Programming Assignment 2 : Group 3

Members : Judah, Ramon, Siddhesh, Yaksh

Our Results :

1. Simulation Component

(a). We have choose to implement SACK and using an initial CW = 4 for this assignment.

Error percentage = 2.5%

Wireless Link = 8 Mbps

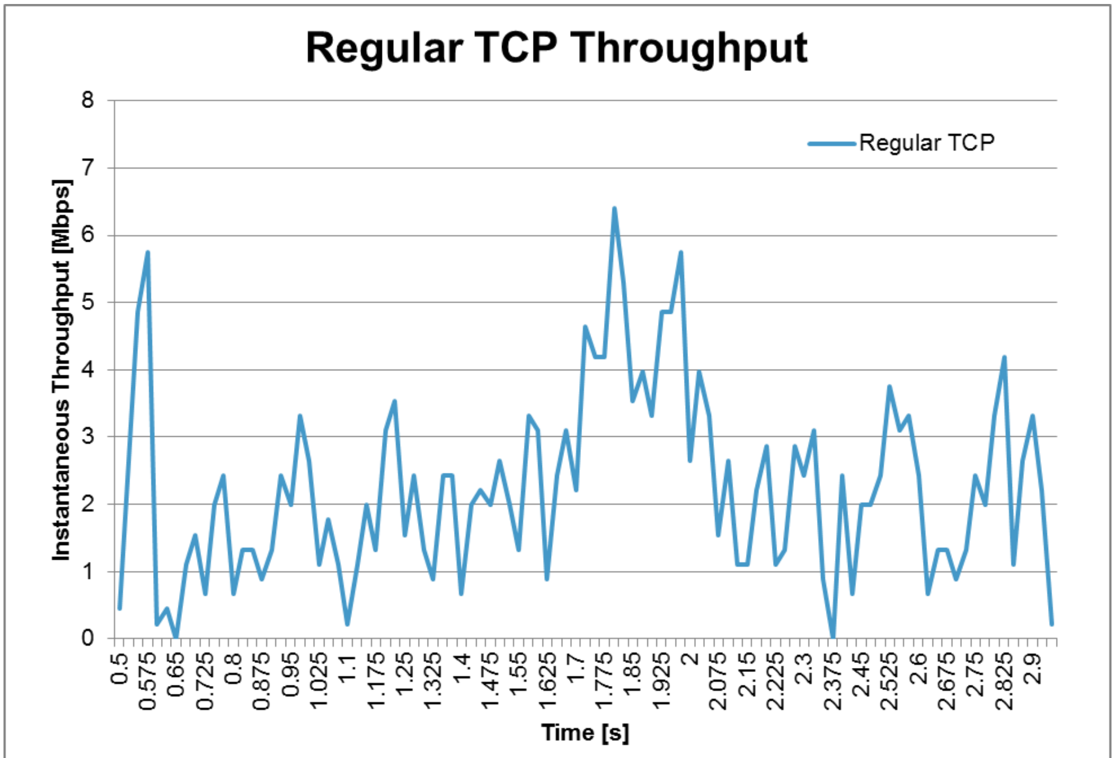
The reasons have been explained in 1.(c) and 1.(d).

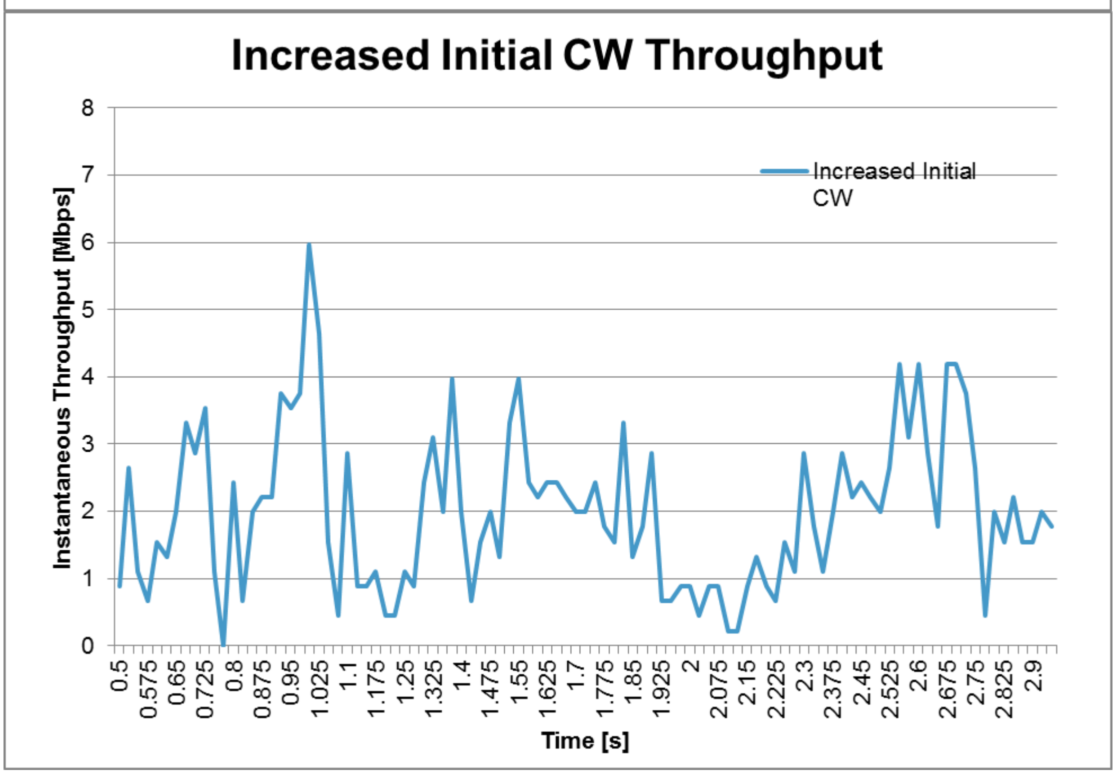
End to end throughput with and without the TCP option is as follows (for a simulation period of 100 seconds and actually transmission time of 99 seconds):

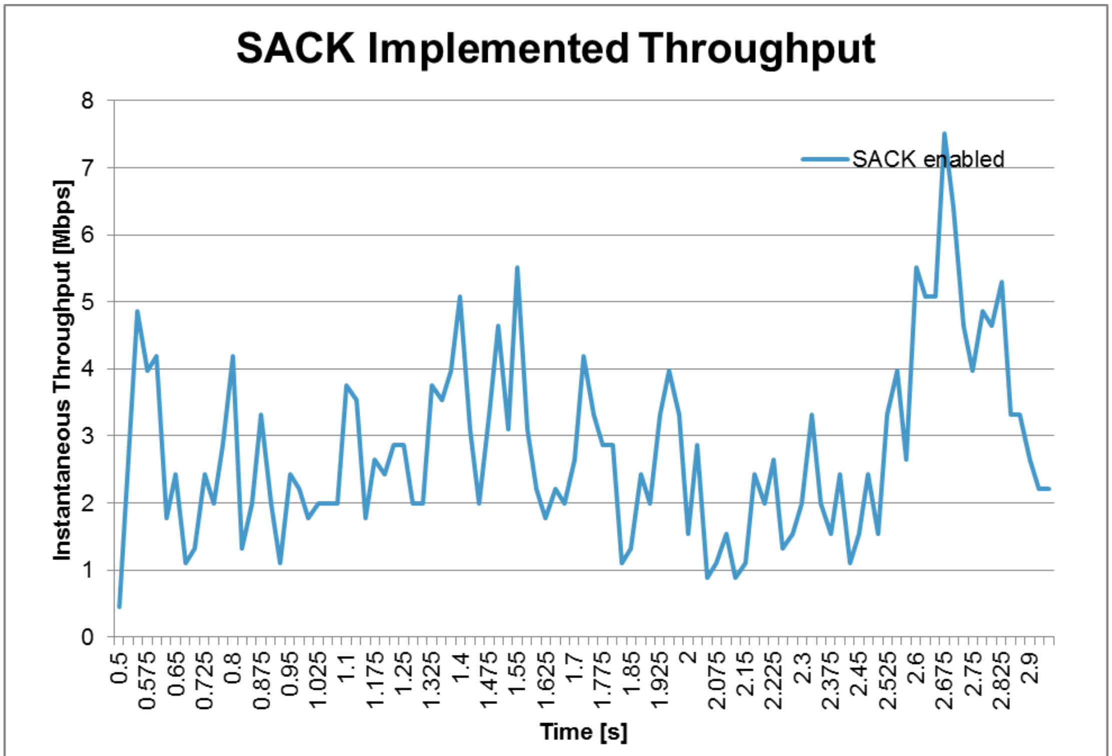
Case :		**Throughput (Mbps)**
With SACK disabled and CW = 1	(No TCP-option implemented)	1.2390
With SACK disabled and CW = 4		1.4070
With SACK enabled and CW = 1		1.7803
With SACK enabled and CW = 4	(Both TCP-option implemented)	1.9202

(b). Instantaneous Throughput (averaged over 25 ms)

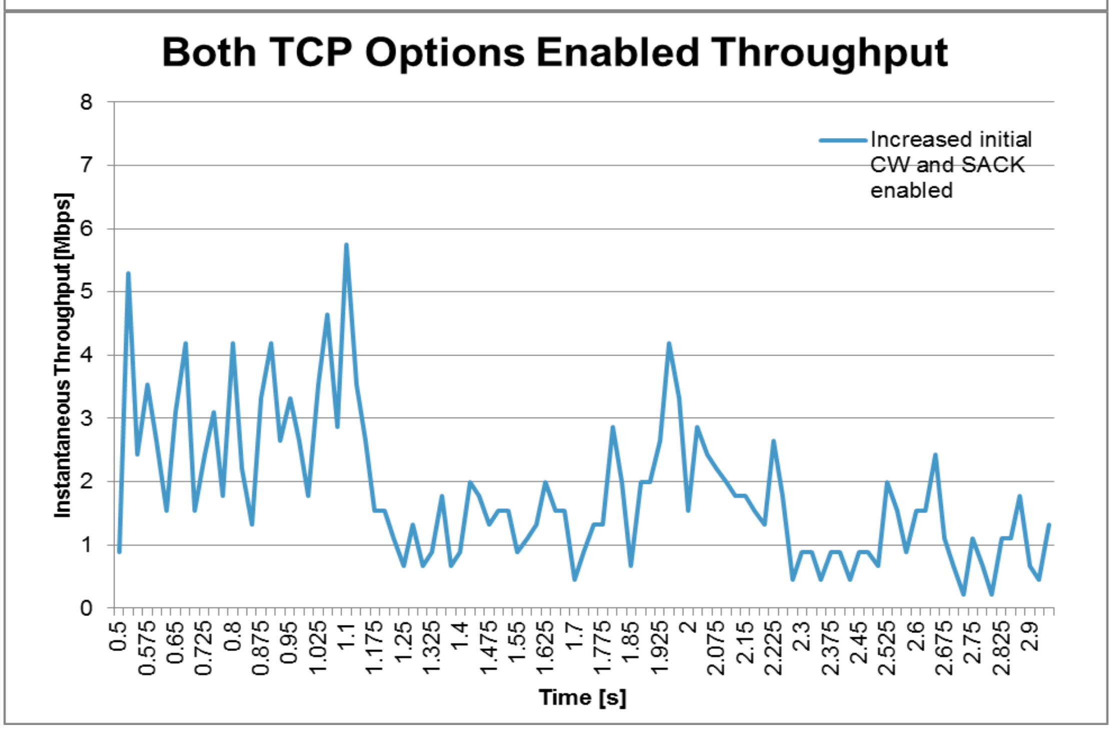
Chart 1 and Chart 2 give the instantaneous throughput results. We have simulated for 100 seconds (actual transmission time is 99 seconds), but we have intentionally selected a smaller period to plot the instantaneous throughput charts below as using all 99 seconds of instantaneous throughput results led to a non-understandable, highly crowded graph. Also, we are attaching the complete instantaneous throughput calculation file in our .zip folder for your reference. These instantaneous throughput plots were changed from 100 ms to 25ms for better interpretation and to show the saw tooth nature of TCP. For the same reason, we are also plotting the CW as a better representation of the saw tooth nature we observed.


