Report on Programming Assignment 2: Understanding TCP

Gaurang Naik (123079009)
Department of Electrical Engineering,
Indian Institute of Technology Bombay,
Powai, Mumbai, INDIA 400076.

September 5, 2014

1 ns3 Simulation

1.1

Figures 1a-1e are the required five graphs. Figure 1a shows the sequence number of the packets transmitted by node 0 as well as the acknowledgement numbers received by node 0 as a function of time. The sequence number increases more or less linearly and so does the acknowledgement number. However, from about 1.1 seconds to 1.2 seconds, the receiver (node 1) sends multiple dupacks. It was observed from Wireshark that during this interval, node 1 sent as many as 21 dupacks. As a result, we observe an almost flat line in this small interval.
Figure 1b shows the average throughput received by node 0 in a one second interval. The average throughput for packets receieved by node 0 is about **4.2 Mbps** which is less than 5 Mbps. This is because of the packet drops due to queue filling up as well as packet errors introduced in the channel as a result of the error rate. This can be seen well in figure 1c and 1d.
As seen in figure 1c, the cwnd at node 0 initially shoots up to a very large value. On closely observing the cwnd values from the tcp-example.cwnd file and the pcap file in Wireshark, it is observed that when the cwnd becomes 17688 bytes, a third dupack arrives and the sender enters the Fast Recovery state. Interestingly, the cwnd immediately becomes 10452 bytes but does not become (cwnd/2). In the Fast Recovery state, TCP NewReno sends a segment for every dupack received from the receiver. During this period, the cwnd continues to increase exponentially since it sends one segment in every RTT. An interesting thing observed is that for every ACK the sender receives, it reduces the cwnd by a very small amount, but continues to remain in Fast Recovery state. Finally at roughly t=1.26 seconds, dupacks stop arriving and an ACK for 95945 arrives and the sender exits fast recovery state and enters Congestion Avoidance state.
Within the simulation time of 10 seconds, the cwnd drops **25 times**. Most of these cwnd drops are due to the queue building up and reaching its limit (10 packets). This can easily be seen by comparing figures 1c and figure 1d. At every instant (except first) when the queue becomes full, it drops a packet and cwnd is halved. The other cwnd drops (at t=7.5s, 8.1s and 9.1s), are because of packet errors introduced because of the error rate.

1.2

The throughput achieved by keeping all other parameters the same and changing the bandwidth from 5 Mbps to 50 Mbps is **13.08 Mbps**. This is significantly lower than 50 Mbps and unlike the 5 Mbps case, where the throughput was less than the bandwidth, but the difference was small.
One posssible reason for the low throughput is that, owing to the high bandwidth, the Bandwidth Delay Product (BDP) is large. As a result the ideal cwnd (which is equal to BDP) is very large. The congestion avoidance

algorithm increases the cwnd to a very high value. When a packet loss or timeout occurs, the cwnd reduces to (cwnd/2), which is also a large value. At this point, the number of unacknowledged packets is cwnd, but the cwnd has reduced to (cwnd/2). So the sender has to wait for (cwnd/2) ACKs to come back. Since, this is large, the amount of time the sender is idle is significant. This leads to poor utilization of the link and the throughput reduces. This reasoning is strengthened by the TCP time sequence graph. Figure 2 shows the zoomed TCP time sequence graph of the sender. The red dots indicate the sequence number. It is seen that on several occasions, the sender waits for a significant amount of time before it sends another segment and consequently the sequence number increases slowly. In order to increase the throughput, the waiting time for the sender at the end of fast recovery state must be reduced. The sender has to wait for (cwnd/2) ACKs to come back. This time can be reduced by reducing cwnd. Since the ideal cwnd is equal to BDP, and Bandwidth is constant, the cwnd can be reduced by reducing the RTT (or delay). This is confirmed by simulating by changing the delay from 5msec to 1msec. The average throughput increases from **13.08 Mbps** to **37.93 Mbps** By further reducing the delay to 0.1msec, throughput increases to **44.2 Mbps**. Thus, clearly the TCP throughput is inversely proportional to the delay (or RTT) as indicated by

$$TCPThroughput = \sqrt{\frac{3}{2}}\frac{MSS}{RTT\sqrt{p}}$$

Further, by increasing the delay to 10msec, the throughput decreases to 6.81 Mbps. Other parameters such as the error rate and queue size were also changed and throughput was observed, while the throughput changed, the change was much smaller as compared to the change observed in case of changing the delay. Thus, **in order to improve the throughput, the delay between the sender and receiver must be reduced.**

1.3

As described in the answer to part 1.2, the throughput has a direct relation with the delay. This relation is because of the BDP. For a fixed bandwidth (5 Mbps in this case), higher the delay (or RTT), higher is the BDP. As a result, the ideal cwnd (equal to BDP), will try to stay close to the BDP. Every time a packet is dropped/lost or timeout occurs, the fast recovery state sets in and when it exits, it waits for (cwnd/2) ACKs to come back. Larger the cwnd, larger is this waiting time and more inefficient the utilization of the link. As a result, TCP throughput is inversely proportional to the RTT for a fixed Bandwidth. The simulated results for delay values of 1msec, 5msec, 10msec, 50msec and 100msec are summarized in table 1.

