CS 331: Computer Networks

Assignment 2

Team 12: Hrriday Ruparel (22110099), Kishan Ved (22110122)

Repository Link

Task 1: Comparison of congestion control schemes (BBR, Westwood, Scalable)

Part A:

Run the client on H1 and the server on H7. Measure the below parameters and summarize the observations for the three congestion schemes as applicable.

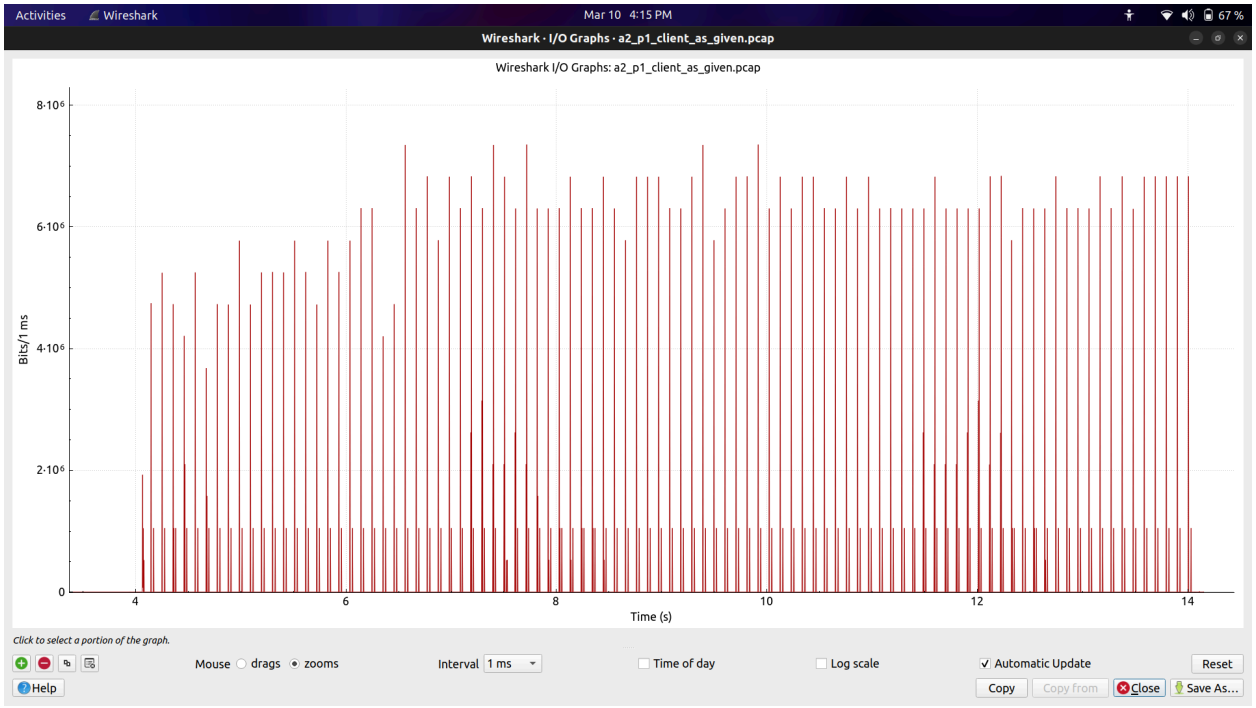
1. Throughput over time (with valid Wireshark I/O graphs)
2. Goodput
3. Packet loss rate
4. Maximum window size achieved (with valid Wireshark I/O graphs).

For this part, we have used time as 10s because running the code for time more than 10s causes the machine to hang and no output occurs.
Command used: iperf3 -c {h7.IP()} -p 5201 -b 10M -P 10 -t 10 -C {ccs} -V

BBR

Throughput:

Wireshark IO graphs generated for client's pcap:
(tcp && ip.src == 10.0.0.1)



Dips show congestion. The throughput drops when congestion occurs and then tries to increase slowly, until it plateaus (max value is reached). BBR maintains high throughput consistently as it models the network's bandwidth and delay. It does not reduce congestion window aggressively on packet loss.

Throughput: Total bytes for TCP packets where destination IP is 10.0.0.7: **125792663 bytes**

Statistics

Measurement	Captured	Displayed	Marked
Packets	5752	3053 (53.1%)	—
Time span, s	16.562	10.211	—
Average pps	347.3	299.0	—
Average packet size, B	21902	41203	—
Bytes	125978536	125792663 (99.9%)	0
Average bytes/s	7,606 k	12 M	—
Average bits/s	60 M	98 M	—

(image is from client's pcap)

Goodput:

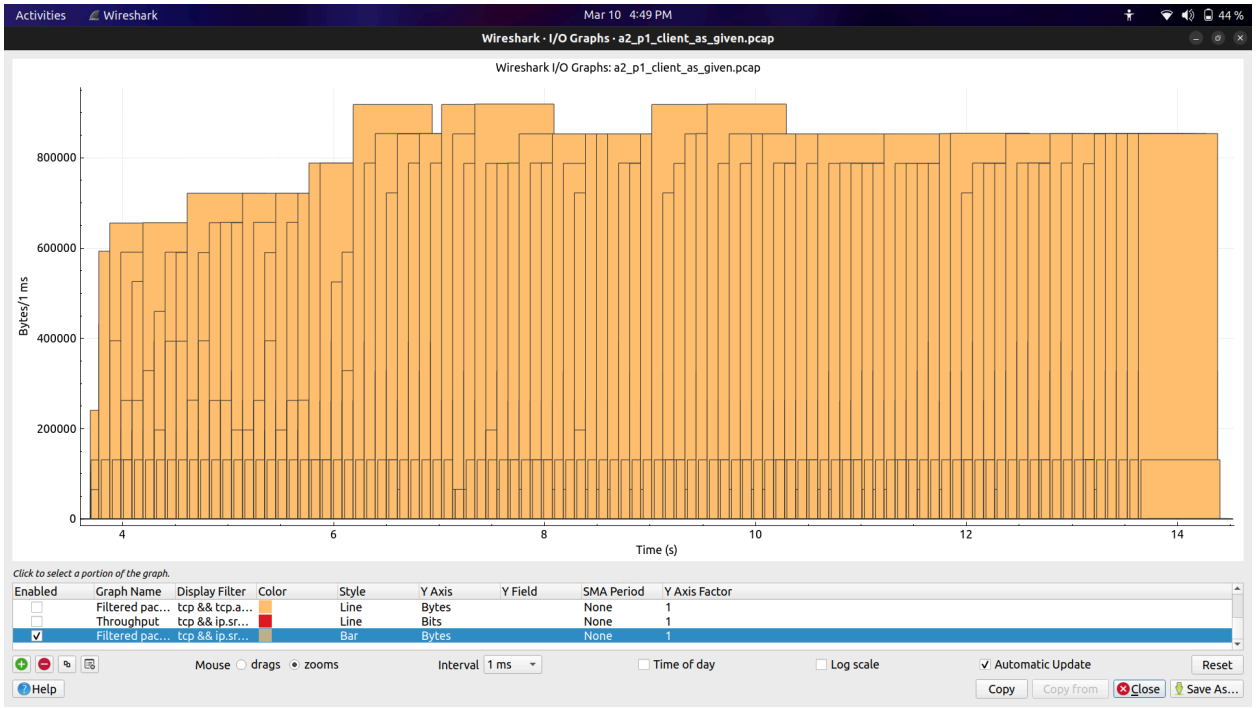
Total goodput bytes for TCP packets sent from sender: 10.0.0.1 (excluding retransmissions): **125572253 bytes**

Packet Loss Rate:

At the sender's (client's) end:
Total Sent: 3053
Distinct packets Sent: 3017
Packets Lost: 36

Packet Loss Rate (Packets lost/Total Packet sent): 1.18

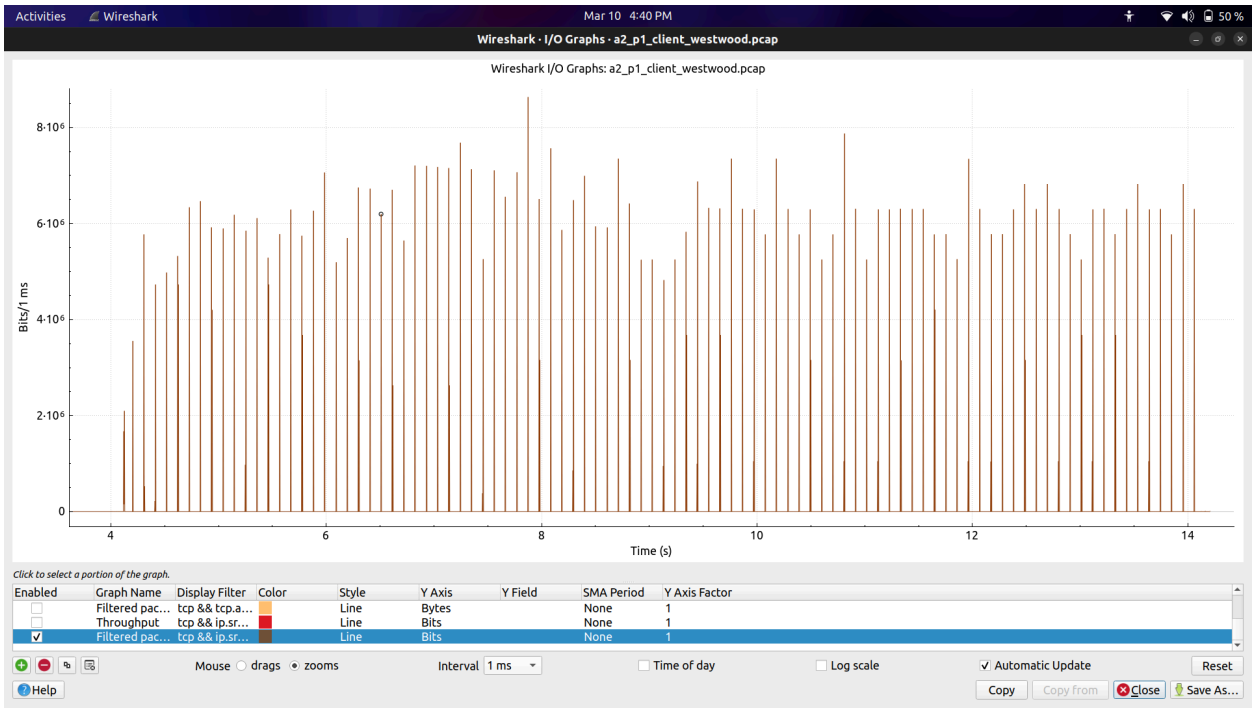
Window size: tcp && ip.dst==10.0.0.7 && tcp.window_size:



Westwood

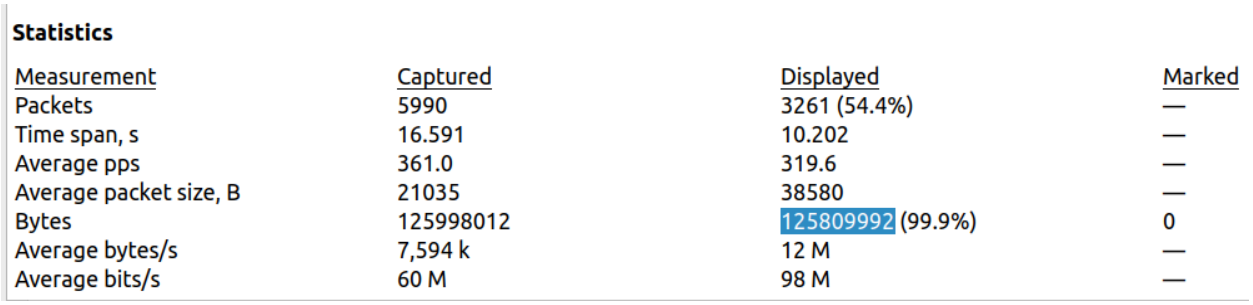
Throughput:

Wireshark IO graphs generated for client's pcap



Westwood adapts congestion window based on end-to-end bandwidth estimation. It recovers better than Reno but may experience slightly fluctuating throughput.