Regular TCP Throughput
Regular TCP
Instantaneous Throughput [Mbps]
Time [s]


Increased Initial CW Throughput
Increased Initial CW
Instantaneous Throughput [Mbps]
Time [s]

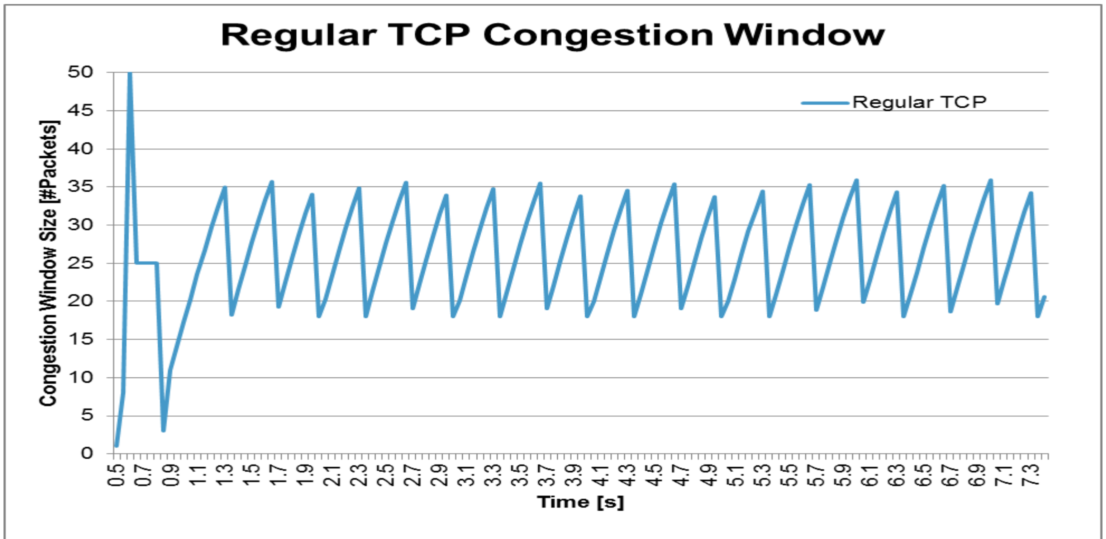


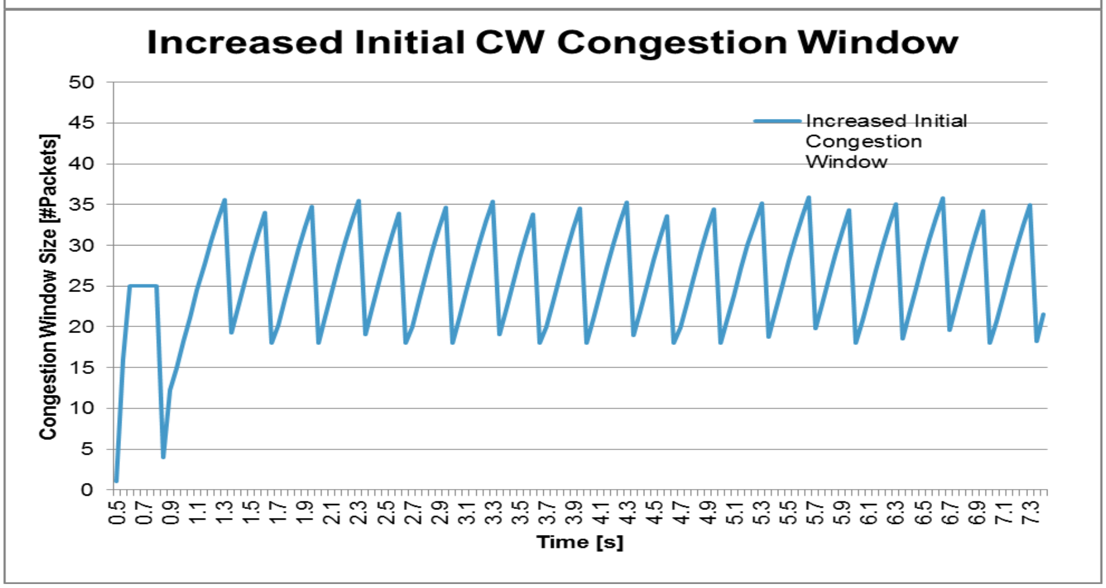
SACK Implemented Throughput
SACK enabled
Instantaneous Throughput [Mbps]
Time [s]

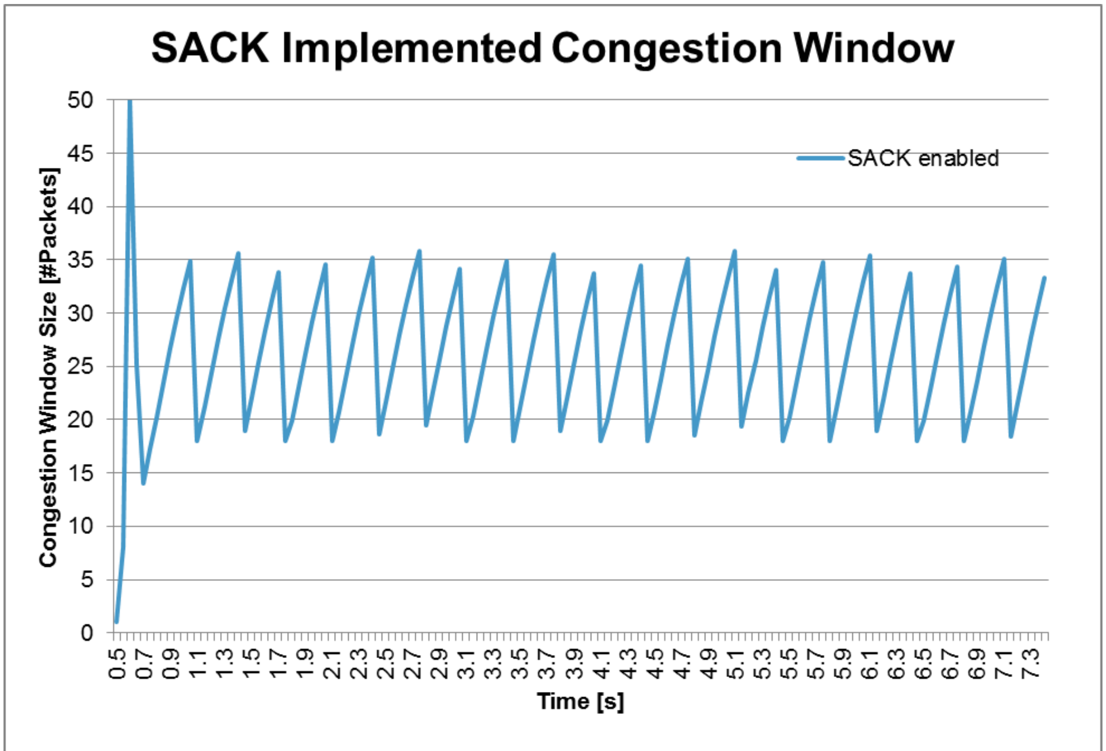


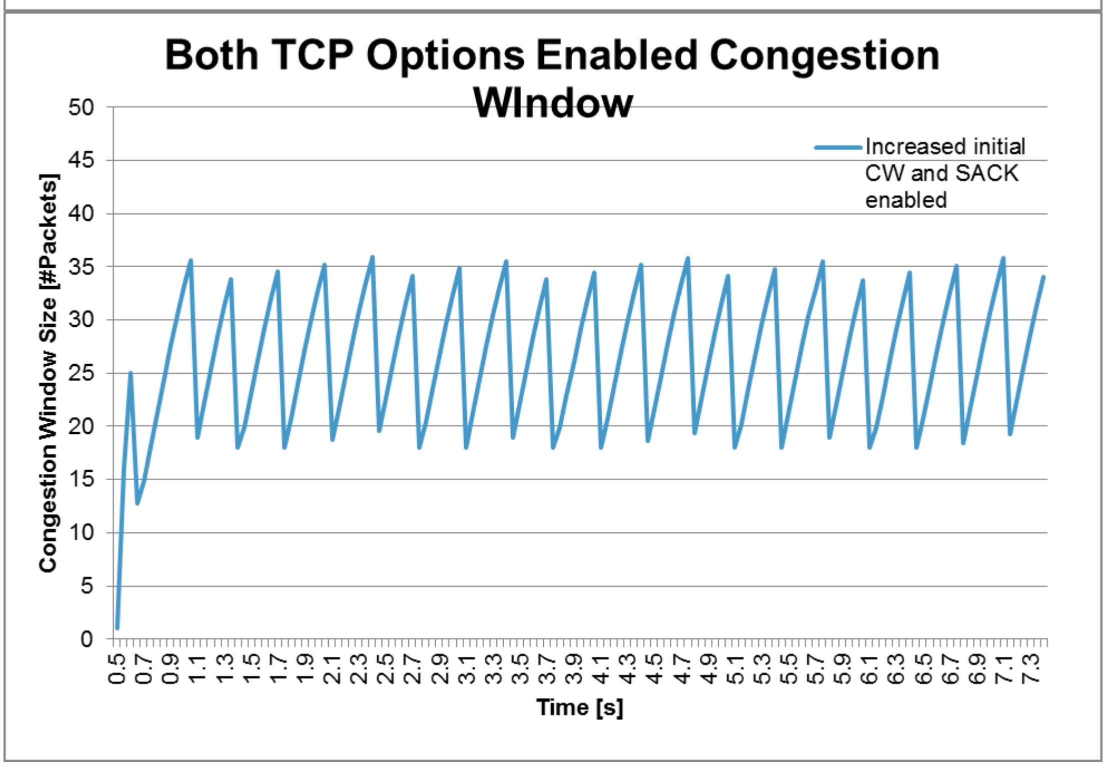
Both TCP Options Enabled Throughput
Increased initial CW and SACK enabled
Instantaneous Throughput [Mbps]
Time [s]

1.