Table 1: Analysis of results of simulation for different values of delay

Delay	**Throughput**
1 msec	4.418 Mbps
5 msec	4.200 Mbps
10 msec	3.662 Mbps
50 msec	0.597 Mbps
100 msec	0.183 Mbps

Figure 3 shows the relationship between the TCP throughput and delay graphically.
From the cwnd plot versus time and the queue occupancy graph, it can be inferred that most of the cwnd at lower delay values (1msec and 5msec) drops occur due to the queue reaching its limit. In case of higher delay values (50msec and 100msec), the queue is rarely full. As a result, the cwnd drops occur mainly due to packet losses manifested in the form of dupacks or timeouts. Moreover, at smaller delay values, the cwnd changes very often, while for large values of delay the cwnd changes less often. This is shown in figures 4a and 4b

1.4

Tabel 2 and figure 5 show the effect of error rate on the throughput. It is clear that the throughput increased with reduction in the error rate. At an error rate of 0.001 (i.e. when every 1000th bit is in error), every second MSS is received in error. As a result, the throughput reduces drastically and is as low as 0.014 Mbps. The Cwnd graph shows that in the slow start itself, there is a packet loss. This is because when node 0 send one segment, it receives the ACK. As a result, node 0 increases its cwnd to 2 MSS, but now the second packet is received in error. The ACKs can be assumed to be received error free since an ACK has only 40 bytes. Node 0 reduces its cwnd again to one MSS. This process keeps repeating as seen from the cwnd graph. Between t=3 and t=8 seconds, node 0 does not send any segments. This is possibly because
As the error rate reduces to 0.0001 every 10000th bit is received in error and the throughput is still low (0.301 Mbps). As the error rate further reduces, the throughput increases, but beyond an error rate of 10^{-6}, the improvement in throughput reduces and the througput saturates. At an error rate of 10^{-7} and 10^{-8}, the throughput is same. Thus, **the throughput increases with decreases in error rate, but saturates beyond a paerticular error rate.**

Table 2: Analysis of results of simulation for different values of error rate

Error Rate	Throughput
0.001	0.014 Mbps
0.0001	0.301 Mbps
0.00001	3.376 Mbps
0.000001	4.200 Mbps
0.0000001	4.288 Mbps
0.00000001	4.288 Mbps

1.5

Table 3 and figure 6 show the effect of queue size on the throughput. When the queue size is 1, the average throughput is extremely low. As the queue size increases, the average throughput increases but saturates beyond a particular queue size. The queue occupany graphs in case of queue size 500 and 1000 reveal that the queue occupancy never increases beyond 110. As a result, further increasing the queue size has no impact on the average throughput.

Table 3: Analysis of results of simulation for different values of queue size

Queue Size	Throughput	Average Queuing Delay	Average Cwnd	Average Queue Occupancy
1	0.173 Mbps	0.00042	1356	0.40
2	3.298 Mbps	0.0006	6141	0.58
5	3.848 Mbps	0.00145	7672	1.60
10	4.200 Mbps	0.0039	9884	4.34
100	4.342 Mbps	0.0355	29374	38.57
500	4.381 Mbps	0.0538	39339	58.15
1000	4.381 Mbps	0.0538	39339	58.15

1.6

Figure 7 shows the variation in Cwnd as a function of time for the three variants, TCP Tahoe, TCP Reno and TCP New Reno. For the standard configuration, retransmissions due to timeouts were not observed in either of these cases. All retransmissions are dupack driven. Every time a timeout occurs, in case of TCP Tahoe, the Cwnd value reduces back to 1 MSS i.e., every time a timeout occurs, TCP Tahoe enters slow start phase.
In case of TCP Reno and NewReno however, when multiple dupacks arrive, the sender enters into fast recovery phase. This can be seen by observing the Cwnd values in case of TCP Reno and TCP NewReno. During simulatio, in each of the cases, the first tim the thrird dupack arrives when the Cwnd value is 17688 bytes. The response of TCP NewReno and TCP Reno differs at this stage.
TCP NewReno keeps increasing the Cwnd in Fast Recovery phase until all ACKs in the window have arrived, while in case of TCP Reno, the moment the first ACK arrives, the server exits fast recovery phase.

