ECE-6610 Programming Assignment 2

Teammate:
Ching-Kai Liang (cliang30@gatech.edu)
Huangwei Fang (hfang@gatech.edu)
Wan-Chen Yeh (wyeh7@gatech.edu)
Yash Shah (shah_yash10@gatech.edu)
Yiling Yin (yyin60@gatech.edu)

Part I: TCP Flavor Experiment

1) Total TCP segments received and throughput. The segment count is for all received segments, including retransmission. Throughput calculation includes TCP header and retransmitted data as well.

   TCP/Tahoe:
   Flow1: 22,317 segments, 446.179 KB/s
   Flow2: 46,236 segments, 558.732 KB/s
   Total: 68,553 segments, 1005 KB/s

   TCP/Reno:
   Flow1: 26,343 segments, 410.726 KB/s
   Flow2: 40,166 segments, 588.079 KB/s
   Total: 66,509 segments, 999 KB/s

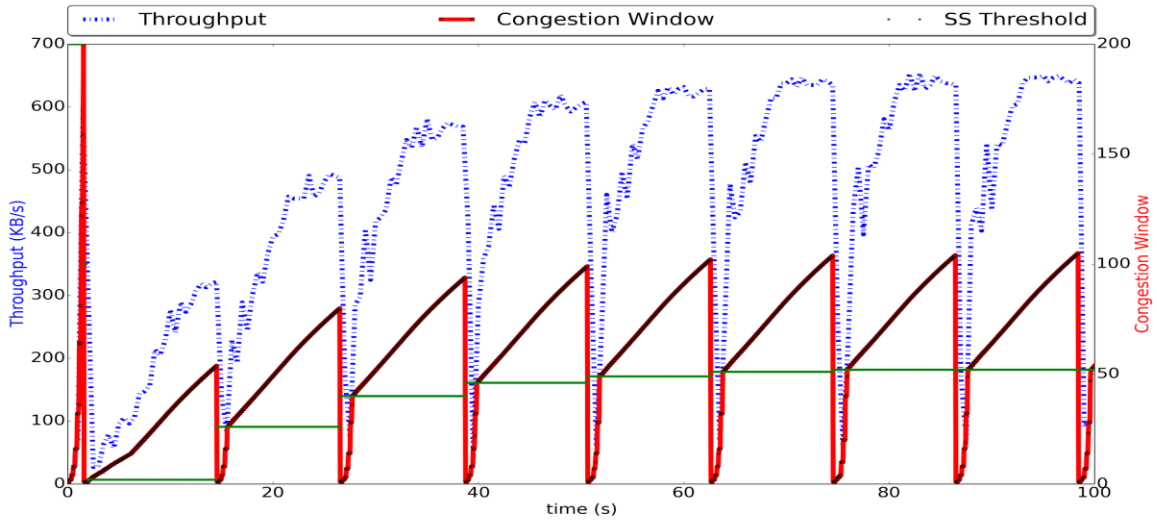
   TCP/New-Reno:
   Flow1: 33,743 segments, 500.647 KB/s
   Flow2: 38,923 segments, 607.633 KB/s
   Total: 72,666 segments, 1108 KB/s

   TCP/Sack:
   Flow1: 29,148 segments, 384.579 KB/s
   Flow2: 44,889 segments, 735.471 KB/s
   Total: 74,037 segments, 1120 KB/s