Throughput: Total bytes for TCP packets where destination IP is 10.0.0.7: **125809992 bytes**



Statistics

Measurement	Captured	Displayed	Marked
Packets	5990	3261 (54.4%)	—
Time span, s	16.591	10.202	—
Average pps	361.0	319.6	—
Average packet size, B	21035	38580	—
Bytes	125998012	125809992 (99.9%)	0
Average bytes/s	7,594 k	12 M	—
Average bits/s	60 M	98 M	—

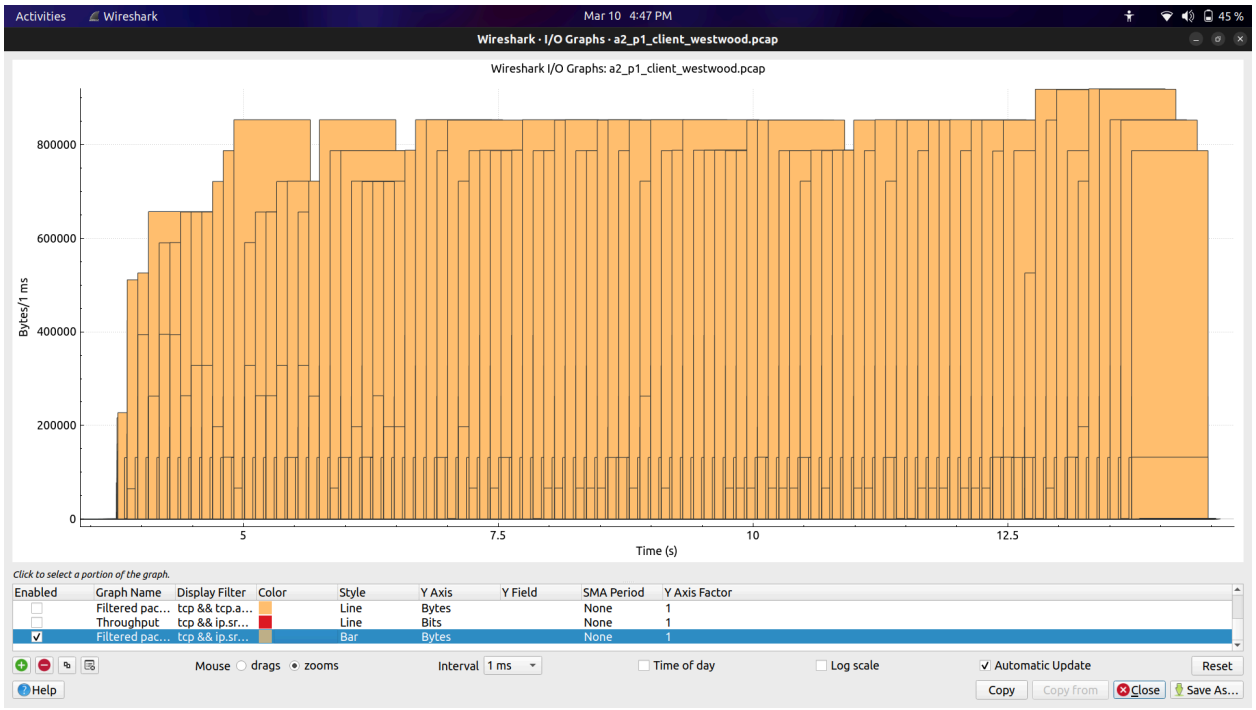
(image is from client's pcap)

Goodput:
Total goodput bytes for TCP packets sent from 10.0.0.1 (excluding retransmissions): **125801269 bytes**

Packet Loss Rate:
At the sender's (client's) end:

Total Sent: 3069
Distinct packets Sent: 3040
Packets Lost: 29
Packet Loss Rate (Packets lost/Total Packet sent): 0.94%

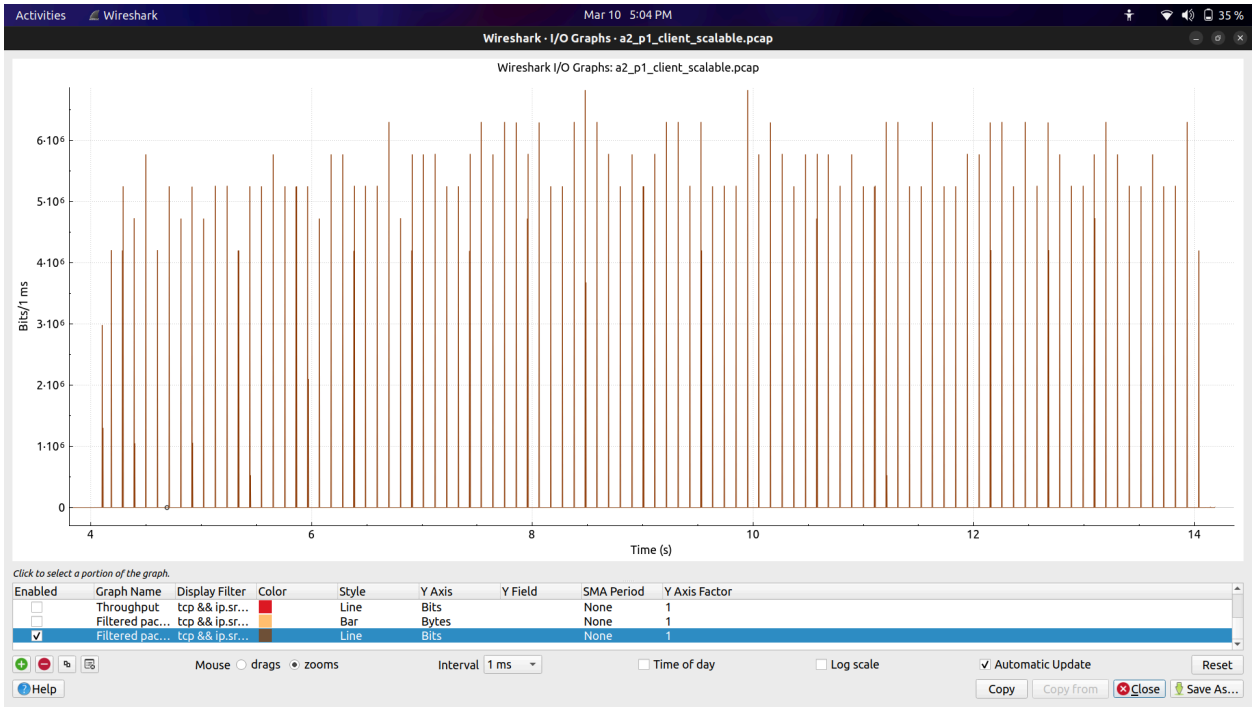
Window size: tcp && ip.dst==10.0.0.7 && tcp.window_size



Scalable

Throughput:

Wireshark IO graphs generated for client's pcap:
(tcp && ip.src == 10.0.0.1)



Scalable is designed for high-speed networks, it aggressively increases the congestion window in high-bandwidth environments but can lead to instability under varying network conditions.

Throughput: Total bytes for TCP packets where destination IP is 10.0.0.7: **125831755 bytes**

Statistics

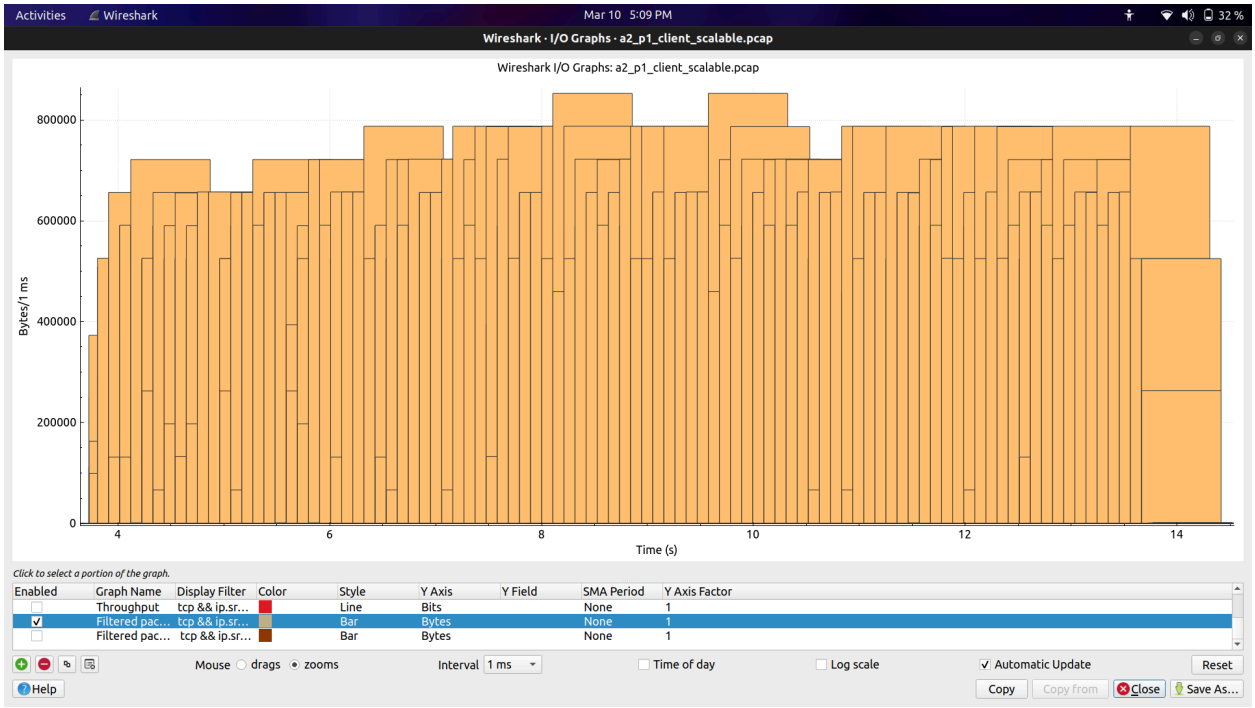
Measurement	Captured	Displayed	Marked
Packets	5969	3152 (52.8%)	—
Time span, s	16.757	10.209	—
Average pps	356.2	308.8	—
Average packet size, B	21113	39921	—
Bytes	126025624	125831755 (99.8%)	0
Average bytes/s	7,520 k	12 M	—
Average bits/s	60 M	98 M	—