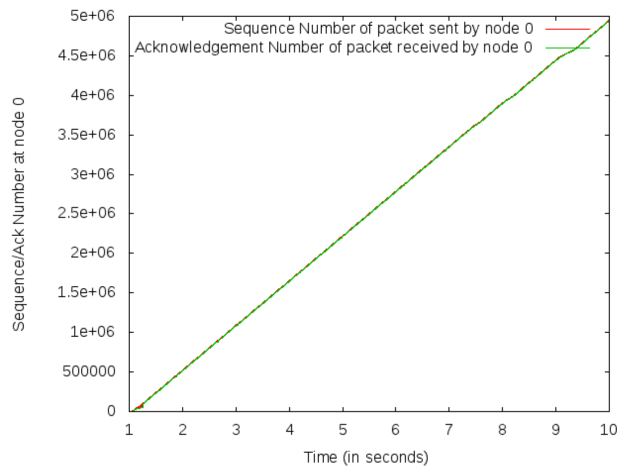
2 Analyzing real-world TCP downloads

2.1

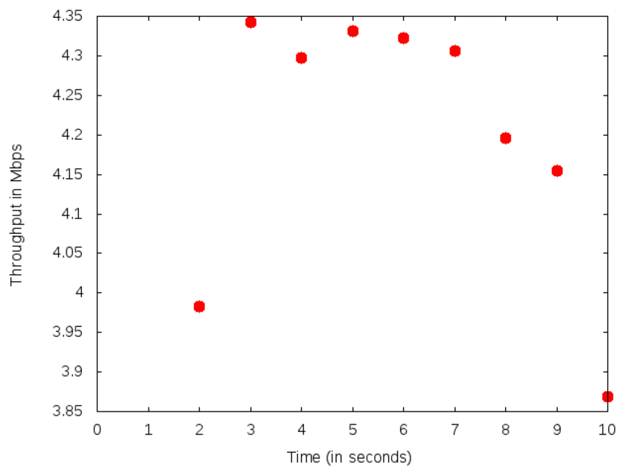
The pcap file was captured on the Ethernet interface eth1 using Wireshark. Once Wireshark started capturing packets, a download of the Adobe Acrobat Reader installer file was started from www.filehippo.com. The size of this installer file was 74 MB. The download was started from an external Internet connection, thus, bypassing the IIT Bombay proxy server. One reason for downloading the file from this particular website was that being an external server, I expected there to be some retransmissions (as suggested on the Project description webpage). The TCP retransmissions were observed using the *tcp.analysis.retransmission* filter in Wireshark. For the Adobe installer file download trace file, the capture duration was 136 seconds. Download of the installer file lasted for 107.8 seconds.
The average TCP throughput during these 107.8 seconds was **6.142 Mbps** and the per second throughput is shown in figure 8. The throughput graph was generated using Wireshark and shows throughput in every 1 second interval for the trace file.
Figure 8 suggests that the actual file transfer must have started at about 15 seconds and lasted until 125 seconds. The trace file when viewed in Wireshark showed that the sender (192.168.1.25) sent the first SYN packet at 16.79 seconds and the receiver (202.159.216.171) sent the FIN packet at 124.58 seconds.
Figure 9 shows the relation between the dupacks sent from 192.168.1.25 and TCP retransmissions sent by 202.159.216.171. There are two source of retransmissions — dupacks and timeouts. Based on the plot, I could infer that all retransmissions were triggered by dupacks. I looked closely at some of the retransmissions, and saw that each retransmission (in cases that I observed) was triggered by a duplicate ACK. Further retransmissons sent all packets from the packet for which the dupack arrived to the packet upto which SACK was sent in each of these dupacks.
The Round Trip Time (RTT) of the TCP flow as a function of time as extracted from Wireshark is shown in figure 10. By zooming in, the **minimum RTT is of the order of 10 microseconds, while the maximum RTT is around 1.16 seconds.** However, the average RTT is lower than the maximum RTT. Since the average throughput for the duration of download is 6.142 Mbps, the transmission delay is small. And since the minimum delay is of the order of microseconds, the propagation delay is also small. Hence, *the major contributor to the average RTT is the propagation and transmission delay.* The time instants at which the RTT shoots up to a very high value are those when there are multiple dupacks and several retransmissions. For instance, from figure 10 it can be seen that the maximum RTT of 1.16 seconds is seen when the sequence number is around 48000000. This occurs at about time 74 seconds when there are larger than average number of retransmissions and dupacks as seen from figure 9. Thus, *the fluctuations in RTT occur due to retransmission which are triggered by dupacks.*