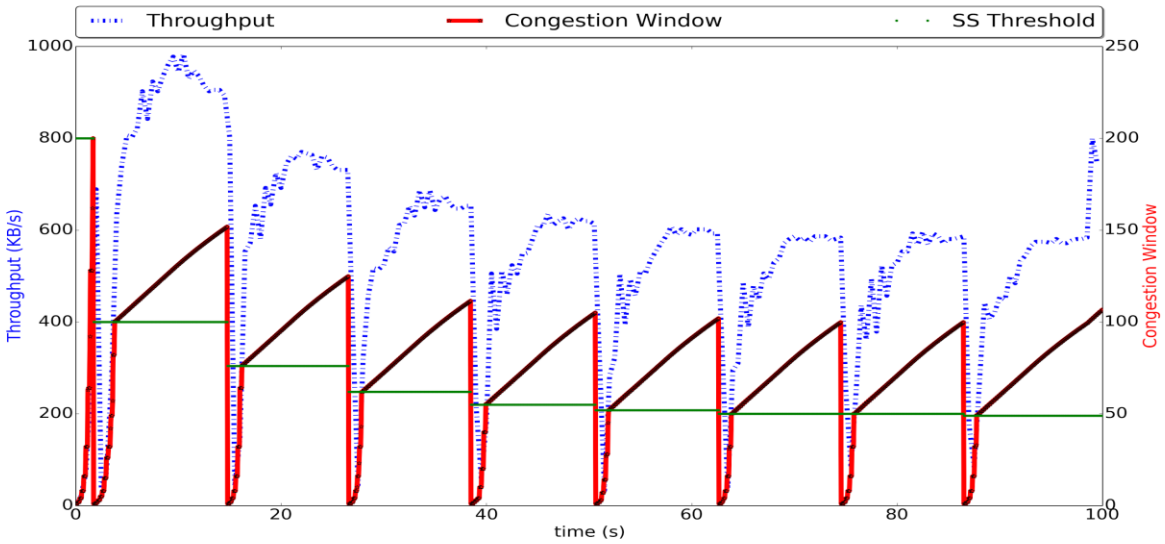
2) Throughput and Congestion Window:

TCP/Tahoe:

Flow 1 (with Slow Start Threshold)



Flow 2 (with Slow Start Threshold)



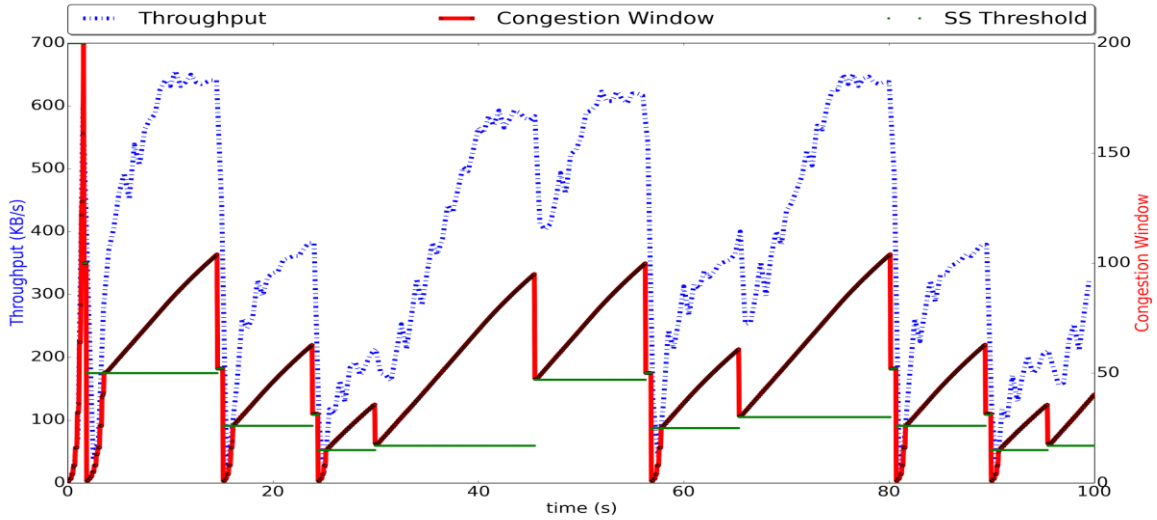
In the above two figures, we can see how TCP/Tahoe works. At the beginning, TCP/Tahoe will enter slow-start until a timeout occurs or a duplicate ACK. When timeout or duplicate ACK occurs, it will reset the congestion window to 1 and then enter slow-start. However, this time it will exit slow-start once the congestion window hits half of the previous size.

