link $n2 $n3 10Mb 10ms DropTail
$ns duplex-link $n3 $n4 10Mb 10ms DropTail
$ns duplex-link $n3 $n6 10Mb 10ms DropTail

# Topology
$ns duplex-link-op $n1 $n2 orient right-down
$ns duplex-link-op $n2 $n5 orient left-down
$ns duplex-link-op $n2 $n3 orient right
$ns duplex-link-op $n3 $n4 orient right-up
$ns duplex-link-op $n3 $n6 orient right-down

#Setup a UDP connection N2-N3
set udp [new Agent/UDP]
$ns attach-agent $n2 $udp
set null [new Agent/Null]
$ns attach-agent $n3 $null
$ns connect $udp $null
$udp set fid_ 23

#Setup a CBR over UDP connection
set cbr [new Application/Traffic/CBR]
$cbr attach-agent $udp
$cbr set type_ CBR
$cbr set packet_size_ 1000
$cbr set rate_ 9mb
$cbr set random_ false

# Setup a TCP connection N5-N6
set tcp56 [new Agent/TCP/Newreno]
$tcp56 set class_ 2
$ns attach-agent $n5 $tcp56
set sink6 [new Agent/TCPSink]
$ns attach-agent $n6 $sink6
$ns connect $tcp56 $sink6
$tcp56 set fid_ 56
```

```
# Setup a CBR over TCP connection
set cbr56 [new Application/Traffic/CBR]
$cbr56 attach-agent $tcp56
$cbr56 set type_ CBR
$cbr56 set packet_size_ 1000
$cbr56 set rate_ 5mb
$cbr56 set random_ false


##############################################################################
######
# Call the finish procedure after 5 seconds simulation time

$ns at 0.2 "$cbr start"
$ns at 0.2 "$cbr56 start"
$ns at 14.5 "$cbr56 stop"
$ns at 14.5 "$cbr stop"
$ns at 15.0 "finish"

# Print CBR packet size and interval
puts "CBR packet size = [$cbr set packet_size_]"
puts "CBR interval = [$cbr set interval_]"

# Run the simulation
$ns run

close $tf
```

cs276hw1__p2_3udpred.tcl

```
# Create a simulator object
set ns [new Simulator]

# Open the nam trace file
set nf [open hw1.nam w]
$ns namtrace-all $nf

#Open the trace file (before you start the experiment!)
set tf [open p23trace.tr w]
$ns trace-all $tf

#Define a 'finish' procedure
proc finish {} {
    global ns nf
    $ns flush-trace
    #Close the NAM trace file
    close $nf
    #Execute NAM on the trace file
```

```
    exec nam hw1.nam &
    exit 0
}

# Agent/TCP - a ``tahoe'' TCP sender
# Agent/TCP/Reno - a ``Reno'' TCP sender
# Agent/TCP/Newreno - Reno with a modification
# Agent/TCP/Sack1 - TCP with selective repeat (follows RFC2018)
# Agent/TCP/Vegas - TCP Vegas
# Agent/TCP/Fack - Reno TCP with ``forward acknowledgment''
# Agent/TCP/Linux - a TCP sender with SACK support that runs TCP congestion control modules from Linux
kernel
# The one-way TCP receiving agents currently supported are:
# Agent/TCPSink - TCP sink with one ACK per packet
# Agent/TCPSink/DelAck - TCP sink with configurable delay per ACK
# Agent/TCPSink/Sack1 - selective ACK sink (follows RFC2018)
# Agent/TCPSink/Sack1/DelAck - Sack1 with DelAck
# The two-way experimental sender currently supports only a Reno form of TCP:
# Agent/TCP/FullTcp

# Insert your own code for topology creation
# and agent definitions, etc. here
###############################################################################
######
 #                                                  N1                N4
 #                 \             /
 #                  \           /
 #            N7------N2--------------N3--------N8
 #                  /           \
 #                 /             \
 #               N5               N6