2.2

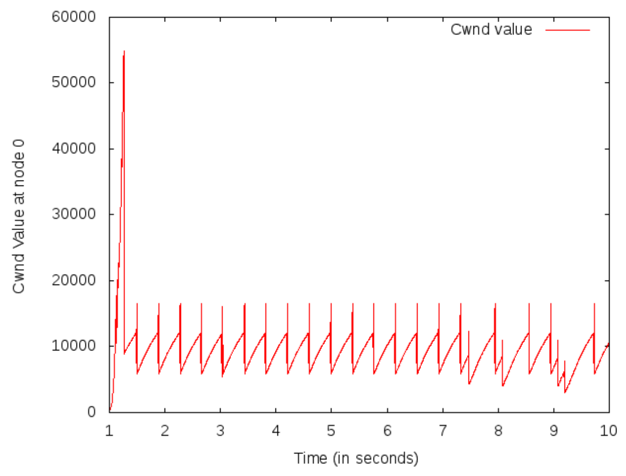
Figure 11 shows the zoomed Time Sequence graph during a TCP retransmission. Similar pattern is observed during all other retransmissions. As seen in the figure, the moment a dupack arrives (shown by a flat line), it contains entries for SACK indicating that those packets are received correctly by the receiver. If at this stage, the server would have reduced the Cwnd to 1 MSS, the sequence number would not have continued increasing at the same rate as earlier. The server enters Fast Recovery state and continues sending packets by increasing the Cwnd value. Thus, the server follows a TCP Reno like TCP variant.
For the recovery mechanism, the sender does perform SACK based recovery. To study this SACK based recovery, a part of capture was studied where there are lot of dupacks received and TCP retransmissions. For instance, at time 96 seconds, the client (192.168.1.25) sends a dupack with number 68876665. In this dupack, it sets the SACK options field with values 68882257 to 68885053. In successive dupacks the right edge of the SACK field keeps moving further and the 8th dupack has the SACK option value 68882257 to 68894839. Thus, the sender now knows that these segment number worth of packets have been received successfully at the client. Each TCP segment has a size of 1398 bytes. After the 8th dupack, the server enters fast retransmission state and sends the segment with sequence number 6887666. Now, in between 68876665 and 68882257, there are 4 segments of 1398 bytes each. Therefore, the sender sends these segments (68876665, 68878063, 68879461, and 68880859) in quick succession, thus performing SACK based recovery.