Goodput:
Total goodput bytes for TCP packets sent from 10.0.0.1 (excluding retransmissions): **125606259 bytes**

Packet Loss Rate:
Total Sent: 3152
Distinct packets Sent: 3110

Packets Lost: 42
Packet Loss Rate (Packets lost/Total Packet sent): 1.33%
tcp && ip.src == 10.0.0.1 && tcp.window_size

Window size: tcp && ip.dst==10.0.0.7&& tcp.window_size



Comparisons based on observations above

Feature	BBR	Westwood	Scalable
Throughput	125,792,663 bytes	125,809,992 bytes	125,831,755 bytes (highest)
Goodput	125,572,253 bytes	125,801,269 bytes (highest)	125,606,259 bytes
Retransmission Overhead	220,410 bytes (moderate)	8,723 bytes (lowest)	225,496 bytes (highest)
Packet Loss Rate	1.18% (moderate)	0.94% (lowest)	1.33% (highest)
Total Packets Sent	3,053	3,069	3,152
Distinct Packets	3,017	3,040	3,110
Congestion Control Approach	Probes bandwidth using RTT and pacing	Estimates bandwidth based on returning ACKs	Aggressive multiplicative increase
Behavior on Congestion	Drops throughput significantly, then recovers gradually	Smooth adaptation, reducing retransmissions	Quickly increases bandwidth usage, leading to higher loss

Window Size Adjustment	Adjusts based on bandwidth and RTT	Adapts using bandwidth estimation	Aggressive increase after congestion events
Best Use Case	Balanced performance and adaptability	Best for networks which need high reliability	Suitable for high-speed networks with tolerable loss

Westwood is the most efficient, with the highest goodput and lowest packet loss, making it ideal for networks that need reliability.

Scalable achieves the highest throughput but suffers from the most retransmissions, making it better for high-speed environments where loss can be tolerated.

BBR is between these 2

Part B:

Run the clients on H1, H3 and H4 in a staggered manner(H1 starts at T=0s and runs for 150s, H3 at T=15s and runs for T=120s, H4 at T=30s and runs for 90s) and the server on H7. Measure the parameters listed in part (a) and explain the observations, for the 3 congestion schemes for all the three flows.

For this part, we have used time as 15s, 12s and 9s, and delay as 1.5s (all times are divided by 10) as if the code is run for a longer duration, the computer hangs. Moreover, different clients are connected via different ports to the server, this is because iperf3 (on Ubuntu) does not support multiple clients on a single server port (this is mentioned in the documentation)

The following code was used to achieve this:

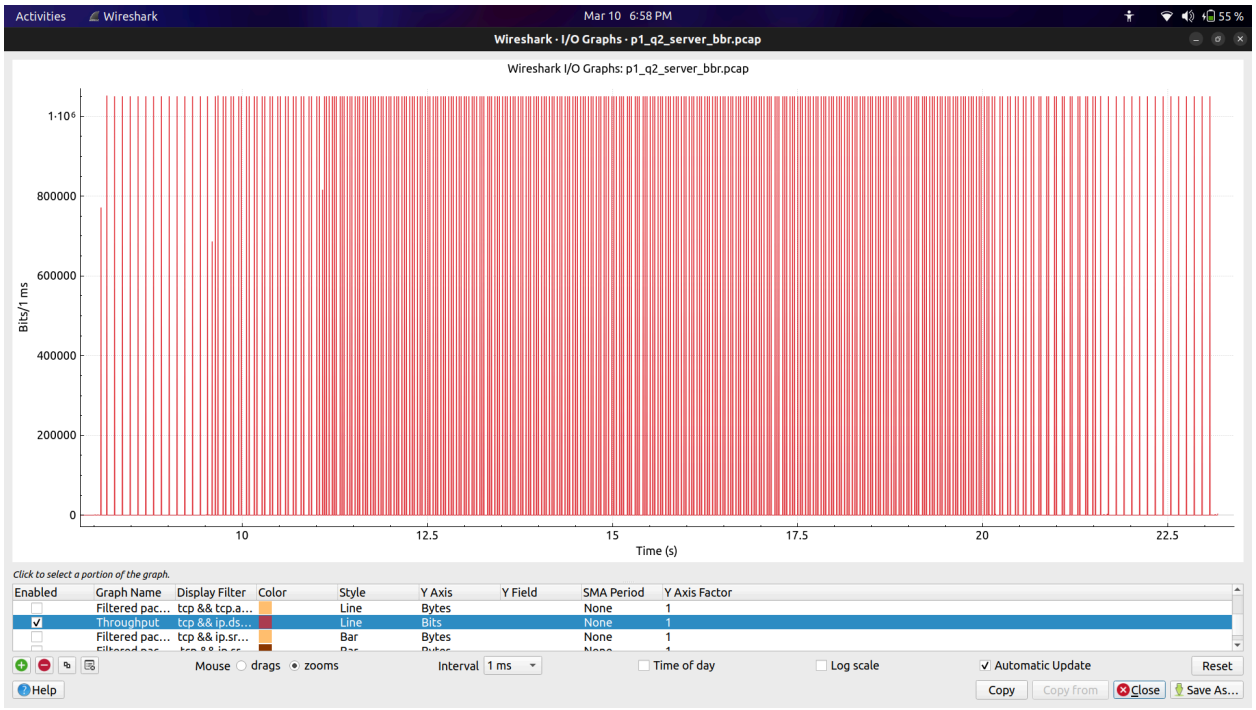
```
h7.cmd("iperf3 -s -p 5201 &")
h7.cmd("iperf3 -s -p 5202 &")
h7.cmd("iperf3 -s -p 5203 &")

client_cmd_h1 = f"iperf3 -c {h7.IP()} -p 5201 -b 10M -P 10 -t 15 -C {ccs} -V &"
client_cmd_h3 = f"iperf3 -c {h7.IP()} -p 5202 -b 10M -P 10 -t 12 -C {ccs} -V &"
client_cmd_h4 = f"iperf3 -c {h7.IP()} -p 5203 -b 10M -P 10 -t 9 -C {ccs} -V &"
```

BBR:

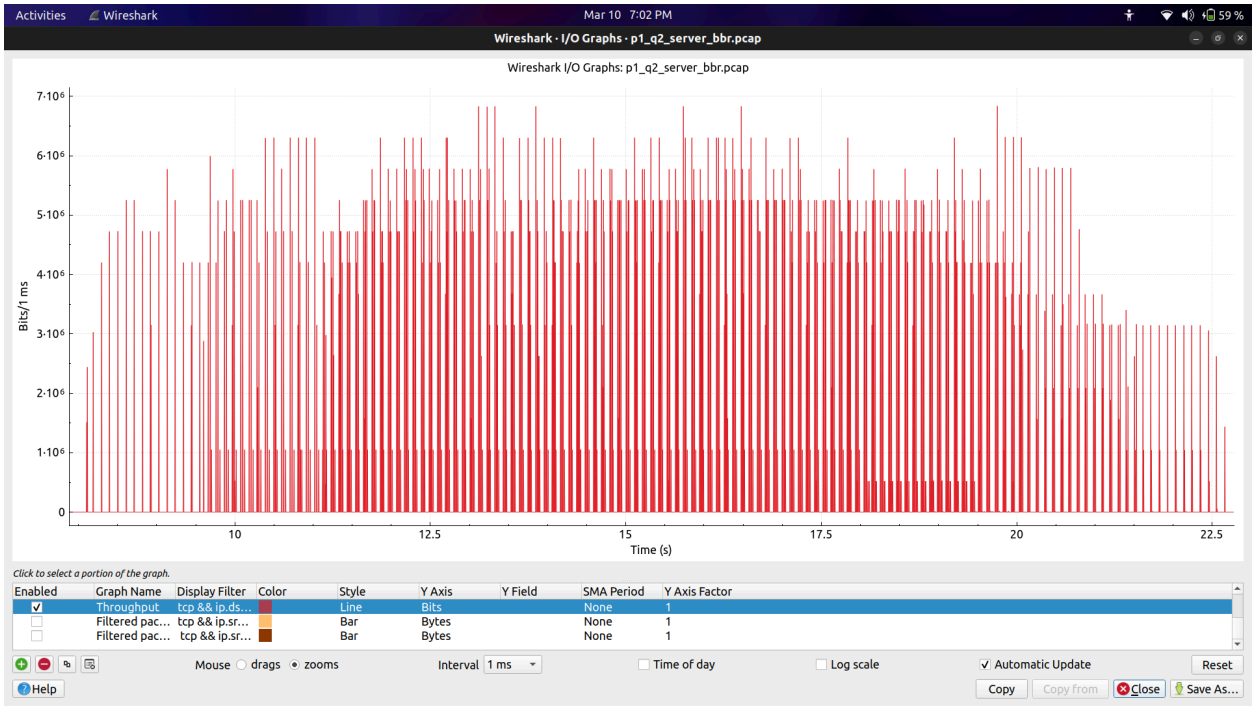
Throughput:

Wireshark IO graphs generated for server's pcap (tcp && ip.dst==10.0.0.7)

On running 1 process per client

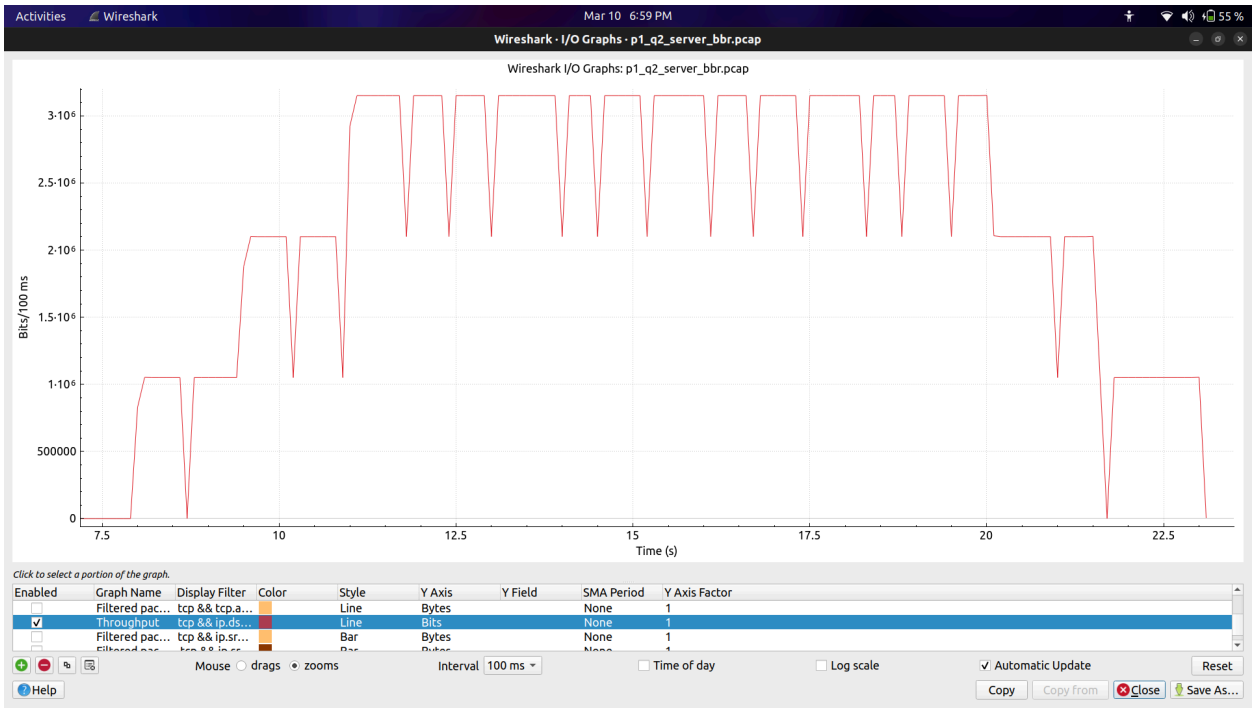


On running 10 processes per client



The same graphs with a change in the y axis's interval:

On running only 1 process per client



On running 10 processes per client

Now, all results are for 10 processes per client
Throughput: Total bytes for TCP packets where destination IP is 10.0.0.7: **419409684 bytes**

Statistics

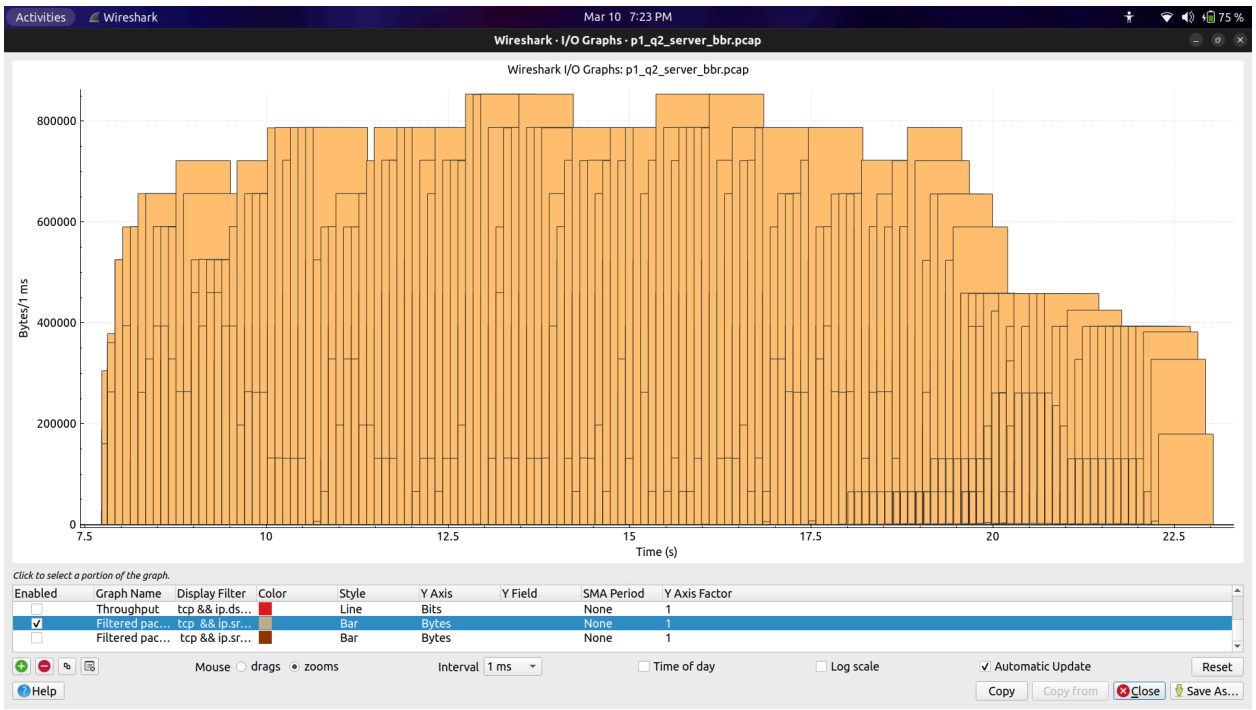
Measurement	Captured	Displayed	Marked
Packets	18464	10489 (56.8%)	—
Time span, s	31.142	23.146	—
Average pps	592.9	453.2	—
Average packet size, B	22747	39986	—
Bytes	419998047	419409684 (99.9%)	0
Average bytes/s	13 M	18 M	—
Average bits/s	107 M	144 M	—

(image is from client's pcap)

Goodput:
Total goodput bytes for TCP packets sent from 10.0.0.1 (excluding retransmissions): **418448836 bytes**

Packet Loss Rate:
Total Sent: 10489
Distinct packets Sent: 10074
Packets Lost: 415
Packet Loss Rate (Packets lost/Total Packet sent): 3.96%

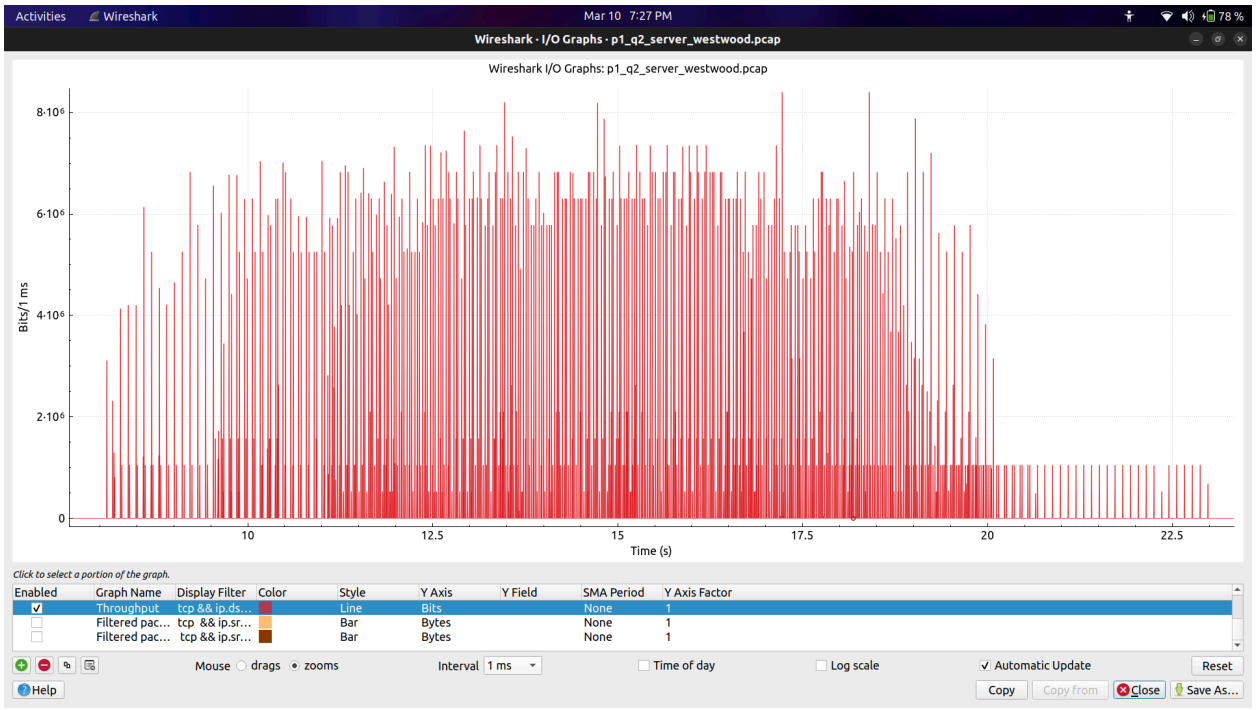
Window size: tcp && ip.dst==10.0.0.7 && tcp.window_size



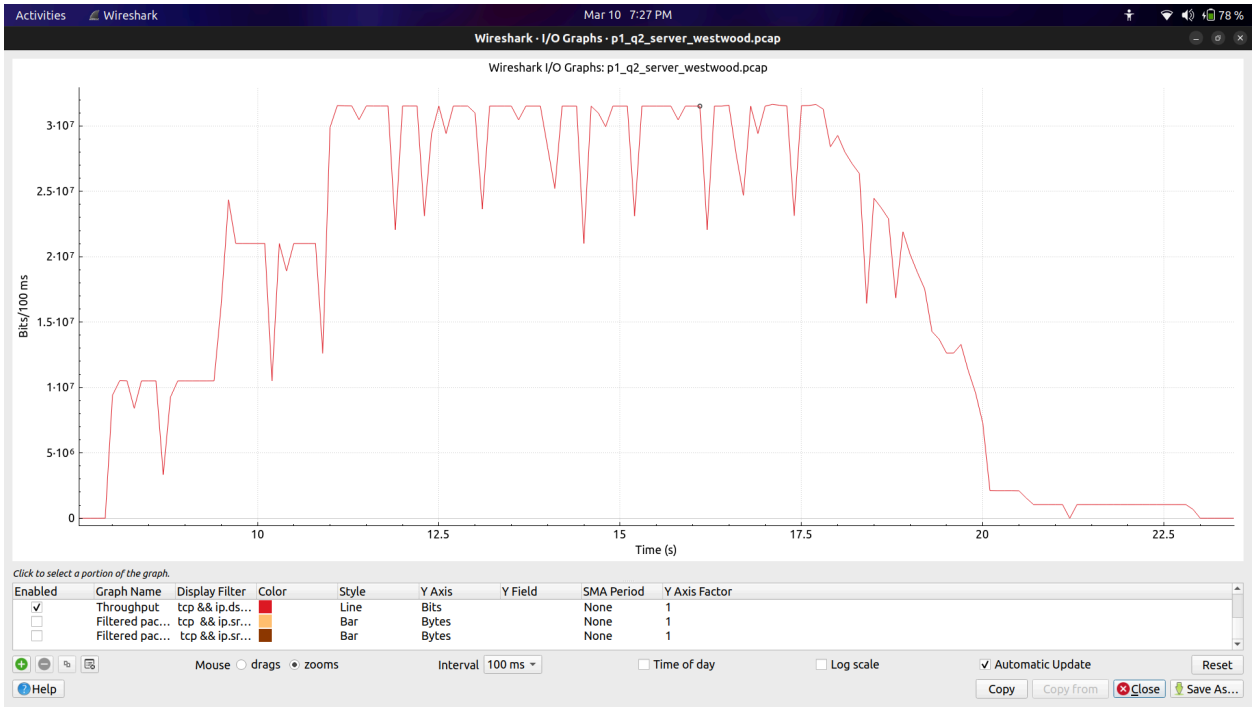
Westwood

Throughput:

Wireshark IO graphs generated for client's pcap



The same graphs with a change in the y axis's interval:



Throughput: Total bytes for TCP packets where destination IP is 10.0.0.7: **3710944969 bytes**

Statistics

Measurement	Captured	Displayed	Marked
Packets	18389	10338 (56.2%)	—
Time span, s	30.710	22.656	—
Average pps	598.8	456.3	—
Average packet size, B	20212	35896	—
Bytes	371686379	371094969 (99.8%)	0
Average bytes/s	12 M	16 M	—
Average bits/s	96 M	131 M	—