Regular TCP Congestion Window
Regular TCP
Congestion Window Size [#Packets]
Time [s]


Increased Initial CW Congestion Window
Increased Initial Congestion Window
Congestion Window Size [#Packets]
Time [s]


SACK Implemented Congestion Window
Congestion Window Size [#Packets]
Time [s]
SACK enabled
0
5
10
15
20
25
30
35
40
45
50
0.5
0.7
0.9
1.1
1.3
1.5
1.7
1.9
2.1
2.3
2.5
2.7
2.9
3.1
3.3
3.5
3.7
3.9
4.1
4.3
4.5
4.7
4.9
5.1
5.3
5.5
5.7
5.9
6.1
6.3
6.5
6.7
6.9
7.1
7.3


Both TCP Options Enabled Congestion WIndow
Congestion Window Size [#Packets]
Time [s]
Increased initial CW and SACK enabled
0
5
10
15
20
25
30
35
40
45
50
0.5
0.7
0.9
1.1
1.3
1.5
1.7
1.9
2.1
2.3
2.5
2.7
2.9
3.1
3.3
3.5
3.7
3.9
4.1
4.3
4.5
4.7
4.9
5.1
5.3
5.5
5.7
5.9
6.1
6.3
6.5
6.7
6.9
7.1
7.3

(c). Give reasons for the choice of 2 options and the corresponding network conditions.

Choice of the 2 options of TCP :

SACK

Without SACK enabled, TCP requires one RTT to recover from each packet loss and this is a big penalty to pay, especially in wireless communication wherein loss is substantially more than wired, often leading to multiple packet loss (due to random loss, burst loss etc). Here, it would take n*RTT to recover from n packets lost - an inefficient mechanism for wireless communication.

With SACK enabled, it allows the transmitter to recover from more than one packet loss in one RTT. In wireless communication, we often see multiple packets losses (especially burst packet losses) and by enabling SACK we are able to detect and recover from these multi-packet losses faster, therefore increasing the performance of TCP.

CW = 4

TCP connection is vulnerable in the beginning of slow start phase (CW =1 or 2). A packet loss during this time window will require TCP to wait for a timeout as there are not enough packets transmitted to cause dupacks. In wireline connection, this is not a problem as losses do not normally occur and unless there is congestion in the network, in which case, it is right for the connection to start out slowly and continue with a low CW until congestion clears out. In a wireless connection, however, a loss can occur for reasons other than network congestions (e.g. random, handoff). It is therefore not uncommon for a packet loss to occur in the beginning of slow start. If such a case occurs, TCP will have to wait for a timeout in order to recover from the loss. The time in which TCP waits for a timeout could have been used to build up the congestion window. By changing the initial congestion window into 4, we eliminate this problem. With initial congestion window set to 4, when a random packet loss occur during the beginning of slow start, there will be other packets transmitted to trigger dupacks and TCP will be able to recover through fast retransmit. In this way, TCP can continue increasing its congestion window and reach its optimal size. This performance improvement at the beginning of the connection is especially important for wireless connections in which connections are usually short-lived.

Network Conditions

We have selected the following network conditions :

Error percentage = 2.5%

Wireless Link = 8 Mbps

Delay of the link = 5 ms

The reason for selecting an error percentage of 2.5% was that is was within the practical range of losses that are experienced practically in wireless connection. By selecting a loss rate lower, we would not have been able to show the effectiveness of the two options that we have enabled. By having too high of loss will lead to a substantial increase in the effectiveness of the TCP options over the no TCP options, however, the raw throughput of the system will be too low for a communication protocol (we need to show that not only is performance increase substantial, but the utilization of the link is also not too low). Hence we have chosen the in intermediate value of 2.5%.

For the bandwidth of the wireless link, we have chosen 8Mbps considering the following factors. First, the wired link is 20Mbps and in any practical situation, we always have the wired link to have a sufficiently greater bandwidth than the wireless bandwidth. Hence our wireless link would have to be substantially less than the wired 20Mbps. Next we compared the performance of the wireless link (throughput comparisons) for different BW choices and observe that for the loss = 2.5% , the throughput of the link does not substantially increase with an increase in bandwidth after a certain BW (here, it was 8Mbps). In other words, we have considered the spectral efficiency to be a factor in selecting the BW of the wireless link. For example, we feel it would be a highly inefficient utilization of the spectrum (a precious resource) to double the bandwidth (to 16Mbps) for a marginal increase in throughput of 25%. We found that from this justification that 8Mbps was the optimal utilization of the bandwidth for the assumed network conditions.

As for the delay of the wireless link channel, we looked at practical scenarios for the average delay of a link, such as the RTT of ping requests over WiFi in different locations and observed that an average delay of less that 1ms was observed in the links. In order to simulate an unfavorable link that leads to a bottle neck at Node 2 due to delays , we wished to select a value that was higher than 2mS. We found from literature that delays up to the range of 8 to 10 ms were not unheard of for a single hop and therefore we took an intermediate value of 5ms to avoid being on extreme ends of the observed values.

Reasoning of the selection of the two options implemented over other options

(we initially interpreted 1(d) to answer this question; we have kept this answer as an additional question):

We feel our choice of options is appropriate as the aim was to choose two options that would have the most significant positive impact on the link performance in these network conditions.
The other options, while all have substantial merit, were not in the top two options to be implemented for this assignment for the following reasons :

Limited transmit : By enabling SACK option, we will be notifying the TX about multiple packet loss in one single ACK packet. This allows the TX to have multiple opportunities to be able to transmit the same packet again, thereby reducing the chances of DUPACKs to mainly those cases where the packet lost is at the beginning of the CW. In the majority of the other cases, due to SACK, we would have captured and prevented 3 Dupacks from occurring and therefore the effectiveness of this option would have not have been as substantial as using another option. Hence we choose SACK over Limited Transmit option.
Timestamp option : Using the timestamp option for every packet in order to calculate a more accurate RTT leads a significant overhead and processing power. Enabling the other options will have a more impact on throughput that having accurate RTT information. Also, using accurate RTT of last packer might not always bring about positive results as the channel condition at a given point of time is not necessarily a function of the past condition, i.e previous packet's RTT may not have any relation to the next packet's channel condition (although the network infrastructure will be nearly the same for consecutive packets, we wish to say that the channel conditions may not be similar for consecutive packets).

Disabling header compression : We wish to consider the network conditions as an important factor and that a lossy link should always perform worse than an ideal link (no loss). If we enable the option of disabling header compression, in situations where the link quality does improve to close to ideal conditions, this option will actually limit the throughput. In fact, in an ideal no loss channel, this option will reduce throughput (as compared to header compression present in regular TCP) due to the increased overhead associated with every packet. Therefore, although each option has its merits, we feel these other options will not be as effectual as the two that were chosen.

1.(d)

From our results we can see that the throughput of the link has increased due to the implementation of the TCP options. For each of the option, we notice that :

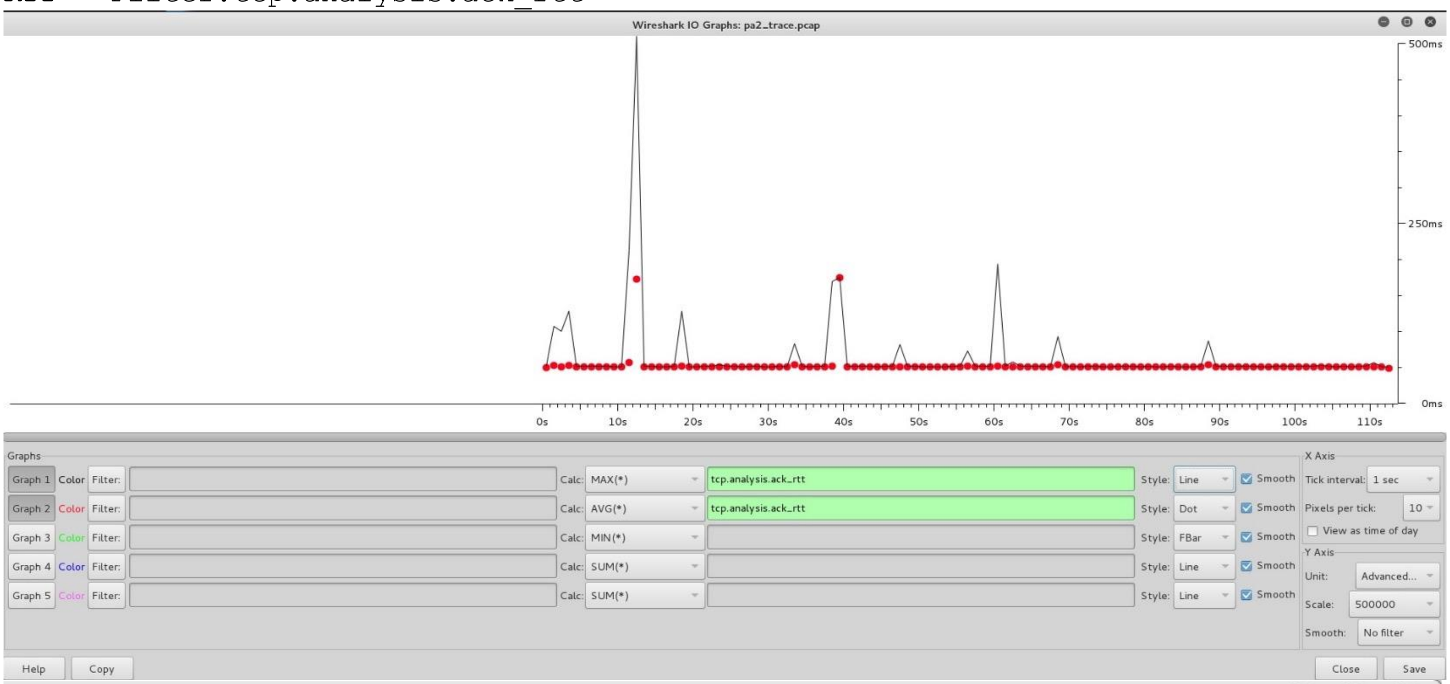
2. SACK : For SACK being enabled, we see that the throughput has substantially increased over no SACK. This can be attributed to less congestions occurring due to the selective ACKs reducing 3 DUPACKs from causing congestion, and thereby allowing the CW to increase. Maintaining a high CW and sustaining it for a longer time period allows the substantial increase in throughput.