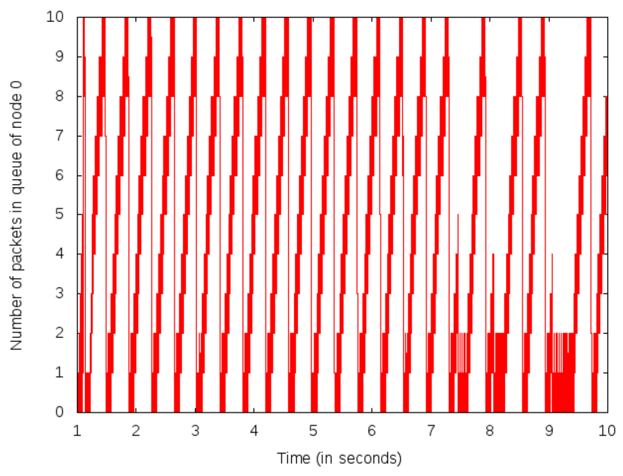
(a) TCP time sequence graph



(b) TCP throughput

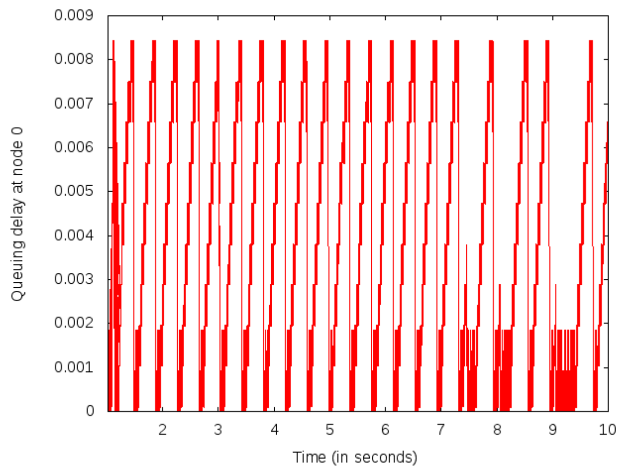


(c) cwnd v/s time



(d) Queue occupancy at node 0

i



(e) Delay of packets in queue at node 0

Figure 1: All five graphs

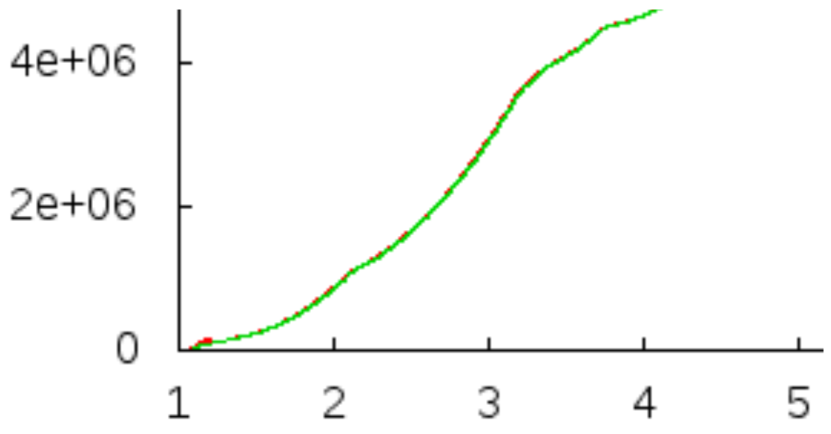


Figure 2: Delay of packets in queue at node 0

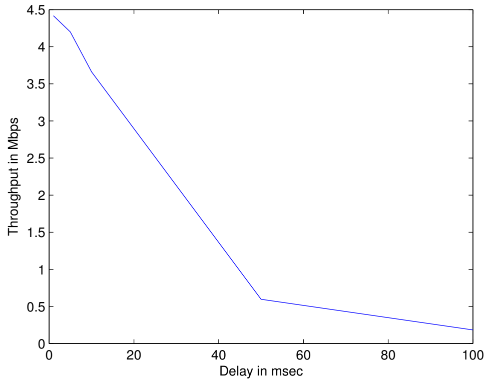
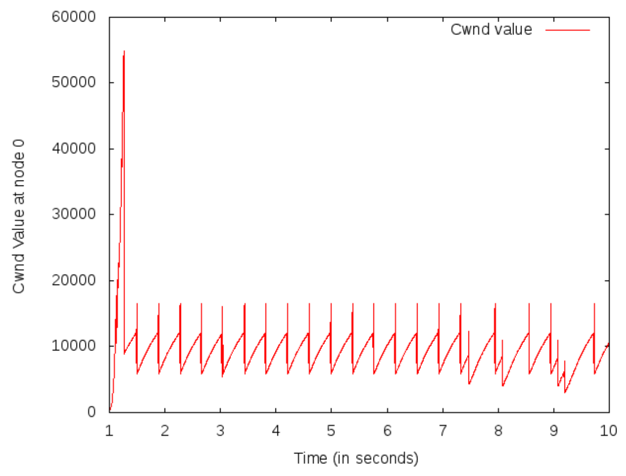
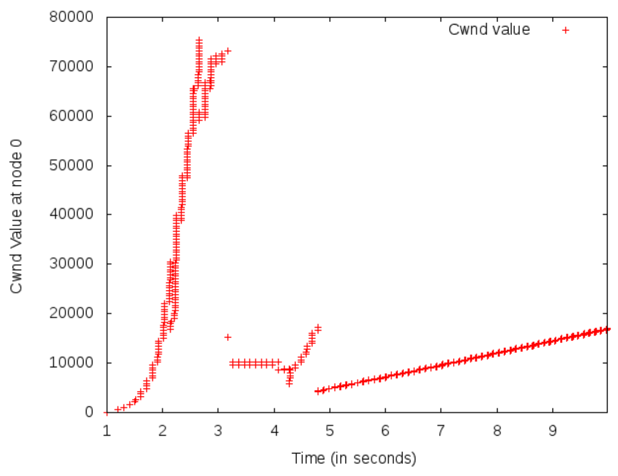