(image is from client's pcap)

Goodput:

Total goodput bytes for TCP packets sent from 10.0.0.1 (excluding retransmissions): **370093729 bytes**

Packet Loss Rate:

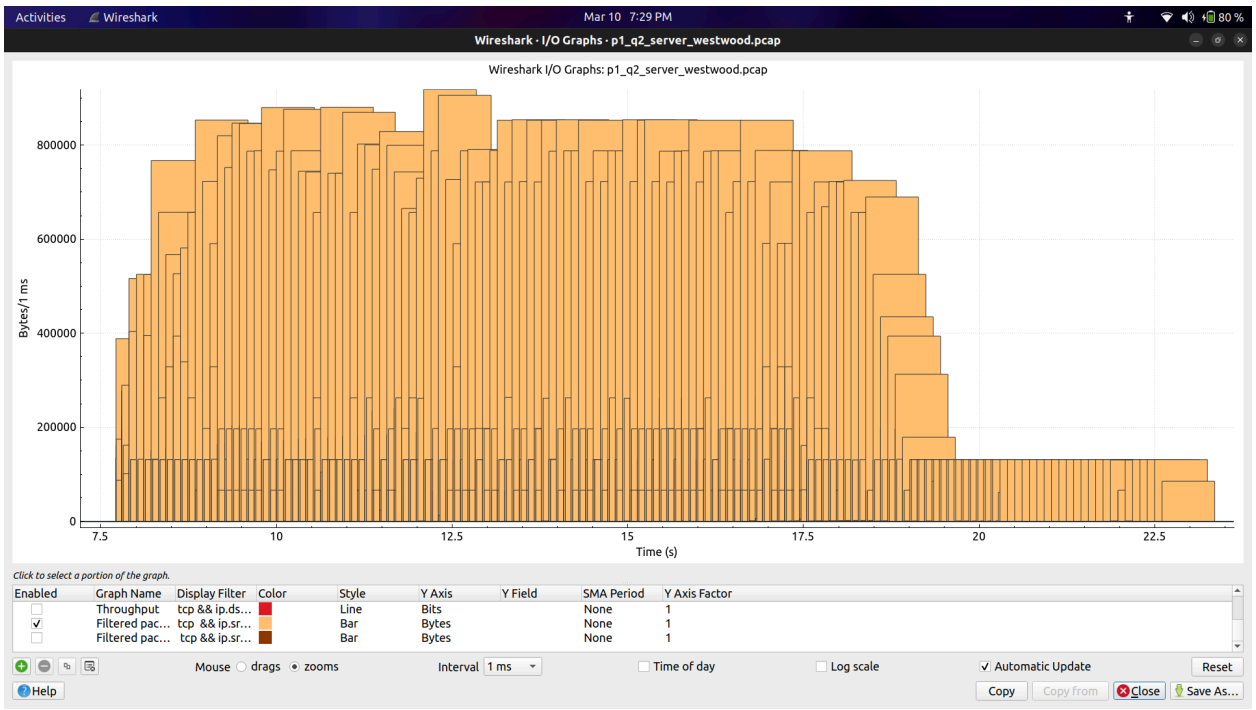
Total Sent: 10338

Distinct packets Sent: 9849

Packets Lost: 489

Packet Loss Rate (Packets lost/Total Packet sent): 4.73%

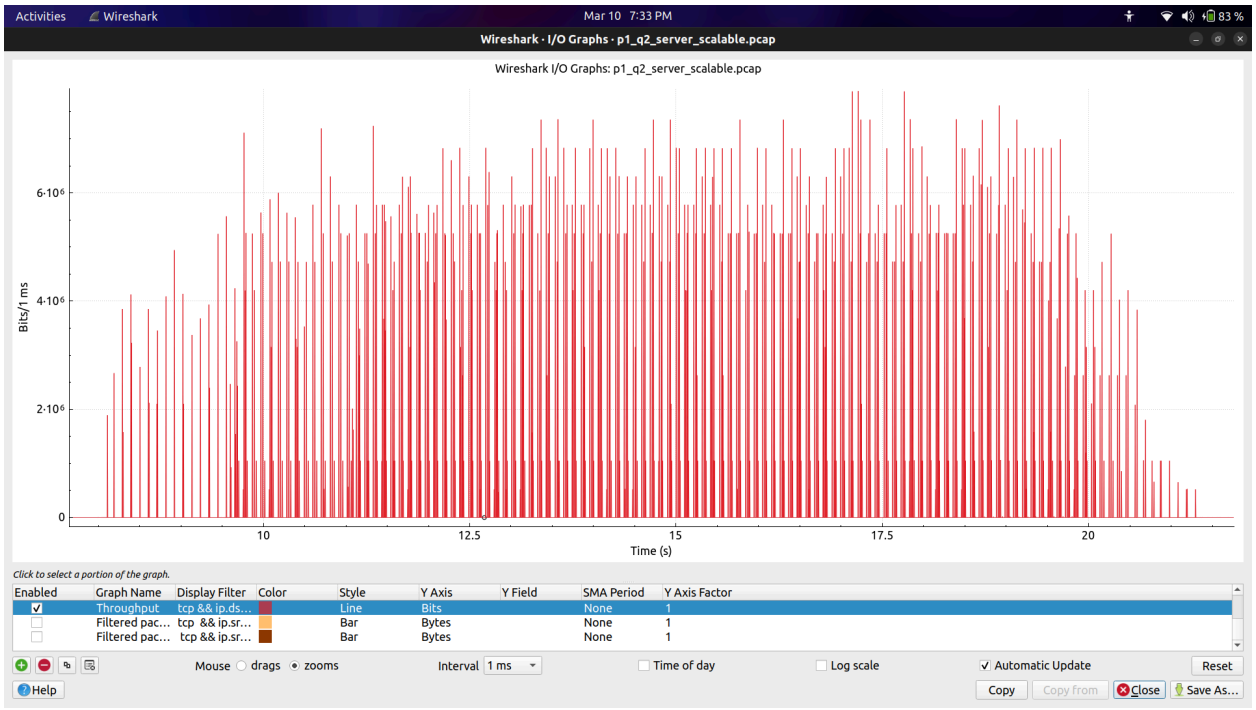
Window size: tcp && ip.dst==10.0.0.7 && tcp.window_size



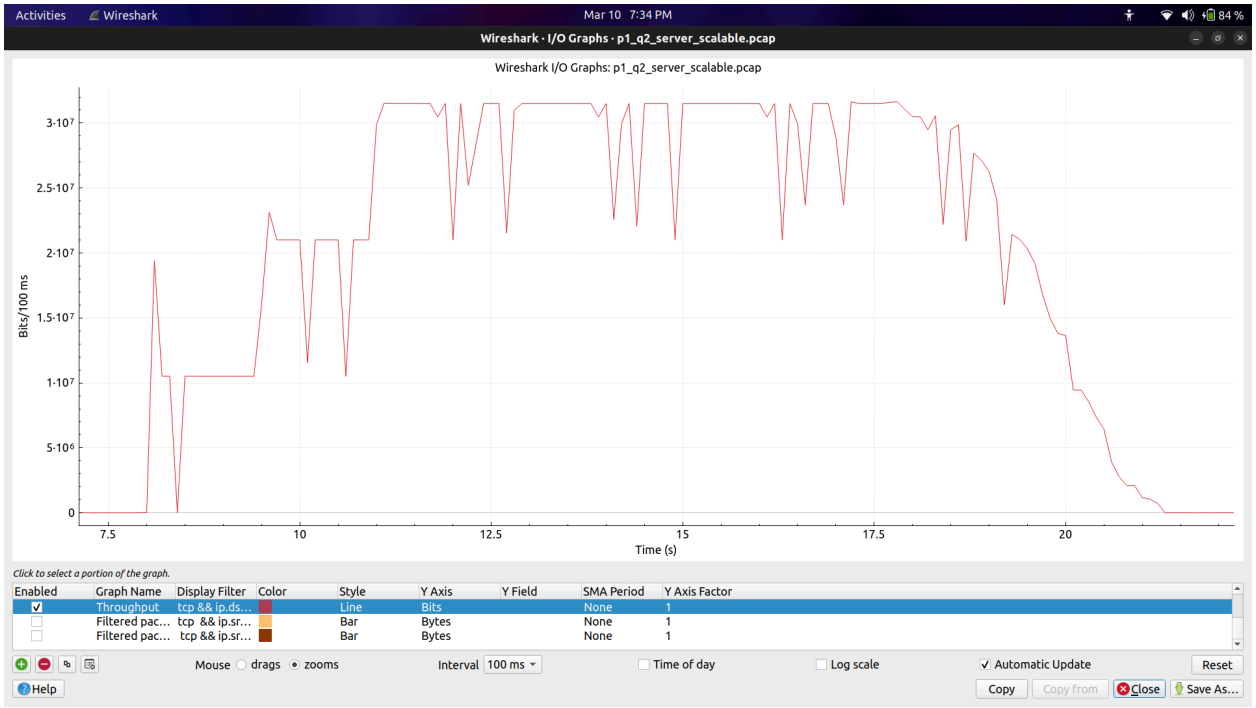
Scalable

Throughput:

Wireshark IO graphs generated for client's pcap



The same graphs with a change in the y axis's interval:



Throughput: Total bytes for TCP packets where destination IP is 10.0.0.7: **386066322 bytes**

Statistics

Measurement	Captured	Displayed	Marked
Packets	18464	10418 (56.4%)	—
Time span, s	31.354	23.120	—
Average pps	588.9	450.6	—
Average packet size, B	20941	37058	—
Bytes	386660298	386066322 (99.8%)	0
Average bytes/s	12 M	16 M	—
Average bits/s	98 M	133 M	—

(image is from client's pcap)

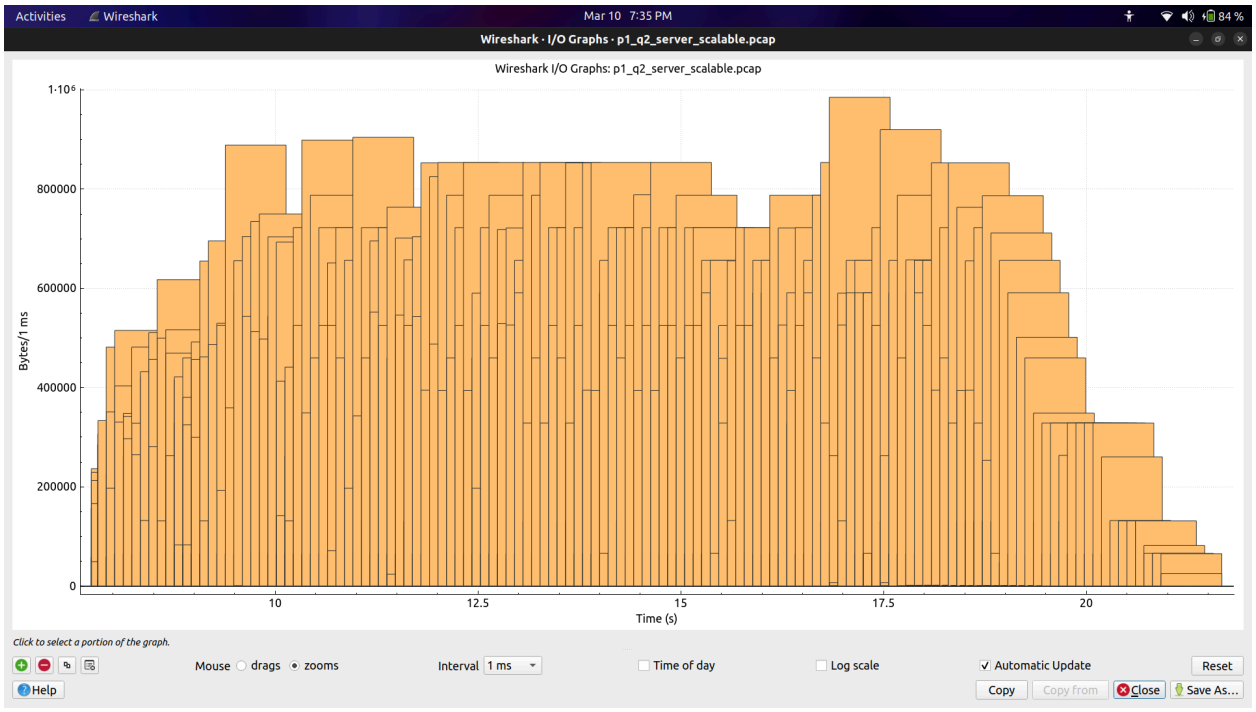
Goodput:

Total goodput bytes for TCP packets sent from 10.0.0.1 (excluding retransmissions): **385072794 bytes**

Packet Loss Rate:

Total Sent: 10418
Distinct packets Sent: 9927
Packets Lost: 491
Packet Loss Rate (Packets lost/Total Packet sent): 4.71%

Window size: tcp && ip.dst==10.0.0.7 && tcp.window_size



Comparisons based on observations above

Performance Metric	BBR	Westwood	Scalable
Throughput (bytes transferred)	419,409,684 bytes (144 Mbps)	371,094,969 bytes (131 Mbps)	386,066,322 bytes (133 Mbps)
Goodput (useful data transferred)	418,448,836 bytes (very low retransmission overhead)	370,093,729 bytes	385,072,794 bytes
Packet Loss Rate	3.96% (415 lost packets) (lowest)	4.73% (489 lost packets)	4.71% (491 lost packets)
Maximum Window Size Achieved	**Highest** – shows aggressive probing for bandwidth	Moderate – adjusts based on bandwidth estimation	Moderate-high – scales multiplicatively but less aggressively than BBR
Adaptability to Staggered Clients	**Best** – Maintains high throughput despite new client arrivals	Poor – conservative adaptation leads to lower bandwidth utilization	Moderate – manages congestion but it's not optimal
Bandwidth Use	**High**	Low	Medium (suboptimal)

Behavior in Congestion	Does not aggressively reduce window on packet loss, leading to higher throughput	Reduces congestion window conservatively, impacting performance	Aggressively increases window, leading to potential instability

BBR is better than the other schemes in terms of throughput, goodput, and bandwidth use. It is well-suited for dynamic network conditions with clients joining at staggered intervals. This is the best choice.

Westwood has the lowest throughput and highest packet loss, as its bandwidth estimation approach is less effective when multiple clients start at different times.

Scalable performs between BBR and westwood (its aggressive window scaling can lead to instability.)

Part C.

Configure the links with the following bandwidths:
I. Link S1-S2: 100Mbps
II. Links S2-S3: 50Mbps
III. Links S3-S4: 100Mbps

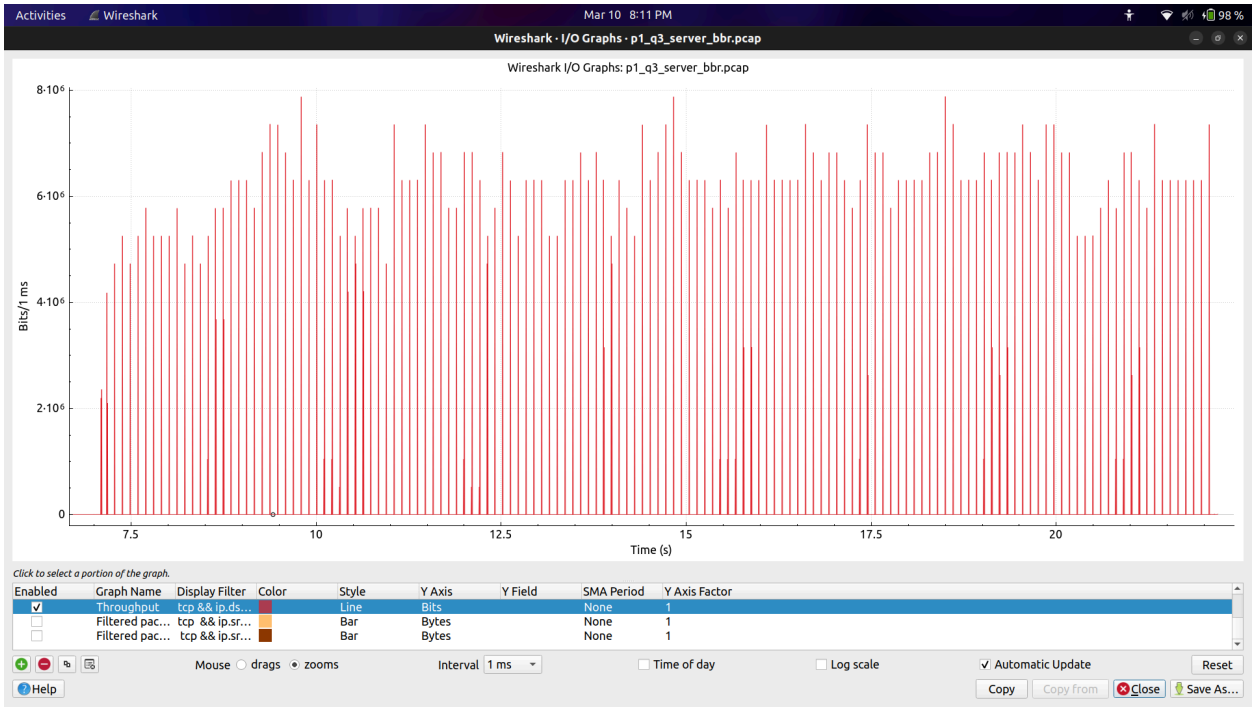
Measure the performance parameters listed in part (a) and explain the observations in the following conditions:
1. Link S2-S4 is active with client on H3 and server on H7

The client has 10 processes that is run for 15 seconds

BBR

Throughput:

Wireshark IO graph



Throughput: 188810458 bytes

Statistics

Measurement	Captured	Displayed	Marked
Packets	8589	4543 (52.9%)	—
Time span, s	22.176	15.187	—
Average pps	387.3	299.1	—
Average packet size, B	22018	41561	—
Bytes	189109625	188810458 (99.8%)	0
Average bytes/s	8,527 k	12 M	—
Average bits/s	68 M	99 M	—

Goodput:

Total goodput bytes for TCP packets sent from 10.0.0.1 (excluding retransmissions): **188474474 bytes**

Packet Loss Rate:

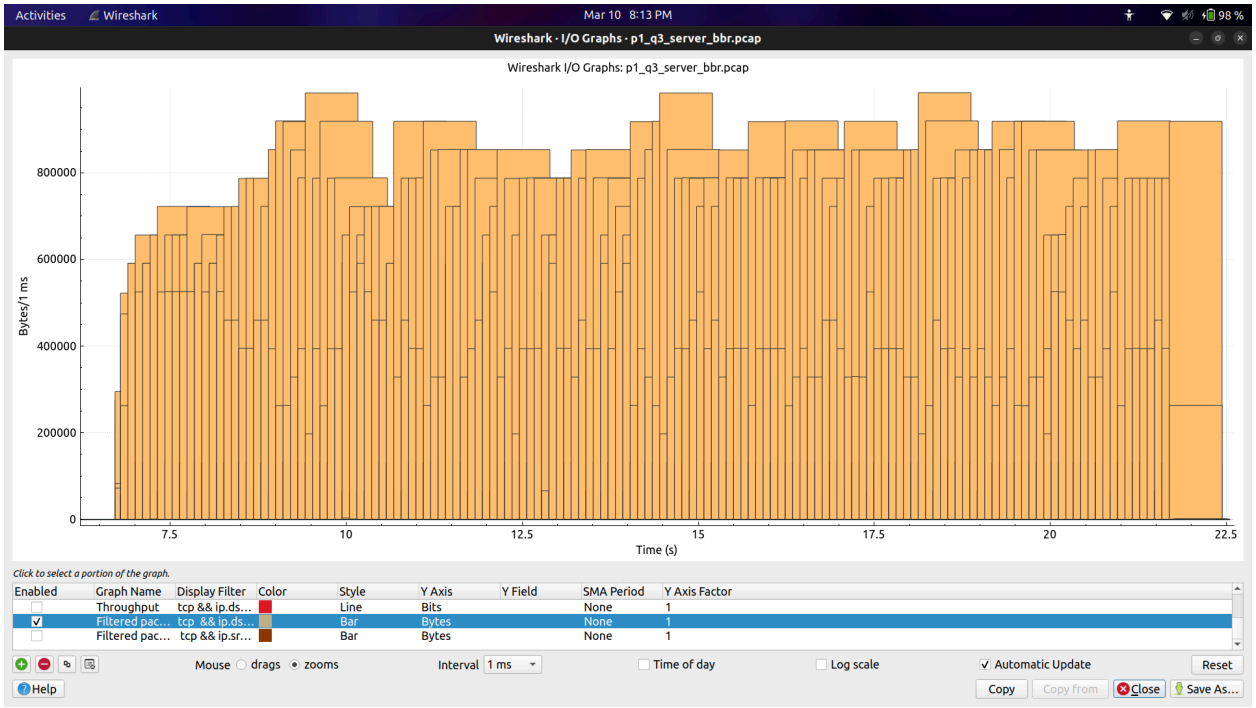
Total Sent: 4543

Distinct packets Sent: 4496

Packets Lost: 47

Packet Loss Rate (Packets lost/Total Packet sent): 1.03%

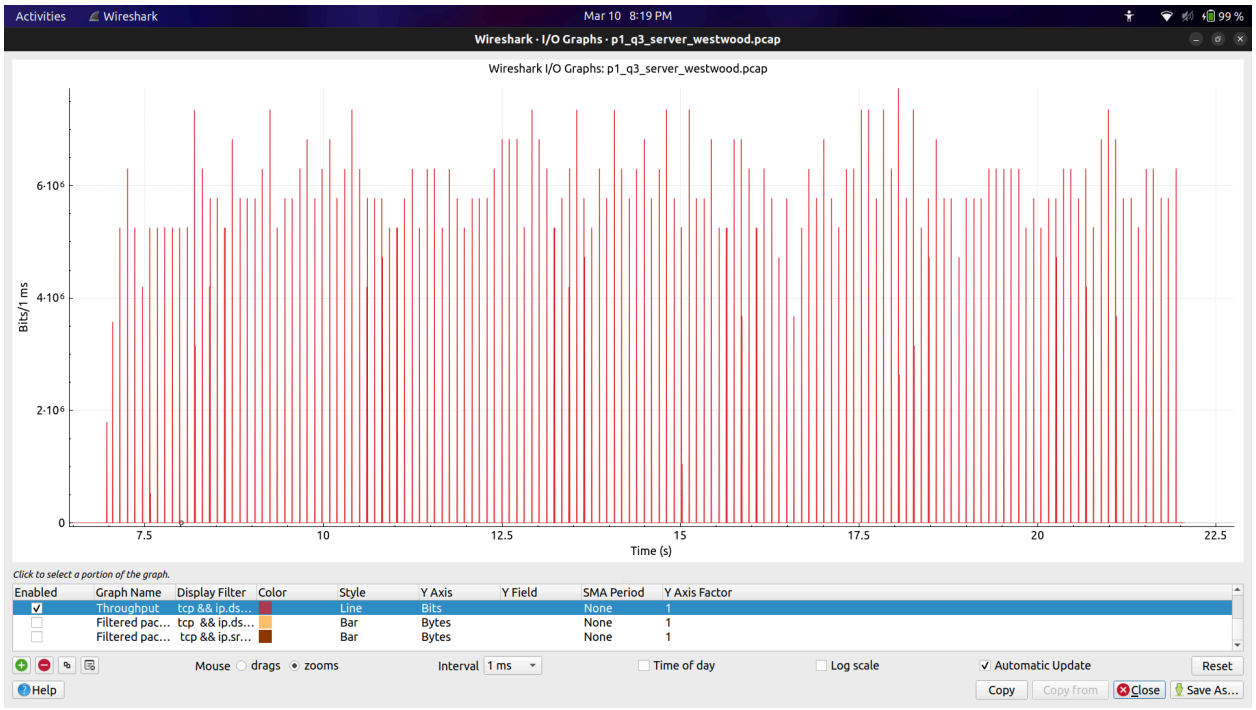
Window size: tcp && ip.dst==10.0.0.7 && tcp.window_size:



Westwood

Throughput:

Wireshark IO graph



Throughput: 188802733 bytes

Statistics

Measurement	Captured	Displayed	Marked
Packets	8540	4598 (53.8%)	—
Time span, s	26.939	15.183	—
Average pps	317.0	302.8	—
Average packet size, B	22142	41062	—
Bytes	189093689	188802733 (99.8%)	0
Average bytes/s	7,019 k	12 M	—
Average bits/s	56 M	99 M	—

Goodput:

Total goodput bytes for TCP packets sent from 10.0.0.1 (excluding retransmissions): **188457109 bytes**

Packet Loss Rate:

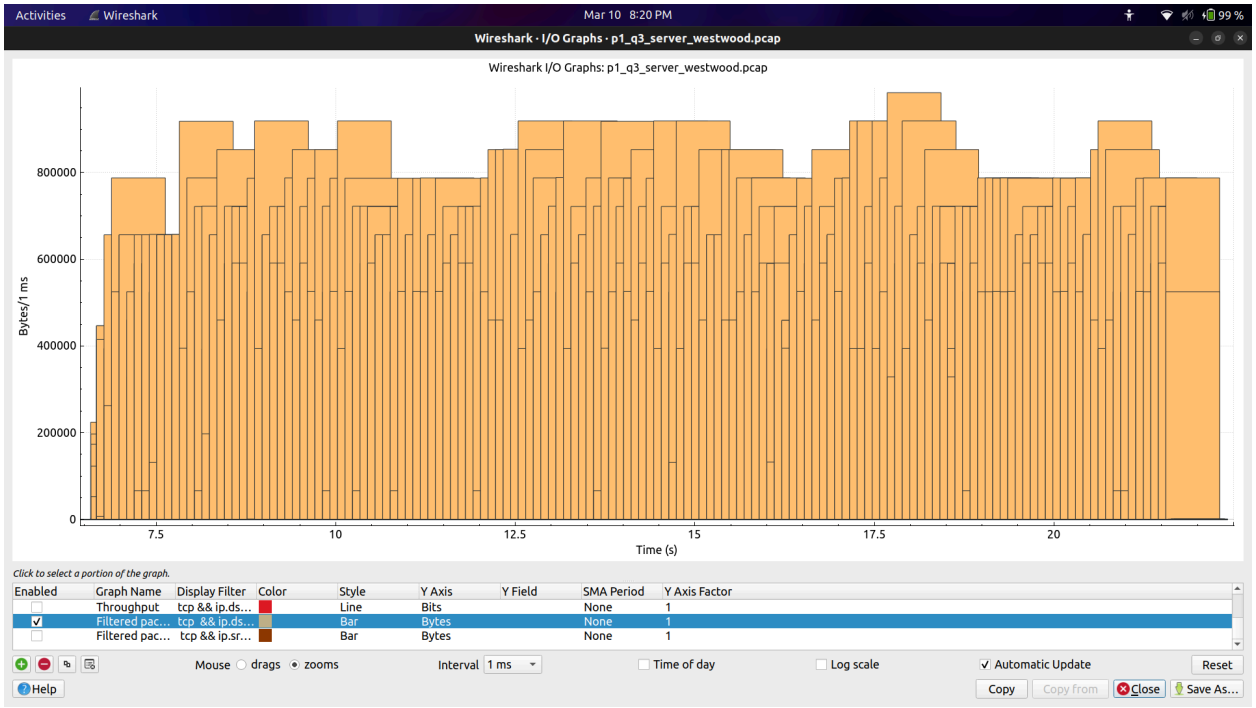
Total Sent: 4598

Distinct packets Sent: 4558

Packets Lost: 40

Packet Loss Rate (Packets lost/Total Packet sent): 0.87%

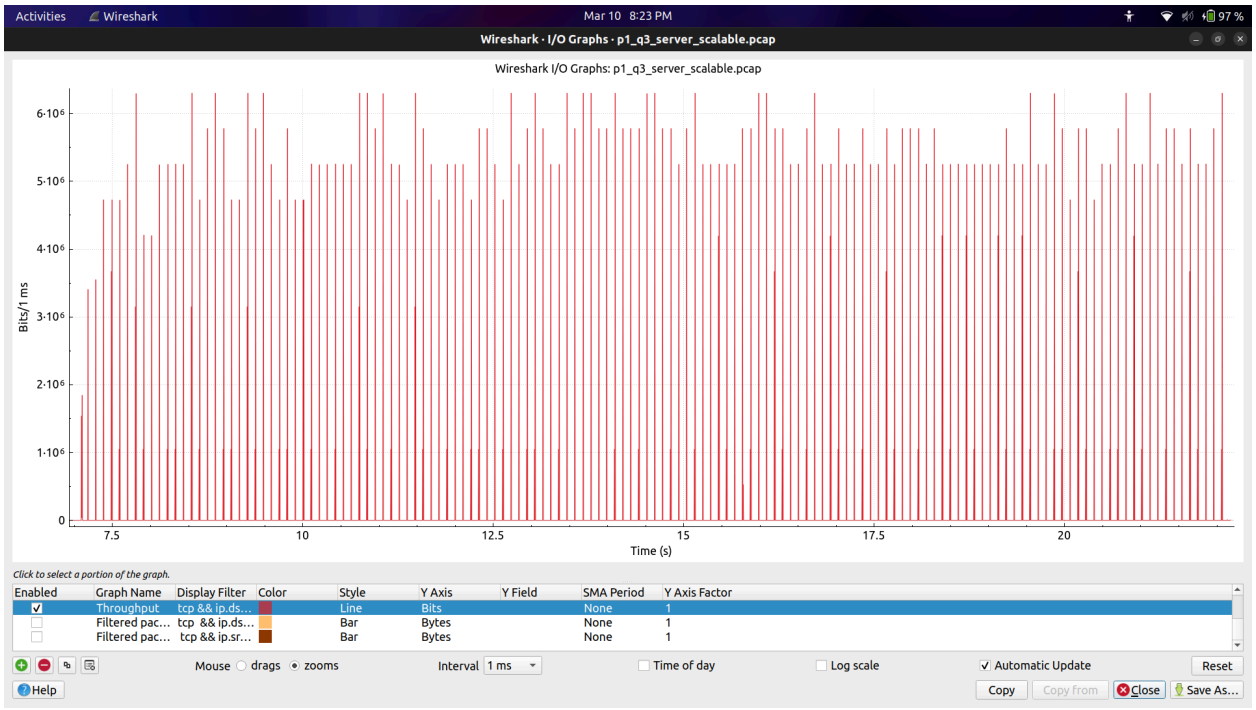
Window size: tcp && ip.dst==10.0.0.7 && tcp.window_size:



Scalable

Throughput:

Wireshark IO graph



Throughput:

Statistics

Measurement	Captured	Displayed	Marked
Packets	8530	4742 (55.6%)	—
Time span, s	22.177	15.186	—
Average pps	384.6	312.3	—
Average packet size, B	22177	39833	—
Bytes	189170191	188889091 (99.9%)	0
Average bytes/s	8,530 k	12 M	—
Average bits/s	68 M	99 M	—

Goodput:

Total goodput bytes for TCP packets sent from 10.0.0.1 (excluding retransmissions): **188528755 bytes**

Packet Loss Rate:

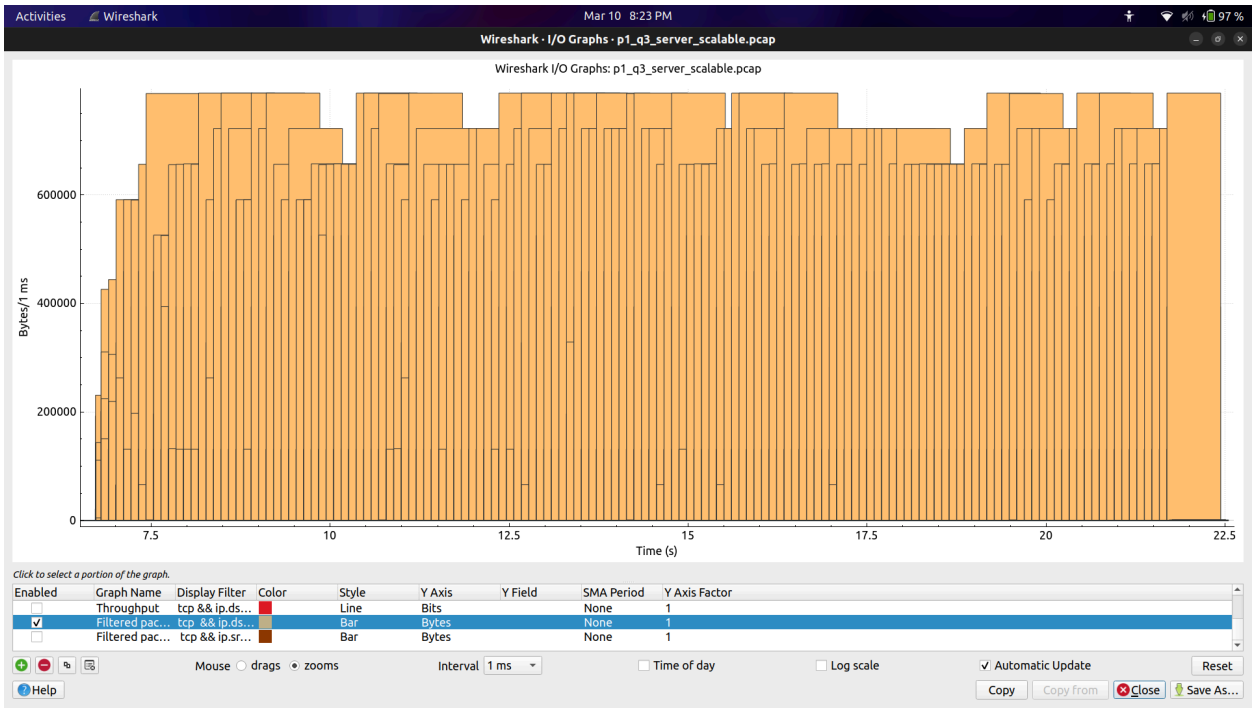
Total Sent: 4742

Distinct packets Sent: 4675

Packets Lost: 67

Packet Loss Rate (Packets lost/Total Packet sent): 1.41%

Window size: tcp && ip.dst==10.0.0.7 && tcp.window_size:



Observations

Difference is observed in packet loss

Westwood had the lowest packet loss (0.87%) – best for reliable data transfers.