An interesting to note is that for flow 1, it enters congestion avoidance after the first duplicate acks. This is because flow 1 experience a burst of duplicate acks, therefore ssthreshold was decreased to half and then immediately reset to 2 due to another duplicate

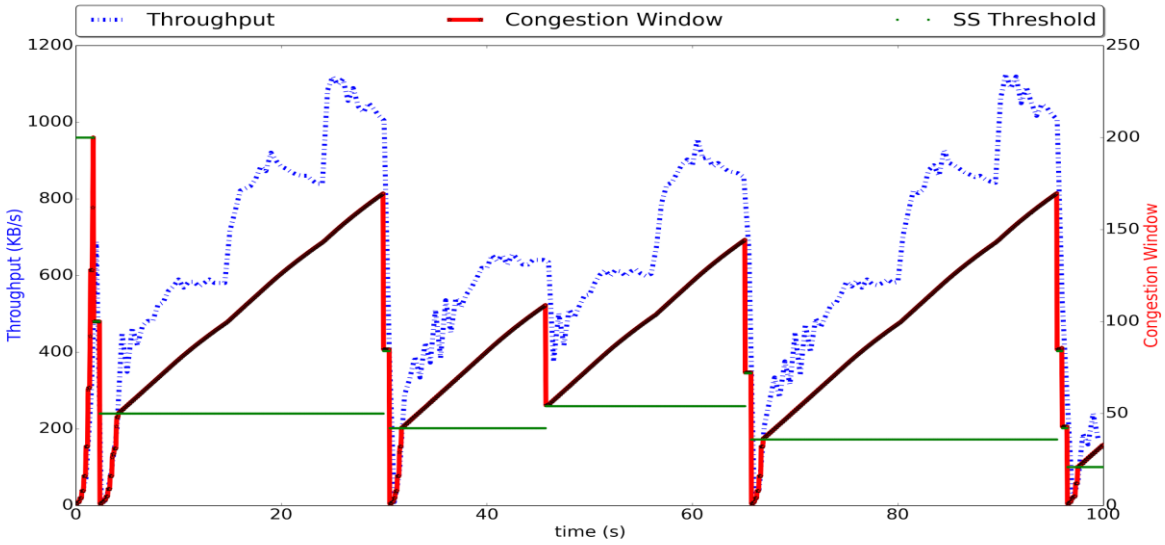
acks, as shown by the SSThreshold(green line). Whereas for flow 2, it did not experience such case (ssthreshold is dropped to 100 after the first duplicate acks).

TCP/Reno:

Flow 1 (with Slow Start Threshold)



Flow 2 (with Slow Start Threshold)



The above two figures shows the behavior of flow 1 and flow 2 using TCP/Reno. Both flow 1 and flow 2 shows the expected behavior under TCP/Reno. When received a duplicate ack, the congestion window and the ss-threshold is reduced to half. When a timeout occurs, the ss-threshold is also reduced to half but the congestion window is reset to 1. Comparing TCP/Reno and TCP/Tahoe, the dominant difference is that TCP/Reno halves the congestion window size instead of reset it to be 1, when the transmitter receives three duplicate ACK's. The duplicate

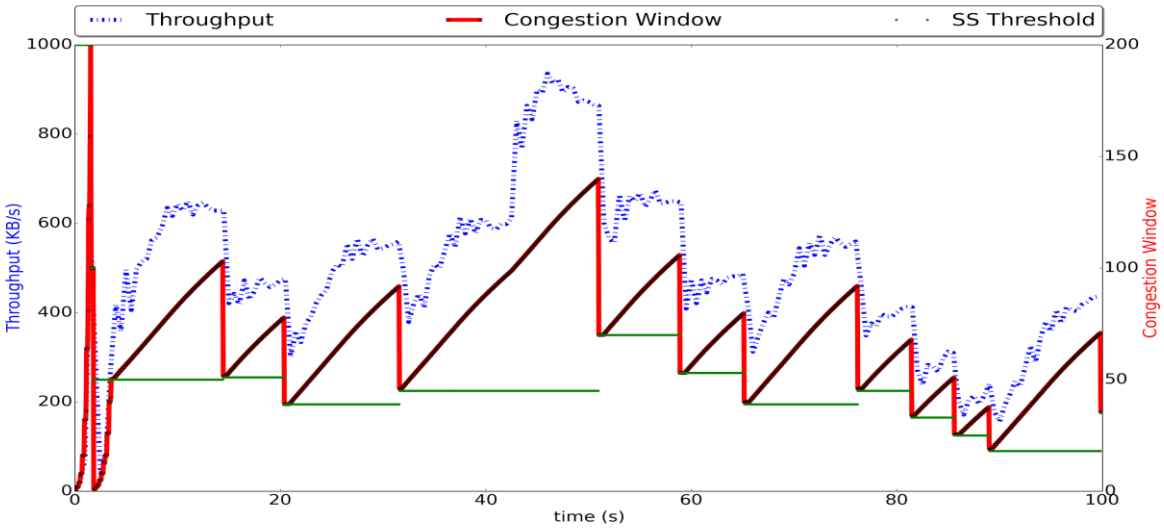
ACKs triggers the fast-retransmit to send back the packet, which is indicated to be lost in three duplicate ACK's.

