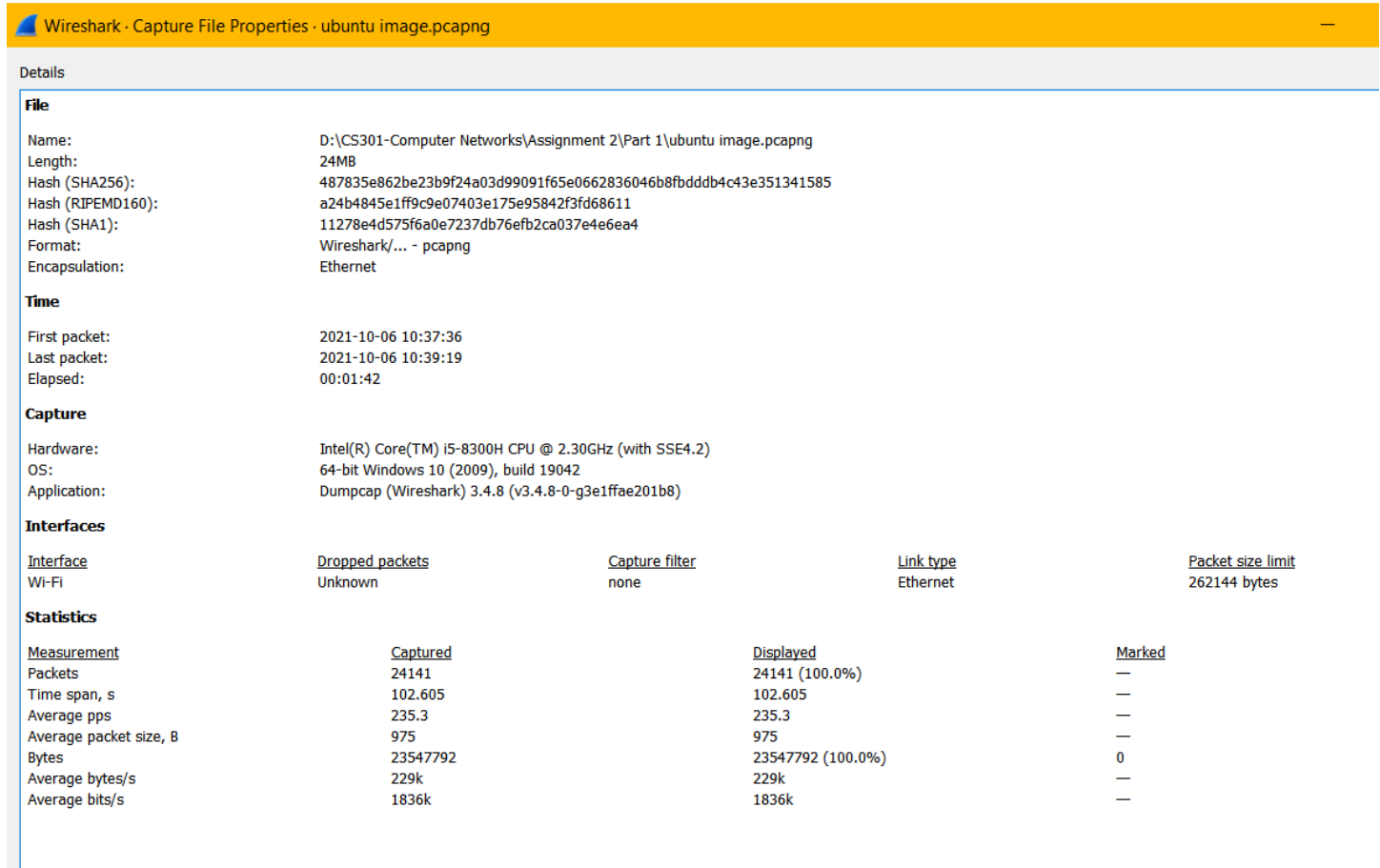


Part 1

Question 1

The Wireshark Capture file for the ubuntu image: ubuntu image.pcapng

At the URL: <https://ubuntu.com/download/desktop>, there are multiple image files, I downloaded the Ubuntu 21.04 file (Size: 2.6 GB).



Wireshark · Capture File Properties · ubuntu image.pcapng

Details

File

Name: D:\CS301-Computer Networks\Assignment 2\Part 1\ubuntu image.pcapng
Length: 24MB
Hash (SHA256): 487835e862be23b9f24a03d99091f65e0662836046b8fbdddb4c43e351341585
Hash (RIPEMD160): a24b4845e1ff9c9e07403e175e95842f3fd68611
Hash (SHA1): 11278e4d575f6a0e7237db76efb2ca037e4e6ea4
Format: Wireshark/... - pcapng
Encapsulation: Ethernet

Time

First packet: 2021-10-06 10:37:36
Last packet: 2021-10-06 10:39:19
Elapsed: 00:01:42

Capture

Hardware: Intel(R) Core(TM) i5-8300H CPU @ 2.30GHz (with SSE4.2)
OS: 64-bit Windows 10 (2009), build 19042
Application: Dumpcap (Wireshark) 3.4.8 (v3.4.8-0-g3e1ffae201b8)

Interfaces

Interface	Dropped packets	Capture filter	Link type	Packet size limit
Wi-Fi	Unknown	none	Ethernet	262144 bytes

Statistics

Measurement	Captured	Displayed	Marked
Packets	24141	24141 (100.0%)	—
Time span, s	102.605	102.605	—
Average pps	235.3	235.3	—
Average packet size, B	975	975	—
Bytes	23547792	23547792 (100.0%)	0
Average bytes/s	229k	229k	—
Average bits/s	1836k	1836k	—

I captured the file for 1 min 42 seconds = 102 seconds and downloaded about 24 MB of the file.

Wireshark · Conversations · ubuntu image.pcapng

Ethernet · 3IPv4 · 45IPv6TCP · 35UDP · 34

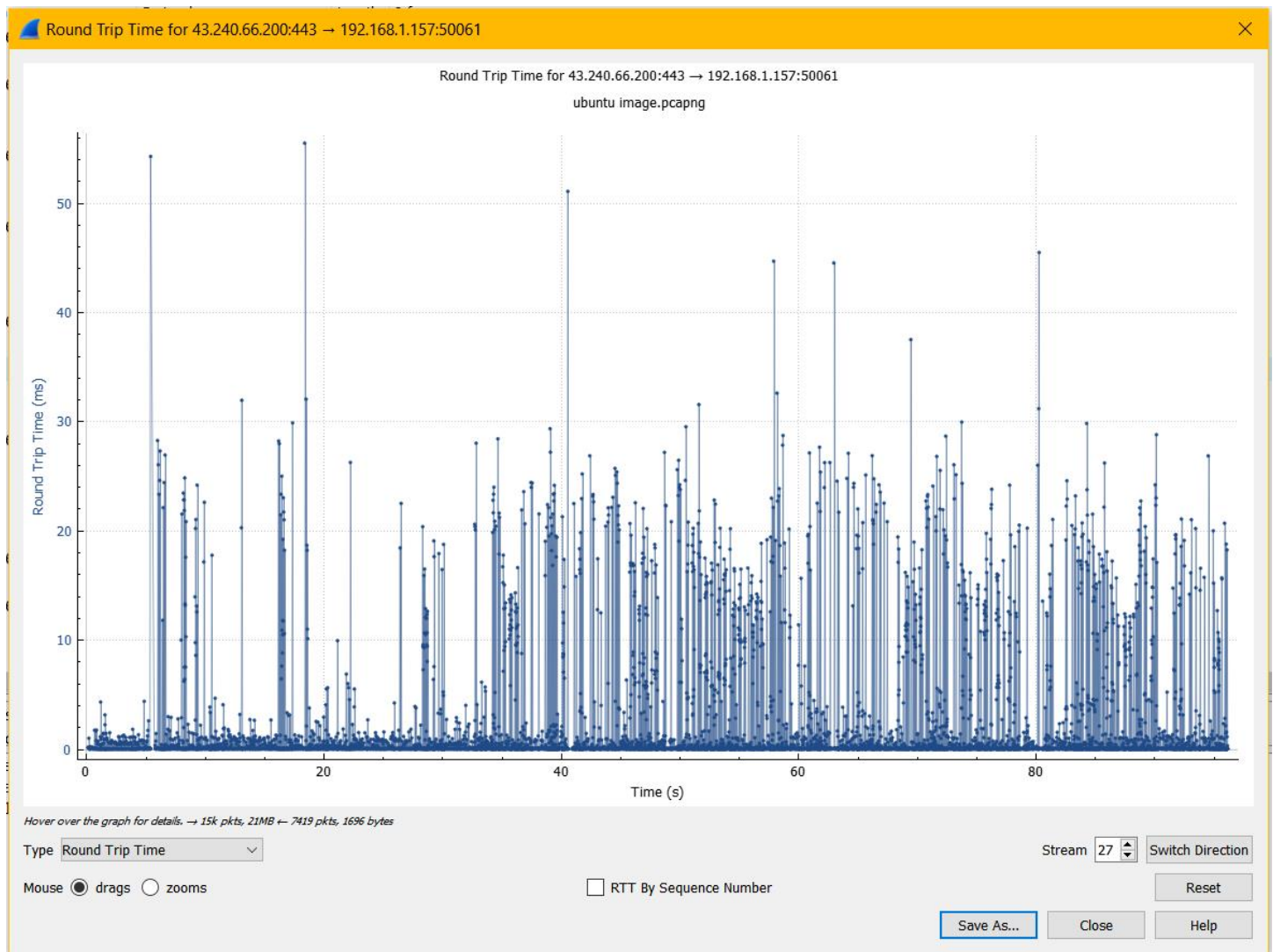
Address A	Port A	Address B	Port B	Packets	Bytes	Packets A → B	Bytes A → B	Packets B → A	Bytes B → A	Rel Start	Duration	Bits/s A → B	Bits/s B → A
192.168.1.157	50061	43.240.66.200	443	22,573	23M	7,419	411k	15,154	22M	6.451179	96.1534	34k	1883k
192.168.1.157	57553	185.125.190.21	443	51	28k	22	3342	29	25k	0.970341	52.4184	510	3902
192.168.1.157	54873	157.240.16.52	443	52	22k	20	1362	32	21k	1.779657	95.2814	114	1789
192.168.1.157	50062	13.76.98.223	443	22	12k	11	3405	11	9493	6.481464	0.3981	68k	190k
192.168.1.157	50063	13.76.98.223	443	22	12k	11	3244	11	9420	6.726554	0.4298	60k	175k
192.168.1.157	59126	18.141.80.142	443	24	9079	13	2793	11	6286	3.203047	14.8360	1506	3389
192.168.1.157	61154	52.74.233.13	443	15	8119	10	4721	5	3398	3.054108	90.0353	419	301
192.168.1.157	55949	54.150.10.110	443	27	7878	14	2215	13	5663	3.225642	60.5808	292	747

In the conversations option of Statistics, we see that about 23 MB of data is coming from the address of 43.240.66.200, thus the IP address of the ubuntu page to download the image file is 43.240.66.200

Since our IP address is 43.240.66.200 and our local IP address is 192.168.1.157, we will plot the RTT for this TCP connection.

As we need to plot the estimated Round-Trip Time (RTT) variation for download, so the direction should be server to the client which means 43.240.66.200 to 192.168.1.157

In Statistics > TCP Stream Graphs > Round Trip Time for the desired IP address, the graph is as follows



This file is also saved as RTT.pdf and shows the RTT for 102 seconds (duration for which the packets are captured).

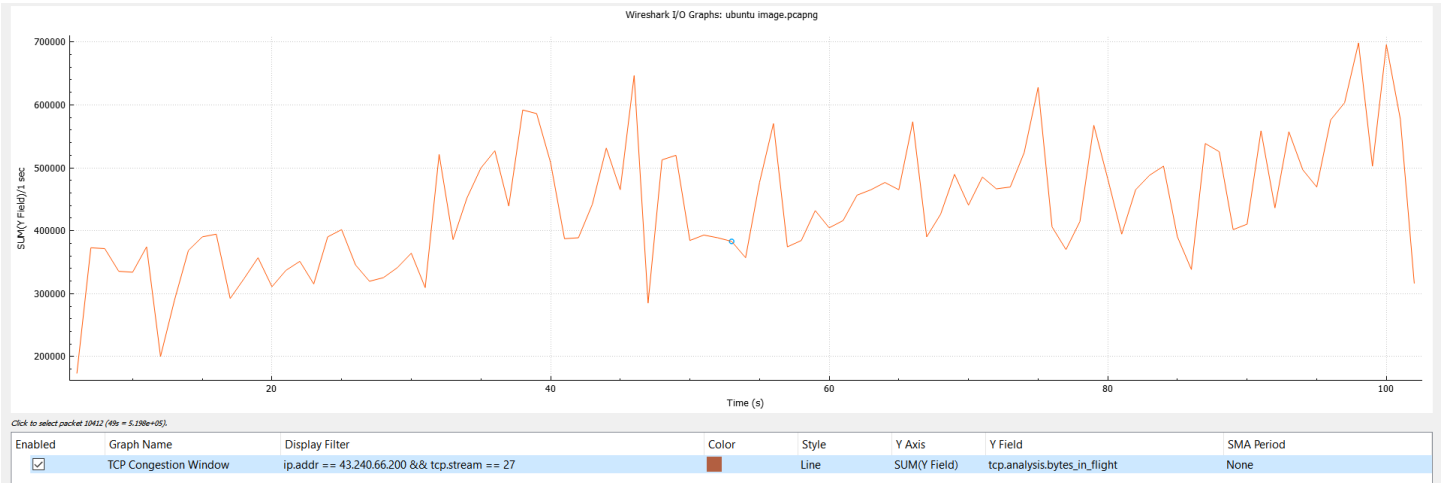
We need to plot the TCP Congestion Window. Due to the transfer of file from the server (Ubuntu) to the local machine (client) (we are downloading the file), the Client is the Receiver Window (rwnd) and the Server is the Congestion Window (cwnd).

In Statistics > I/O Graphs, we can plot the TCP congestion window using appropriate filters

The IP address is the same and the packet transmission is done from Stream number 27 (verified from the packet). This can be verified from the below Window Scaling images which link the IP address and TCP stream. The congestion window can only be calculated for a stream and not for a particular IP address as there can be multiple TCP streams to transfer the same file. (Different packets coming from different streams)

As mentioned in the question, we need to sum the bytes in each RTT (sum the tcp.analysis.bytes_in_flight), the same thing is done in the advanced field of Y-Axis and corresponding field.

The Y-axis is the sum of bytes_in_flight which essentially counts the bytes delivered from server to client (Download is happening for the file, so bytes are transferred from the server (src) to client (dst)). The X-axis is the time with which the congestion window varies.



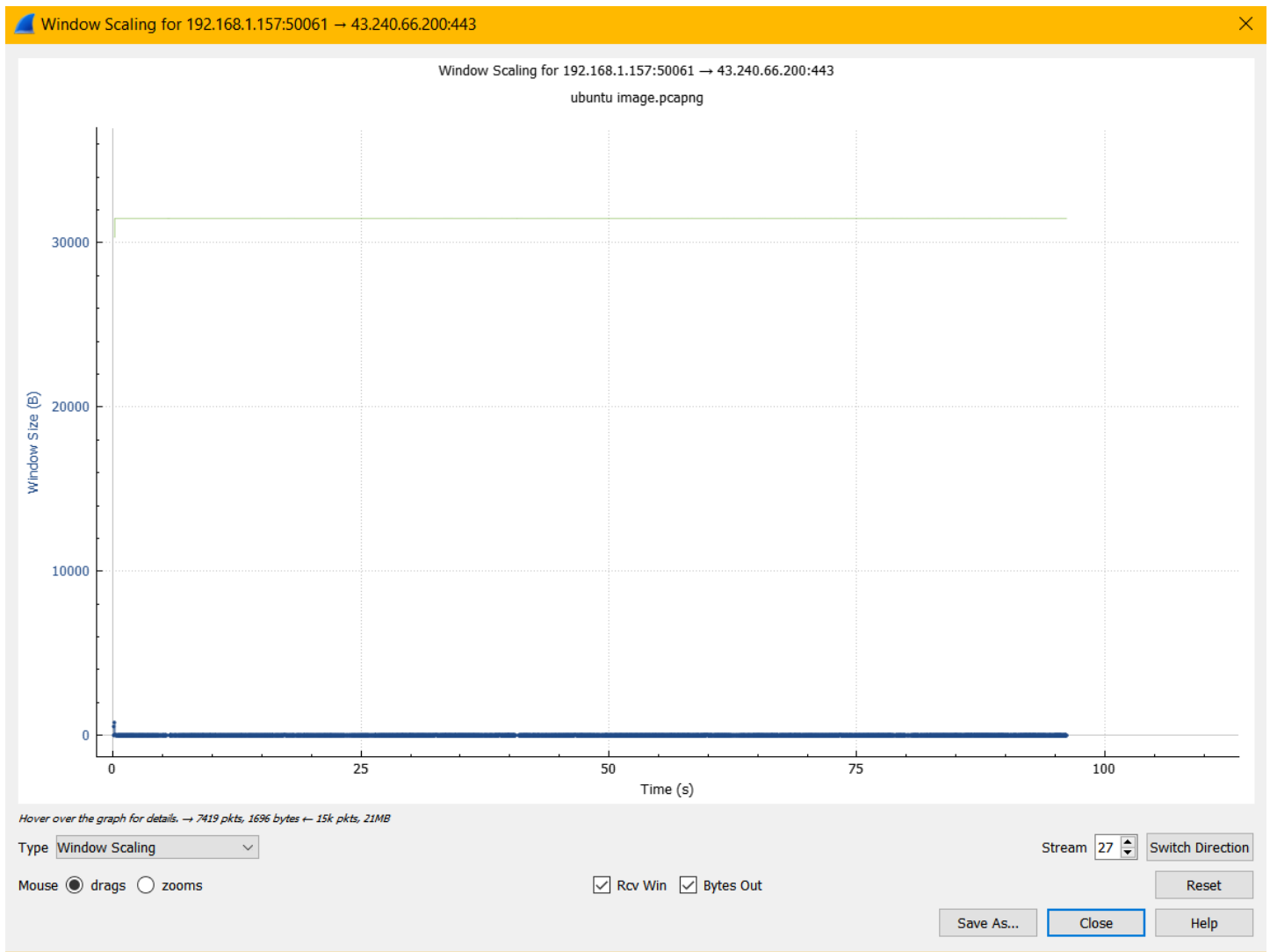
The above file is saved as tcp cwnd.pdf

In the above graph, the value of the sum of bytes delivered is taken, the next few sets of graphs will take into account the actual window value and scalable size to give the window size across which the bytes are delivered.

Another way to show the Congestion window is by showing the Windows Scale factor of the TCP window at the server end. This can be taken as an enhanced result to the first graph shown above.

In Statistics > TCP Stream Graphs > Window Scaling, we get a graph. Since we need the TCP congestion window for download, so we need to print the window at the Server.

The graph shows us the receiver window which is for the client (Client is the receiver in download), but we need the TCP Congestion Window at Server, thus we switch the direction of the graph. The output is (client to server)



Here the Rcv Win i.e., Receiver Window is marked by the Green Line. As the graph is switched, it gives the window size at the server which is needed.

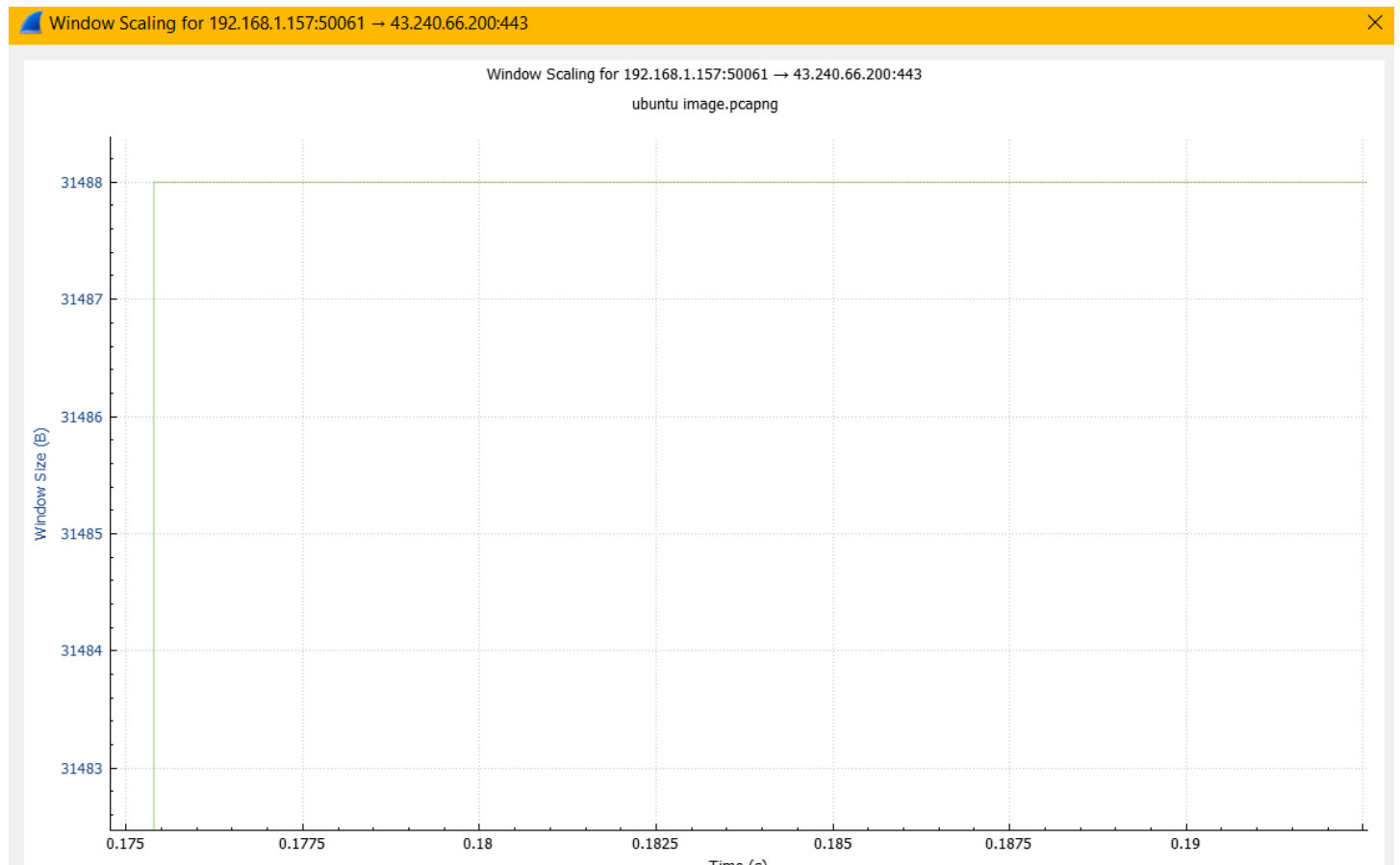
On zooming, the value for this window is fixed at 31,488 bytes. In a very short period (in about 0.176 s), the congestion window goes up from 30,336 bytes to 31,488 bytes. There is a standard scaling factor of 256 applied on the window to give the size shown in the above graph.