Figure 3: TCP throughput as a function of delay



(a) 10msec



(b) 50msec

Figure 4: Cwnd variation graph for small and large value of delay

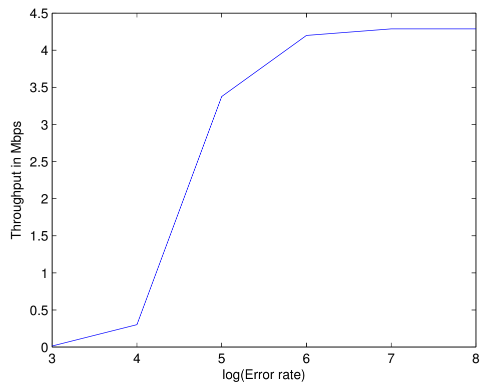


Figure 5: TCP throughput as a function of error rate

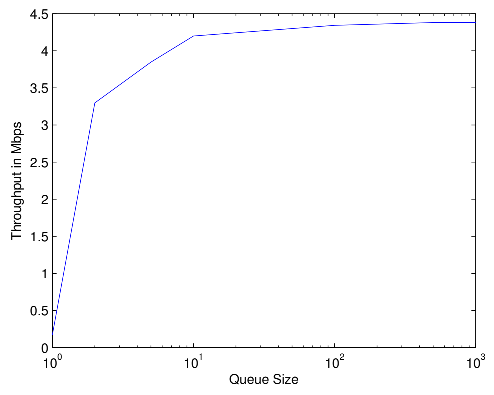
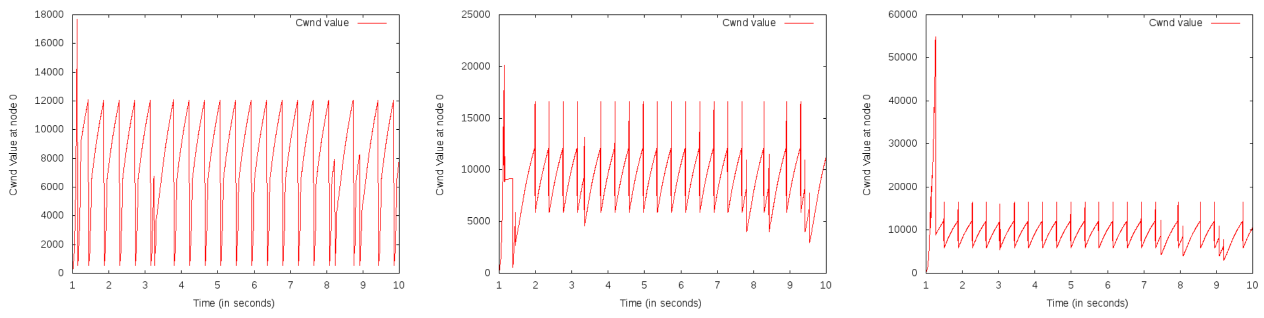
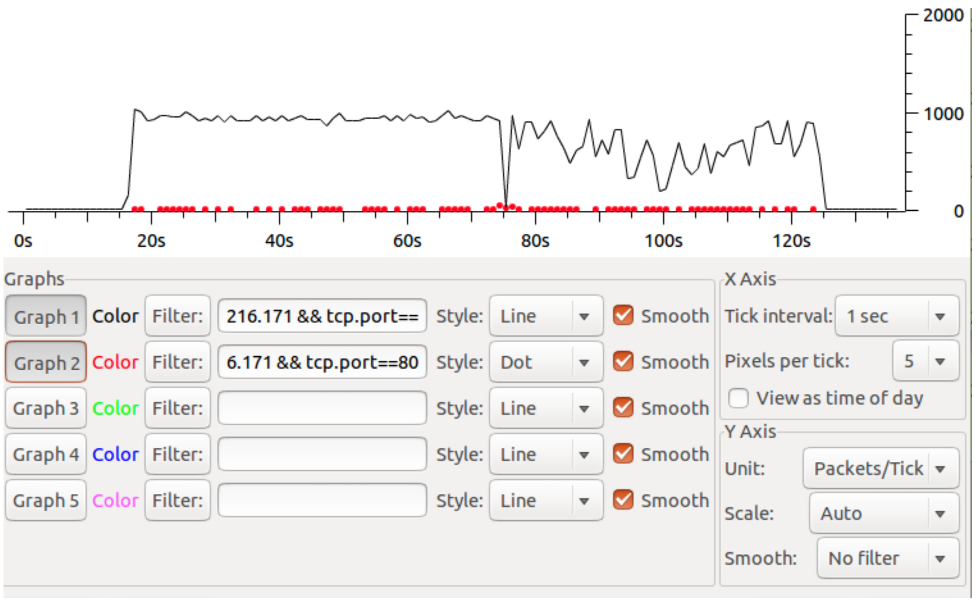


Figure 6: TCP throughput as a function of queue size



(a) Tahoe (b) Reno (c) NewReno

Figure 7: Cwnd variation for different TCP variants



Graphs

Graph 1 Color Filter: 216.171 && tcp.port== Style: Line Smooth ✓
Graph 2 Color Filter: 6.171 && tcp.port==80 Style: Dot Smooth ✓
Graph 3 Color Filter: Style: Line Smooth ✓
Graph 4 Color Filter: Style: Line Smooth ✓
Graph 5 Color Filter: Style: Line Smooth ✓