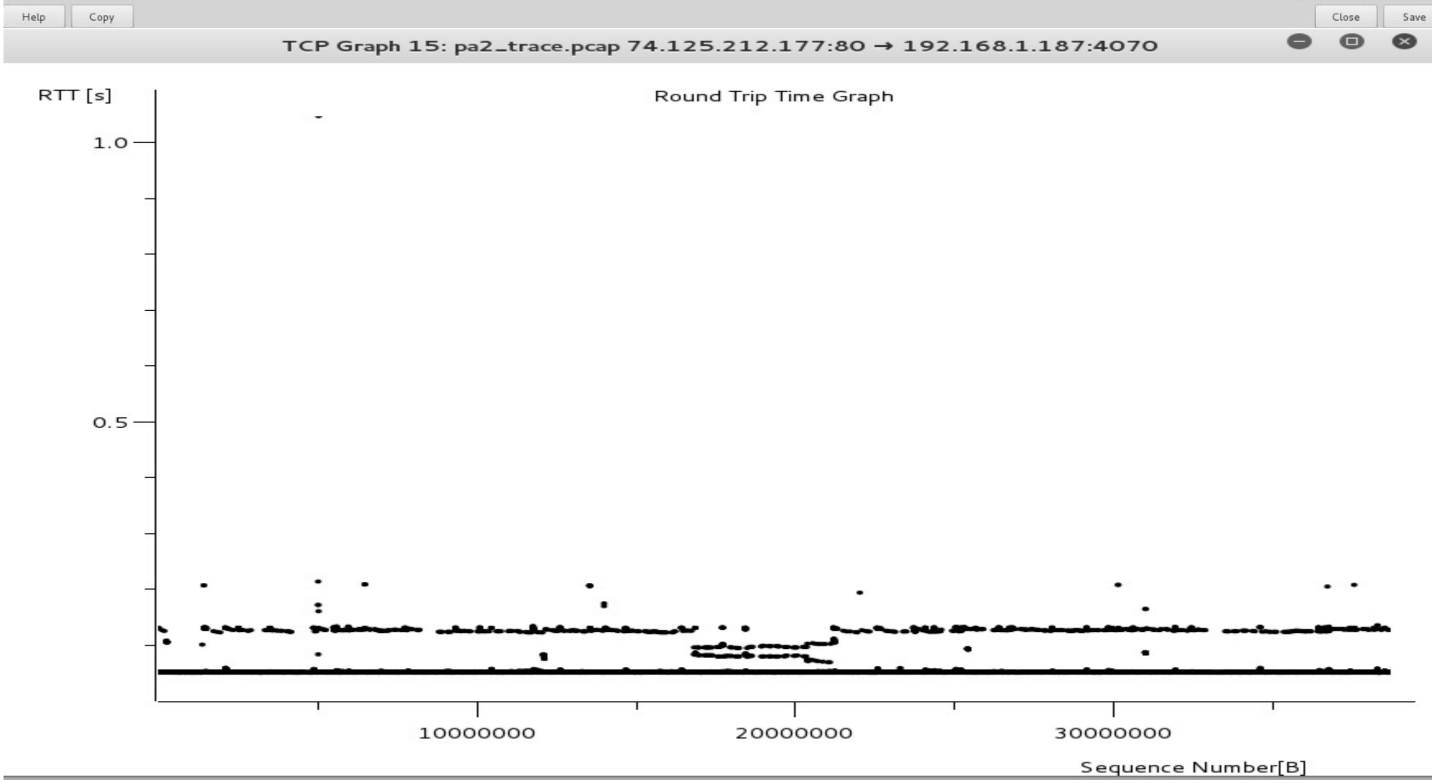
3. CW = 4. We observe an increase in throughput here as well and notice that it is not as large as SACK. The throughput increase is due to the larger CW which prevents timeouts from occurring when the CW is less than 4. As a result, traffic is continuously flowing in to the network, even after a timeout, unlike when CW =1, where one packet loss creates a timeout, due to lack of 3 Dupacks. Also, we notice that the throughput increase is not as high as SACK option enabled as the CW = 4 option takes effect after congestion, in the slow start mode only, whereas SACK's effect is present throughout all three modes of the TCP connection, therefor it leads a higher increase in throughput.

WIRESHARK ANSWERS

2a.
local host: 192.168.1.187
remote host: 74.125.212.177
filter: http.request.uri
uri:
/videoplayback?sparams=id%2CexpilYNUluQ0pPVXMLAAAAaEpreV8zbkR0cElzCw
(TCP Stream FLow)

2b.
RTT = Filter:tcp.analysis.ack_rtt



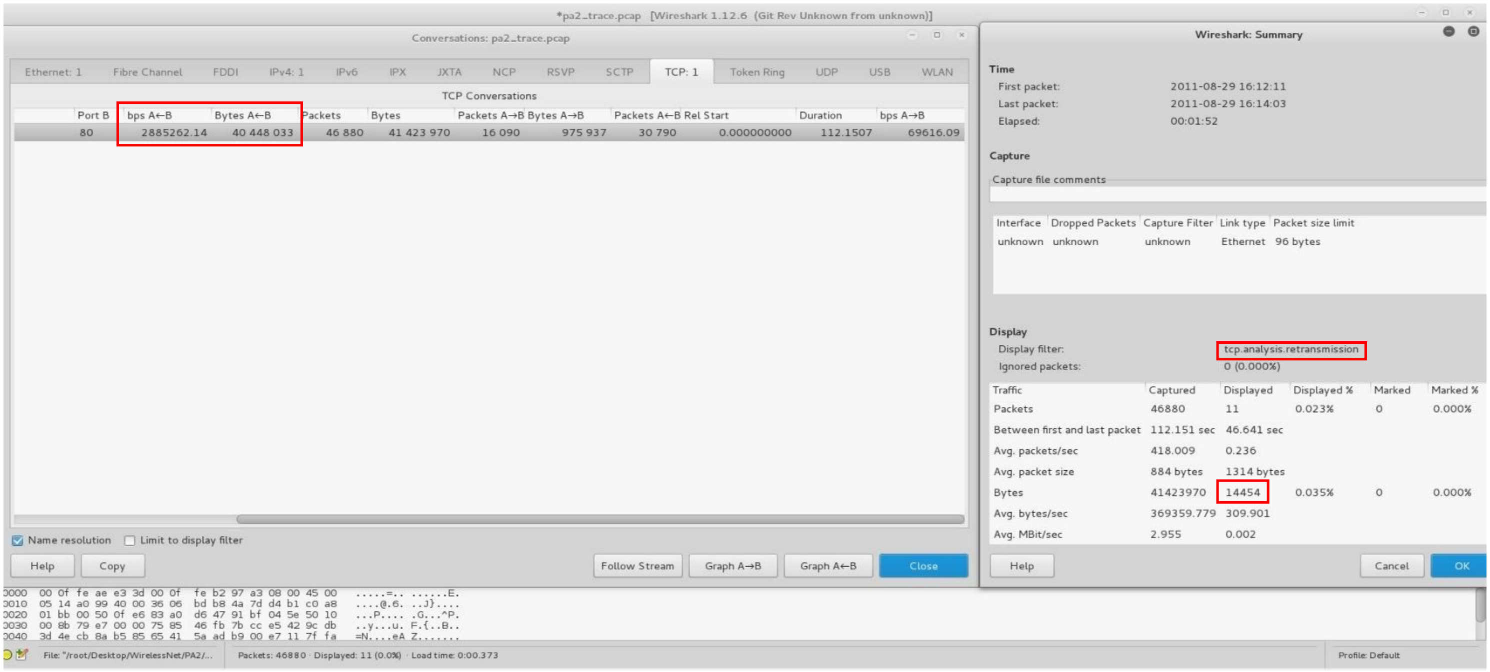


For actual downlink throughput=

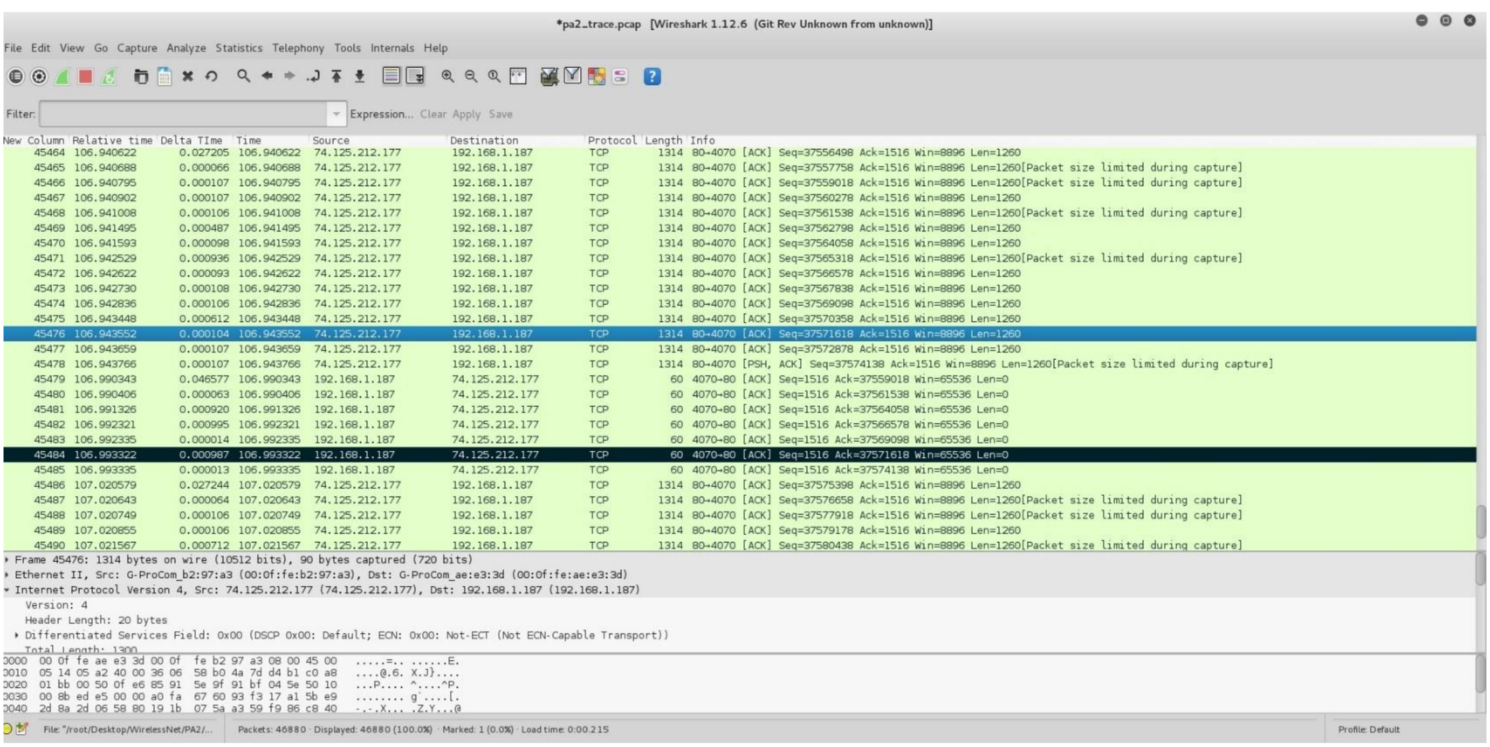
(((Total #bytes - #retx bytes)*8)/(1,000,000*Total Duration)) Mbps

= (((40448033-14454)*8)/(1,000,000*112.057))

= 2.88423 Mbps



To confirm that the Avg. Throughput value isn't off by a lot, we can calculate instantaneous throughput of one packet with respect to its sequence and ACK Number. In the figure below, packet with sequence number 37571618 is ACKed at 106.993322, so Throughput= 2.8092 Mbps, which is not that different from the Avg. Throughput.