Window Size = Window * Scaling Factor (generally 256)

The actual window is $31,488 / 256 = 123$ which is the size of a TCP Congestion Window. We are plotting the Window size; however, it is just a scaled-up value, however, the behaviour of the scaled value is the same as the Window value.

Since this much is enough for the server to start delivering packets and maintain a continuous throughput, it remains constant.

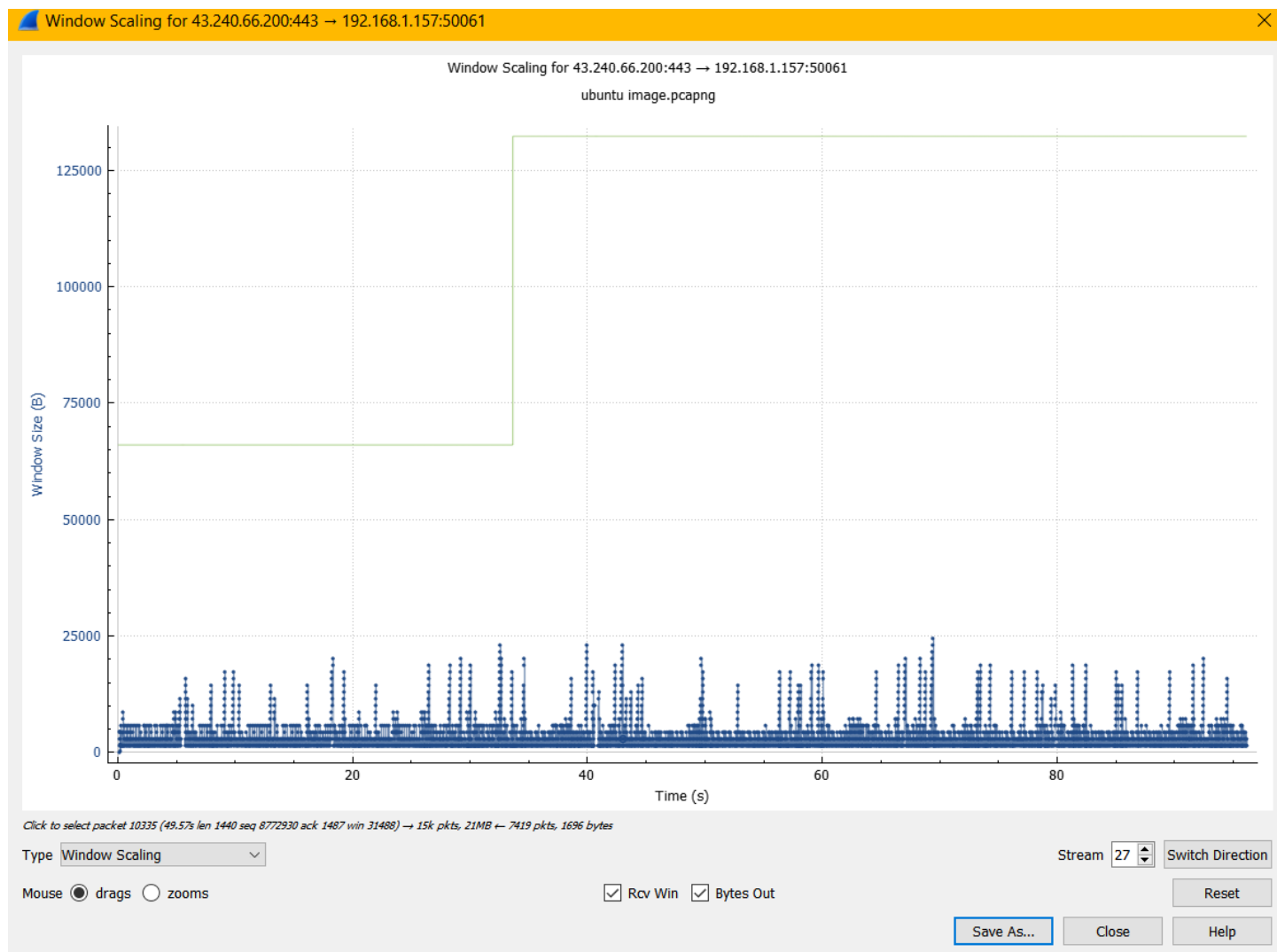
Another aspect is that the difference in the ack numbers and bytes delivered needs to be considered, which is considered in this case as the ack number is above and bytes out is below, so logically the difference is considered.



The above file is saved as client_to_server.pdf

The TCP congestion window size for download is 31,488 bytes which varies initially as the above plots suggest.

If you look at the other side (Server to client):



In both directions of the graph, we can see that a statement is written below the graph, which states that the server has sent 15k packets of size 21 MB and the client has sent 7419 packets of size 1696 bytes in total.

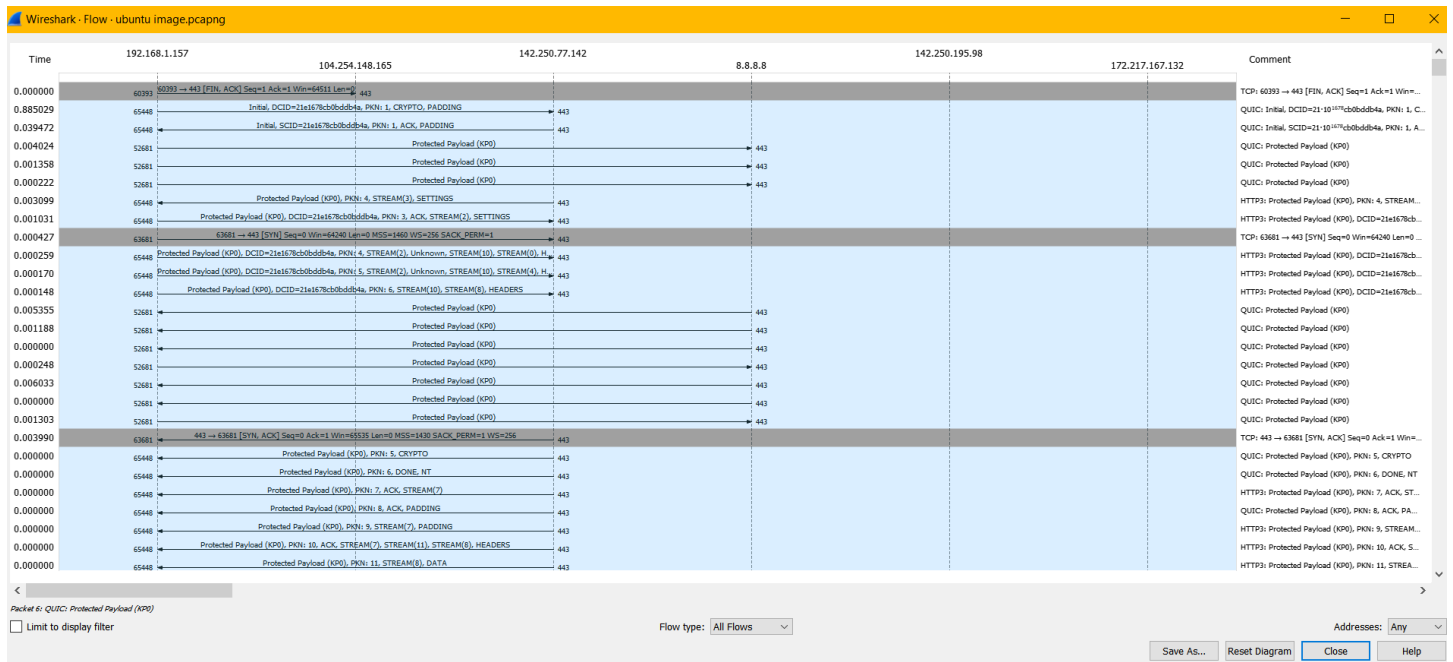
The Bytes out as shown by the blue skyline in the graph verify that the sum of bytes delivered over each RTT both by the server and client is verified in both directions of the graph.

The line at the bottom verifies that each packet contributes to the sum and its validity is expressed by taking the cumulative sum of all the packets.

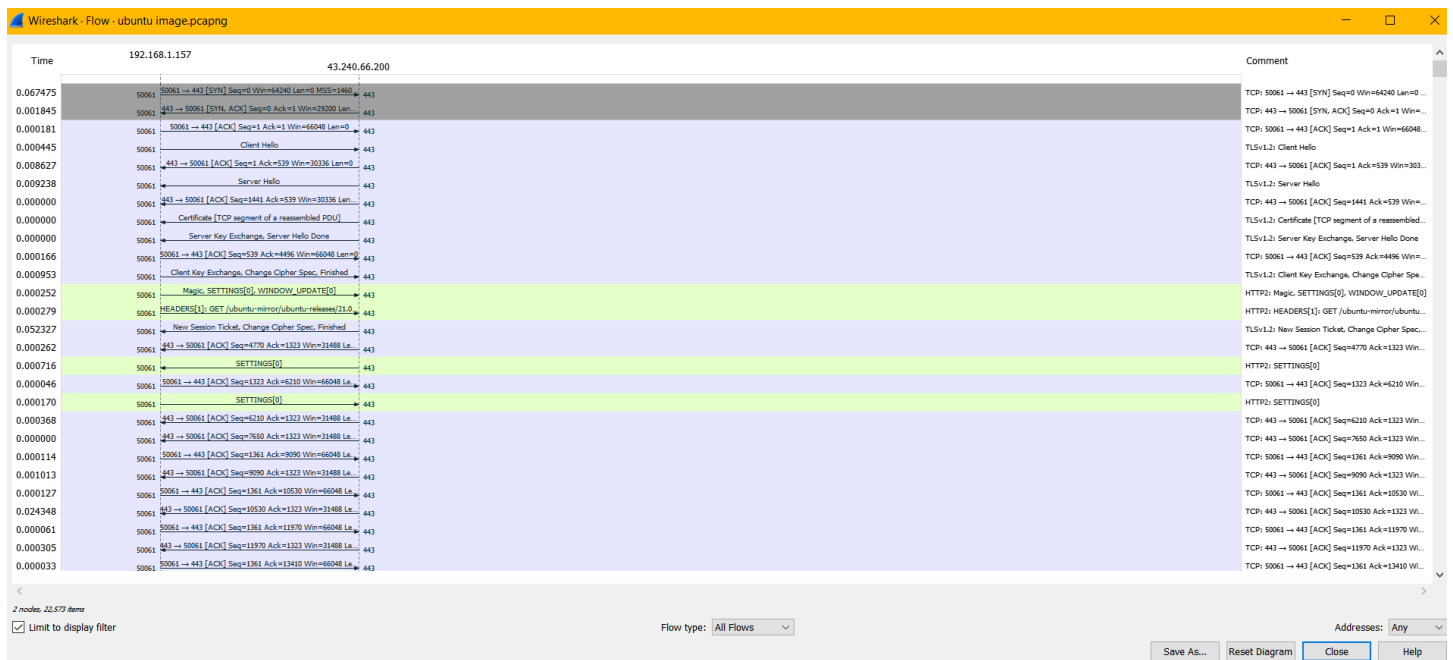
Adding on, this graph shows the client as the receiver and the Rcv Win shown by the Green Line indicates that the window size increases abruptly in between to become twice.

The above file is saved as server_to_client.pdf

The Flow Graph is



Since we want only the end-points of the connection, we will select the Limit to Display filter (Dialog Box – Bottom Left), we get

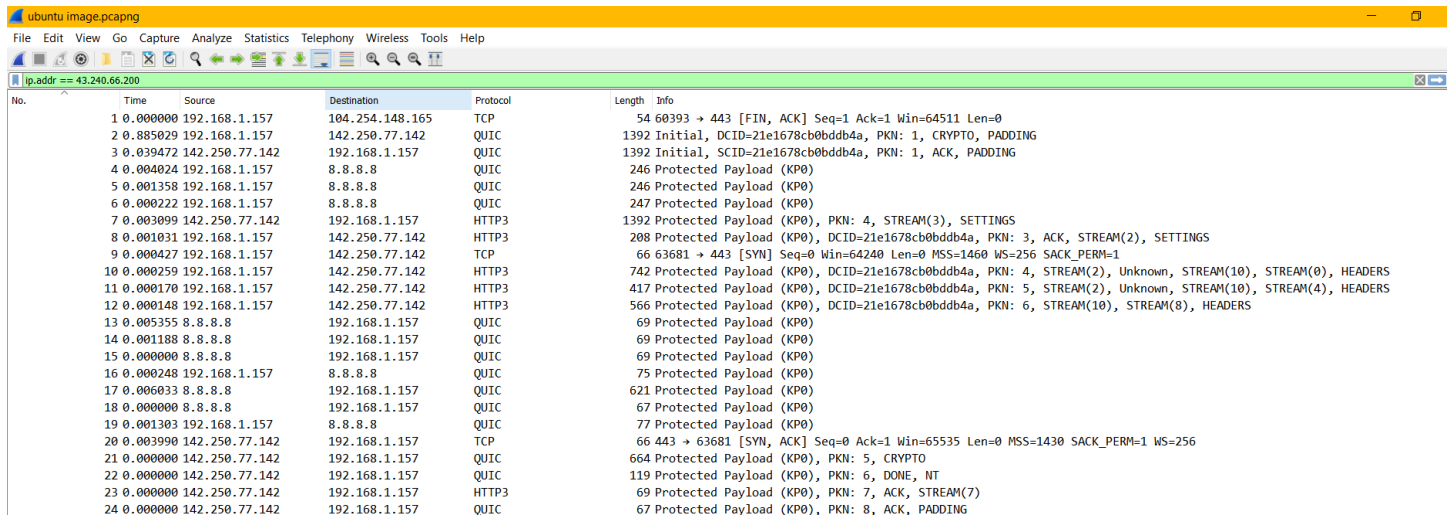


This file is also saved as flow graph.pdf

We need the average throughput of the packets from the IP address 43.240.66.200 as we need to show the throughput for the downloaded file only, not for all the packets.

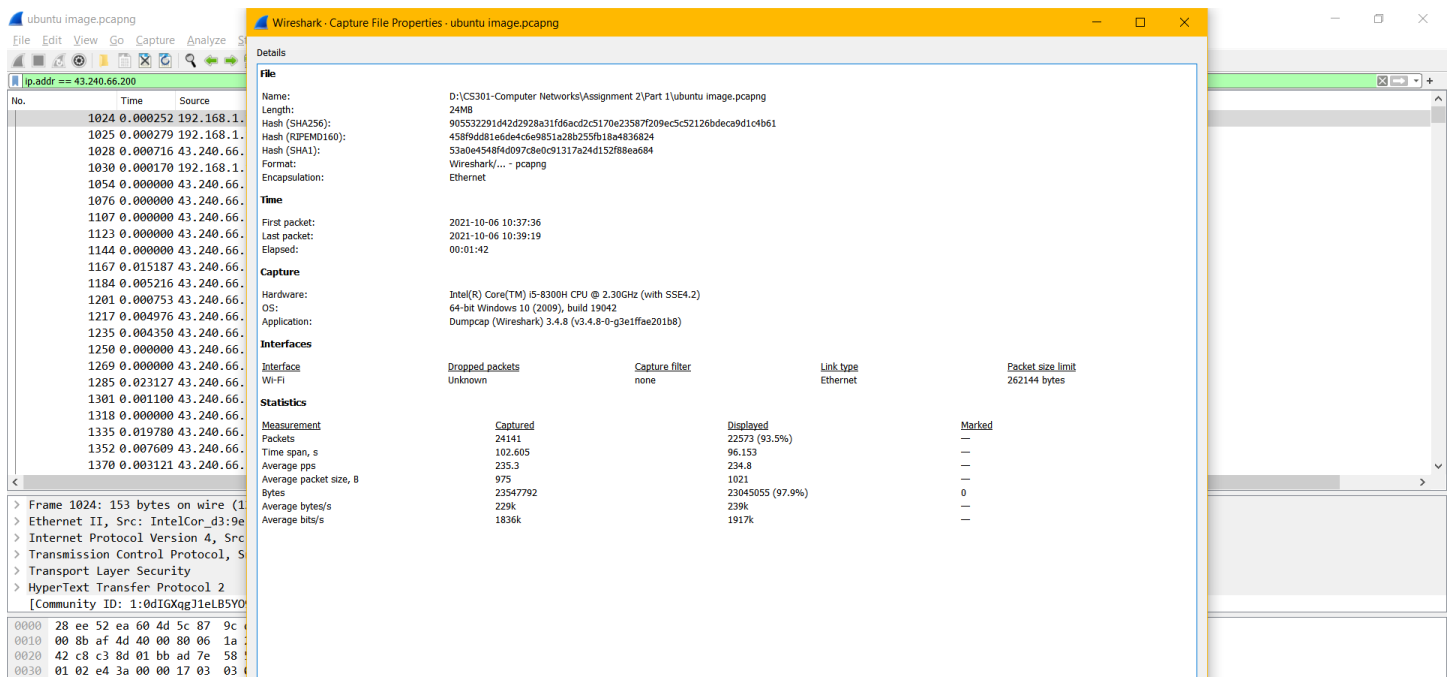
For this, we apply the query: `ip.addr == 43.240.66.200` to display the required packet only

We get the following sequence of packets,



No.	Time	Source	Destination	Protocol	Length	Info
1	0.000000	192.168.1.157	104.254.148.165	TCP	54	60393 → 443 [FIN, ACK] Seq=1 Ack=1 Win=64511 Len=0
2	0.885029	192.168.1.157	142.250.77.142	QUIC	1392	Initial, DCID=21e1678cb0b4db4a, PKN: 1, CRYPTO, PADDING
3	0.039472	142.250.77.142	192.168.1.157	QUIC	1392	Initial, SCID=21e1678cb0b4db4a, PKN: 1, ACK, PADDING
4	0.004024	192.168.1.157	8.8.8.8	QUIC	246	Protected Payload (KP0)
5	0.001358	192.168.1.157	8.8.8.8	QUIC	246	Protected Payload (KP0)
6	0.000222	192.168.1.157	8.8.8.8	QUIC	247	Protected Payload (KP0)
7	0.003099	142.250.77.142	192.168.1.157	HTTP3	1392	Protected Payload (KP0), PKN: 4, STREAM(3), SETTINGS
8	0.001031	192.168.1.157	142.250.77.142	HTTP3	208	Protected Payload (KP0), DCID=21e1678cb0b4db4a, PKN: 3, ACK, STREAM(2), SETTINGS
9	0.000427	192.168.1.157	142.250.77.142	TCP	66	63681 → 443 [SYN] Seq=0 Win=64240 Len=0 MSS=1460 WS=256 SACK_PERM=1
10	0.000259	192.168.1.157	142.250.77.142	HTTP3	742	Protected Payload (KP0), DCID=21e1678cb0b4db4a, PKN: 4, STREAM(2), Unknown, STREAM(10), STREAM(0), HEADERS
11	0.000170	192.168.1.157	142.250.77.142	HTTP3	417	Protected Payload (KP0), DCID=21e1678cb0b4db4a, PKN: 5, STREAM(2), Unknown, STREAM(10), STREAM(4), HEADERS
12	0.000148	192.168.1.157	142.250.77.142	HTTP3	566	Protected Payload (KP0), DCID=21e1678cb0b4db4a, PKN: 6, STREAM(10), STREAM(8), HEADERS
13	0.005355	8.8.8.8	192.168.1.157	QUIC	69	Protected Payload (KP0)
14	0.001188	8.8.8.8	192.168.1.157	QUIC	69	Protected Payload (KP0)
15	0.000000	8.8.8.8	192.168.1.157	QUIC	69	Protected Payload (KP0)
16	0.000248	192.168.1.157	8.8.8.8	QUIC	75	Protected Payload (KP0)
17	0.006033	8.8.8.8	192.168.1.157	QUIC	621	Protected Payload (KP0)
18	0.000000	8.8.8.8	192.168.1.157	QUIC	67	Protected Payload (KP0)
19	0.001303	192.168.1.157	8.8.8.8	QUIC	77	Protected Payload (KP0)
20	0.003990	142.250.77.142	192.168.1.157	TCP	66	443 → 63681 [SYN, ACK] Seq=0 Ack=1 Win=65535 Len=0 MSS=1430 SACK_PERM=1 WS=256
21	0.000000	142.250.77.142	192.168.1.157	QUIC	664	Protected Payload (KP0), PKN: 5, CRYPTO
22	0.000000	142.250.77.142	192.168.1.157	QUIC	119	Protected Payload (KP0), PKN: 6, DONE, NT
23	0.000000	142.250.77.142	192.168.1.157	HTTP3	69	Protected Payload (KP0), PKN: 7, ACK, STREAM(7)
24	0.000000	142.250.77.142	192.168.1.157	QUIC	67	Protected Payload (KP0), PKN: 8, ACK, PADDING