Moreover, if we look close to the congestion window size curve around 30s' in flow 2, we can also see the congestion window size reduces twice. This is because three duplicate acks causing both the congestion window and the ss-threshold to reduce to half, and then, a packet timeout immediately occurred. This result in resetting the congestion window to 1 and also half the ss-threshold again.

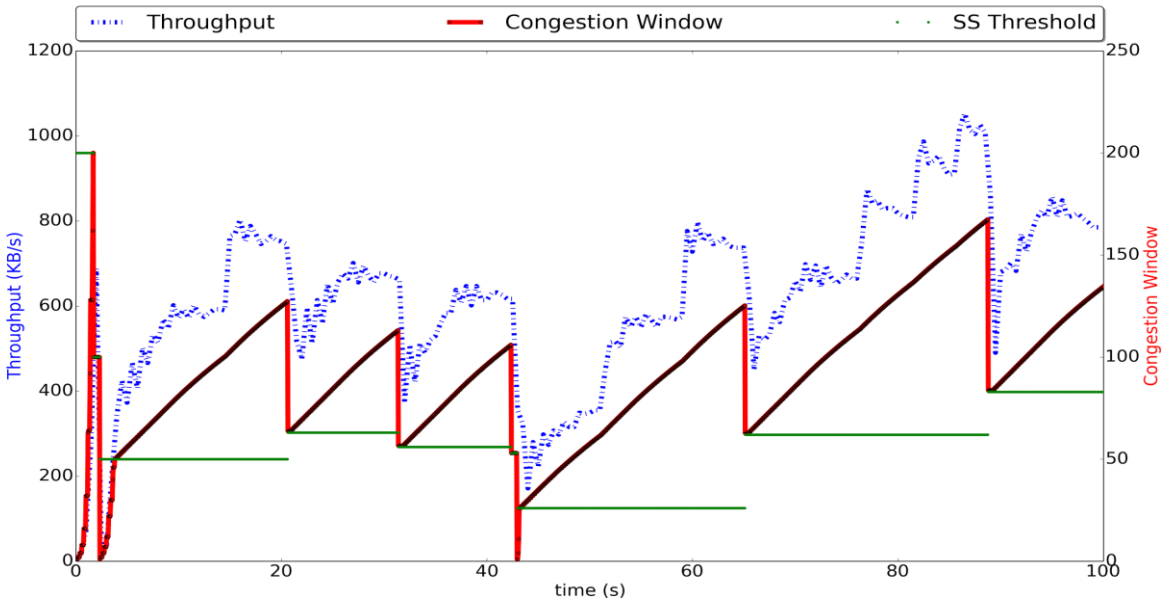
Comparing flow 1 with flow 2, flow 2 has better performance than flow 1 on throughput. According to the changes of congestion window size, we can see that flow 1 has more packet losses than flow 2, which contributes the lower throughput in flow 1. Due to the differences in link bandwidth and delay, the lower bandwidth and higher delay in flow 2 make the transmitter quickly overflow the queues and buffers of nodes. Therefore, flow 2 has less packet dropped in buffer of each node.