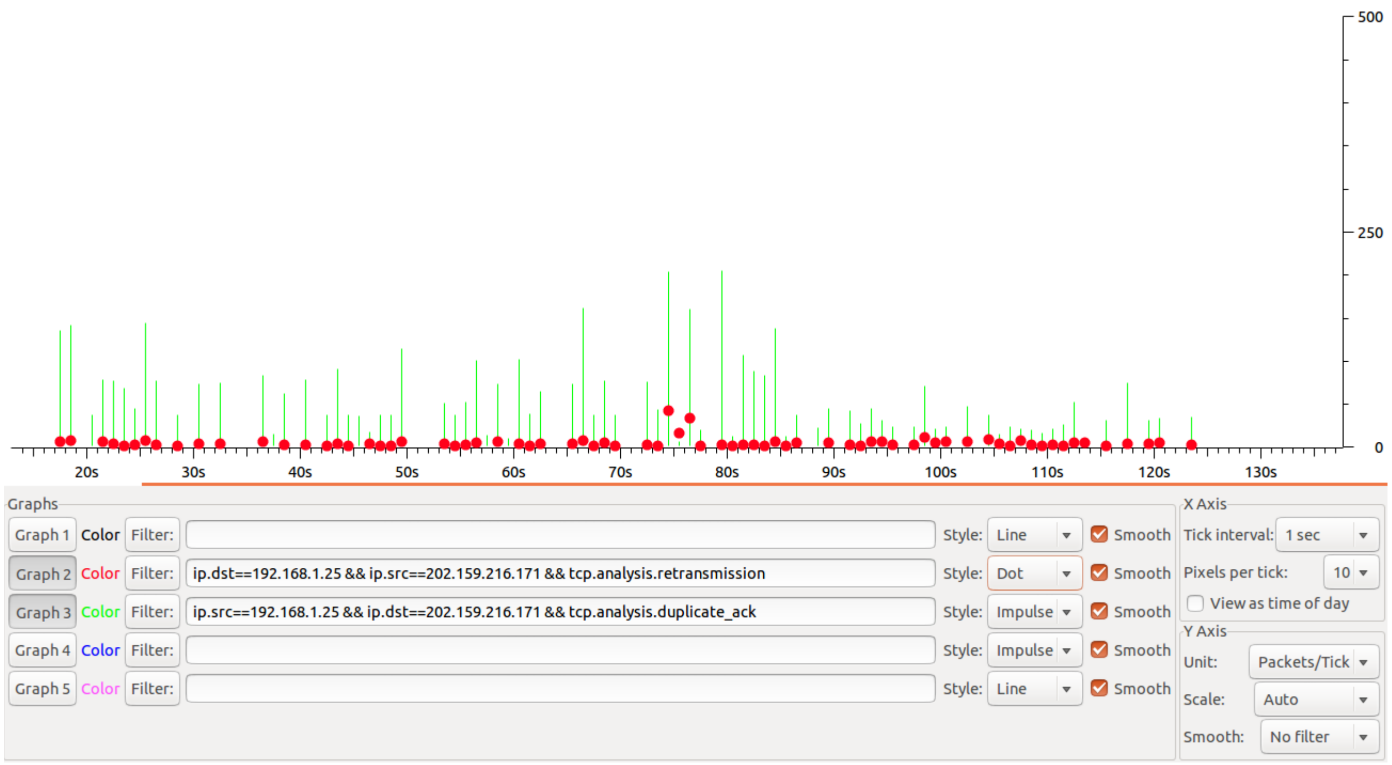
X Axis

Tick interval: 1 sec
Pixels per tick: 5
☐ View as time of day

Y Axis

Unit: Packets/Tick
Scale: Auto
Smooth: No filter

Figure 8: TCP throughput on a per second basis



Graphs

Graph 1 Color Filter: Style: Line Smooth ✓
Graph 2 Color Filter: ip.dst==192.168.1.25 && ip.src==202.159.216.171 && tcp.analysis.retransmission Style: Dot Smooth ✓
Graph 3 Color Filter: ip.src==192.168.1.25 && ip.dst==202.159.216.171 && tcp.analysis.duplicate_ack Style: Impulse Smooth ✓
Graph 4 Color Filter: Style: Impulse Smooth ✓
Graph 5 Color Filter: Style: Line Smooth ✓