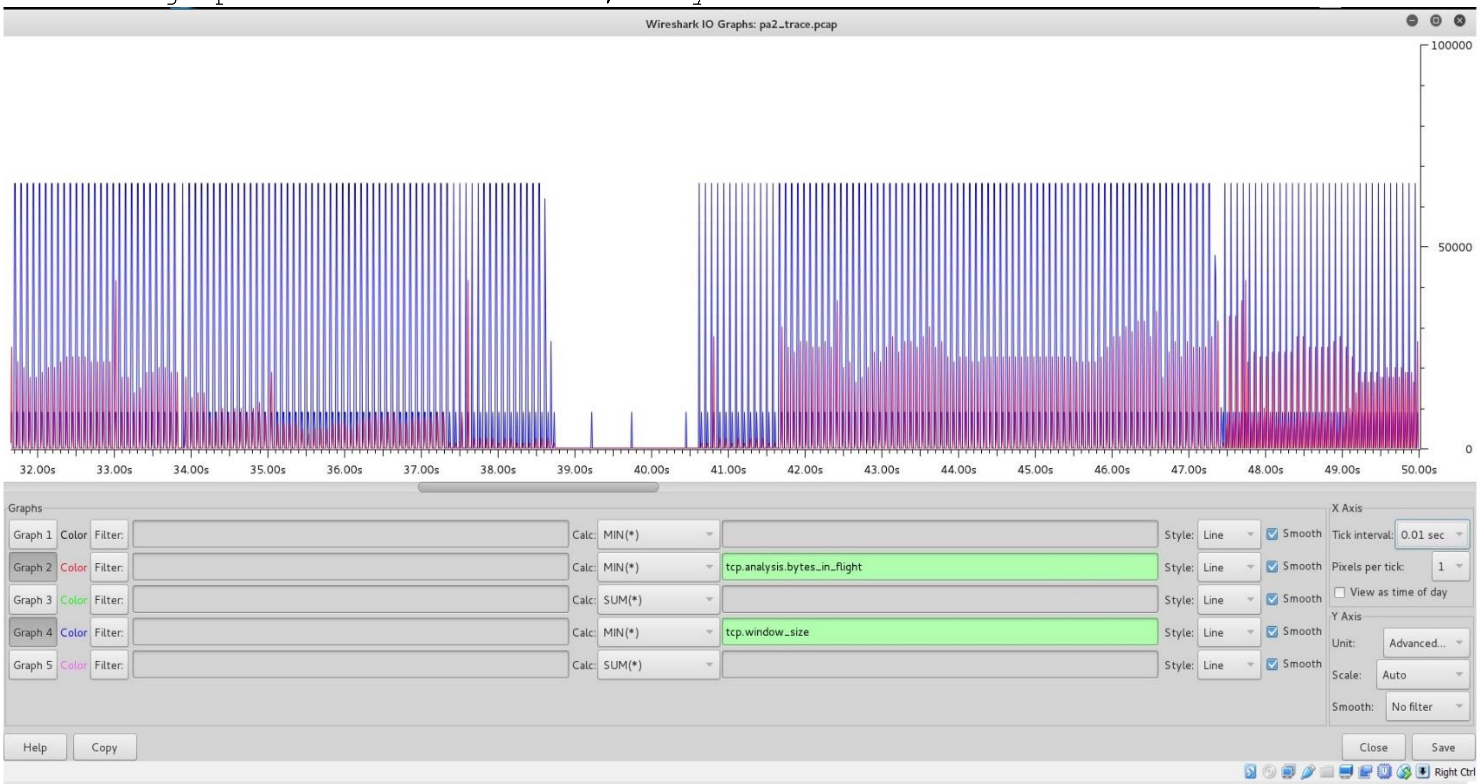


Congestion Window:

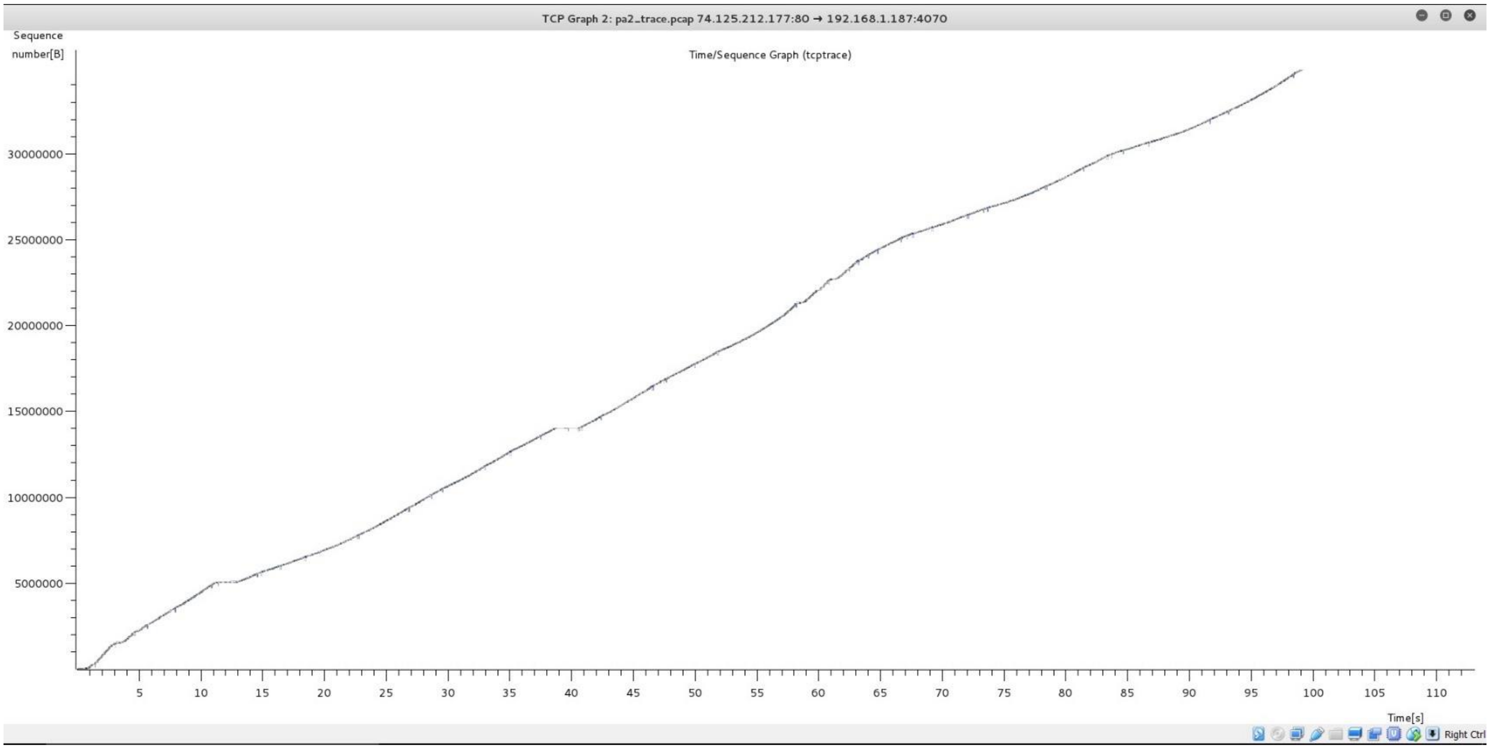
Filter: tcp.analysis.bytes_in_flight

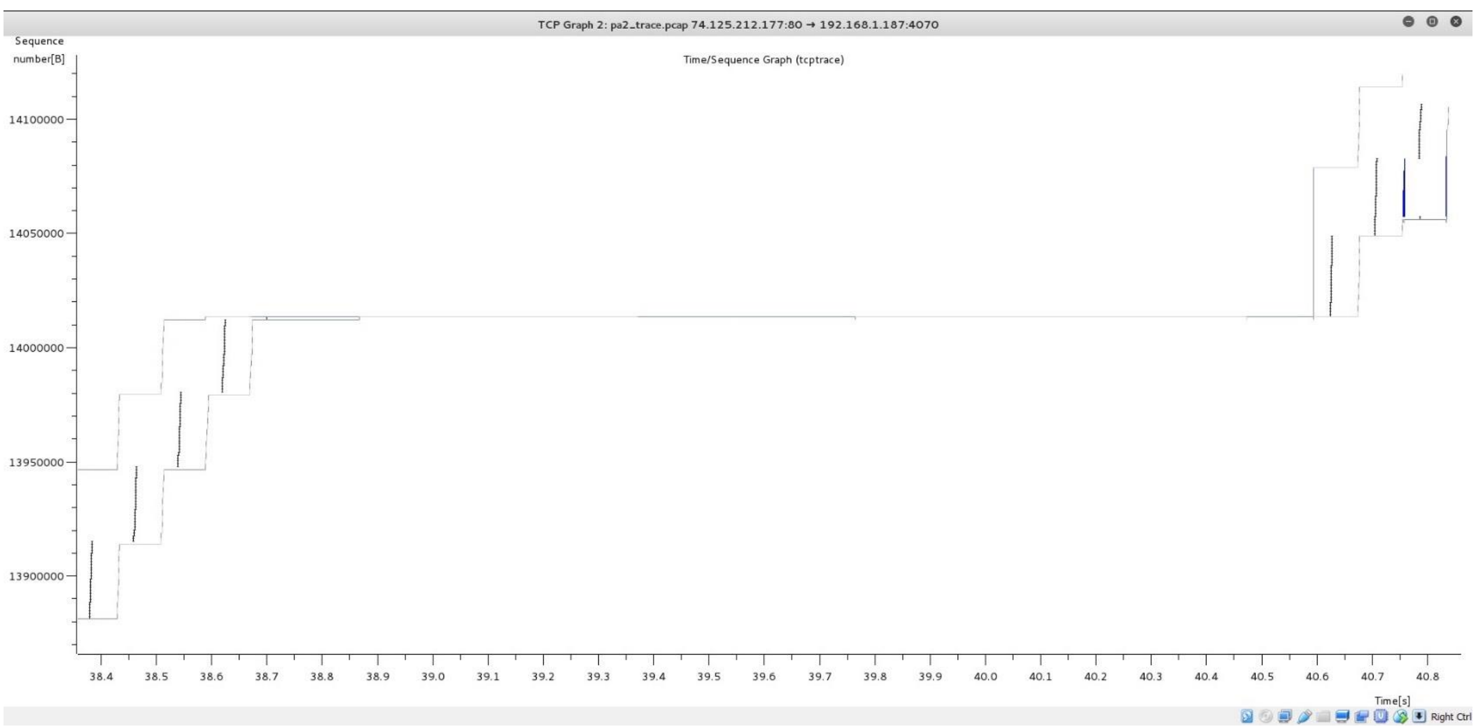
We can't 'get' that value directly from the capture file, as it is not advertized. bytes_in_flight are un-ACKed bytes for Wireshark. So, this gives us a good estimate of the congestion window progression. Also, if bytes_in_flight is below the advertized window size, it can also mean that the receiver ACKed before the receive window was full.

Both the graphs below are the same, only with different Tick Interval.



Another Graph, time-sequence number graph, can give a better idea about congestion and how fast is the recovery. Both Plots below are the same with the second plot being a zoomed in version of the first one around time=40.0 Seconds where Zero Window Occurs.





2c.