TCP/New-Reno:

Flow 1 (with Slow Start Threshold)



Flow 2 (with Slow Start Threshold)

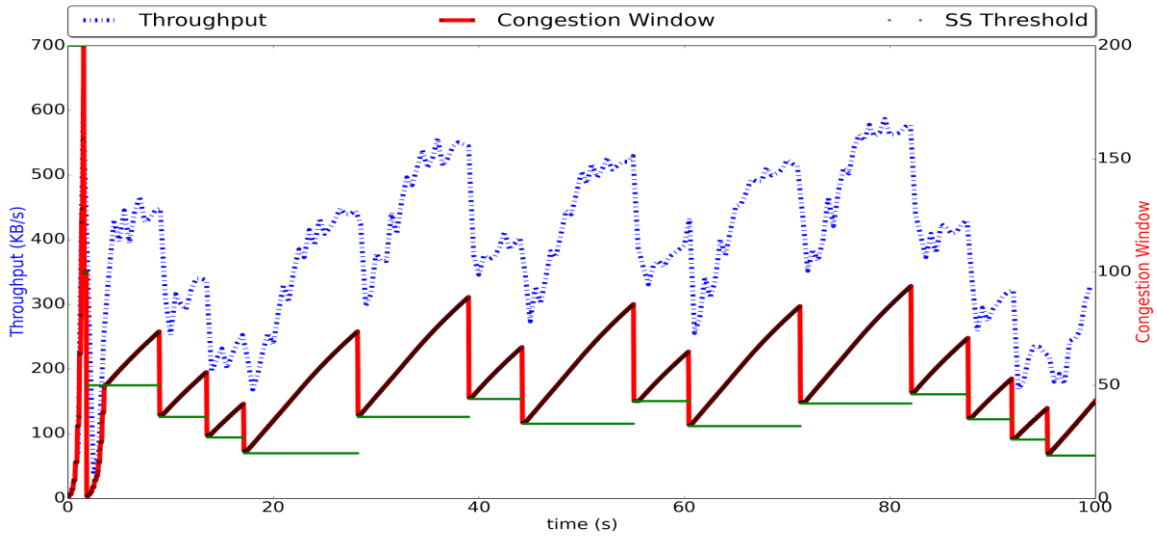


The TCP/Newreno is similar to TCP/Reno in slow start and fast-retransmit except for the mechanism of recovery (the TCP/Newreno has the smart recovery). In TCP/Newreno, the fast-recovery starts from the duplicate ACKs and ends when all of the packets are acknowledged. In other words, the sender keeps re-sending the packets which were non-ACKed. In the mean time, the congestion window remains additive increasing until the unACKed packets are all ACKed (then exiting the fast-recovery mode) or timeout. When it exits the fast-recovery, the congestion window will reduce to the slow start threshold (ssthresh).

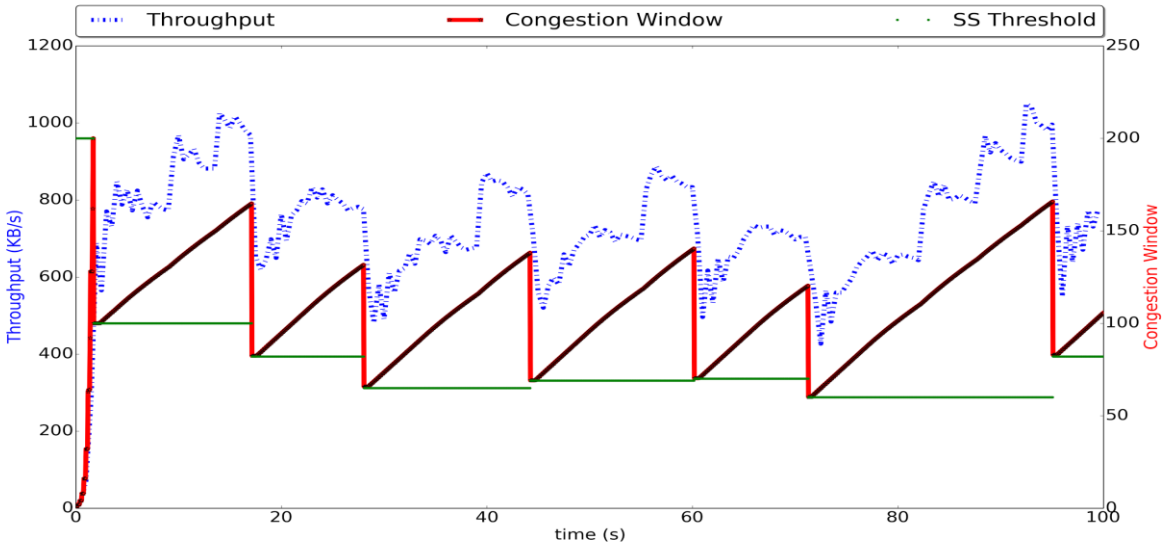
From Flow 2, we can observe that the curve has more something like glitches when the congestion window drops to 1 in TCP/Reno (because it exits fast-recovery mode once receiving any ACK). The strategy of retransmitted packets in TCP/Newreno is different from that in TCP/Reno in order to overcome the problem in multiple packet losses. Therefore, there will be less timeout in TCP/Newreno than in TCP/Reno. For example in Flow 1, there are six timeouts in TCP/Reno, but only one timeouts in TCP/Newreno. This means that TCP/Newreno can obtain some information from partial ACKs, which are the indications of another lost packets. Hence, the TCP/Newreno outperforms the traditional TCP/Reno especially at high loss condition. After timeout, the two strategies both execute slow start and then the AIMD.