X Axis

Tick interval: 1 sec
Pixels per tick: 10
☐ View as time of day

Y Axis

Unit: Packets/Tick
Scale: Auto
Smooth: No filter

Figure 9: Relation between dupacks and TCP retransmissions

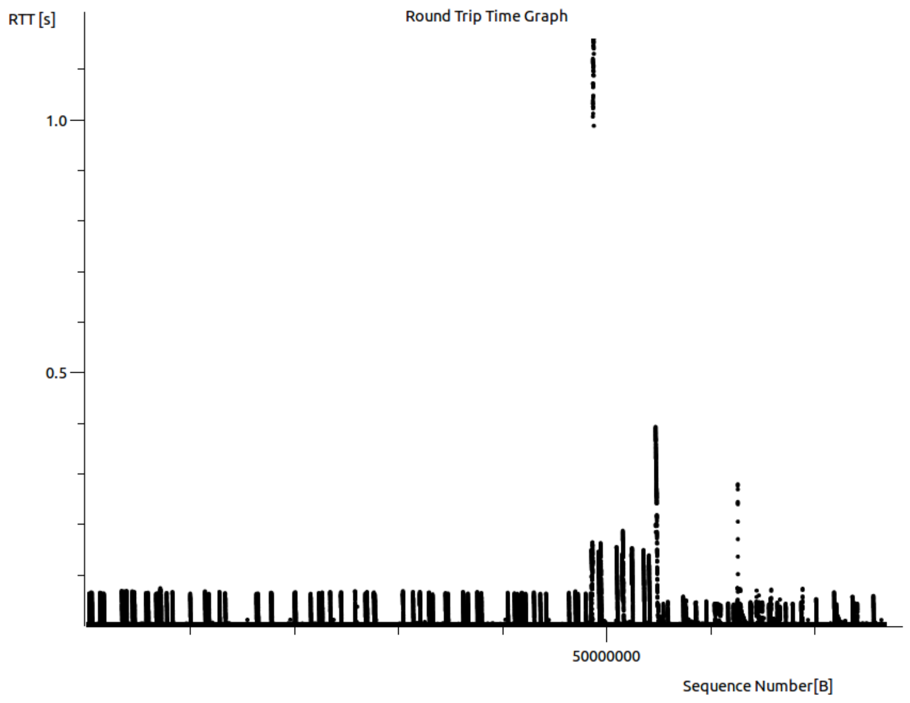


Figure 10: RTT as a function of time

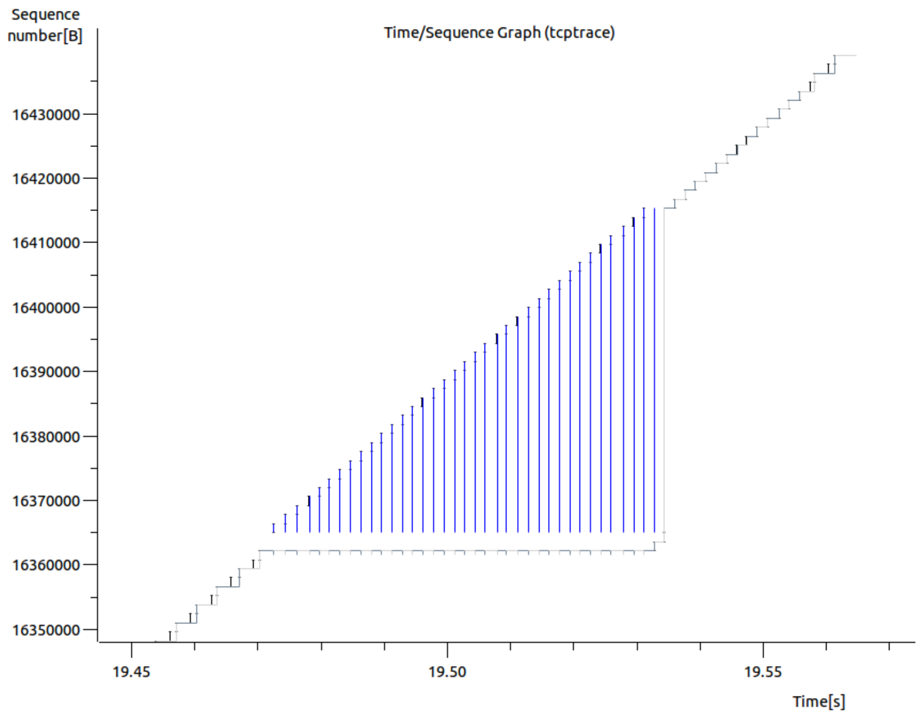


Figure 11: Zoomed Time Sequence graph during TCP retransmissions