In Statistics > Capture File Properties, we get



File				
Name:	D:\CS301-Computer Networks\Assignment 2\Part 1\ubuntu image.pcapng			
Length:	24MB			
Hash (SHA256):	905532291d42d2928a31f0eacd2c5170e23587209ec5c52126bdce9d1e4b61			
Hash (RIPEMD160):	458f9dd81e6de4c6e9851a28b255fb18a4836824			
Hash (SHA1):	53a0e4548f4d097c8e0c91317a24d152f88ee684			
Format:	Wireshark/... - pcapng			
Encapsulation:	Ethernet			
Time				
First packet:	2021-10-06 10:37:36			
Last packet:	2021-10-06 10:39:19			
Elapsed:	00:01:42			
Capture				
Hardware:	Intel(R) Core(TM) i5-8300H CPU @ 2.30GHz (with SSE4.2)			
OS:	64-bit Windows 10 (20H2), build 19042			
Application:	Dumpcap (Wireshark) 3.4.8 (v3.4.8-0-g3e1f0e201b8)			
Interfaces				
Interface	Dropped packets	Capture filter	Link type	Packet size limit
Wi-Fi	Unknown	none	Ethernet	262144 bytes
Statistics				
Measurement	Captured	Displayed	Marked	
Packets	24141	22573 (93.5%)	---	
Time span, s	102.605	96.153	---	
Average pps	235.3	234.8	---	
Average packet size, B	975	1021	---	
Bytes	23547792	23045055 (97.9%)	0	
Average bytes/s	229k	239k	---	
Average bits/s	1836k	1917k	---	

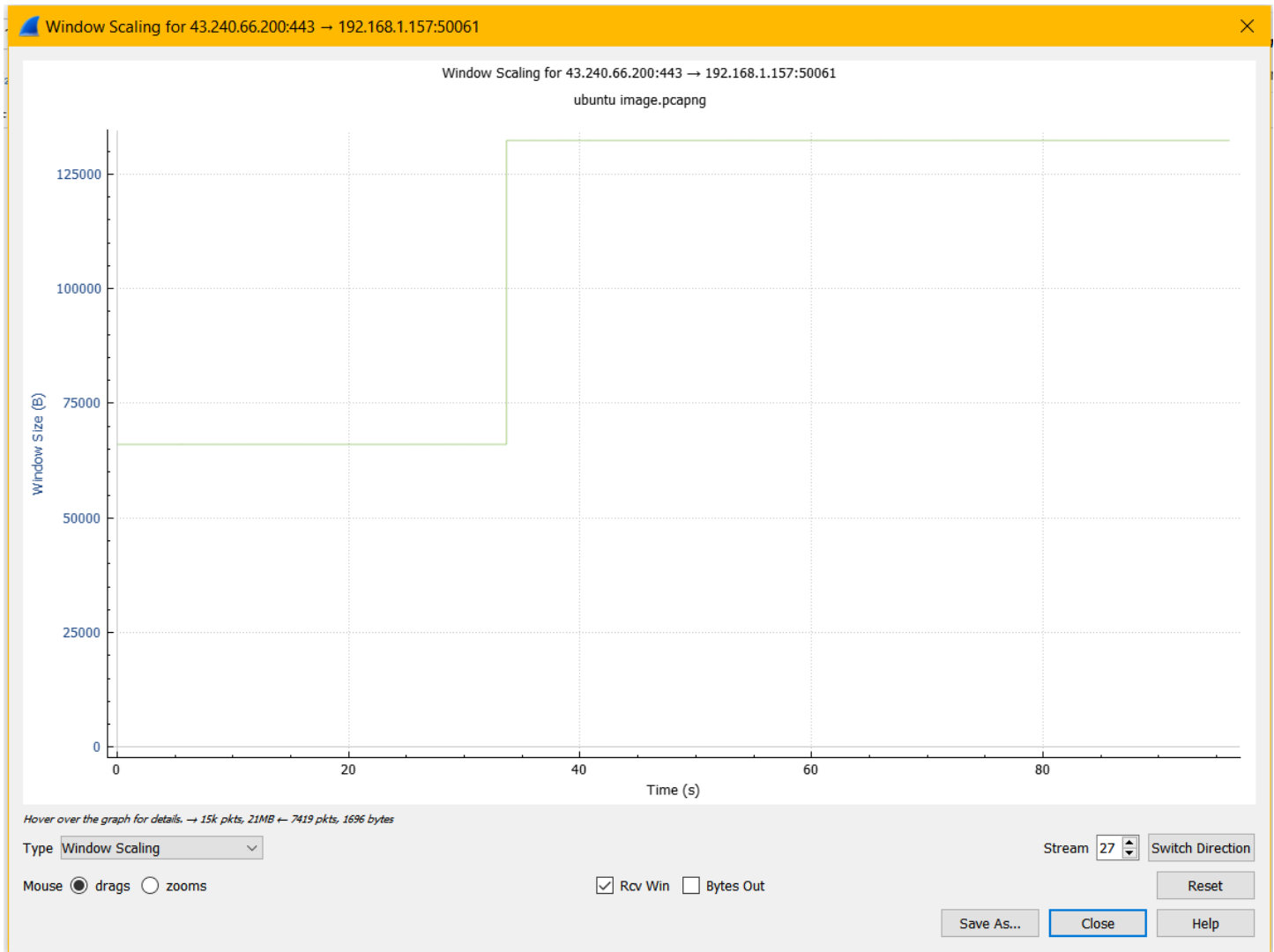
Since the displayed shows all the needed packets (93.5% packets belong to the file transfer), the throughput for the displayed packets is our answer.

The average throughput is the average bytes/s which for the displayed packets is 239k = 239,000 bytes/s.

Thus, the average throughput is 239k bytes/s.

In Statistics > TCP Stream Graphs > Window Scaling, we get the needed graph

The receiver is the client who is downloading the file. Thus, the Rcv Win (Green Line) in the following graph is the variation of the receiver congestion window size with time. Client is 192.168.1.157



This graph shows the client as the receiver and the Rcv Win shown by the Green Line indicates that the window size increases abruptly in between to become twice. The window scale factor is 256 throughout the packet transmission.

The initial window size is 66,048 bytes after which it jumps to 132,352 bytes. This variation occurs because the server sees that it needs to send more data to the user and hence increases the window size to transmit more packets in the same interval of time.

The above file is saved as receive_window.pdf

Initial Window Size

1019 0.000000 43.240.66.200	192.168.1.157	TLSv1.2	1494 Certificate [TCP segment of a reassembled PDU]
1020 0.000000 43.240.66.200	192.168.1.157	TLSv1.2	229 Server Key Exchange, Server Hello Done
1022 0.000166 192.168.1.157	43.240.66.200	TCP	54 50061 → 443 [ACK] Seq=539 Ack=4496 Win=66048 Len=0
1023 0.000953 192.168.1.157	43.240.66.200	TLSv1.2	180 Client Key Exchange, Change Cipher Spec, Finished
1024 0.000252 192.168.1.157	43.240.66.200	HTTP2	153 Magic, SETTINGS[0], WINDOW_UPDATE[0]
1025 0.000279 192.168.1.157	43.240.66.200	HTTP2	613 HEADERS[1]: GET /ubuntu-mirror/ubuntu-releases/21.04/ubuntu-21.04-desktop-amd64.iso
1026 0.052327 43.240.66.200	192.168.1.157	TLSv1.2	328 New Session Ticket, Change Cipher Spec, Finished
1027 0.000262 43.240.66.200	192.168.1.157	TCP	54 443 → 50061 [ACK] Seq=4770 Ack=1323 Win=31488 Len=0
1028 0.000716 43.240.66.200	192.168.1.157	HTTP2	1494 SETTINGS[0]
1029 0.000046 192.168.1.157	43.240.66.200	TCP	54 50061 → 443 [ACK] Seq=1323 Ack=6210 Win=66048 Len=0
1030 0.000170 192.168.1.157	43.240.66.200	HTTP2	92 SETTINGS[0]

[TCP Segment Len: 0]
Sequence Number: 539 (relative sequence number)
Sequence Number (raw): 2910738397
[Next Sequence Number: 539 (relative sequence number)]
Acknowledgment Number: 4496 (relative ack number)
Acknowledgment number (raw): 3453734923
0101 = Header Length: 20 bytes (5)
> Flags: 0x010 (ACK)
Window: 258
[Calculated window size: 66048]
[Window size scaling factor: 256]
Checksum: 0x1549 [unverified]
[Checksum Status: Unverified]
Urgent Pointer: 0
> [SEQ/ACK analysis]

The value shown is 66,048 after scaling by 256, so the window = 258 as shown in the image.

Final Window Size

ubuntu image.pcapng					
File Edit View Go Capture Analyze Statistics Telephony Wireless Tools Help					
ip.addr == 43.240.66.200					
No.	Time	Source	Destination	Protocol	Length Info
24131	0.000055	192.168.1.157	43.240.66.200	TCP	54 50061 → 443 [ACK] Seq=1697 Ack=21789090 Win=132352 Len=0
24132	0.014375	43.240.66.200	192.168.1.157	TCP	1494 443 → 50061 [ACK] Seq=21789090 Ack=1697 Win=31488 Len=1440[Reassembly error, protocol TCP: New fragment over]
24133	0.000278	43.240.66.200	192.168.1.157	TCP	1494 443 → 50061 [ACK] Seq=21790530 Ack=1697 Win=31488 Len=1440[Reassembly error, protocol TCP: New fragment over]
24134	0.000056	192.168.1.157	43.240.66.200	TCP	54 50061 → 443 [ACK] Seq=1697 Ack=21791970 Win=132352 Len=0
24135	0.001896	43.240.66.200	192.168.1.157	TCP	1494 443 → 50061 [ACK] Seq=21791970 Ack=1697 Win=31488 Len=1440[Reassembly error, protocol TCP: New fragment over]
24136	0.000000	43.240.66.200	192.168.1.157	TCP	1494 443 → 50061 [ACK] Seq=21793410 Ack=1697 Win=31488 Len=1440[Reassembly error, protocol TCP: New fragment over]
24137	0.000058	192.168.1.157	43.240.66.200	TCP	54 50061 → 443 [ACK] Seq=1697 Ack=21794850 Win=132352 Len=0
24138	0.017647	43.240.66.200	192.168.1.157	TCP	1494 443 → 50061 [ACK] Seq=21794850 Ack=1697 Win=31488 Len=1440[Reassembly error, protocol TCP: New fragment over]
24139	0.000000	43.240.66.200	192.168.1.157	TCP	1494 443 → 50061 [ACK] Seq=21796290 Ack=1697 Win=31488 Len=1440[Reassembly error, protocol TCP: New fragment over]
24140	0.000063	192.168.1.157	43.240.66.200	TCP	54 50061 → 443 [ACK] Seq=1697 Ack=21797730 Win=132352 Len=0
24141	0.005009	43.240.66.200	192.168.1.157	TCP	1494 443 → 50061 [ACK] Seq=21797730 Ack=1697 Win=31488 Len=1440[Reassembly error, protocol TCP: New fragment over]

[TCP Segment Len: 0]
Sequence Number: 1697 (relative sequence number)
Sequence Number (raw): 2910739555
[Next Sequence Number: 1697 (relative sequence number)]
Acknowledgment Number: 21794850 (relative ack number)
Acknowledgment number (raw): 3475525277
0101 = Header Length: 20 bytes (5)
> Flags: 0x010 (ACK)
Window: 517
[Calculated window size: 132352]
[Window size scaling factor: 256]
Checksum: 0x8fe1 [unverified]
[Checksum Status: Unverified]
Urgent Pointer: 0
> [SEQ/ACK analysis]
> [Timestamps]
[Community ID: 1:0dIGXqJ1eLB5Y09natMnt+JQW0=]

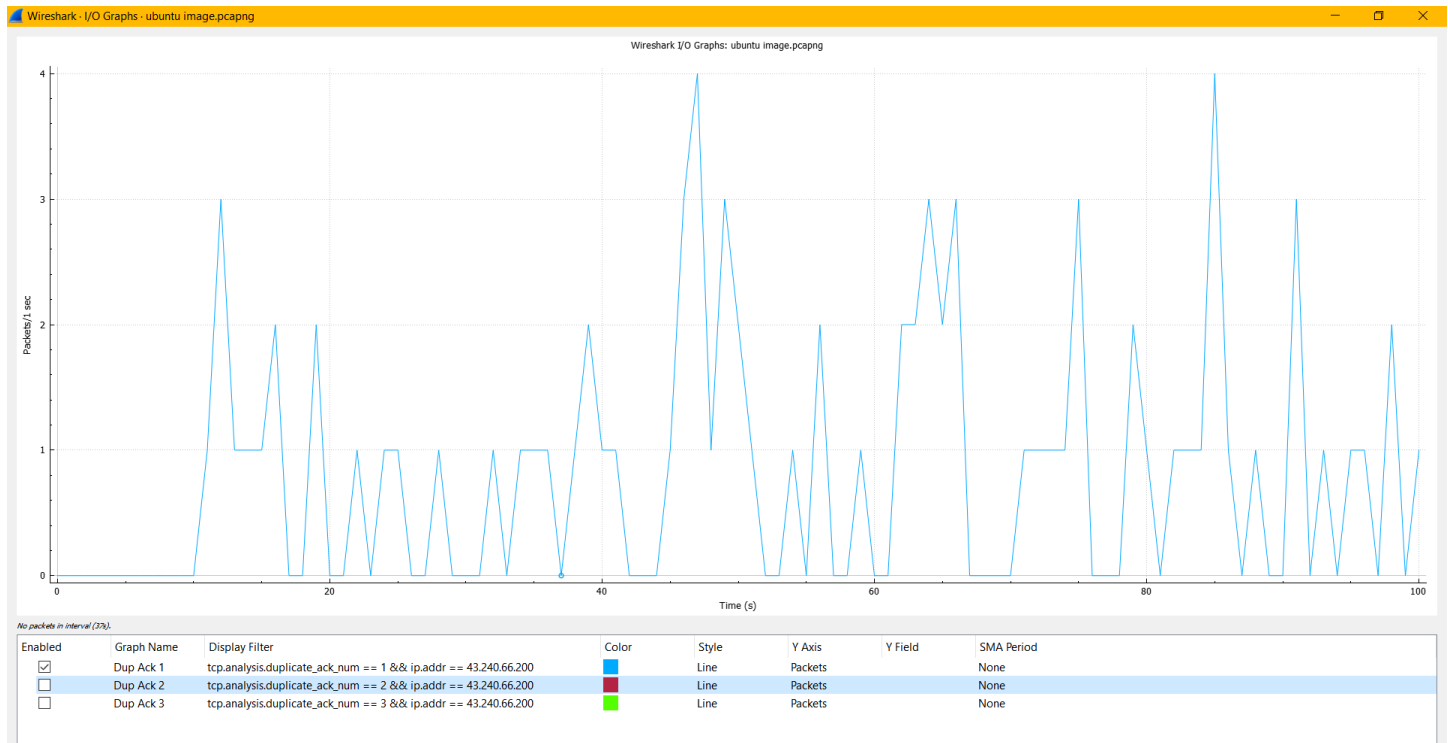
0000	28 ee 52 ea 60 4d 5c 87	9c d3 9e a1 08 00 45 00	(.R..M\.....E:
0010	00 28 cc 41 40 00 80 06	fd 90 c0 a8 01 9d 2b f0	..(A@.....+.
0020	42 c8 c3 8d 01 bb ad 7e	5c 63 cf 28 4e 9d 50 10	B.....~\c(N-P

The value shown is 132,352 after scaling by 256, so the window = 517 as shown in the image.

This will always show the value at the source. Here, the receiver is the client (as we are downloading), thus the receiver congestion window size graph is plotted. (The behaviour will be the same as the Window value of 258 and 517, just with a different scale). IP Address of the client is 192.168.1.157

We will use Wireshark I/O Graphs to plot the number of duplicate acknowledgement packets. As we deal with the download of the file, the IP address is 43.240.66.200

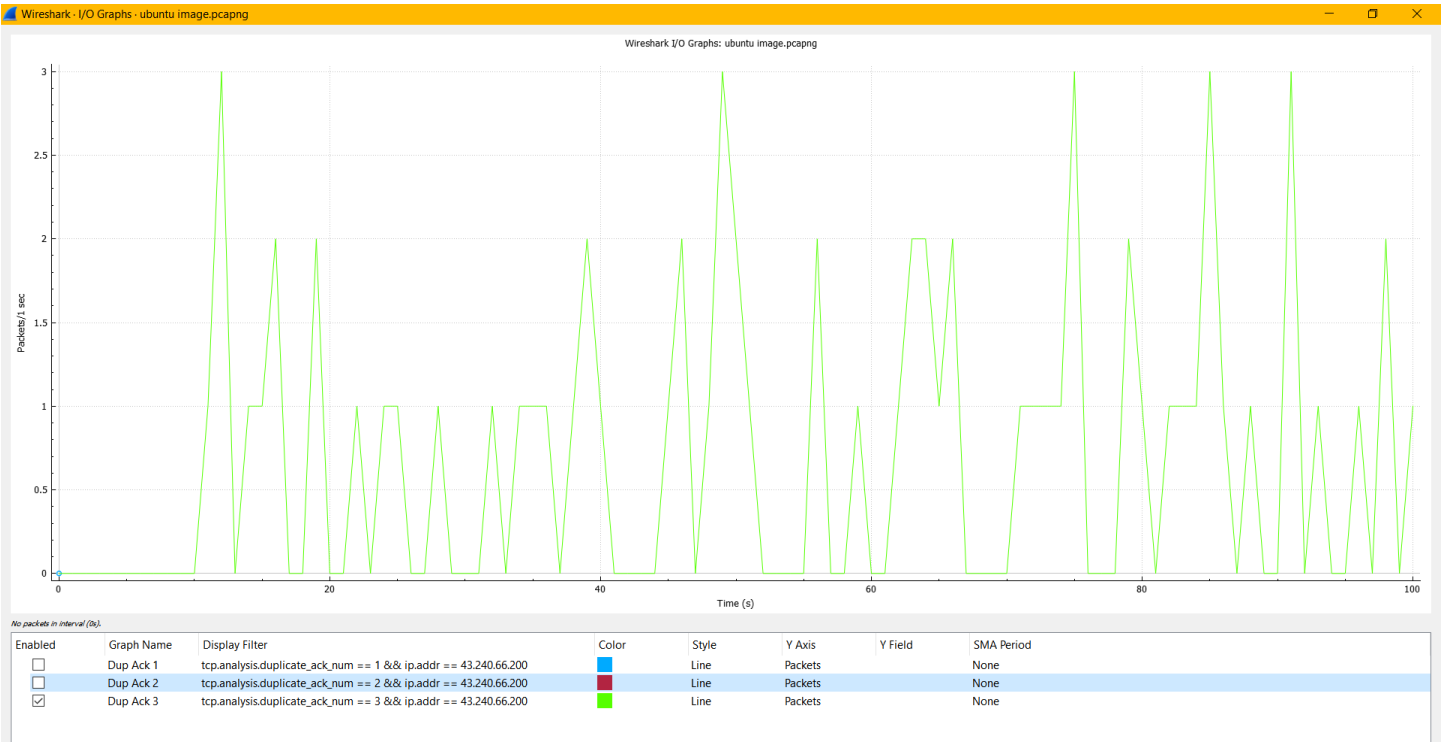
1 – Duplicate Ack – Query Applied: tcp.analysis.duplicate_ack_num == 1 && ip.addr == is 43.240.66.200



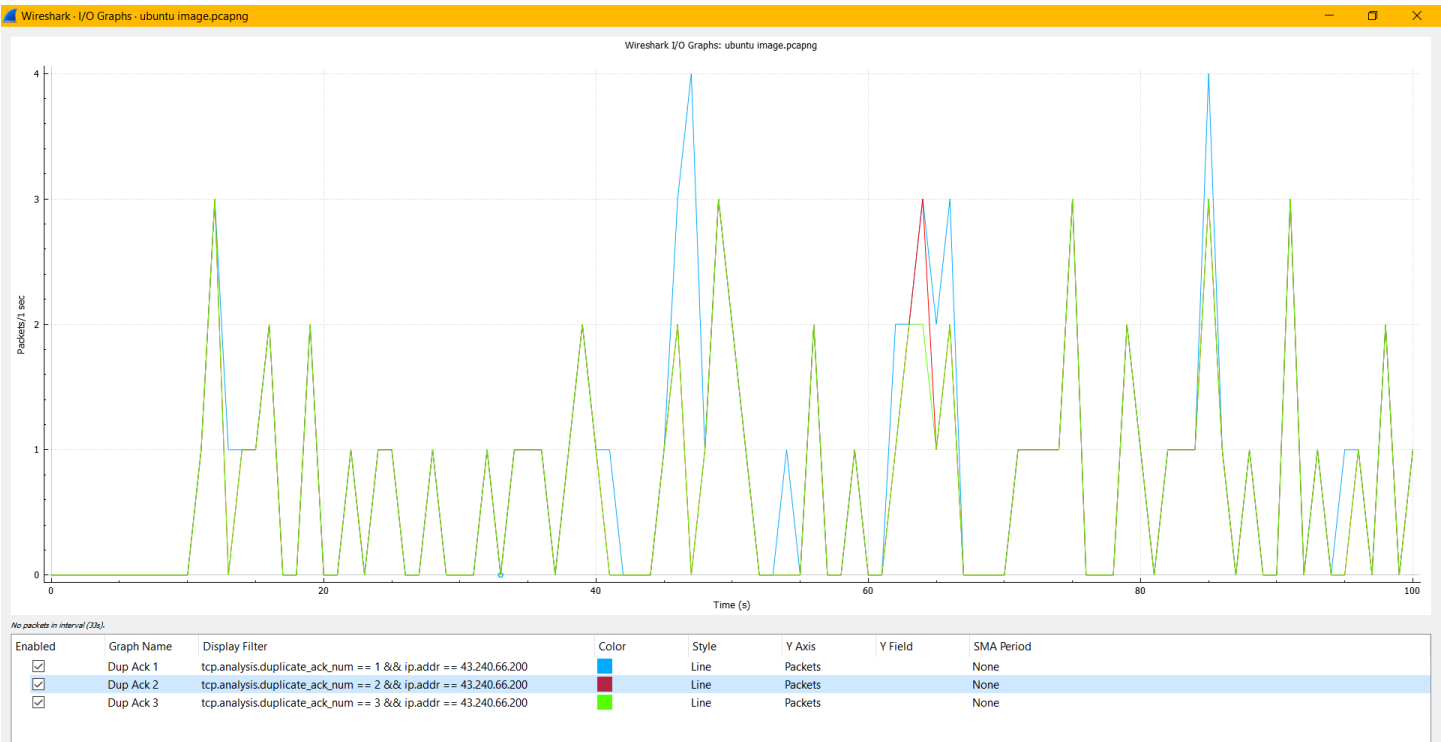
2 – Duplicate Ack – Query Applied: tcp.analysis.duplicate_ack_num == 2 && ip.addr == is 43.240.66.200



3 – Duplicate Ack – Query Applied: tcp.analysis.duplicate_ack_num == 3 && ip.addr == is 43.240.66.200



Combined



The above file is saved as duplicate ack.pdf

Question 2

The Wireshark Capture file for the 3-way handshake: 3-Way TCP Handshake.pcapng

The following file was downloaded to identify the 3 Way Handshake

<https://www.nirfindia.org/nirfpdfcdn/2021/framework/Research.pdf>

```
mastershubham@LAPTOP-8Q15SHE6: /mnt/d
mastershubham@LAPTOP-8Q15SHE6:/mnt/d$ dig www.nirfindia.org

; <<>> DiG 9.11.3-1ubuntu1.7-Ubuntu <<>> www.nirfindia.org
;; global options: +cmd
;; Got answer:
;; ->>HEADER<<- opcode: QUERY, status: NOERROR, id: 1223
;; flags: qr rd ad; QUERY: 1, ANSWER: 1, AUTHORITY: 0, ADDITIONAL: 0
;; WARNING: recursion requested but not available

;; QUESTION SECTION:
;www.nirfindia.org.                IN      A

;; ANSWER SECTION:
www.nirfindia.org.                0       IN      A      115.124.102.160

;; Query time: 120 msec
;; SERVER: 172.30.112.1#53(172.30.112.1)
;; WHEN: Tue Oct 05 17:57:37 IST 2021
;; MSG SIZE rcvd: 68

mastershubham@LAPTOP-8Q15SHE6:/mnt/d$
```

The IP address of the NIRF Website is 115.124.102.160.