TCP/Sack:

Flow 1 (with Slow Start Threshold)



Flow 2 (with Slow Start Threshold)

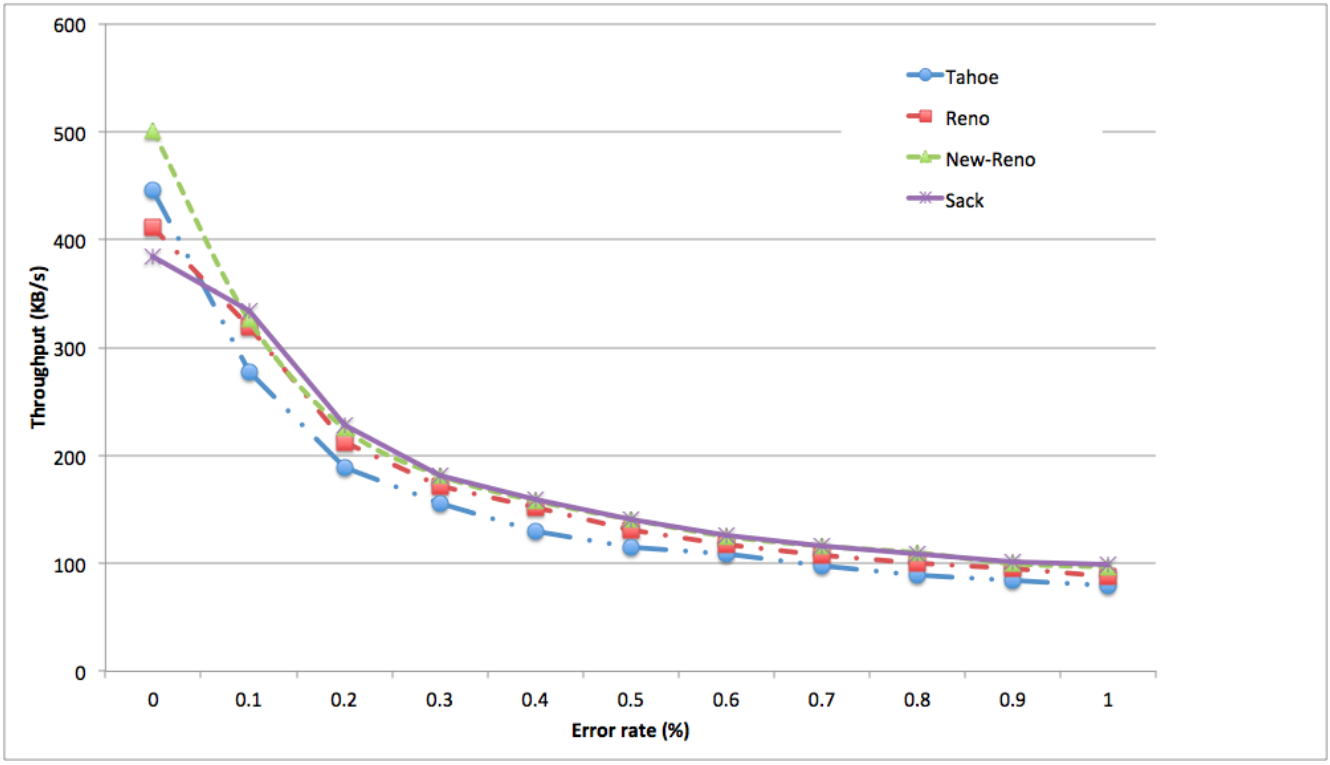


The above two figures shows the congestion window vs throughput plot for flow 1 and flow 2 using TCP/Sack. Comparing with other TCP flavors, we can see that TCP/Sack reduces the congestion window to half on almost every instance of the observed drop. This happens due to the ACK implementation in TCP/Sack. For TCP/Sack, the format of acknowledgement includes the original ACK and the selective ACK (EG. ACK{ack 4, sack 5-7} packets 5-7 received after 4, packet 5 missing). The SACK byte informs the Tx that