$ns color 23 Blue
$ns color 14 Green
$ns color 56 Red

# Create six nodes
set n1 [$ns node]
set n2 [$ns node]
set n3 [$ns node]
set n4 [$ns node]
set n5 [$ns node]
set n6 [$ns node]
set n7 [$ns node]
set n8 [$ns node]

# Create a duplex link between the nodes
$ns duplex-link $n1 $n2 1.5Mb 10ms RED
```

```
$ns duplex-link $n2 $n5 1.5Mb 10ms RED
$ns duplex-link $n2 $n3 1.5Mb 10ms RED
$ns duplex-link $n3 $n4 1.5Mb 10ms RED
$ns duplex-link $n3 $n6 1.5Mb 10ms RED
$ns duplex-link $n7 $n2 1.5Mb 10ms RED
$ns duplex-link $n3 $n8 1.5Mb 10ms RED

# Topology
$ns duplex-link-op $n1 $n2 orient right-down
$ns duplex-link-op $n2 $n5 orient left-down
$ns duplex-link-op $n2 $n3 orient right
$ns duplex-link-op $n3 $n4 orient right-up
$ns duplex-link-op $n3 $n6 orient right-down
$ns duplex-link-op $n3 $n8 orient right
$ns duplex-link-op $n2 $n7 orient left

# #Setup a UDP connection N2-N3
# set udp [new Agent/UDP]
# $ns attach-agent $n2 $udp
# set null [new Agent/Null]
# $ns attach-agent $n3 $null
# $ns connect $udp $null
# $udp set fid_ 23

# #Setup a CBR over UDP connection
# set cbr [new Application/Traffic/CBR]
# $cbr attach-agent $udp
# $cbr set type_ CBR
# $cbr set packet_size_ 1000
# $cbr set rate_ 8mb
# $cbr set random_ false

# Setup a UDP connection N1-N4
set udp14 [new Agent/UDP]
$udp14 set class_ 2
$ns attach-agent $n1 $udp14
set sink4 [new Agent/Null]
$ns attach-agent $n4 $sink4
$ns connect $udp14 $sink4
$udp14 set fid_ 14

# Setup a UDP connection N5-N6
set udp56 [new Agent/UDP]
$ns attach-agent $n5 $udp56
set sink6 [new Agent/Null]
$ns attach-agent $n6 $sink6
$ns connect $udp56 $sink6
$udp56 set fid_ 56
```

```
# Setup a UDP connection N7-N8
set udp78 [new Agent/UDP]
$ns attach-agent $n7 $udp78
set sink8 [new Agent/Null]
$ns attach-agent $n8 $sink8
$ns connect $udp78 $sink8
$udp78 set fid_ 78

# Setup a CBR over UDP connection
set cbr14 [new Application/Traffic/CBR]
$cbr14 attach-agent $udp14
$cbr14 set type_ CBR
$cbr14 set packet_size_ 1000
$cbr14 set rate_ 1mb
$cbr14 set random_ false

# Setup a CBR over UDP connection
set cbr56 [new Application/Traffic/CBR]
$cbr56 attach-agent $udp56
$cbr56 set type_ CBR
$cbr56 set packet_size_ 1000
$cbr56 set rate_ 1mb
$cbr56 set random_ false

# Setup a CBR over UDP connection
set cbr78 [new Application/Traffic/CBR]
$cbr78 attach-agent $udp78
$cbr78 set type_ CBR
$cbr78 set packet_size_ 500
$cbr78 set rate_ 0.6mb
$cbr78 set random_ false

##############################################################################
######
# Call the finish procedure after 5 seconds simulation time

# $ns at 1.0 "$cbr start"
$ns at 0.0 "$cbr14 start"
$ns at 0.1 "$cbr56 start"
$ns at 0.2 "$cbr78 start"
$ns at 9.5 "$cbr14 stop"
$ns at 9.5 "$cbr56 stop"
$ns at 1.3 "$cbr78 stop"
# $ns at 9.5 "$cbr stop"
$ns at 10.0 "finish"

# Print CBR packet size and interval
```

```
# puts "CBR packet size = [$cbr set packet_size_]"
# puts "CBR interval = [$cbr set interval_]"

# Run the simulation
$ns run

close $tf
```