Local IP address: 192.168.1.157

Since we have a TCP handshake, the following query is applied:

(tcp) && (ip.addr == 115.124.102.160)

which results in the following screenshot

No.	Time	Source	Destination	Protocol	Length	Info
16	0.180926	192.168.1.157	115.124.102.160	TCP	66	55330 → 443 [SYN] Seq=0 Win=64240 Len=0 MSS=1460 WS=256 SACK_PERM=1
17	0.000738	192.168.1.157	115.124.102.160	TCP	66	63777 → 443 [SYN] Seq=0 Win=64240 Len=0 MSS=1460 WS=256 SACK_PERM=1
18	0.023370	115.124.102.160	192.168.1.157	TCP	66	443 → 63777 [SYN, ACK] Seq=0 Ack=1 Win=29200 Len=0 MSS=1440 SACK_PERM=1 WS=128
19	0.000150	192.168.1.157	115.124.102.160	TCP	54	63777 → 443 [ACK] Seq=1 Ack=1 Win=132352 Len=0
20	0.000445	192.168.1.157	115.124.102.160	TLSv1.2		571 Client Hello
21	0.005236	115.124.102.160	192.168.1.157	TCP	66	443 → 55330 [SYN, ACK] Seq=0 Ack=1 Win=29200 Len=0 MSS=1440 SACK_PERM=1 WS=128
22	0.000098	192.168.1.157	115.124.102.160	TCP	54	55330 → 443 [ACK] Seq=1 Ack=1 Win=132352 Len=0
23	0.000437	192.168.1.157	115.124.102.160	TLSv1.2		571 Client Hello

In the above figure, 443 is the standard port number for HTTPS (Secure) connection from the server-side.

If you closely observe the packets 17, 18, and 19, they comprise of 3 Way TCP Handshake.

This handshake consists of 3 steps:

1. Step 1 (SYN) : (Packet Number 17)

- a) In the first step, the client wants to establish a connection with the server, so it sends a segment with SYN(Synchronize Sequence Number) which informs the server that the client is likely to start communication and with what sequence number it starts segments with.
- b) This happens with 63777 -> 443 SYN Request (Packet Number 17). As it is an HTTPS request (Specified in URL), the information is sent to port 443 on the server.

2. Step 2 (SYN + ACK): (Packet Number 18)

- a) The server responds to the client request with SYN-ACK signal bits set. Acknowledgement(ACK) signifies the response of segment it received and SYN signifies with what sequence number it is likely to start the segments with.
- b) This happens with 443 -> 63777 [SYN, ACK] Request (Packet Number 18). It is directed to the same port from where Packet 17 arrived.
- c) The bits for SYN = 0 and ACK = 1 is that connection is acknowledged and it is Synchronized with the Client.

3. Step 3 (ACK) : (Packet Number 19)

- a) In the final part client acknowledges the response of the server and they both establish a reliable connection with which they will start the actual data transfer.
- b) This happens with 63777 -> 443 ACK Request (Packet Number 19). This indicates acknowledgement of the server-side connection and reliability on both sides to begin data transfer.

Steps 1, 2 establish the connection parameter (sequence number) for one direction and it is acknowledged. Steps 2, 3 establish the connection parameter (sequence number) for the other direction and it is acknowledged. With these, full-duplex communication is established.

Note:

There exists a packet number 16 before the 3-way handshake. Initial sequence numbers are randomly selected while establishing connections between client and server. The client-side chose to establish a connection from port 55330 (chosen at random) to the server. However, this failed and the connection could not be established.

Next, another port 63777 was chosen (again, randomly) and as it was able to successfully establish the connection, so 3 packets were used in the TCP Handshake.

It might be possible that the initial connections fail due to unavailability, bandwidth, and a variety of reasons, however, on a successful port (63777 in my case), it will take a maximum of 3 packets for the connection.

Thus, we can conclude that packets 17, 18, 19 comprise the TCP 3-Way Handshake.

Question 3

The Wireshark Capture file for the 3-way handshake: ping host.pcapng

The IP address of the IIT Bhilai Web Page is 103.147.138.100

For our host, we will be using the IIT Bhilai Web Server.

The result of the Ping is: (The command used is: ping <IP Address> (On the CMD of Windows))

```
C:\WINDOWS\system32\cmd.exe

C:\Users\shubham gupta>ping 103.147.138.100

Pinging 103.147.138.100 with 32 bytes of data:
Reply from 103.147.138.100: bytes=32 time=74ms TTL=47
Reply from 103.147.138.100: bytes=32 time=75ms TTL=47
Reply from 103.147.138.100: bytes=32 time=95ms TTL=47
Reply from 103.147.138.100: bytes=32 time=74ms TTL=47

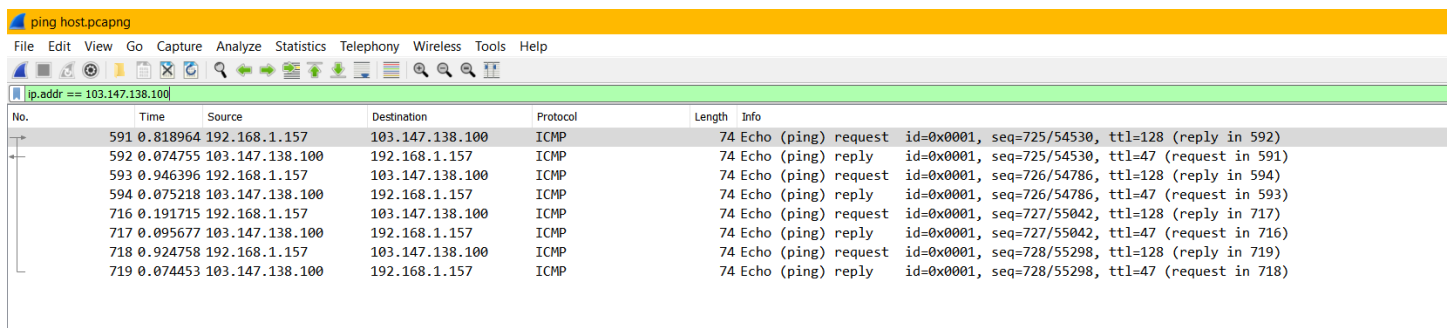
Ping statistics for 103.147.138.100:
    Packets: Sent = 4, Received = 4, Lost = 0 (0% loss),
Approximate round trip times in milli-seconds:
    Minimum = 74ms, Maximum = 95ms, Average = 79ms

C:\Users\shubham gupta>
```

Since we need to deal with the packets from the IIT Bhilai host, the following Query is applied:

ip.addr == 103.147.138.100

We get the following result:



No.	Time	Source	Destination	Protocol	Length	Info
591	0.818964	192.168.1.157	103.147.138.100	ICMP	74	Echo (ping) request id=0x0001, seq=725/54530, ttl=128 (reply in 592)
592	0.074755	103.147.138.100	192.168.1.157	ICMP	74	Echo (ping) reply id=0x0001, seq=725/54530, ttl=47 (request in 591)
593	0.946396	192.168.1.157	103.147.138.100	ICMP	74	Echo (ping) request id=0x0001, seq=726/54786, ttl=128 (reply in 594)
594	0.075218	103.147.138.100	192.168.1.157	ICMP	74	Echo (ping) reply id=0x0001, seq=726/54786, ttl=47 (request in 593)
716	0.191715	192.168.1.157	103.147.138.100	ICMP	74	Echo (ping) request id=0x0001, seq=727/55042, ttl=128 (reply in 717)
717	0.095677	103.147.138.100	192.168.1.157	ICMP	74	Echo (ping) reply id=0x0001, seq=727/55042, ttl=47 (request in 716)
718	0.924758	192.168.1.157	103.147.138.100	ICMP	74	Echo (ping) request id=0x0001, seq=728/55298, ttl=128 (reply in 719)
719	0.074453	103.147.138.100	192.168.1.157	ICMP	74	Echo (ping) reply id=0x0001, seq=728/55298, ttl=47 (request in 718)

As we can see in the result, the only packets generated by the ping command are ICMP echo requests and ICMP echo reply packets.

Ping is used for testing the reachability of hosts on the IP. It solely uses the ICMP protocol to operate and transmit packets. As we can see only 2 types of packets, Echo (ping) request followed by Echo (ping) reply are used. Echo-Request packets are used for querying the host and reply packets are for sending information from the host to the client.

The program reports errors, packet loss, and a statistical summary of the results, typically including the minimum, maximum, the mean round-trip times, and standard deviation of the mean.

There are 4 requests followed by 4 replies in this case with all the ICMP messages having the same length and ID Value.

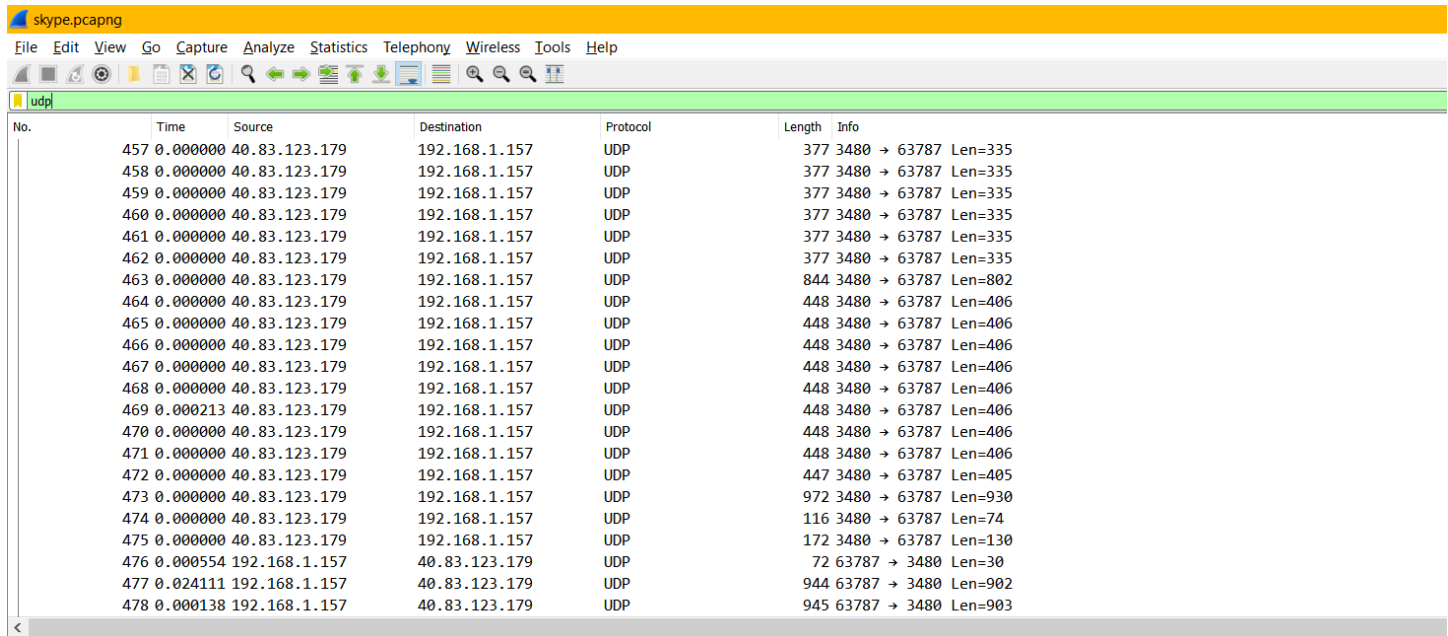
Adding on, the TTL of the request is 128 and the reply is 47 (again standard values) during the transmission of ICMP messages.

Question 4

The Wireshark Capture file for the skype call: skype.pcapng

Since all the UDP communication is happening from a single IP address, thus the IP address of skype is 40.83.123.179

Applying filter: udp, we get



No.	Time	Source	Destination	Protocol	Length	Info
457	0.000000	40.83.123.179	192.168.1.157	UDP	377	3480 → 63787 Len=335
458	0.000000	40.83.123.179	192.168.1.157	UDP	377	3480 → 63787 Len=335
459	0.000000	40.83.123.179	192.168.1.157	UDP	377	3480 → 63787 Len=335
460	0.000000	40.83.123.179	192.168.1.157	UDP	377	3480 → 63787 Len=335
461	0.000000	40.83.123.179	192.168.1.157	UDP	377	3480 → 63787 Len=335
462	0.000000	40.83.123.179	192.168.1.157	UDP	377	3480 → 63787 Len=335
463	0.000000	40.83.123.179	192.168.1.157	UDP	844	3480 → 63787 Len=802
464	0.000000	40.83.123.179	192.168.1.157	UDP	448	3480 → 63787 Len=406
465	0.000000	40.83.123.179	192.168.1.157	UDP	448	3480 → 63787 Len=406
466	0.000000	40.83.123.179	192.168.1.157	UDP	448	3480 → 63787 Len=406
467	0.000000	40.83.123.179	192.168.1.157	UDP	448	3480 → 63787 Len=406
468	0.000000	40.83.123.179	192.168.1.157	UDP	448	3480 → 63787 Len=406
469	0.000213	40.83.123.179	192.168.1.157	UDP	448	3480 → 63787 Len=406
470	0.000000	40.83.123.179	192.168.1.157	UDP	448	3480 → 63787 Len=406
471	0.000000	40.83.123.179	192.168.1.157	UDP	448	3480 → 63787 Len=406
472	0.000000	40.83.123.179	192.168.1.157	UDP	447	3480 → 63787 Len=405
473	0.000000	40.83.123.179	192.168.1.157	UDP	972	3480 → 63787 Len=930
474	0.000000	40.83.123.179	192.168.1.157	UDP	116	3480 → 63787 Len=74
475	0.000000	40.83.123.179	192.168.1.157	UDP	172	3480 → 63787 Len=130
476	0.000554	192.168.1.157	40.83.123.179	UDP	72	63787 → 3480 Len=30
477	0.024111	192.168.1.157	40.83.123.179	UDP	944	63787 → 3480 Len=902
478	0.000138	192.168.1.157	40.83.123.179	UDP	945	63787 → 3480 Len=903

In Statistics > I/O Graphs, we can plot the interpacket interval using the necessary filters

The Display filter will be: udp && ip.addr == 40.83.123.179 as we need to display the UDP packets from the skype IP address.

The interpacket interval can be displayed from the frame.time_delta filter as time_delta gives the intermediate interval between the frames. In UDP transmission, the frames are the packets themselves.

Note:

The standard interval is 1 second, however, we are reducing it to 100 milliseconds for a more granular, finer, and realistic result. We get results that can be better interpreted.