the mentioned set of packets with the included seq no. has also been received after the associated ack seq. no. This enables the Tx to resend the missing packets only, without the need to resend all following packet. Due to this mechanism, the probability of timeout occurring is small due to the efficient window handling mechanism allowing for better overall throughput when compared to other flavors. In case of duplicate ACK and timeout, the

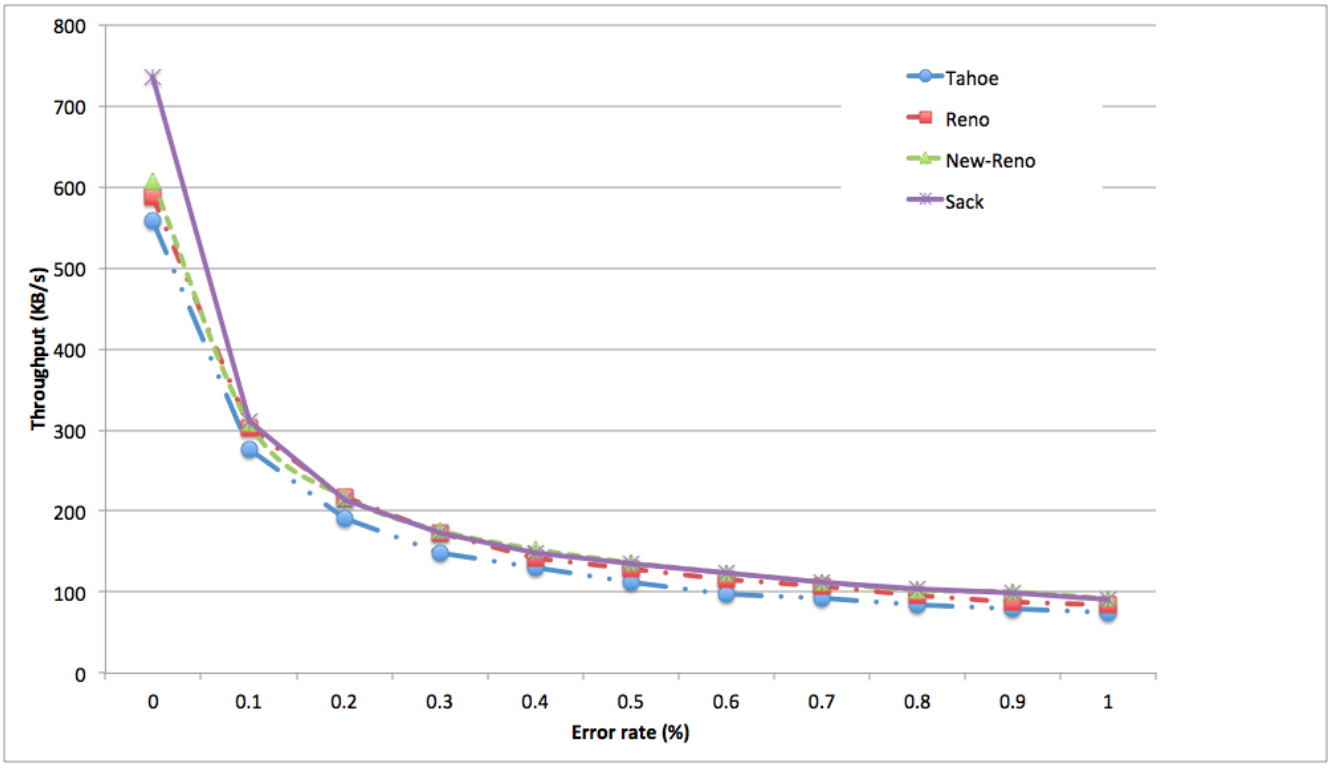
behaviour mechanism of SACK is similar to Reno/NewReno, i.e. on 3 duplicated ACK congestion window drops to half and on timeout window drops to 1.