Zero Window

filter: tcp.analysis.zero_window

From Receivers point of view (192.168.1.187,4070,74.125.212.177,80)

Receivers Side:

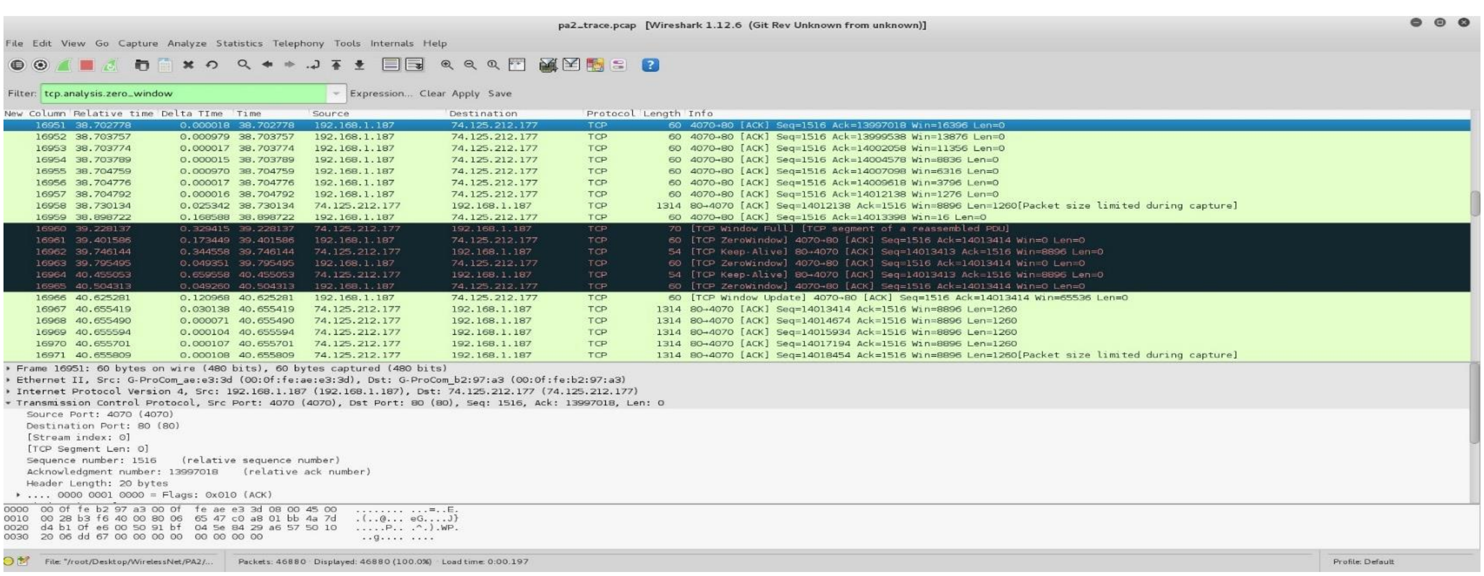
Through TCP ACK frames, the client informs the server of how much room is in this buffer. If the TCP Window Size goes down to 0, the client will not be able to receive any more data until it processes and opens the buffer up again. In this case, Protocol Expert will alert a "Zero Window" in Expert View. The receiver keeps doing this till it can process more data.

Possible Reason for Zero Window:

It could be that the machine is running too many processes at that moment, and its processor is maxed. Or it could be that there is an error in the TCP receiver, like a Windows registry misconfiguration.

Senders Side:

The sending station is trying to see if the remote peer is dead(using Keep Alives), if the connection is still open and in use, or may just need to keep the connection open instead of suffering another handshake overhead. If the target does not respond, the sender may send several Keep Alives before finally sending a TCP reset to kill the socket. This is a good thing, since we don't want open/unused TCP connections staying open and hogging resources forever.

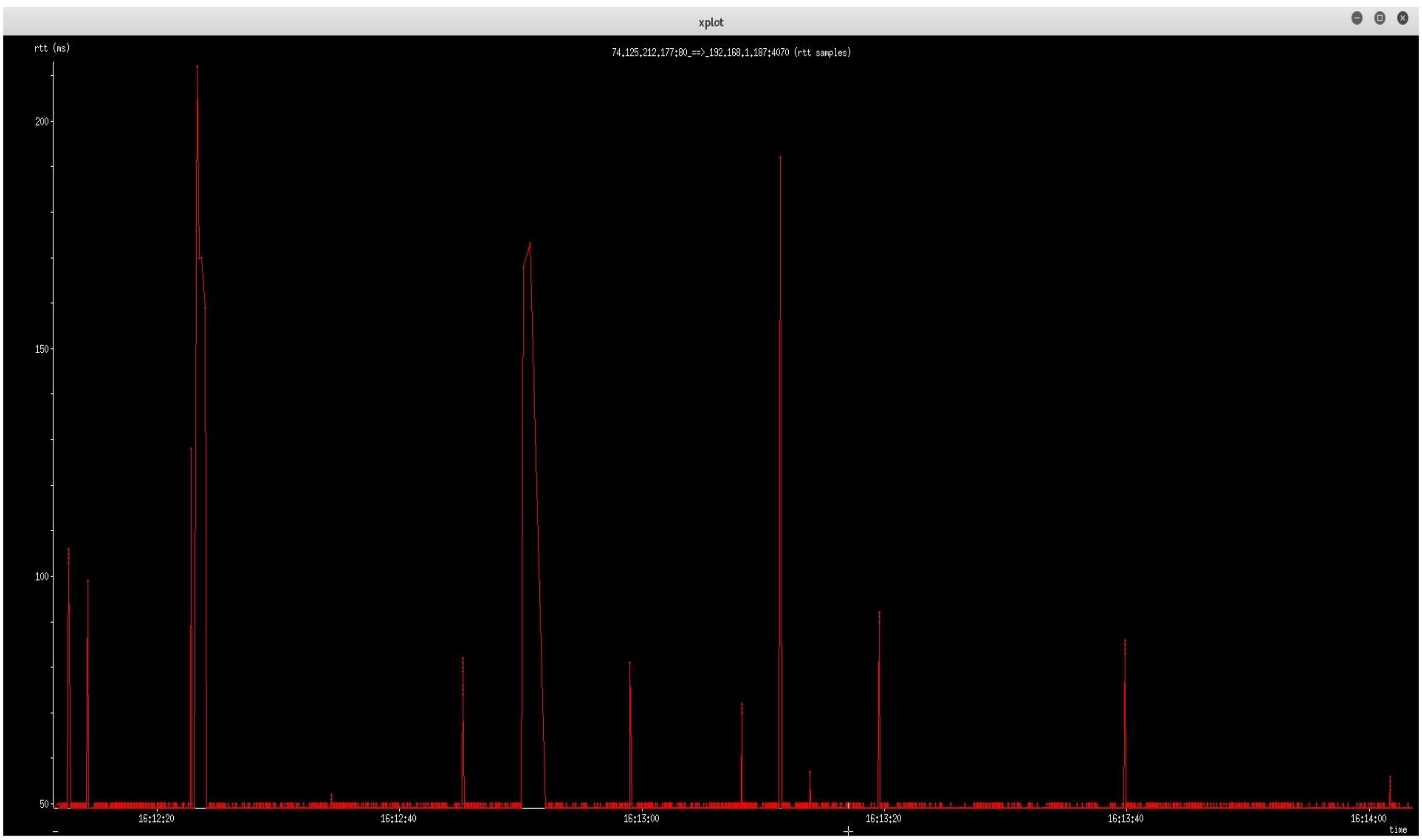


TCPTRACE ANSWERS

2a.
local host: 192.168.1.187
remote host: 74.125.212.177

Command:
tcptrace -e pa2_trace.pcap
(for uri extraction, this extracts content of tcp stream into a file which contains the uri)
uri: /videoplayback?sparams=id%2CexpilYNUluQ0pPVXMLAAAAaEpreV8zbkR0cElzCw

2b.
RTT
Commands:
tcptrace -G pa2_trace.pcap (Creates all the graphs)
xplot.org b2a_rtt.xpl (plots rtt for b->a)



Downlink Throughput:

Command:

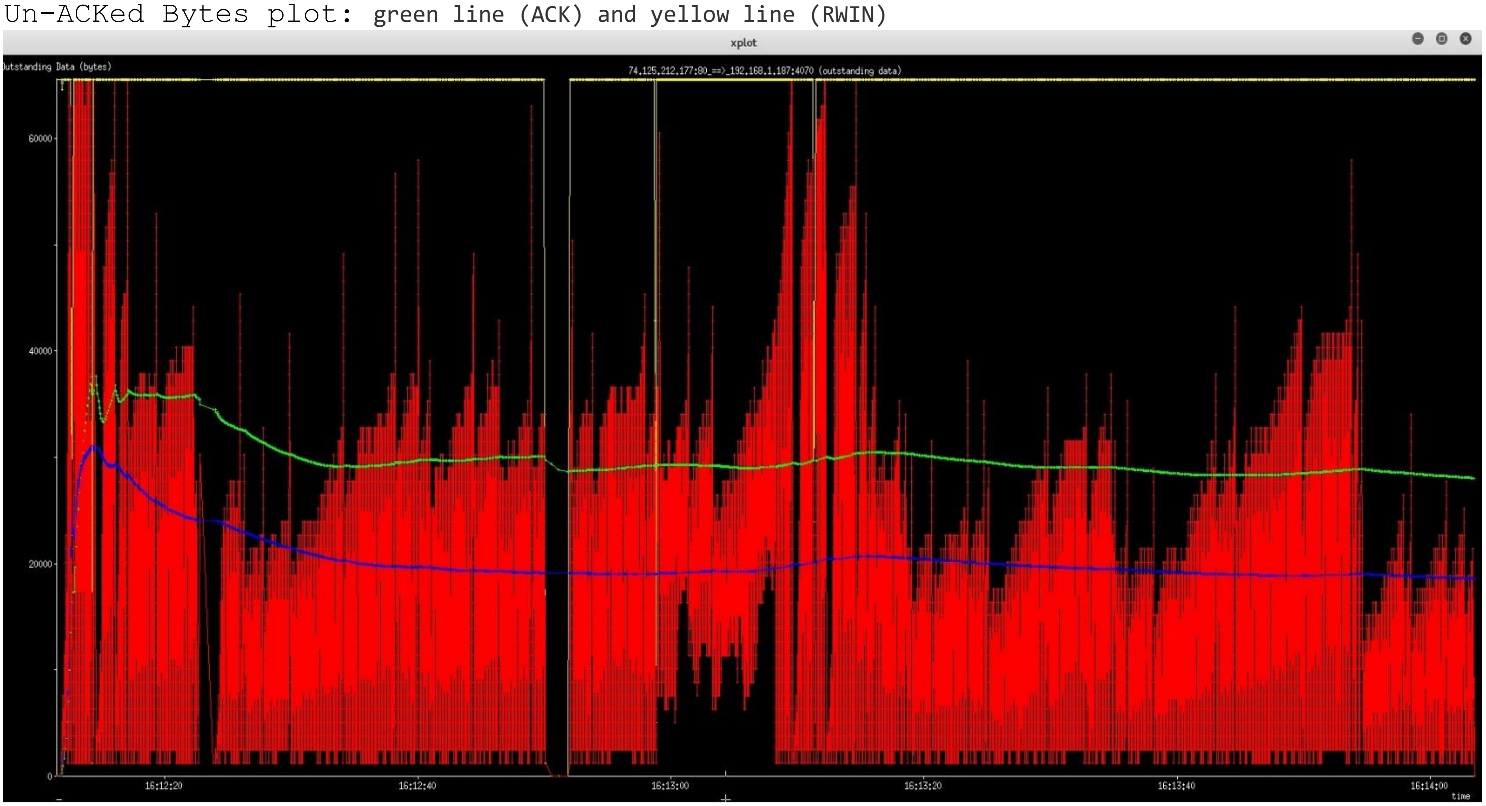
tcptrace -r -l pa2_trace.pcap

```
46880 packets seen, 46880 TCP packets traced
elapsed wallclock time: 0:00:00.052966, 885096 pkts/sec analyzed
trace file elapsed time: 0:01:52.150733
TCP connection info:
1 TCP connection traced:
TCP connection 1:
	host a:        192.168.1.187:4070
	host b:        74.125.212.177:80
	complete conn: yes
	first packet:  Mon Aug 29 16:12:11.356700 2011
	last packet:   Mon Aug 29 16:14:03.507433 2011
	elapsed time:  0:01:52.150733
	total packets: 46880
	filename:      pa2_trace.pcap
   a->b:			      b->a:
     total packets:         16090           total packets:         30790
     ack pkts sent:         16089           ack pkts sent:         30790
     pure acks sent:        16086           pure acks sent:            4
     sack pkts sent:         1466           sack pkts sent:            0
     dsack pkts sent:           0           dsack pkts sent:           0
     max sack blks/ack:         2           max sack blks/ack:         0
     unique bytes sent:      1515           unique bytes sent:  38719841
     actual data pkts:          2           actual data pkts:      30785
     actual data bytes:      1515           actual data bytes:  38785361
     rexmt data pkts:           0           rexmt data pkts:          52
     rexmt data bytes:          0           rexmt data bytes:      65520
     zwnd probe pkts:           0           zwnd probe pkts:           0
     zwnd probe bytes:          0           zwnd probe bytes:          0
     outoforder pkts:           0           outoforder pkts:          16
     pushed data pkts:          1           pushed data pkts:        578
     SYN/FIN pkts sent:       1/1           SYN/FIN pkts sent:       1/1
     req 1323 ws/ts:          Y/N           req 1323 ws/ts:          Y/N
     adv wind scale:            1           adv wind scale:            6
     req sack:                  Y           req sack:                  Y
     sacks sent:             1466           sacks sent:                0
     urgent data pkts:          0 pkts      urgent data pkts:          0 pkts
     urgent data bytes:         0 bytes     urgent data bytes:         0 bytes
     mss requested:          1260 bytes     mss requested:          1380 bytes
     max segm size:          1260 bytes     max segm size:          1260 bytes
     min segm size:           255 bytes     min segm size:            16 bytes
     avg segm size:           757 bytes     avg segm size:          1259 bytes
     max win adv:           65536 bytes     max win adv:            8896 bytes
     min win adv:              16 bytes     min win adv:            8896 bytes
     zero win adv:              3 times     zero win adv:              0 times
     avg win adv:           65251 bytes     avg win adv:            8896 bytes
     initial window:         1515 bytes     initial window:         2197 bytes
     initial window:            2 pkts      initial window:            3 pkts
     ttl stream length:      1515 bytes     ttl stream length:  38719841 bytes
     missed data:               0 bytes     missed data:               0 bytes
     truncated data:         1443 bytes     truncated data:     37677121 bytes
     truncated packets:         2 pkts      truncated packets:     30784 pkts
```