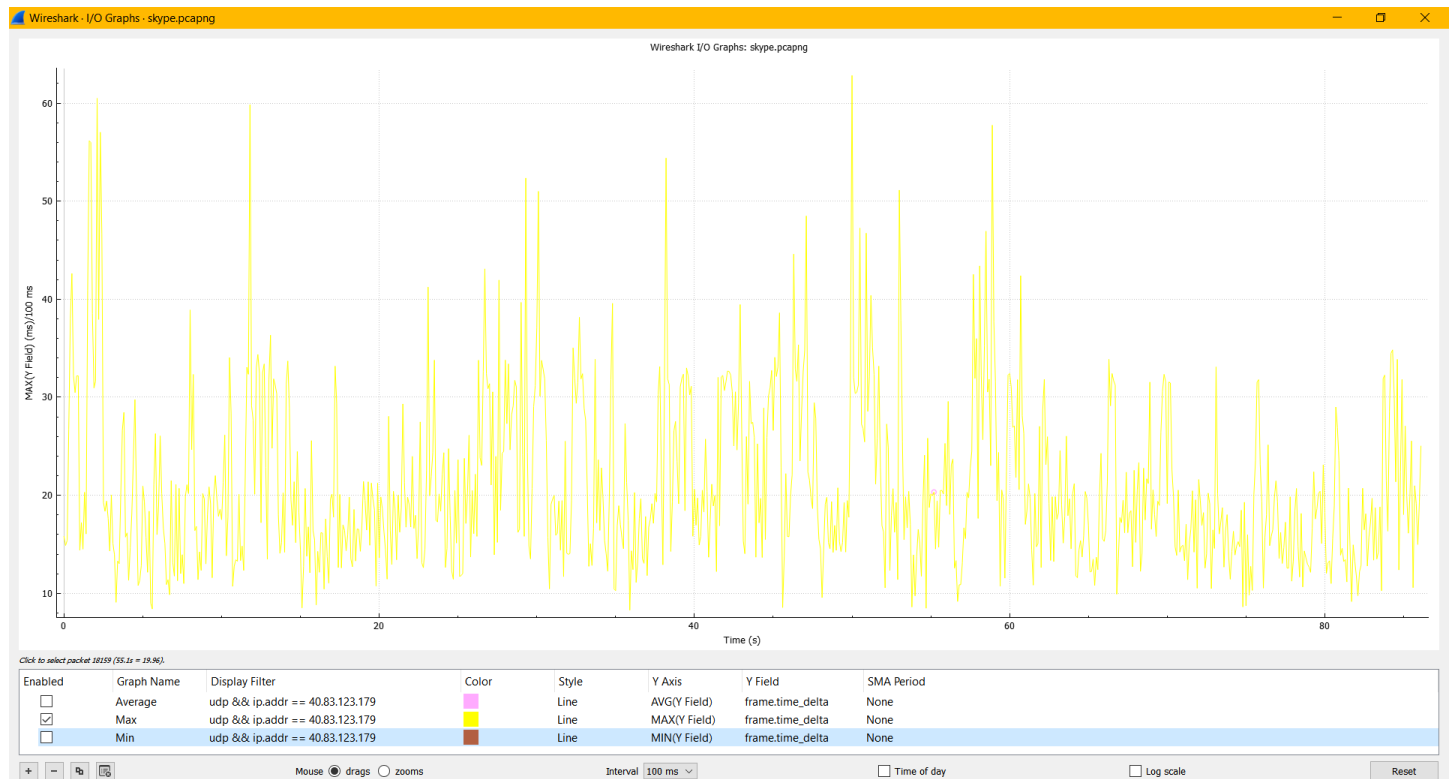
Average Interpacket Interval

AVG (Y Field) and Y Field == frame.time_delta



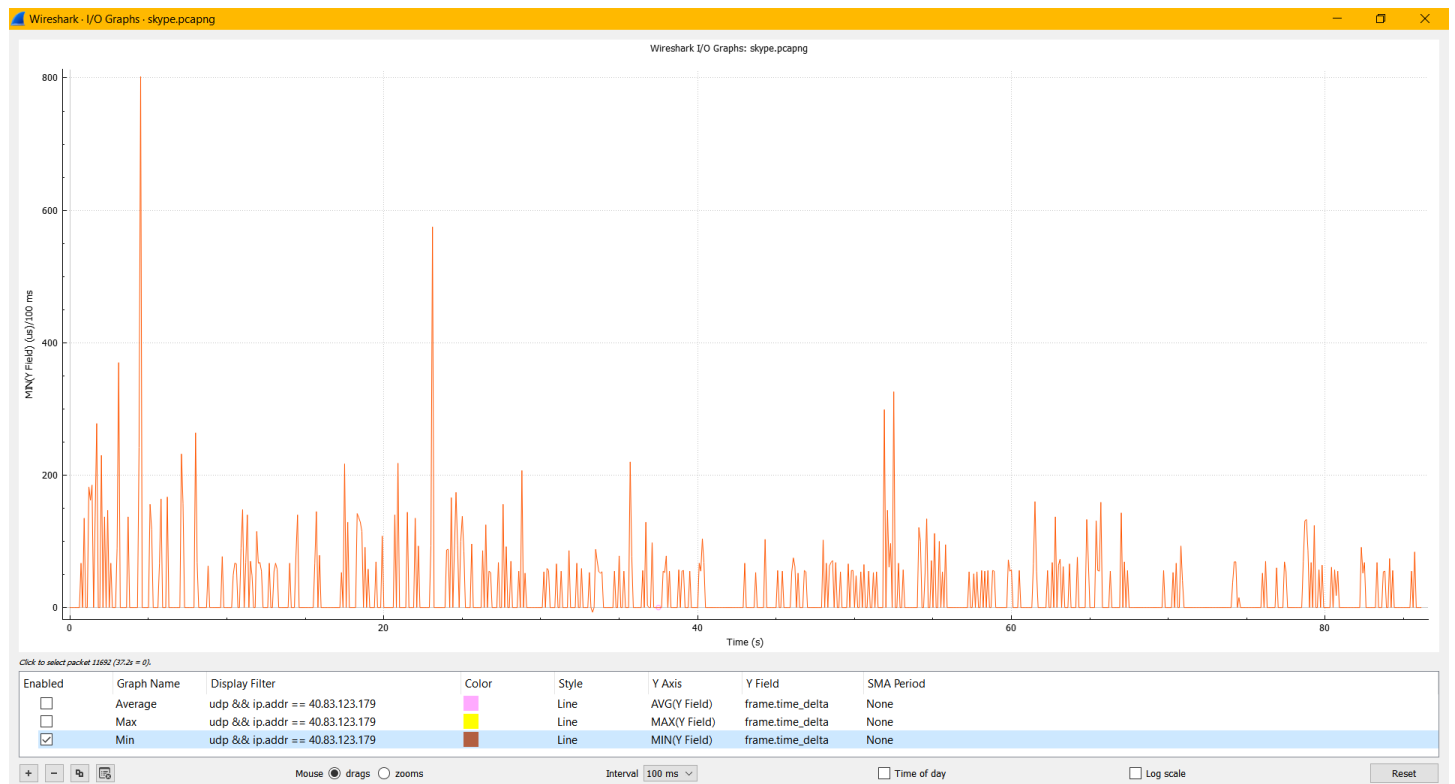
Max Interpacket Interval

MAX (Y Field) and Y Field == frame.time_delta

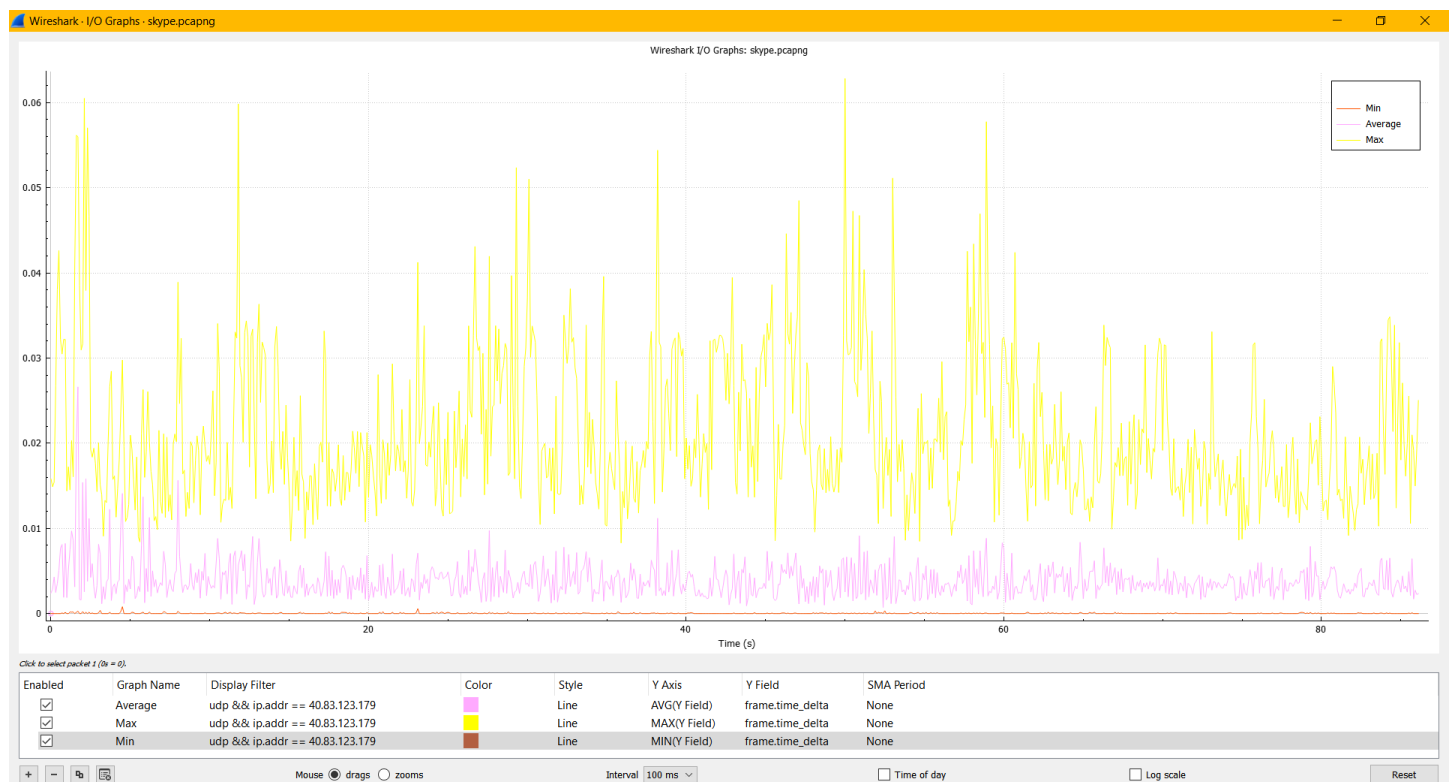


Min Interpacket Interval

MIN (Y Field) and Y Field == frame.time_delta



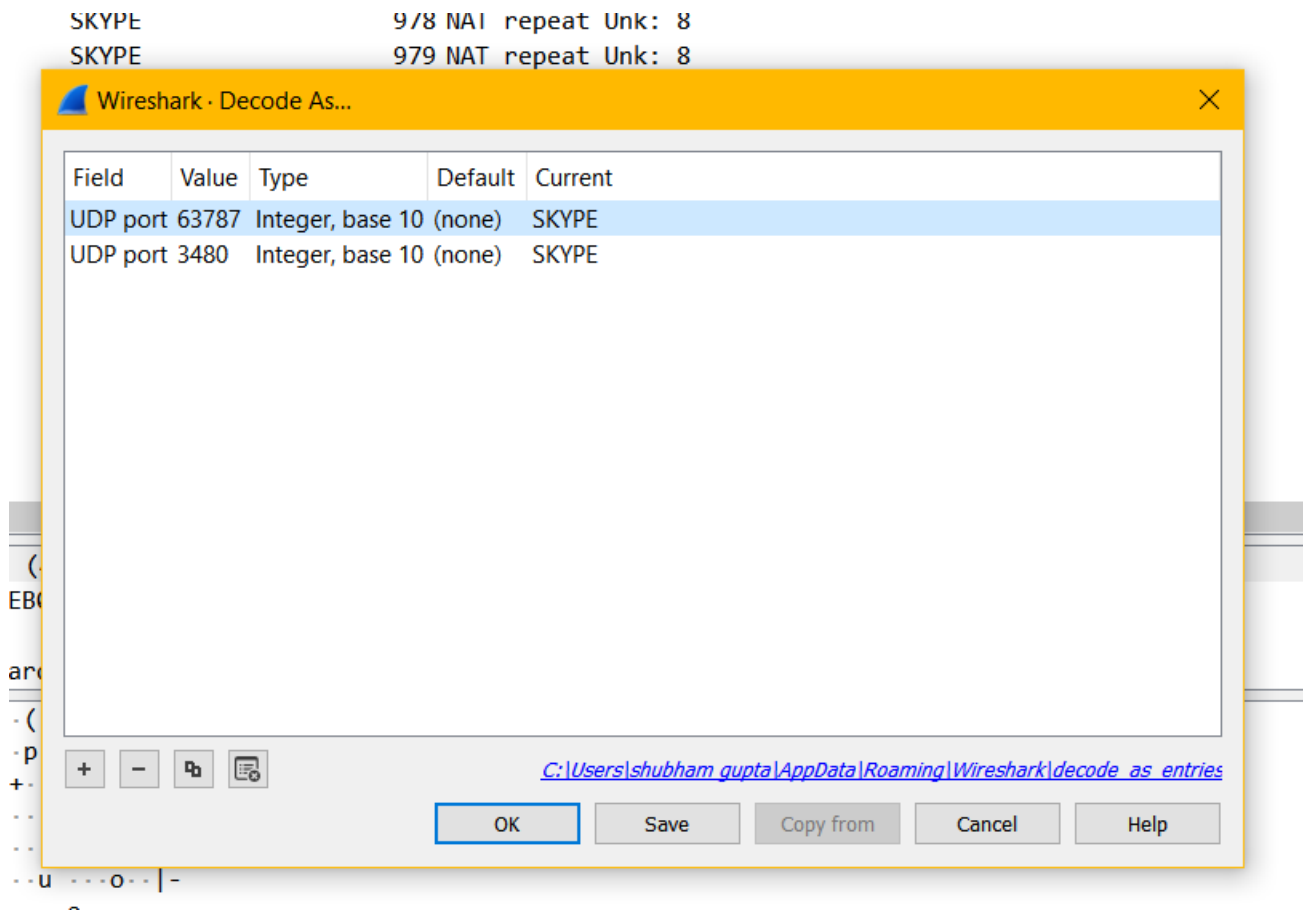
Combined



The above file is saved as interpacket interval.pdf

We need to plot the jitter incurred for the skype traffic. For this, we need to decode the captured UDP packets into SKYPE packets.

This is done using Analyse >Decode As an option, we get a dialog box



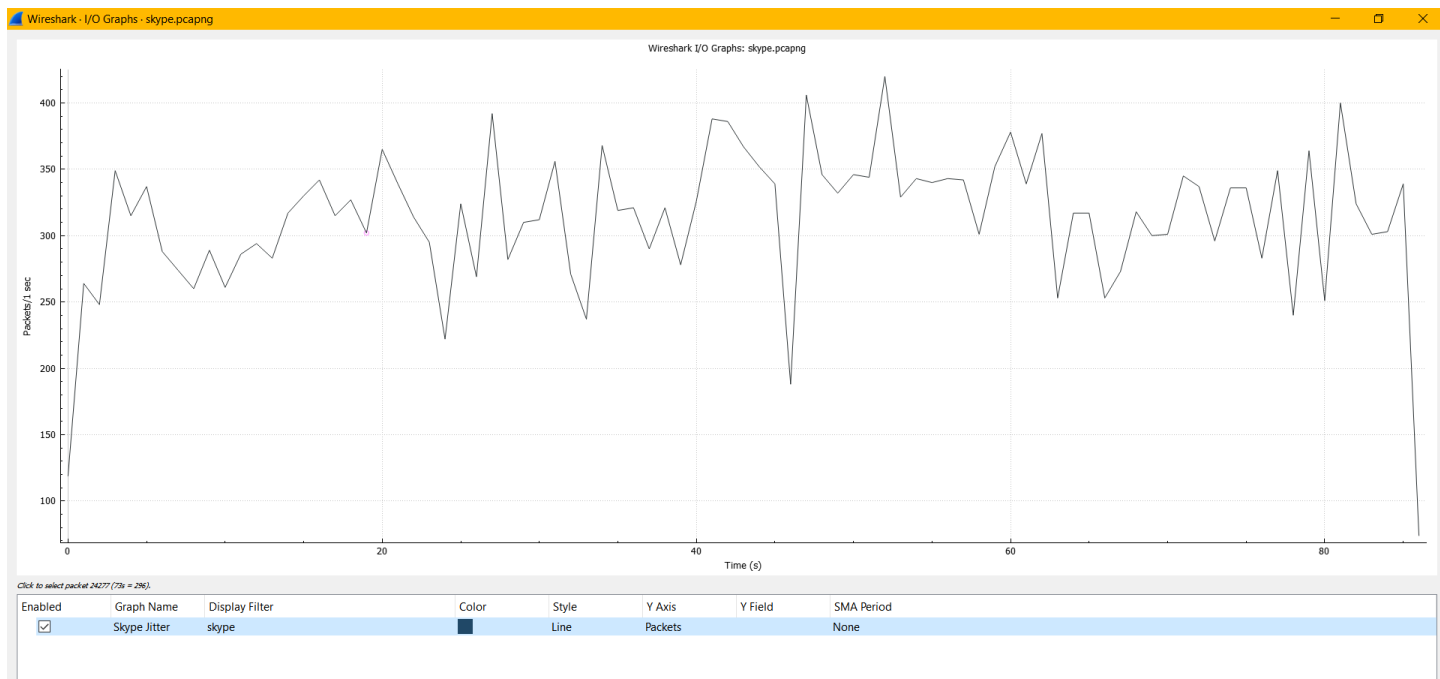
On clicking OK, we will see SKYPE packets in the pcapng file. We need to set the UDP ports at both client and server to the SKYPE ports and the connection is transformed.

With the captured traffic rendering the skype protocol, we can now plot the jitter.

Come to Statistics > I/O Graphs, we need to apply the necessary filters to get the Skype jitter.

Display filter: skype (we only need the skype traffic), select the Y-axis as packets, we get

There is no need to explicitly define the IP Address as we are only considering the skype traffic (it will come from a single IP address only). The filtering has already been covered.



The above file is also saved as skype jitter.pdf.

If you closely observe the above graph, the interval between the spikes represents the time when the client is not communicating with the skype server. This is the jitter. The wide fluctuation in the number of packets also shows the delay in networks due to congestion, inactivity, and a variety of reasons.

If the scale is reduced from 1 second to 100 milliseconds, we see the following figure

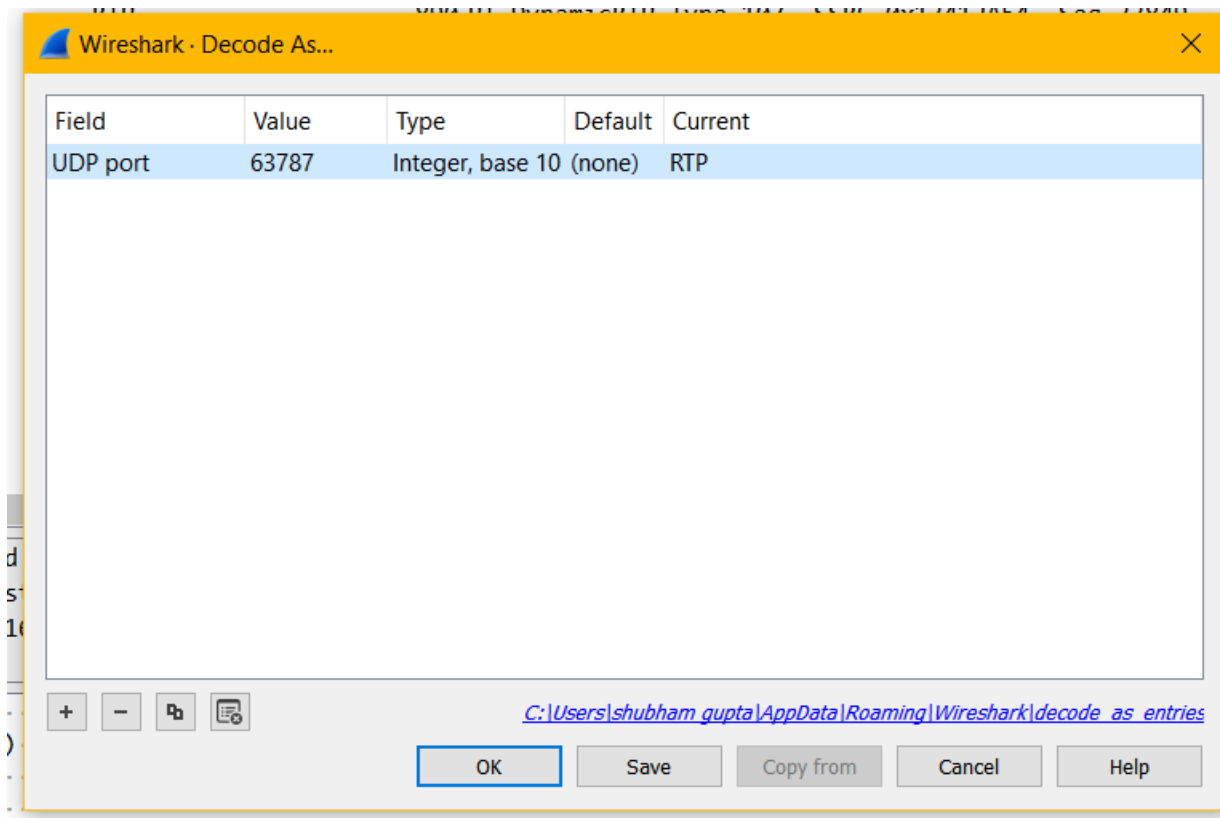


As the time is very short, there is a wide fluctuation. We can conclude that as the scale becomes smaller there is a lot of fluctuation in the signal. The jitter can be better understood on a small scale of time.

Another way to calculate the jitter is to convert the UDP packets to RTP packets. Wireshark has an inbuilt RTP Stream analysis that plots the jitter graph directly.

For this, we need to decode the captured UDP packets into RTP packets.

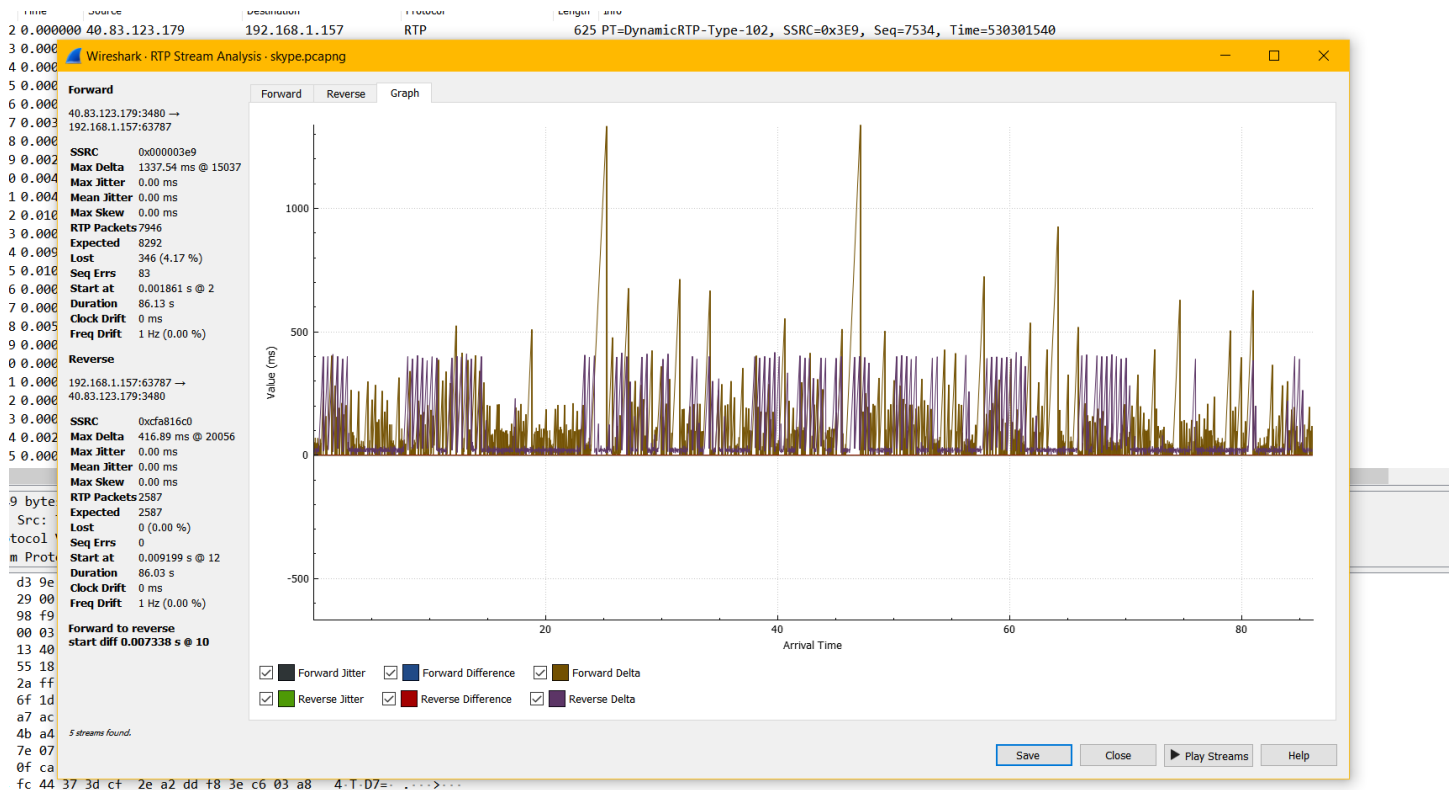
This is done using Analyse >Decode As an option, we get a dialog box,



Once this is done, we will see a host of RTP packets (post decoding from the UDP packets) like the image:

54	0.000000	40.83.123.179	192.168.1.157	RTP	625	PT=DynamicRTP-Type-102, SSRC=0x3E9, Seq=7536, Time=530301540
55	0.000000	40.83.123.179	192.168.1.157	RTP	625	PT=DynamicRTP-Type-102, SSRC=0x3E9, Seq=7537, Time=530301540
56	0.000000	40.83.123.179	192.168.1.157	RTP	617	PT=DynamicRTP-Type-102, SSRC=0x3E9, Seq=7538, Time=530301540, Mark
57	0.003671	40.83.123.179	192.168.1.157	RTP	1015	PT=DynamicRTP-Type-102, SSRC=0x3E9, Seq=7539, Time=530301876
58	0.000000	40.83.123.179	192.168.1.157	RTP	1014	PT=DynamicRTP-Type-102, SSRC=0x3E9, Seq=7540, Time=530301876, Mark
59	0.002653	192.168.1.157	40.83.123.179	RTP	890	PT=DynamicRTP-Type-107, SSRC=0x13412A54, Seq=33860, Time=1144809988
60	0.004691	40.83.123.179	192.168.1.157	RTCP	172	Sender Report (PSE:Unknown [Malformed Packet])
61	0.004248	192.168.1.157	40.83.123.179	RTP	891	PT=DynamicRTP-Type-107, SSRC=0x13412A54, Seq=33861, Time=1144809988
62	0.010567	192.168.1.157	40.83.123.179	RTP	124	PT=DynamicRTP-Type-102, SSRC=0xCFA816C0, Seq=12547, Time=3041827804
63	0.000200	192.168.1.157	40.83.123.179	RTP	891	PT=DynamicRTP-Type-107, SSRC=0x13412A54, Seq=33862, Time=1144809988, Mark
64	0.009213	192.168.1.157	40.83.123.179	RTP	907	PT=DynamicRTP-Type-107, SSRC=0x13412A54, Seq=33863, Time=1144812868
65	0.010719	192.168.1.157	40.83.123.179	RTCP	92	Generic RTP Feedback [Malformed Packet]
66	0.000273	192.168.1.157	40.83.123.179	RTCP	64	Receiver Report
67	0.000173	192.168.1.157	40.83.123.179	RTP	907	PT=DynamicRTP-Type-107, SSRC=0x13412A54, Seq=33864, Time=1144812868
68	0.005068	40.83.123.179	192.168.1.157	RTP	355	PT=DynamicRTP-Type-102, SSRC=0x3E9, Seq=7541, Time=530306476
69	0.000000	40.83.123.179	192.168.1.157	RTP	355	PT=DynamicRTP-Type-102, SSRC=0x3E9, Seq=7542, Time=530306476
70	0.000000	40.83.123.179	192.168.1.157	RTP	355	PT=DynamicRTP-Type-102, SSRC=0x3E9, Seq=7543, Time=530306476
71	0.000000	40.83.123.179	192.168.1.157	RTP	355	PT=DynamicRTP-Type-102, SSRC=0x3E9, Seq=7544, Time=530306476
72	0.000000	40.83.123.179	192.168.1.157	RTP	355	PT=DynamicRTP-Type-102, SSRC=0x3E9, Seq=7545, Time=530306476, Mark
73	0.000000	40.83.123.179	192.168.1.157	RTP	395	PT=DynamicRTP-Type-102, SSRC=0x3E9, Seq=7546, Time=530306508, Mark

Select Telephony > RTP > Stream Analysis, a dialog box will open, where we need to select the Graph Tab on top. We will get the following image:



The above file is also saved as jitter graph.pdf

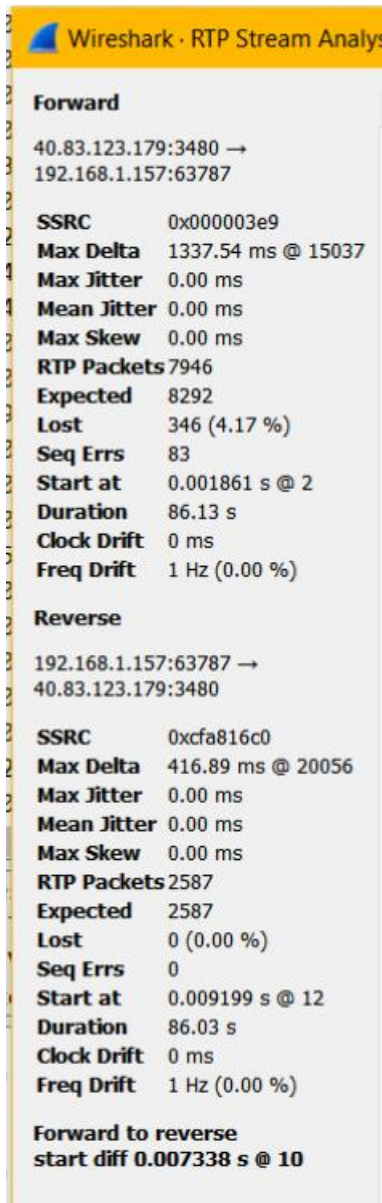
This is essentially the jitter graph of the RTP packets which were previously decoded in the pcapng file. Wireshark has a mechanism to compute the jitter graph for the RTP protocol and not for the UDP protocol. The decoding of packets is done based on the timestamps i.e., the real-time value of the arrival time.

Using this timestamp and delay values the jitter is calculated which is the time when there is no communication between the client and server due to late ACKs or bandwidth fluctuations.

The time difference and delta between the server delays and client delays come within the calculation of jitter which is plotted in the above image.

I have taken a closer snip of the values obtained and we get the max and mean jitter to be 0. The primary reason for this can be seen above. The decoding of all the packets is done in Dynamic-RTP format. If you look at the various RTP Payload formats - https://en.wikipedia.org/wiki/RTP_payload_formats, we can see that the payload type assigned by Wireshark are bigger than 96 (> 96). In such situations, the payload is automatically allocated as dynamic.

The reason why the packet decoding is done dynamically is that Skype is a proprietary and encrypted protocol developed by Microsoft (<https://www.skype.com/en/>) which nobody has been able to decode or reverse-engineer. Skype is one of the oldest protocols that use VoIP and UDP as the transport layer mechanism to transfer data. Microsoft has ensured that nobody can break the proprietary part of the protocol and continuously makes changes to prevent this. Microsoft has made additions to the base IETF standards which are not recognized by Wireshark.



As Wireshark is unable to understand the mechanism of the packets (due to encryption and is unable to understand them, it assigns them to the dynamic payload. It can only capture the necessary ethernet frames and dissect them to get more data.

There are a variety of payloads in the RTP including audio, video, the mixture of both, formats, bitrate, etc. As packets are encrypted and cannot be reverse-engineered to understand the data, we cannot conclusively allocate the data to the packets making it dynamic (it can be anything among the 36 different payloads).

Wireshark cannot decode Dynamic RTP packets for the sole reason that it fails to understand the data present in the packets, so it is unable to allocate a fixed byte stream associated with it. With this, a dynamic tab is allocated to it (essentially, it means unknown or it has not been included as part of the functionality).

As mentioned in the previous definition of jitter, we must know the data and bytes of a packet to compute the jitter value. Since we have no clue about this data, we cannot calculate the value. Thus, a nominal value of 0 is assigned.

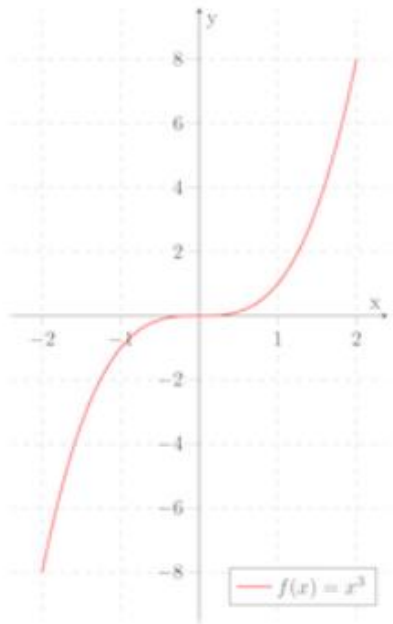
Part 2

Question 1

TCP Cubic

Algorithm Details

Cubic is an algorithm designed to handle high congestion present in networks (situation of excess packets than handling). Initial algorithms like Tahoe and Reno increased window size exponentially to a limit followed by a linear increase to solve the issue. Modern systems even after this kind of increase are unable to sustain this congestion. Cubic uses the mathematical cubic function ($f(x) = x^3$) to solve the congestion issue without overloading the network.



If you look at the above graph, if $x \rightarrow 0$, the value is negligible (point of inflection) otherwise it has a very high slope (both positive \rightarrow concave shape) and (negative \rightarrow convex shape).

The reason for implementing this algorithm is that, consider the case where the congestion window is grown as a cubic function of time since the last packet drop, and the inflection point is set to be the size of the congestion window at the last drop, we get a scenario where:

- The window can grow exponentially fast,
- The congestion window begins growing very slowly when the last drop window is approximately achieved
- If a drop occurred in the last time and currently there are no drops the increase is very fast

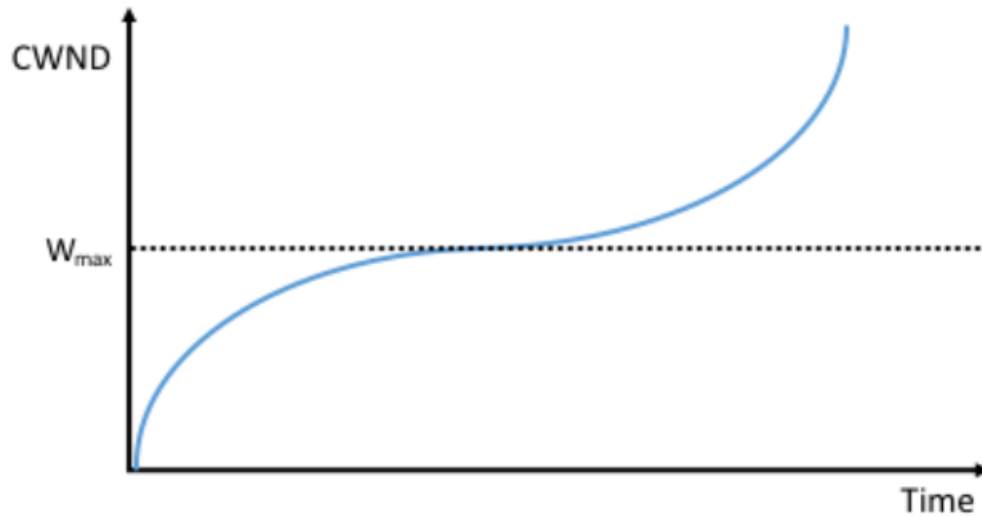
Concave justifies that the window grows quickly and can then slow down very fast (like a catchup game, where the catching speed reduces when the target is nearby)

Convex justifies moving onto an “exploratory phase” in the case of newly available bandwidth and zero packet drop.

Inflection point justifies the point where there are packet drops in the network after which the packet transmission is significantly reduced and further increased in the case of no packet dropped.

CUBIC adjusts its congestion window at regular intervals, based on the amount of time that has elapsed since the last congestion event (the arrival of a duplicate ACK), rather than only when ACKs arrive that depends only on the RTT). This allows CUBIC to behave fairly when competing with short-RTT flows, which will have ACKs arriving more frequently.

The congestion window behaviour is as follows:



If you look at the above function that represents the relation of CWND with W_{max} and Time, TCP Cubic reaches the maximum congestion window size achieved just before the last congestion event as a target (denoted W_{max}).

The start is fast but the growth rate is slow as you get close to W_{max} , be cautious and have near-zero growth when close to W_{max} , followed by an increase in the growth rate as you move away from W_{max} . The latter phrase is essentially probing for a new achievable W_{max} .

This results in computing the congestion window as a function of time t since the last congestion event. The mathematical expression for this CWND function is:

$$CWND(t) = C \times (t - K)^3 + W_{max} \quad \text{where} \quad K = \sqrt[3]{W_{max} \times (1 - \beta) / C}$$

In the above expression, C is the scaling constant and β is the multiplicative decrease factor (Generally $\beta = 0.7$ and $C = 0.4$).

Suitable Scenarios

Cubic is advantageous in the scenario of parallel linking when there are multiple queues on the same link. If there are multiple streams of data on the same link (same src and dst) there is an interaction/linking among the queues while sending the data. Cubic reduces the packet transmission or backs off only if there is a packet drop.

In multiple streams, the linking is made such that the non-cubic streams will take the hit of the packet drop due to which the cubic stream never backs off. The concavity of the graph will come into action and it will reach the target as per the graph behaviour.

The parallel streams will prevent the full blocking of the queue that will prevent the packet drop. In this sort of linked parallel transmission, the burden of overhead and blocking is given to the linked streams which are non-cubic whereas the cubic stream will only deal with the full utilization of the bandwidth to send maximum packets through a larger congestion window.

Very useful and stable in scenarios that have high latency and tremendous bandwidth where multiple streams are transmitting data simultaneously over long fat networks. It gives a full performance scenario without dealing with all the overhead and drastic changes.

It is a very stable solution for high latency and congested networks, especially over multiple hops. Since most data is transmitted in parallel streams today, it is considered as one of the best TCP variants in modern TCP systems with explicit reference to the scenarios and high-speed networks.

Failure Scenarios

If there is a single cubic stream in the connection, then all the overhead has to be performed by the cubic whose behaviour is not very good and shows a sudden window size drop in the case of packet loss.

Large queue sizes also cause bad performance for the cubic algorithm as if there is packet loss at a large window size, then the drastic reduction in the window size can impact RTT and performance significantly.

Not useful in cases where there are single streams that send low bandwidth data from point to point, especially if there is a large queue. If there is a backoff due to a break for whatever reason, it would take a lot of time to recover and send the data even if it is less.

It does not ensure any sort of fairness in the transmission of packets. The wide fluctuation in the transmission of packets can lead to significant packet drops followed by drastic drops in window size leading to a recovery mechanism. This repeats in a cycle which can be easily avoided by bringing fairness and will not bring overhead to keep fluctuating the transmission.

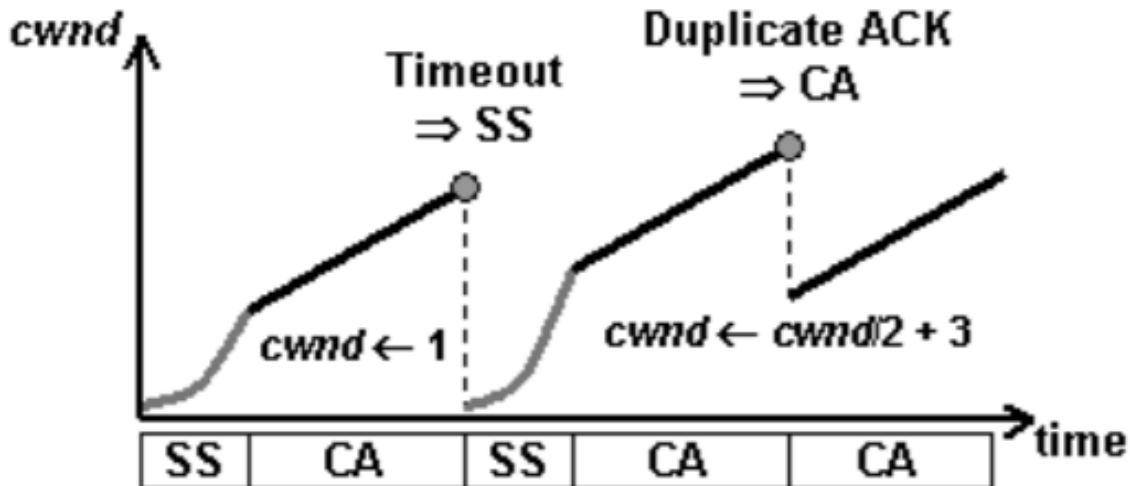
TCP Reno

Algorithm Details

TCP Reno is a fast recovery-based algorithm that retransmits packets after the retransmit timeout has occurred or 3 duplicate acknowledgements have arrived. A single retransmission timeout can cause multiple retransmissions of packets in order or out-of-order.

In TCP Reno, the window size is cyclically changed in a typical situation. The window size continues to be increased until packet loss occurs. It has two phases in increasing its window size: slow start phase and congestion avoidance phase.

Its behaviour is described as:



SS: Slow start
CA: Collision avoidance

It uses an AIMD (Additive Increase Multiplicative Decrease) mechanism to vary the window size based on the network congestions and packet drops. The brief working of the algorithm in the case of retransmission is that it shows if cwnd (congestion window) is less than the threshold value (that is represented using variable ssthresh) then congestion window is increment by one otherwise it enters the slow start. If one or two acknowledgments are received, then the threshold value is set half of the congestion window, but if more than two acknowledgments are received then it indicates the congestion. For each duplicate acknowledgment received increase congestion window.

As a pseudo-code

```

if (cwnd < ssthresh)
    cwnd = cwnd + 1 # slow start
else if (cwnd >= ssthresh)
    cwnd = cwnd + 1/cwnd # congestion avoidance
    if (duplicate ACK)
        If (duplicate ACK == (1 || 2))
            cwnd = ssthresh # packet delayed/ out-of-packet received
            ssthresh = cwnd/2
        else (duplicate ACK > 2)
            cwnd = cwnd + Number (ACK) # packet loss due to congestion
            ssthresh = cwnd/2

```

Suitable Scenarios

It is best suited for situations when the packet loss is are very small. If there is one packet loss in a very long time, then the fast retransmit mechanism can be used to recover and keep the connection continued. It is very useful in low bandwidth low congestion scenarios with very low or almost zero packet drop where there are no multiple packet drops.

It brings minimal overhead compared to the modern TCP variants and is suited for scenarios that have low compute resources and do not demand high-performance computing (HPC) systems.

It was a major part of the TCP system in the 1990s as there were low bandwidth and less data transfer across the internet.

Failure Scenarios

It can detect only a single packet loss at a time and do the necessary retransmission. In the situation of multiple packet losses, it can detect only a single loss at a time and thus will not be able to retransmit the other lost packets. Essentially, in the case of high packet loss, the performance degrades and cannot cope up with the transmission.

In the scenario of a small window size with a packet loss, then enough acknowledgements (min 3) will not come for the packet to be retransmitted. We will have no other option but to wait for the retransmission timeout that increases RTT as well as hampers performance.

Another factor in the case of multiple packet loss is that repeated timeouts can cause retransmission of the same packet multiple times. It's handling of this scenario is poor and cannot be justified even after fast retransmission.

It is not suitable for highly interactive applications where there is a need for constant packet transmission and packet drops cannot be afforded. High congestion leads to a lot of packet drops and the interaction will be very poor for the user. Multiple packet drops are a complete wasted variant.

Modern systems deal with high congestion networks with multiple packets being dropped and thus, are not suited and outdated.

TCP Vegas

Algorithm Details

Vegas detects congestion by increasing RTT and takes decisions about congestion based on the Base RTT value of the packets. At the beginning of a new connection, there is no idea of the available bandwidth and an exponential increase can cause congestion.

In this kind of scenario, Vegas controls its window size based on the RTT. If observed RTTs become large, Vegas recognizes that the network begins to be congested, and throttles the window size. If RTTs become small, on the other hand, the sender host of TCP Vegas determines that the network is relieved from the congestion, and increases the window size again. Hence, the window size in an ideal situation is expected to be converted to an appropriate value.

The window size update function in Vegas is:

$$cwnd(t + t_A) = \begin{cases} cwnd(t) + 1, & \text{if } diff < \frac{\alpha}{base_rtt} \\ cwnd(t), & \text{if } \frac{\alpha}{base_rtt} \leq diff \leq \frac{\beta}{base_rtt} \\ cwnd(t) - 1, & \text{if } \frac{\beta}{base_rtt} < diff \end{cases} ; \quad diff = \frac{cwnd(t)}{base_rtt} - \frac{cwnd(t)}{rtt}$$

where $rtt[sec]$ is an observed round-trip time, $base_rtt[sec]$ is the smallest value of observed RTTs, $(t + t_A)$ is the time to send the next packet whereas t is the current time, and α, β are fixed constants.

It keeps track of when each segment was sent and it also calculates an estimate of the RTT by keeping track of how long it takes for the acknowledgment to get back. Whenever a duplicate acknowledgement is received it checks to see if the (current time segment transmission time) > RTT estimate; if it is then it immediately retransmits the segment without waiting for 3 duplicate acknowledgements or a coarse timeout

Congestion is checked by decreasing the sender rate in contrast to the expected rate resulting in large queues at the sender's end. If there is a lot of distance between the current and expected rate then the increase is significant and a low distance will cause almost no change in the sender's rate.

It keeps a significant check on the bandwidth before increasing or decreasing the transmission rate. The window size is incremented every other time an ACK packet is received and the window size remains unchanged if the RTTs of consecutive packets are the same. If the window size is appropriately controlled such that the packet loss does not occur in the network, the throughput degradation due to the throttled window can be avoided.

This is a mechanism to improve performance while maintaining continued fairness in the transmission. The principal idea of Vegas is fairness, so it minimizes unnecessary transmission and tries to scale it to all aspects of the network. Congestion is avoided in this case as it is done once in a while and not uniformly which does not consume excess network resources.

Suitable Scenarios

It is very useful for interoperability with various TCP variants that use loss-based/estimation methods such as TCP Tahoe, etc as it can improve their performance by adding the delay factor for the retransmission instead of waiting for 3 duplicate acknowledgements or timeout. It detects congestion very early in the system before experiencing packet drops (drop will lead to a timeout, which Vegas avoids for performance) and hence the inter-operability will help the loss-based algorithms to take the necessary steps to tackle the congestion.

A major scenario for Vegas is misbehaving connections. Early congestion packet congestion and a fair/ uniform transmission of packets detect misbehaviour in connections early. Misbehaviour in bandwidth availability but packet delay can be easily checked. In the early stages of congestion, packets are delayed but still reach the receiver. Later on, they are dropped as the connection is unable to handle the old data.

An advantageous scenario for Vegas is where fairness is needed for transmission in packets. It transmits packets uniformly even if the network is highly congested. High congestion is dealt with fast multiple packet loss detection and retransmission to keep connection fail-safe. If the data requirement is consistent and continued but not very high, it is the best TCP delay variant to turn to. It keeps the transmission consistent and does not fluctuate based on the network congestion. In this case, if we cannot afford a packet drop but a packet delay, then Vegas will ensure that packets are not dropped.

Another area suited for Vegas is unknown network connections. As the connection is unknown, it can be good or corrupted. The corrupted/misbehaviour is explained previously and for a good connection using packet delay to check the connection is good. Delay will lead to retransmission making it fail-safe and even in the event of the initial packet reaching, its duplicate nature will inform the receiver to discard it, causing a secure transmission.

Failure Scenarios

Vegas tries to maintain fairness in its connections. If there are multiple Vegas connections, then trying to distribute a fair share of the bandwidth to each of the connections becomes almost technically impossible. In such a scenario the algorithm in the attempt to keep a fair share to all the connections degrades performance severely.