3) Throughput vs Error rate

Flow 1:



Flow 2:



From the above two figures, we observe that for all the four TCP flavors, the overall throughput will decrease as the error rate increase, which is quite reasonable. The following are the discussions for all of the TCP flavors:

TCP/Tahoe has the lowest throughput among the four, since it will always reduce the congestion window to 1 in response to timeout and three duplicate ACKs. This aggressive congestion control will cause the congestion window to shrink very fast and frequent in high error rate, which result in the lowest throughput.

TCP/Reno has a slightly better throughput than TCP/Tahoe, since it implements fast recovery, which would only halve the congestion window in the observation of three duplicate ACKs.

TCP/NewReno, since it will not exit fast recovery until all the outstanding packets are ACKed, has a better performance than Reno in high error rate. SACK has the highest throughput among the four. It's because that SACK is able to deal with multiple losses using selective ACK, which could effectively avoid the retransmission of packets that are already received and guarantee its performance under high error rates.

For flow1, since it flows through links with higher bandwidth and a lower delay, the transmitter can quickly overflow the buffers at each node, which would render more packets

to be dropped for flow1. This result in a lower throughput for all the four flavors under low error rate. With higher error rate, however, the overall throughputs for any given flavor in flow1 is close to that in flow2. This is because under high error rate, it is the error rate that will primarily determine the throughput but not the link's bandwidth or the delay.

Part II: Experimental component

1. IP addresses of the local and remote end-hosts.

a) local host: 192.168.1.187
b) remote end host: 74.125.212.177

2. Port numbers of the local and remote end-hosts.

a) local host: 4070
b) remote end host: 80

3. Maximum segment size used in either direction.

a->b 1260 bytes
b->a 1260 bytes

4. Total bytes and unique bytes sent by the remote end-host.

Total bytes
b->a 38785361 bytes
Unique bytes
b->a 38719841 bytes

5. Number of unique data packets sent by the remote end-host.

unique data packets : (ttl_pkts-rexmt_pkts)
=> 30785-52= 30733 packets

6. Average downlink throughput. (The value reported by tcptrace may not be correct. Calculate the value by yourself.)

value in trace :- 345248 Bps (unique bytes/time)(38719841 / 112.151)
calculated value :- 345832 Bps (ttl bytes/time)(38785361 / 112.121)

TcpTrace calculates the throughput value based on the unique packets received, here we have calculated the throughput based on the total packets received.

7. Average RTT in either direction.

a->b 30.8 ms
b->a 40.9 ms

8. Minimum end-to-end RTT. (Study how tcptrace measures RTT.)

a->b 30.6ms
b->a 49.9ms

RTT is calculated by tcptrace by studying the timestamp associated with each transmitted and received packets for all transmissions falling under the following 2 categories

1) ACK value should be 1 greater than the last packets sequence number,
2) Does not consider transmissions where data is retransmitted,
   This is done to invalidate retransmission data set and reduce the possibility of retransmission ambiguity problem to be considered in the calculation RTT value

9. TCP flavors used by both end-hosts. How do you identify the flavors?

Based on the generated congestion and timeline plots of the tcp_trace.pcap, we assume that the TCP flavour being used is TCP NewReno with SAck enabled, based on the following two observations:

1) We observe that for every time out the congestion window drops to 1 .
2) In the event of 3 duplicate ACK received, the congestion window is reduced to half its current size (as observed from the congestion window plot) .
3) On observation of the Time-Line trace, the packet flow behaviour observed suggests that Sack is enable for this connection.

10. Maximum congestion control window size used by the remote end-host.

Since congestion window at remote host cannot be directly determined, it is estimated by using outstanding unACKed data

max Cwnd observed: 65521 bytes