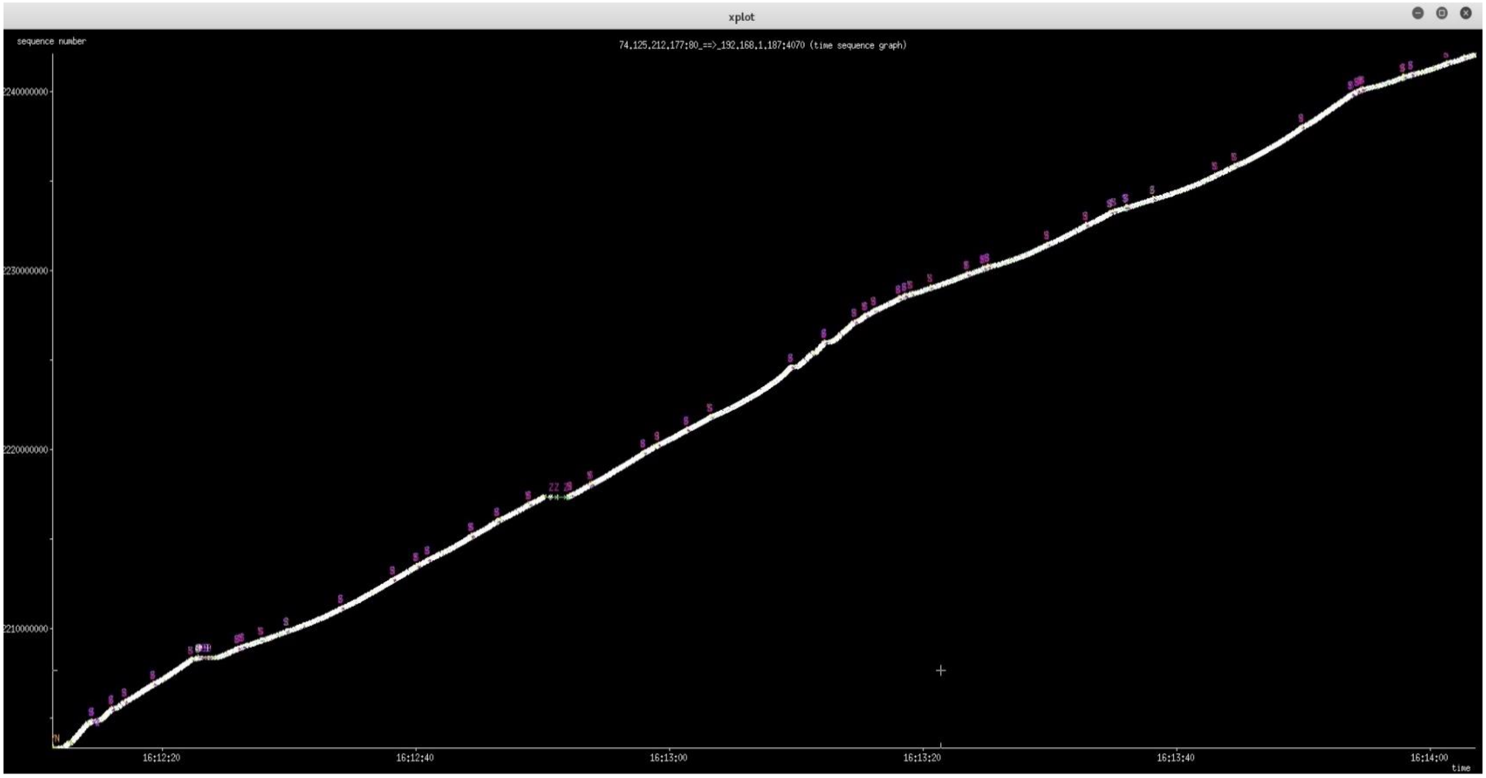
data xmit time:	0.000 secs	data xmit time:	111.603 secs
idletime max:	708.8 ms	idletime max:	708.9 ms
throughput:	14 Bps	throughput:	345248 Bps
RTT samples:	3	RTT samples:	14596
RTT min:	30.6 ms	RTT min:	49.1 ms
RTT max:	31.1 ms	RTT max:	212.6 ms
RTT avg:	30.8 ms	RTT avg:	49.9 ms
RTT stdev:	0.3 ms	RTT stdev:	3.6 ms
RTT from 3WHS:	31.1 ms	RTT from 3WHS:	49.8 ms
RTT full_sz smpls:	2	RTT full_sz smpls:	1
RTT full_sz min:	30.6 ms	RTT full_sz min:	49.8 ms
RTT full_sz max:	31.1 ms	RTT full_sz max:	49.8 ms
RTT full_sz avg:	30.9 ms	RTT full_sz avg:	49.8 ms
RTT full_sz stdev:	0.0 ms	RTT full_sz stdev:	0.0 ms
post-loss acks:	0	post-loss acks:	52
segs cum acked:	1	segs cum acked:	16086
duplicate acks:	2	duplicate acks:	1435
triple dupacks:	0	triple dupacks:	52
max # retrans:	0	max # retrans:	1
min retr time:	0.0 ms	min retr time:	79.9 ms
max retr time:	0.0 ms	max retr time:	158.0 ms
avg retr time:	0.0 ms	avg retr time:	88.1 ms
sdv retr time:	0.0 ms	sdv retr time:	22.5 ms

Throughput : b->a
Downlink Throughput= (unique bytes sent * 8)/1,000,000
= 2.775Mbps (According to tcptrace, Wireshark reports slightly different value)

Congestion Window: (Explanation same as Wireshark)
Command:
xplot.org b2a_owin.xpl (plots unacked bytes for b->a)
Un-ACKed Bytes plot: green line (ACK) and yellow line (RWIN)



Time Sequence Plot:



2c.

Zero Window

Command:

tcptrace -r -l pa2_trace.pcap

The output of the above command gives the Zero Window Advertisement(if any).The output is given in part 2b and we can observe that the Local Host (192.168.1.187), makes a Zero Window Adv 3 times.

From Receivers point of view (192.168.1.187,4070,74.125.212.177,80)

Iperf

- Measure the bi-directional bandwidth between client and server at 3 different locations

Location 1

```
C:\Users\joo5819\Downloads\iperf-3.0.11-win64>iperf3.exe -c 192.168.1.83 -d
send_parameters:
{
        "tcp":  true,
        "omit": 0,
        "time": 10,
        "parallel":     1,
        "len":  131072,
        "client_version":       "3.1b3"
}
Connecting to host 192.168.1.83, port 5201
SO_SNDBUF is 212992
[  4] local 192.168.1.90 port 49226 connected to 192.168.1.83 port 5201
[ ID] Interval           Transfer     Bandwidth
[  4]   0.00-1.01   sec  2.75 MBytes  22.9 Mbits/sec
[  4]   1.01-2.00   sec  2.00 MBytes  16.9 Mbits/sec
[  4]   2.00-3.01   sec  2.38 MBytes  19.8 Mbits/sec
[  4]   3.01-4.00   sec  2.75 MBytes  23.2 Mbits/sec
[  4]   4.00-5.01   sec  2.88 MBytes  24.0 Mbits/sec
[  4]   5.01-6.01   sec  2.50 MBytes  20.8 Mbits/sec
[  4]   6.01-7.01   sec  2.62 MBytes  22.2 Mbits/sec
[  4]   7.01-8.00   sec  2.50 MBytes  21.1 Mbits/sec
[  4]   8.00-9.00   sec  2.62 MBytes  22.0 Mbits/sec
send_results
{
        "cpu_util_total":       22.3658,
        "cpu_util_user":        0.455258,
        "cpu_util_system":      21.9105,
        "sender_has_retransmits":       0,
        "streams":      [{
                        "id":   1,
                        "bytes":        26738688,
                        "retransmits":  -1
```

```
        "streams":      [{
                        "id":   1,
                        "bytes":        26738688,
                        "retransmits":  -1,
                        "jitter":       0,
                        "errors":       0,
                        "packets":      0
                }]
}
[  4]   9.00-10.00  sec  2.50 MBytes  21.0 Mbits/sec
- - - - - - - - - - - - - - - - - - - - - - - - -
[ ID] Interval           Transfer     Bandwidth
[  4]   0.00-10.00  sec  25.5 MBytes  21.4 Mbits/sec                  sender
[  4]   0.00-10.00  sec  25.5 MBytes  21.4 Mbits/sec                  receiver

iperf Done.
```