It spends a lot of resources in maintaining fairness whereas if 1 or 2 connections take more bandwidth, it will not impact the transmission severely. The overhead to try and maintain this fairness backfires and cannot be used very well if there are a lot of parallel and simultaneous Vegas (10-12) connections.

In low congestion networks with high bandwidth, Vegas brings fairness into play to transmit the packets which are not needed when an exponential increase in transmission can transfer large data in a very small amount of time. To maintain fairness, it keeps fixed window size with additive increments.

This scenario will create a huge mess if the availability is for a short period. A fast exponential transmission will utilize the connection and complete the job within a short span. Vegas in the ideal situation of fairness will not pounce on the advantage and fail to complete the job.

Vegas uses an estimate of the propagation delay, baseRTT, to adjust its window size, a TCP Vegas connection needs to be able to have an accurate estimation. Persistent connections sometimes reroute their path which calculates baseRTT inaccurately causing a substantial decrease in throughput with time.

Another issue with Vegas is that when packets in the network are propagating slowly (situation before we approach packet delays), however, they are not delayed nor dropped, there is a need to keep a few packets in the network to improve the forward propagation of the packets.

Vegas tries to ensure fairness, which keeps the packet window size consistent and the same number of packets in the network which leads to a persistent connection. This does not help as the need to lighten the congestion in the network is not achieved.

TCP Africa

Algorithm Details

TCP Africa is a hybrid TCP congestion control algorithm whose primary objective is to meet end-to-end expectations in networks whose bandwidth exceeds 100 Mbps. It uses an aggressive and scalable window increase rule to allow quick utilization of available bandwidth. packet round trip time measurements technique to predict imminent congestion events.

It uses two parallel mechanisms simultaneously, one being the packet loss mechanism and packet delay mechanism to detect congestion and adjust suitable parameters to handle it.

- Scalable window sizing to make maximal use of available bandwidth. The window is increased when there is bandwidth and no packet is dropped. In a fast congestion building scenario, it uses a scalable congestion window avoidance rule whereas, in a slow congestion building scenario, it uses the TCP-Reno congestion window avoidance rule (it is derived from TCP-Reno).
- Packet RTT measurement to predict early congestion in the network (similar to the Vegas algorithm described above) that uses packet delay to measure congestion and not drop.

Moving onto the mathematical description of the algorithm, we define a few quantities, the current high accuracy RTT estimate – $aRTT$, the current minimum RTT in the congestion path – $minRTT$ (purpose – queueing delay estimation), and current window congestion size – W .

Congestion is detected from the following metric:

$$\frac{W(aRTT - minRTT)}{aRTT} \geq \alpha.$$

In the above expression, the quantity $(aRTT - minRTT)$ gives us an estimate of the queueing delay of the network. Since the overall RTT is $minRTT + (aRTT - minRTT)$, the quantity $(aRTT - minRTT)/aRTT$ is the proportion of the RTT that is due to queueing delay rather than propagation delay.

Since TCP maintains an average sending rate of $W/aRTT$ packets per second, a slight extension gives $W(aRTT - minRTT)/aRTT$ is an estimate of the number of packets that the protocol currently has in the queue. α is a constant real parameter ($\alpha > 1$). The choice of α determines how sensitive the protocol is to delay. Generally, $\alpha = 1.65$ which is optimal under most (82%) of the network conditions.

Combining both the above congestions avoidance protocols (fast and slow), we obtain the protocol:

```

if ( $W(aRTT - minRTT) < \alpha \times aRTT$ ) {
     $W = W + fast\_increase(W)/W$ 
} else {
     $W = W + 1/W$ 
}

```

If you look closely at the else block of the above code segment, it corresponds to the congestion window protocol of TCP-Reno.

The function $fast_increase(W)$ generally uses a modified rule to increase W based on the congestion. Generally, $fast_increase(W) = 0.01 \times W$. It is a very efficient function because it always the window size by 1% and takes an apt decision for the window size based on the congestion bottleneck.

The use of delay metrics to predict early congestion can be derived from the same congestion metric as described above. The switch between the fast and congestion depends on the amount of queueing delay and current RTT. So, flows with small RTT don't affect the congestion protocol. If $aRTT/W$ is small, then it impacts the queueing delay and switched to the slow mode.

This has a wide-ranging impact on the network by giving rise to improved RTT bias performance as compared to other high-speed loss-based TCP protocols. It outperforms Reno as the long RTT flow is using a scalable increase rule that allows faster than one packet per round trip time increase

The base mechanism for Africa is to use packet loss as a sign of congestion. However, using delay in packet transmission as congestion helps to make better decisions on the variation of the W gives efficient and optimized performance.

Suitable Scenarios

TCP Africa was built for the situation where the access point transfer link was more than 100 Mbps. It overshadows other protocols with brilliant performance where tremendous bandwidth and uninterrupted connections are required. HPC systems and Cloud platforms that provide bandwidth to run 1000s of servers simultaneously where we cannot expect even a 0.01% drop in Quality of Service (QoS) are suited scenarios and Africa meets the expectation and requirements of such scenarios.

Using a combination of both loss and delay to detect congestion, its packet drop rate is almost zero and it takes necessary steps to deal with congestion far earlier than the scenario hits.

Another scenario where Africa works are in the case of UDP emulation. It is slightly inefficient than the UDP transmission because it obeys all the TCP protocol rules but gives a very high utilization to transfer highly important data without losses.

In scenarios where UDP-like transmission is required but we can compromise on the time but not correctness by up to 10% (not exactly UDP, but an emulation where 90% of the real parameters are achieved), Africa is the best-suited algorithm by far.

Failure Scenarios

TCP Africa is a complicated protocol dealing with multiple functionalities it performs a lot of computation on every single packet by increasing/decreasing window size, high accuracy RTT, AIMD, *fast_increase(W)* function, etc, it takes unreasonable equipment for all the functionality and brings immense overhead.

The immense overhead TCP Africa brings with it cannot be handled if the processing power is less. It cannot be used in systems that don't provide the necessary resources. It fails in scenarios where we have low compute resources and can do the necessary computation on a subset of packets, but not all; which gives a reduced performance.

Africa is also a delay-based protocol and it is very sensitive to reverse-path congestion that occurs due to the returning stream of ACKs. This can lead to an inaccurate calculation because of the ACK noise.

It can impact the congestion detection scheme forcing it to switch to a slow TCP Reno congestion handling mechanism even in a situation where bandwidth to increase packets is available. This forceful slow congestion handling will make it harder to transmit packets by reducing the algorithm's ability to acquire bandwidth quickly.

Comparison

Cubic vs Reno vs Vegas vs Africa

TCP Variants & Year	Base variant	Mod. @ ¹	CD ²	N/w Env. ³	Added/Changed Modes or Features	Congestion Avoidance Method $cwnd_{new}$
TCP Reno 1990	Tahoe	Sender	Loss	Wired	Slow Start plus Congestion Avoidance plus Fast Retransmit and Fast Recovery	AI $cwnd_{new} = \alpha + \beta$ $\alpha = cwnd_{old}, \beta = \text{Quotient}$
TCP-Vegas 1995	Reno	Sender	Delay	Wired	Utilizes Bottleneck buffer as a primary feedback for the Congestion Avoidance and secondary for the Slow Start.	AI $cwnd_{new} = \alpha + \beta$ $\alpha = cwnd_{old}, \beta = \text{Constant}$
TCP Cubic 2008	BIC	Sender	Loss	HS/LD	The congestion window control as a cubic function of time elapsed since a last congestion event	Equation Based $cwnd$
TCP Africa 2005	Vegas, HS-TCP	Sender	LD	HS/LD	Exchange between fast of HS-TCP and slow start of NewReno mode depending on the Vegas-type network state estimation	AI $cwnd_{new} = \alpha + \beta$ $\alpha = cwnd_{old}, \beta = \text{Quotient}$

Among all the protocols the most basic is TCP Reno that follows a simple protocol to avoid congestion windows by altering the window size and fast retransmission of the dropped packets. Reno does this by setting the ssthresh value to half of the current cwnd instead of setting the cwnd to 1.

Vegas enhances the use of Reno by extending the functionality into both wired and wireless environments. This is done by calculating baseRTT and comparing it with currentRTT (recently received ACK). If $\text{currentRTT} \ll \text{baseRTT}$, then cwnd is incremented; else it is decremented.

Cubic uses a modern mechanism for packet loss handling which can quickly and effectively use bandwidth for large transmission and slow down in high congestion scenarios. It can work in an optimized scenario in all sorts of networks congestion and uses packet loss as a congestion handling mechanism.

The later and more modern algorithms all exhibit the hybrid version model where different features are taken from different algorithms and are implemented into one. It essentially expands functionality and minimized failure scenarios.

Africa combines the HS-TCP and NewReno (Enhanced Reno protocol that can handle multiple packets dropped simultaneously) protocols for the packet drop and Vegas for the packet delay method-based congestion handling.

All these TCP variants consist of two modes in their Congestion avoidance algorithms and operate depending on some constraints and conditions. The above table uses two distinct components to define the algorithm for a TCP Variant.

The first component and second component of the function, $f(cwnd)$ is defined as α and β to represent the increment policies of additive and multiplicative, respectively. α in the additive increase which depends on the old cwnd size. And β in the additive increase depends on whether it is zero, constant, scaling, quotient, or other.

Inferences on all the 4 variants

Modern requirements demand the use of high bandwidth and large transmission of data consistently. TCP Africa is the best for such purposes. It uses the delay to make fair decisions and combines 3 protocols, HSTCP, NewReno, and Vegas to make the best decisions.

High compute resources are becoming very common these days and people use HPC systems for regular computations and professional and college work. They are not very costly to buy, set up, and deploy which makes Africa the best possible choice to implement today. We are moving towards cloud computing, VMs, Smartphones, 5G, etc which make the use of TCP Africa become a common phenomenon in the near future.

If you want to transmit data over a high bandwidth channel consistently with multiple users, TCP Vegas is the best as it maintains fairness during the entire transmission and consistently updates window size if it sees more bandwidth. It does not allow packets to be dropped and retransmits packets when they are delayed. It will ensure all packets reach you within the span of time.

Checking corrupted/unknown connections before transmitting packets can be done effectively using Vegas. The best bet you have while venturing into unknown territory where you need data consistently not at a rapid pace is Vegas. It brings fairness into play and will not disregard any connection.

If you have a known connection which experiences unknown bursts of congestion and there is a need for high transmission of data, TCP Cubic is the variant to turn to. It estimates the best transmission in situations of packet drops. As the congestion bursts are unknown, the cubic function will best utilize the bandwidth and adjust cwnd for efficient output.

It was built for high congestion and high throughput networks. It is the most advanced TCP variant that deals with packet loss handling. It will not bring fairness and it would not matter, as we need out material in the high congestion scenario; regardless of the transmission to the other users (Selfish/greedy mechanism).

TCP Reno has become an outdated variant and no longer has any practical use in modern systems. Being one of the first TCP variants, it was apt for the 1990s with few connections, low bandwidth, low compute resources, minimal packet drops, fast retransmission, and packet recovery. It almost has no use in the current scenarios.

However, with old systems and low compute resources where we need to transfer low bandwidth/data packets across a point-to-point (P2P) link, Reno can come in handy.

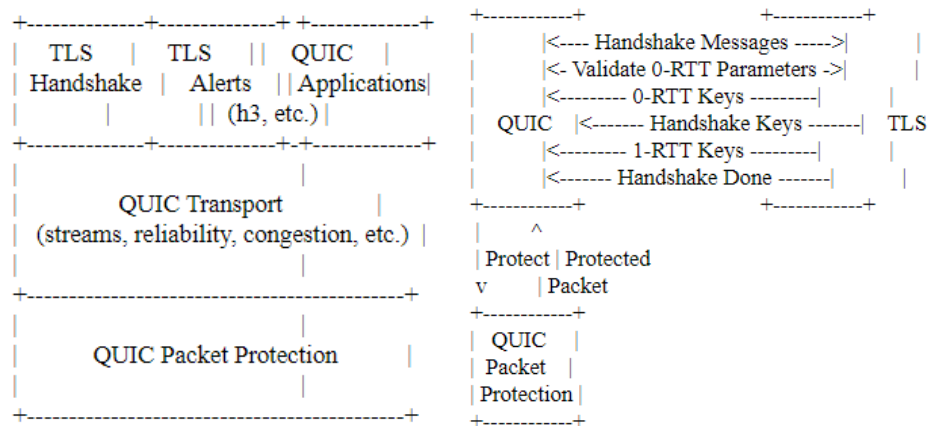
Overall, we have described the 4 variants, how it handles different situations, and a comparative study in the end with a few inferences about the modern outlook of all the TCP variants.

Question 2

The full form of QUIC is **Quick UDP Internet Connections** with the sole purpose of reducing the latency of TCP. In a simple description, it combines TCP+TLS+HTTP/2 implemented on UDP

The integration of TCP with the lowest levels of the machinery makes tinkering impossible and QUIC is built over UDP giving it a lot of flexibility to tinker and play with. Its major application is with flow-controlled streams for structured communication, low-latency connection establishment, and network path migration. It gives the added advantage of TLS for security measures to implement confidentiality, integrity, and loss detection.

QUIC wants to maintain packet correctness similar to TCP but cannot compromise on the speed (It needs a low latency). It integrates the methodology HTTP uses to reduce the latency. HTTP2 uses TLS (Transport Layer Security) to encrypt data before sending it through the Transport Layer. QUIC makes the exchange of setup keys and supported protocols part of the initial handshake process to incorporate TLS to encrypt the data. This gives zero RTT to establishing the TLS encryption into the QUIC stream. Combining multiple requests into a single step reduces latency.



The above images show that the Handshaking, Validation of RTT parameters, TLS keys are all shared in the initial handshake mechanism. It uses a co-operation mechanism coupled with 2 major interactions:

- The TLS component sends and receives messages via the QUIC component, with QUIC providing a reliable stream abstraction to TLS.
- The TLS component provides a series of updates to the QUIC component, including new packet protection keys to install and state changes such as handshake completion, the server certificate, etc.

TLS is incorporated into the packets which are further transmitted as a QUIC Stream.

TCP to incorporate TLS uses extra packets to exchange the setup keys which brings extra overhead to add the TLS. TCP uses a 3 Way handshake to only establish the connection. This is followed by a packet exchange for the security keys in the exchange mechanism. Sometimes TCP also establishes multiple connections just to exchange the security keys. It is inefficient if it can be integrated into a single step.

As previously stated, tinkering can only be done with UDP and not TCP. UDP does not have a loss recovery mechanism whereas QUIC distributes data into streams and flow control and data recovery is maintained at each stream. If there is an error in a stream, it can serve the other streams independently. This improves the performance of error-prone links.

It does not wait for the error-correction mechanism to transmit the packets; this is essentially the head-of-line blocking problem. The transmission persists and the stream where the error had been detected will be rectified in due course of time. It does not let the performance/user experience suffer due to network issues/failures.

The issue with TCP is that it detects an error followed by a hold on the transmission of the remaining packets. It rectifies the error based on the error-correction code. This is followed by transmitting the packets kept on hold.

QUIC encrypts packets individually and not as a stream. This saves a lot of overhead and time in the decryption time, where we do not have to wait for the remaining packets and assemble them followed by decryption. With single packet encryption, out-of-order decryption can also be done. The single-packet encryption can do all this in a single handshake making it more effective.

TCP encrypts data in a byte-stream and there is no knowledge about this stream by any layer except the transport layer. The extra overhead of assembling the out-of-order packets and then decrypting them is ineffective.

QUIC improves performance in network-switching events by including a connection identifier that uniquely identifies the connection to the server regardless of source. This allows the connection to be re-established simply by sending a packet, which always contains this ID, as the original connection ID will still be valid even if the user's IP address changes

TCP in the scenario of a network switching event results in a lengthy, computation-intensive event. First, there is a timeout that waits for packet transmission to complete and closes the TCP connection. After this, new connections are established by a 3 Way Handshake to the new connection. ACK packets are sent to the new network or changed IP address and then a connection is re-established. This completes the network switching.

QUIC does not need to be implemented in the OS kernel. The protocol stack of QUIC is hosted on a particular application which makes transportation and implementation very easy as it does not deal with the OS complexities. It makes forward error correction (FEC) easy by implementing error codes on error-prone and misbehaving connections beforehand.

Forward error correction (FEC) is a method of obtaining error control in data transmission in which the source sends redundant data and the destination recognizes only the portion of the data that contains no apparent errors. It is a very simple mechanism to check misbehaving connections. Adding new and exclusive functionality to the QUIC version is easy as OS updates and complex mechanism is not needed.

TCP is implemented within the low levels of the OS kernel and supports packet transfer from multiple applications into the Transport Layer. TCP is used when the number of packet losses is significant. Applications sometimes jump from UDP to TCP to arrest the slide of packet losses to complete the data transfer. This switching causes latency. It could lead to the blocking of old UDP connections to establish new TCP connections. This can also cause the head-of-line (HOL) blocking problem.

QUIC transfers streams over UDP, so switching does not hamper latency. It is implemented over the HTTP/2 stack. Since HTTP stack is used by most applications, it is placed over the libraries in a modularized manner which causes almost no effect on the error correction mechanism in HTTP and latency.

QUIC Pros/Cons

Pros

- There is no other transport layer protocol like QUIC which is fast and ensures reliable data transfer.
- It was built for modern technologies but uses the HTTP stack as a fallback option when UDP is not available. Thus, it shows compatibility with old versions of browsers and HTTP/1.1 protocols.
- Encryption of singular packets and not as a stream. Does not need to wait for the assembly of the out-of-order packets stream for decryption and can assemble and render data to the user post decryption even if a few packets fail to reach or packet error codes are detected. This tagging of packets is a very useful inaccurate calculation of RTT time and takes the necessary steps in the handling of network congestion.
- No unnecessary connections are created during the handshake. The connection ACK and TLS security happen within a single handshake.
- QUIC has a stable connection for people on the move and recovers from packet loss much faster than TCP
- Implements a Forward Error Correction (FEC) mechanism that is not implemented in UDP and TCP with no negative implementation consequences in real-world applications.
- Multiplexes data into a multiple QUIC over UDP streams and decreases Head-of-Line (HOL) blocking significantly which TCP and UDP are unable to mitigate.
- Fast network migration and connection switching capabilities without a full closing and re-establishment procedure, increasing transfer speed with minimal latency.
- It is not integrated with the hardware to process data and send it forward. It stays independent of the hardware and is defined for the use of a single application. It is only required at the application level which makes compatibility less complex; updating and inculcating changes easier.

Cons

- As it is a new protocol, to migrate QUIC into old applications, we need to write new code into the HTTP/2 framework to take full advantage of it.
- New protocols bring security concerns with them. If QUIC is being incorporated into an expensive software that has wide-ranging problems, the last problem we want to run into is security issues. QUIC security mechanism is not defined very clearly, so dealing with this problem is dicey.
- QUIC uses a combination of TCP+TLS+HTTP/2 which is a combination of 3 different protocols stacks. Troubleshooting such a scenario will require the troubleshooting of all the 3 different protocol stacks. Small issues will require a massive amount of troubleshooting and fixing bugs can take a lot of time, effort, and energy for debugging.
- It is of not much use in the scenario of UDP throttling. UDP is an unreliable connection that goes on and off during the packet transmission. If UDP is unavailable, it switches to HTTP which makes the packet transfer inefficient. If HTTP time is over 60%, then TCP transmission is faster and it brings 100% reliability to the fore; rendering the QUIC protocol useless.

Part 3

Compilation Data

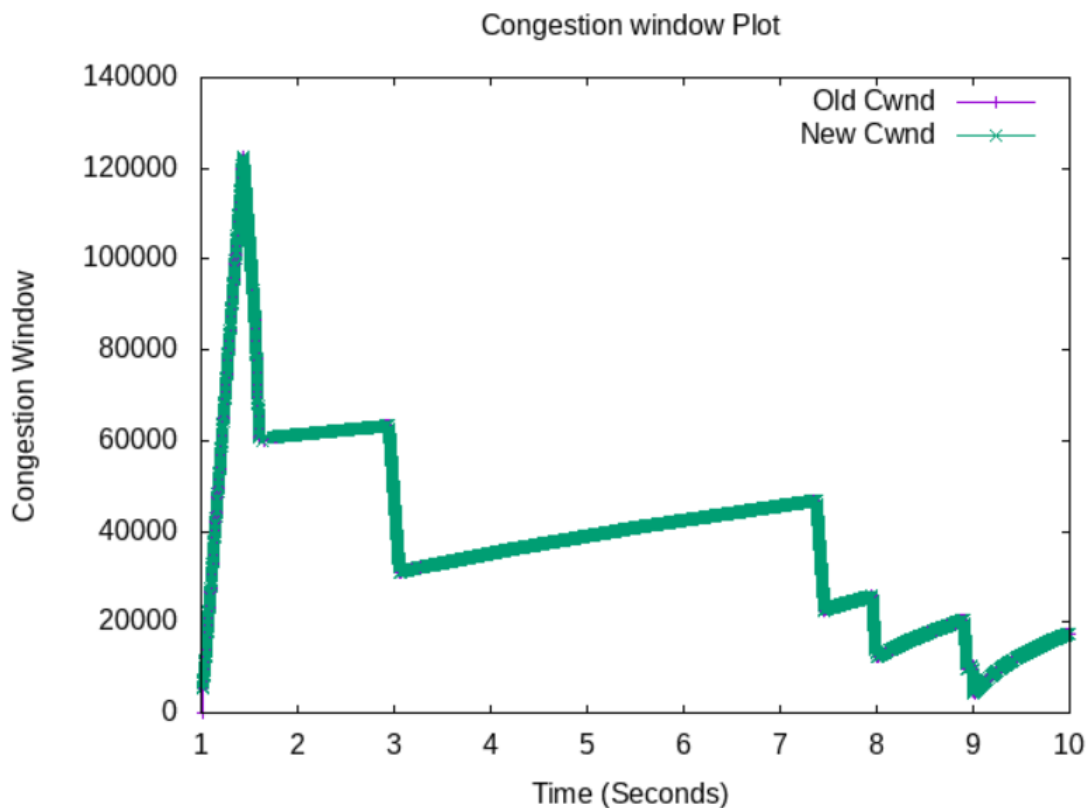
```
mastershubham@LAPTOP-8Q15SHE6:/mnt/d/ns-allinone-3.34/ns-3.34$ ./waf --run scratch/TCP_Demo
Waf: Entering directory `/mnt/d/ns-allinone-3.34/ns-3.34/build'
[2604/2675] Compiling scratch/TCP_Demo.cc
[2635/2675] Linking build/scratch/TCP_Demo
Waf: Leaving directory `/mnt/d/ns-allinone-3.34/ns-3.34/build'
Build commands will be stored in build/compile_commands.json
'build' finished successfully (1m30.017s)
mastershubham@LAPTOP-8Q15SHE6:/mnt/d/ns-allinone-3.34/ns-3.34$
```

On compiling TCP_Demo.cc file for the NewReno TCP variant, this is the result obtained. The build is completed and the necessary files are generated.

We get the following files

- tcp-demo-0-0.pcap
- tcp-demo-1-0.pcap
- tcp-demo.tr
- tcp-demo.cwnd

On running gnu plot CW.plt, we get the following plot (stored as CW.png)



All the above files are stored in the NewReno folder of Part 3 of the Tar ball folder.

```

mastershubham@LAPTOP-8Q15SHE6:/mnt/d/ns-allinone-3.34/ns-3.34$ ./waf --run scratch/TCP_Demo
Waf: Entering directory `/mnt/d/ns-allinone-3.34/ns-3.34/build'
[2604/2675] Compiling scratch/TCP_Demo.cc
[2635/2675] Linking build/scratch/TCP_Demo
Waf: Leaving directory `/mnt/d/ns-allinone-3.34/ns-3.34/build'
Build commands will be stored in build/compile_commands.json
'build' finished successfully (58.712s)
mastershubham@LAPTOP-8Q15SHE6:/mnt/d/ns-allinone-3.34/ns-3.34$

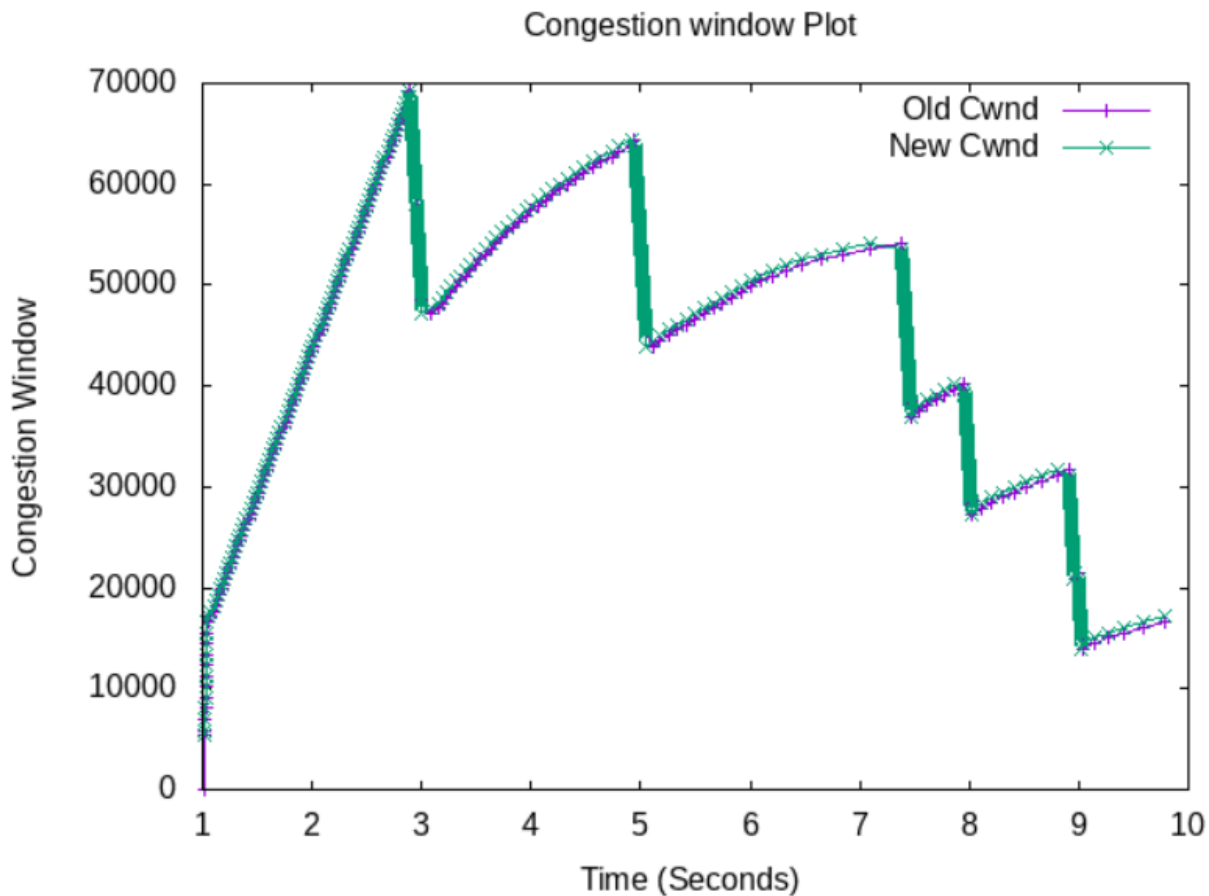
```

On compiling TCP_Demo.cc file for the Cubic TCP variant, this is the result obtained. The build is completed and the necessary files are generated.

We get the following files

- tcp-demo-0-0.pcap
- tcp-demo-1-0.pcap
- tcp-demo.tr
- tcp-demo.cwnd

On running gnu plot CW.plt, we get the following plot (stored as CW.png)



All the above files are stored in the Cubic folder of Part 3 of the Tar ball folder.

Question 1

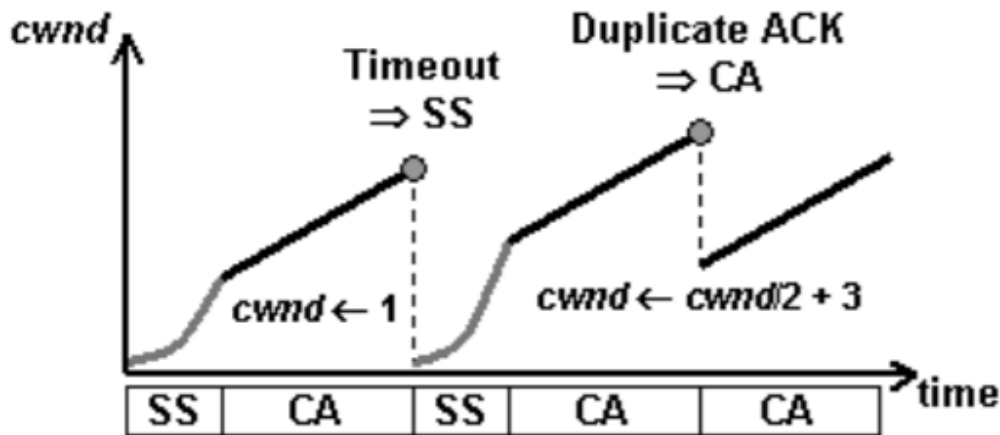
Both the algorithms NewReno and cubic reduce cwnd 6 (SIX) times. (The answer looks like 5 but if you look at the last drop carefully (the one that occurs just before the 9 seconds), you will see that there is a slight increment (only once which can be verified by the cross and dash symbol present there)). It is a negligible increment but it needs to be considered.

The remaining drops are a fully linear graph and have no negligible increments in between, thus the number of reductions in the cwnd can be safely concluded as 6.

To understand the behaviour in the reduction of cwnd, we need to explore the algorithms.

TCP NewReno follows the same cwnd mechanism as TCP Reno. I have described the behaviour of the TCP Reno cwnd as part of my answer to Question 1 of Part 2. I am copying the same below.

The behaviour of cwnd of Reno/NewReno is:

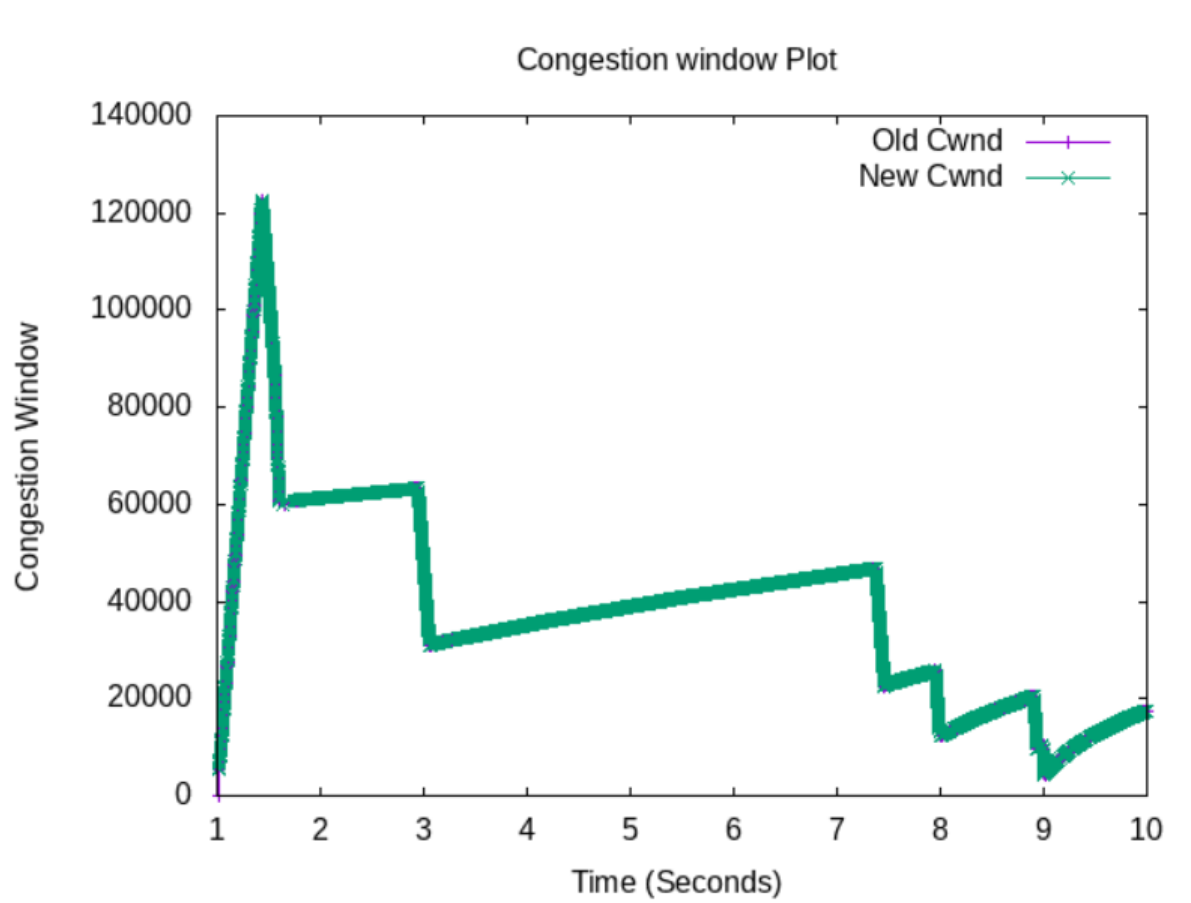


SS: Slow start
CA: Collision avoidance

The Pseudo-Code of Reno/NewReno is:

```
if (cwnd < ssthresh)
    cwnd = cwnd + 1 # slow start
else if (cwnd >= ssthresh)
    cwnd = cwnd + 1/cwnd # congestion avoidance
    if (duplicate ACK)
        If (duplicate ACK == (1 || 2))
            cwnd = ssthresh # packet delayed/ out-of-packet received
            ssthresh = cwnd/2
        else (duplicate ACK > 2)
            cwnd = cwnd + Number (ACK) # packet loss due to congestion
            ssthresh = cwnd/2
```

If you look at the cwnd obtained by gnu plot of the NS-3 simulation of the NewReno connection:



We can see that in each cwnd drop, there is a multiplicative decrease of half (part of the AIMD protocol of TCP NewReno). The 1st drop-in cwnd is from 121000 to about 600500 which is about half. As the exact scale values are not present, I am using intuition to make the decision.

The cwnd is reduced by half if duplicate ACKs are coming in. Whether duplicate ACKs are 1, 2, or more than 2, the cwnd is reduced to half.

Just before the 9th second, there are 2 drops spaced at the interval of 1 packet. This means that after the 1st duplicate ACK, the cwnd came down from 20000 to 10000, and after the 2nd duplicate ACK, the cwnd came down from 10000 to 5000 (reduced by half, as stated by the above Pseudo-Code).

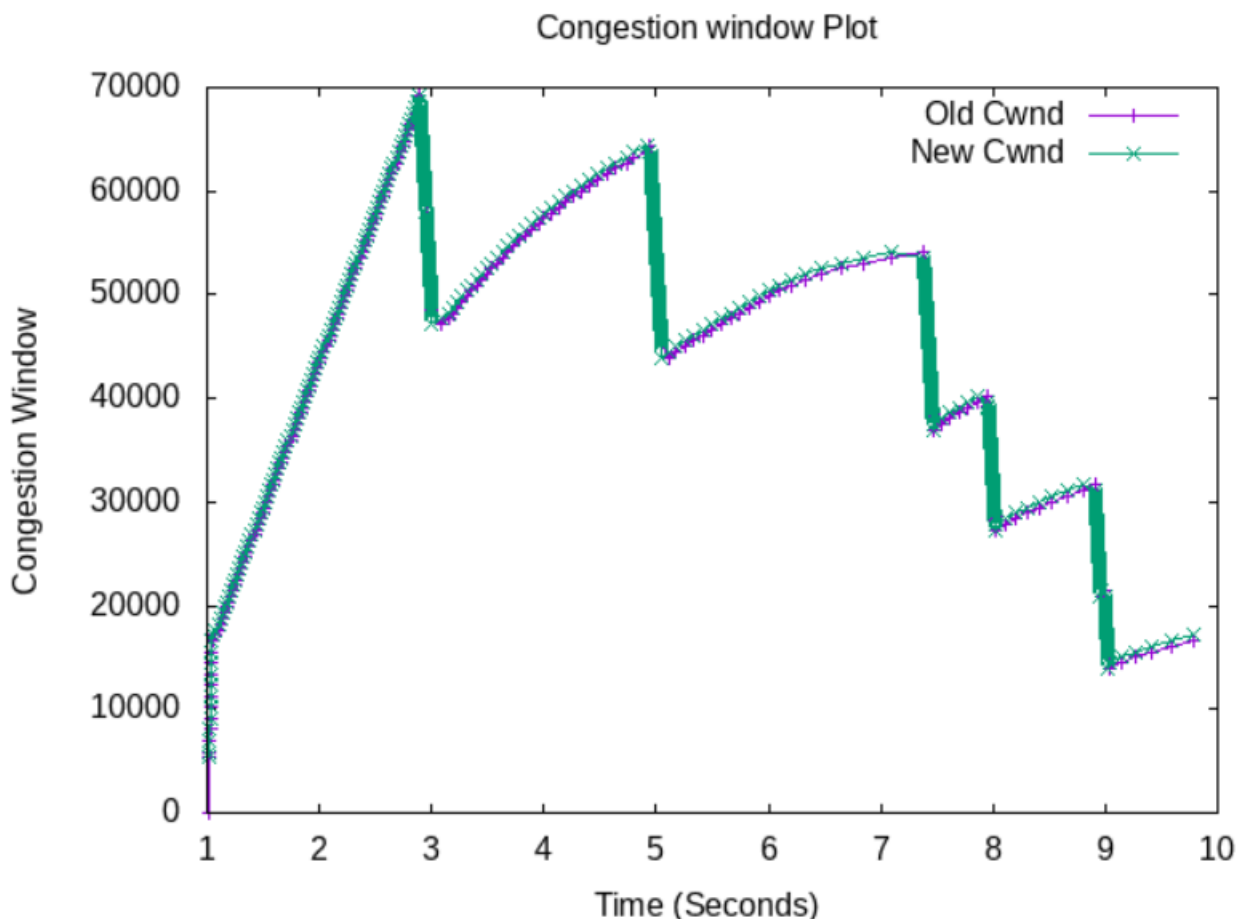
For the 1st 4 drops in cwnd, there was only a single duplicate ACK because the Additive Increase mechanism takes over post the cwnd drop. The last 2 drops are spaced by the difference of a single packet which must be the 2nd duplicate ACK as the cwnd is drastically reduced after the duplicate ACK is received.

So, there are 6 total cwnd drops in the NewReno NS-3 simulation.

Coming to the TCP Cubic variant, I have mentioned the CWND formula above and pasted the same below.

$$CWND(t) = C \times (t - K)^3 + W_{max} \quad \text{where} \quad K = \sqrt[3]{W_{max} \times (1 - \beta) / C}$$

If you look at the cwnd obtained by gnu plot of the NS-3 simulation of the Cubic connection:



TCP Cubic does not follow the AIMD mechanism to control the cwnd. The increment in the congestion window is described by the cubic function mentioned above, however, the decrement in the window size is done using the multiplicative decrease.

In the above cubic expression, C is the scaling constant and β is the multiplicative decrease factor (Generally $\beta = 0.7$). As $\beta = 0.7$, we can see that the cwnd is dropped by the factor of β whenever the cwnd. The meaning is that cwnd is always reduced by the constant factor of $\beta = 0.7$.

If you compare both the graphs i.e., the NewReno and Cubic, the last 4 cwnd drops happen at the same time. This happens because of the increase in the congestion in the network as well as the receipt of duplicate ACKs in the network (**Note:** There are 2 duplicate ACKs just before the 9th second as previously described).

However, there is a major change in the 1st 2 cwnd drops as they happen at different times. In NewReno as previously mentioned, cwnd is dropped when there is a duplicate ACK. Cubic does not follow this protocol and decides SACK (Selective ACK).

Cubic notices the fact that responding to the ACK will force it to reduce cwnd size when it has a lot of bandwidth available. When the 2nd instance of the duplicate ACK comes at $t = 3$, then Cubic knows that it needs to ACK the 1st instance otherwise the congestion control mechanism to drop the cwnd size severely. It immediately sends the 1st packet on receiving the 2nd instance.

The 2nd instance of duplicate ACK received at $t = 3$ is sent after 2 seconds. It is not immediately sent as Cubic knows it can exploit available bandwidth till it reaches the W_{max} and after closely acquiring the available bandwidth, it sends the packet selectively (SACK, again). This happens around $t = 5$.

CUBIC is an intelligent protocol that keeps changing decisions based on the network scenario at the interval of each millisecond making it difficult to deterministically predict the behaviour of the algorithm. It acquires more bandwidth before the SACK because if the cwnd size is low, it will be very small after the multiplicative decrease is applied. This helps it to collect more bandwidth to recover from the congestion to transmit more packets.

In the last 4 duplicate ACK where the `duplicate_ack_num == 1`, it does not see the necessity to consume more bandwidth as most of the transmission is complete. It immediately transmits the packet and reduces cwnd by the $\beta = 0.7$ factor.

Question 2

Among the TCP variants of NewReno and Cubic, the TCP Variant CUBIC is better. Cubic uses a modern mechanism for packet loss handling which can quickly and effectively use bandwidth for large transmission and slowdown in high congestion scenarios. It can work in an optimized scenario in all sorts of networks congestion and uses packet loss as a congestion handling mechanism.

If the above graphs are looked at, TCP Cubic makes intelligent decisions on when to capture bandwidth and uses SACK only when it feels that delaying it will backfire. It maintains a certain degree of fairness while capturing the bandwidth. In the graph, cwnd for Cubic varies from 3000-6000 whereas NewReno varies from 2000-8000 for the same interval.

Reduction cwnd by half for every duplicate ACK in NewReno is not a very good technique for the high congestion scenario. The current scenario transmits a lot of data in a very short time about 50000 bytes in a second. Cubic is designed for transmission in high congestion scenarios and intelligently adapts to the network scenarios by predating bandwidth exponentially when required.

NewReno fluctuation is too much for long computations and long connections. NewReno is just an extension of Reno to handle multiple packet losses simultaneously. It is a protocol to handle low bandwidth in short-duration connections.

A small comparative study:

TCP Variants & Year	Base variant	Mod. @ ¹	CD ²	N/w Env. ³	Added/Changed Modes or Features	Congestion Avoidance Method $cwnd_{new}$
TCP New Reno 1999	Reno	Sender	Loss	Wired	Fast Recovery resistant to multiple losses	AI $cwnd_{new} = \alpha + \beta$ $\alpha = cwnd_{old}$, $\beta = \text{Quotient}$
TCP Cubic 2008	BIC	Sender	Loss	HS/LD	The congestion window control as a cubic function of time elapsed since a last congestion event	Equation Based $cwnd$

NewReno treats every event as a new event and takes decisions. It does not account for any data from the previous congestion scenario and uses it to make better decisions in the future. For this case, the bandwidth goes up beyond 100000 and then comes down following AIMD after every duplicate ACK with no fairness in maintaining bandwidth consistently.

Cubic is the best protocol for this case as it is designed to deal with a high congestion network where a large amount of data needs to be transferred. A fair mechanism to deal with duplicate ACK is by using SACK when required to control the bandwidth fluctuations.

NewReno using the same algorithm in every situation for such high bandwidth scenario is the perfect recipe for failure. Its purpose was never intended for transmission exceeding 1 Mbps; due to which the graph is not distributing packets fairly and fluctuating the cwnd by large amounts.

NewReno was built for the 1990s before the advent of the 21st century and is suited for old systems and conditions prevalent at that time. We are in 2021, 20 years after the advent of the 21st century. Cubic came in 2008 and has gone significant updates. It has been prevalent for the past 10 years and is well apt for this case where we reasonably deal with high bandwidth.

Thus, we can conclude that the CUBIC TCP Variant is better.