Location 2

```
{
        "cpu_util_total":       3.74733,
        "cpu_util_user":        0.59573,
        "cpu_util_system":      3.1516,
        "sender_has_retransmits":       0,
        "streams":      [{
                        "id":   1,
                        "bytes":        19136512,
                        "retransmits":  -1,
                        "jitter":       0,
                        "errors":       0,
                        "packets":      0
                }]
}
get_results
{
        "cpu_util_total":       0.768734,
        "cpu_util_user":        0.106403,
        "cpu_util_system":      0.662319,
        "sender_has_retransmits":       -1,
        "streams":      [{
                        "id":   1,
                        "bytes":        19136512,
                        "retransmits":  -1,
                        "jitter":       0,
                        "errors":       0,
                        "packets":      0
                }]
}
[  4]   9.00-10.00  sec  1.88 MBytes  15.7 Mbits/sec
- - - - - - - - - - - - - - - - - - - - - - - - -
[ ID] Interval           Transfer     Bandwidth
[  4]   0.00-10.00  sec  18.2 MBytes  15.3 Mbits/sec                  sender
[  4]   0.00-10.00  sec  18.2 MBytes  15.3 Mbits/sec                  receiver

iperf Done.

C:\Users\yakshdeep\Desktop\iperf-3.0.11-win64>
```

Location 3

```
{
        "tcp":  true,
        "omit": 0,
        "time": 10,
        "parallel":     1,
        "len":  131072,
        "client_version":       "3.1b3"
}
Connecting to host 192.168.1.85, port 5201
SO_SNDBUF is 212992
[  4] local 192.168.1.90 port 56965 connected to 192.168.1.85 port 5201
[ ID] Interval           Transfer     Bandwidth
[  4]   0.00-1.00   sec  3.75 MBytes  31.4 Mbits/sec
[  4]   1.00-2.02   sec  4.12 MBytes  34.1 Mbits/sec
[  4]   2.02-3.01   sec  4.50 MBytes  38.1 Mbits/sec
[  4]   3.01-4.00   sec  4.25 MBytes  35.8 Mbits/sec
[  4]   4.00-5.00   sec  4.50 MBytes  37.8 Mbits/sec
[  4]   5.00-6.00   sec  4.62 MBytes  38.6 Mbits/sec
[  4]   6.00-7.00   sec  4.00 MBytes  33.7 Mbits/sec
[  4]   7.00-8.01   sec  4.75 MBytes  39.5 Mbits/sec
[  4]   8.01-9.02   sec  4.25 MBytes  35.2 Mbits/sec
send_results
{
        "cpu_util_total":       41.7309,
        "cpu_util_user":        0.761585,
        "cpu_util_system":      40.9694,
        "sender_has_retransmits":       0,
        "streams":      [{
                        "id":   1,
                        "bytes":        44957696,
                        "retransmits":  -1,
                        "jitter":       0,
                        "errors":       0,
                        "packets":      0
                }]
}
get_results
{
        "cpu_util_total":       1.14028,
        "cpu_util_user":        0.129477,
        "cpu_util_system":      1.01081,
        "sender_has_retransmits":       -1,
        "streams":      [{
                        "id":   1,
                        "bytes":        44955104,
                        "retransmits":  -1,
                        "jitter":       0,
                        "errors":       0,
                        "packets":      0
                }]
}
[  4]   9.02-10.00  sec  4.12 MBytes  35.2 Mbits/sec
- - - - - - - - - - - - - - - - - - - - - - - - -
[ ID] Interval           Transfer     Bandwidth
[  4]   0.00-10.00  sec  42.9 MBytes  36.0 Mbits/sec                  sender
[  4]   0.00-10.00  sec  42.9 MBytes  35.9 Mbits/sec                  receiver
```

- Change TCP window size appropriately to get better bandwidth.

Window Size: 5000 bytes:

Transfer Data Size(Mb)	*Bandwidth(Mb/s)*
11.5	*9.61*
13.9	*11.7*
12.1	*10.1*
13.9	*11.6*

Window Size 10000 bytes:

Transfer Data Size(Mb)	*Bandwidth(Mb/s)*
16.8	*14.1*
15.3	*12.8*
11.8	*9.9*
15.1	*12.7*

Widow Size 16357 bytes:

Transfer Data Size(Mb)	*Bandwidth(Mb/s)*
18.3	*15.3*
19.2	*16.1*
14.8	*12.4*
16.9	*14.2*

Window Size 25000 bytes:

Transfer Data Size(Mb)	*Bandwidth(Mb/s)*
14.7	*12.3*
17.2	*14.4*
15.7	*13.2*
18.9	*15.8*

Window Size 40000 bytes:

Transfer Data Size(Mb)	*Bandwidth(Mb/s)*
23.7	*19.9*
21.7	*18.2*
22.5	*18.8*
24.5	*20.5*

Window Size 50000 bytes:

Transfer Data Size(Mb)	*Bandwidth(Mb/s)*
20.4	*17.1*
22.9	*19.2*
19.6	*16.4*
24.2	*20.3*

Window Size 60000 bytes:

Transfer Data Size(Mb)	*Bandwidth(Mb/s)*
23.6	*19.8*
25.9	*21.7*
22.1	*18.6*
22.8	*19.2*

Explanation

For best Results, tcp window size = Bandwidth delay product. In our case, the bandwidth that we had was 34Mbps & average delay of 10ms, so BDP= 42500 Bytes.

Thus, we can see from the above results that the bandwidth keeps improving till 40KB (window size) and remains somewhat constant for window sizes >40KB.

- **Compare the throughput observed when using bi-directional UDP and a TCP**

  UDP Result

```
C:\Users\yakshdeep\Desktop\iperf-3.0.11-win64>iperf3.exe -c 24.72.251.234 -u -b 10000m
Connecting to host 24.72.251.234, port 5201
[  4] local 24.72.250.122 port 58139 connected to 24.72.251.234 port 5201
[ ID] Interval           Transfer     Bandwidth       Total Datagrams
[  4]   0.00-1.02   sec  1.88 MBytes  15.5 Mbits/sec  241
[  4]   1.02-2.02   sec  1.98 MBytes  16.6 Mbits/sec  253
[  4]   2.02-3.00   sec  2.15 MBytes  18.3 Mbits/sec  275
[  4]   3.00-4.01   sec  2.23 MBytes  18.6 Mbits/sec  285
[  4]   4.01-5.00   sec  2.22 MBytes  18.7 Mbits/sec  284
[  4]   5.00-6.01   sec  2.02 MBytes  16.8 Mbits/sec  259
[  4]   6.01-7.01   sec  1.93 MBytes  16.3 Mbits/sec  247
[  4]   7.01-8.01   sec  1.72 MBytes  14.4 Mbits/sec  220
[  4]   8.01-9.00   sec  1.72 MBytes  14.5 Mbits/sec  220
[  4]   9.00-10.00  sec  2.05 MBytes  17.2 Mbits/sec  262
- - - - - - - - - - - - - - - - - - - - - - - - -
[ ID] Interval           Transfer     Bandwidth       Jitter    Lost/Total Datagrams
[  4]   0.00-10.00  sec  19.9 MBytes  16.7 Mbits/sec  3.479 ms  1/2544 (0.039%)
[  4] Sent 2544 datagrams

iperf Done.
```

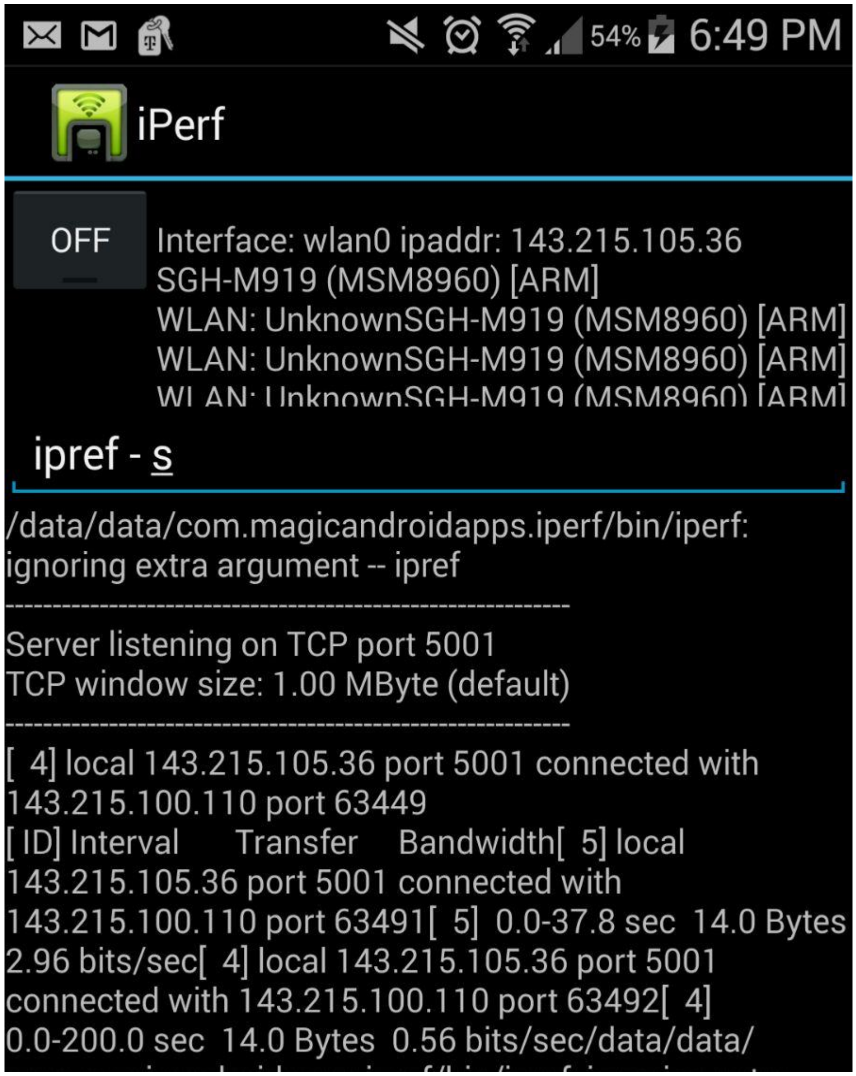
  TCP Result:

```
send_results
{
        "cpu_util_total":       1.21578,
        "cpu_util_user":        0,
        "cpu_util_system":      1.21578,
        "sender_has_retransmits":       0,
        "streams":      [{
                        "id":   1,
                        "bytes":        14362144,
                        "retransmits":  -1,
                        "jitter":       0,
                        "errors":       0,
                        "packets":      0
                }]
}
get_results
{
        "cpu_util_total":       1.97622,
        "cpu_util_user":        0.69041,
        "cpu_util_system":      1.28581,
        "sender_has_retransmits":       -1,
        "streams":      [{
                        "id":   1,
                        "bytes":        14316164,
                        "retransmits":  -1,
                        "jitter":       0,
                        "errors":       0,
                        "packets":      0
                }]
}
[  4]   9.00-10.00  sec  1.72 MBytes  14.4 Mbits/sec
- - - - - - - - - - - - - - - - - - - - - - - - -
[ ID] Interval           Transfer     Bandwidth
[  4]   0.00-10.00  sec  13.7 MBytes  11.5 Mbits/sec                  sender
[  4]   0.00-10.00  sec  13.7 MBytes  11.5 Mbits/sec                  receiver
```

  UDP performs better than TCP in the tested network conditions since in UDP mode, iperf has no flow control whereas TCP does (limiting bandwidth). Since the data rate is not high in this case, UDP performs better than TCP.

  For higher data rates, it is imperative that TCP performs better.

- **Install iperf in your phone(server) . Analyze the bandwidth measured when the server is moving.**



The speeds are terrible when the server is mobile this is probably due to the server getting farther away from the access point.