BBR was slightly worse (1.03%), still decent.

Scalable had the highest packet loss (1.41%), likely due to its aggressive window size adjustments.

Useage

Use Westwood if packet loss matters (e.g., file transfers, video streaming).

Scalable gives slightly better throughput but at the cost of more lost packets.

BBR is between the 2

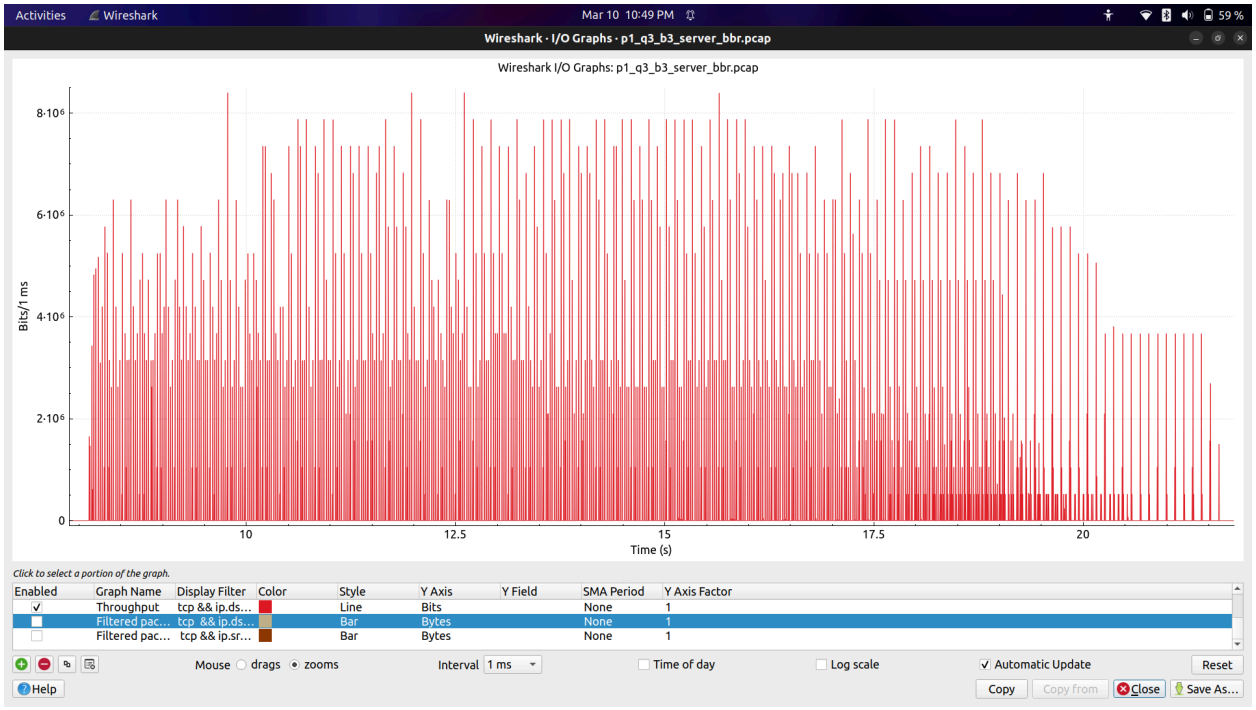
Part C.2c

Running client on H1, H3 and H4 and server on H7

BBR

Throughput:

Wireshark IO graph



Throughput:

Statistics

Measurement	Captured	Displayed	Marked
Packets	18321	10530 (57.5%)	—
Time span, s	27.653	19.662	—
Average pps	662.5	535.5	—
Average packet size, B	23144	40214	—
Bytes	424029627	423458267 (99.9%)	0
Average bytes/s	15 M	21 M	—
Average bits/s	122 M	172 M	—

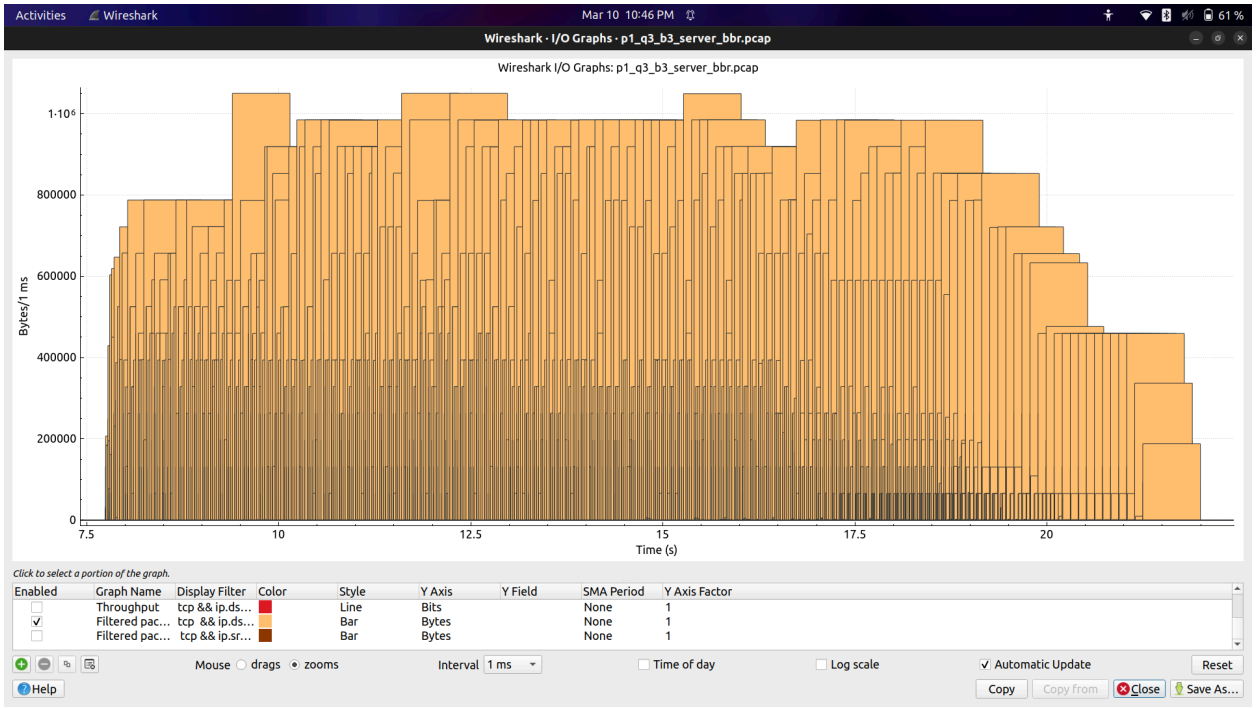
Goodput:

Total goodput bytes for TCP packets sent from 10.0.0.1 (excluding retransmissions): **422473971 bytes**

Packet Loss Rate:

Total Sent: 10530
Distinct packets Sent: 10091
Packets Lost: 439
Packet Loss Rate (Packets lost/Total Packet sent): 4.17%

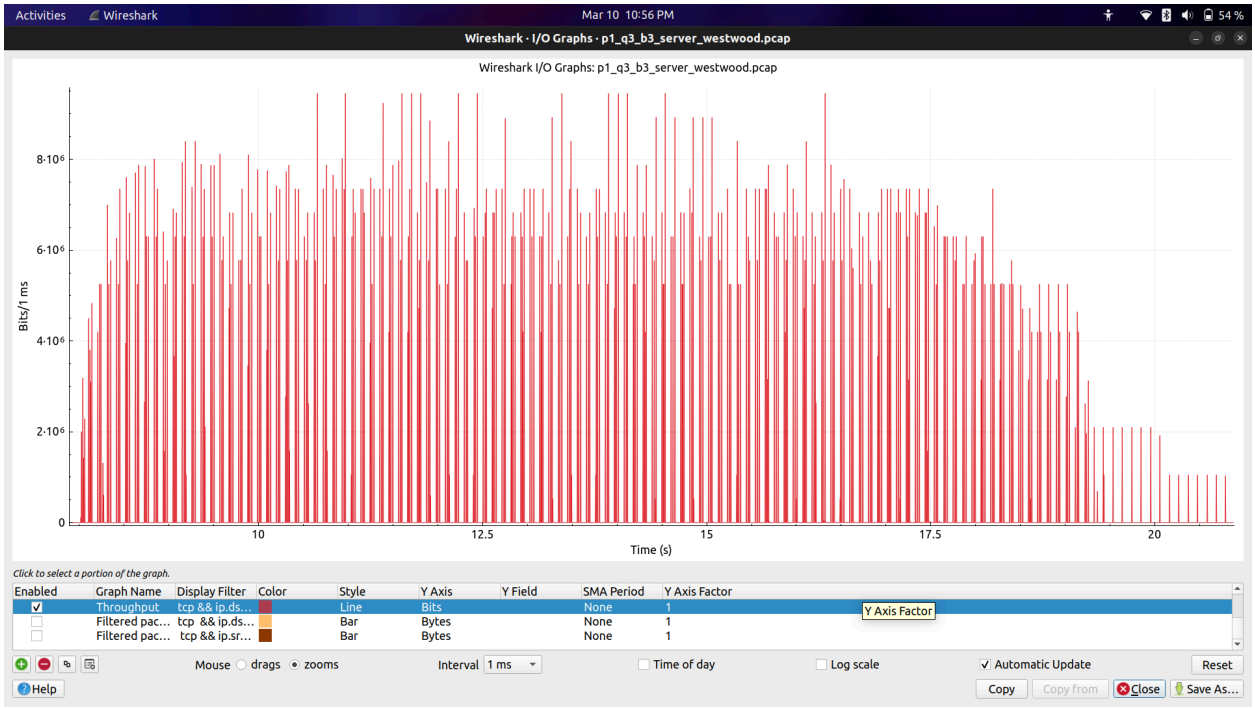
Window size: tcp && ip.dst==10.0.0.7 && tcp.window_size:



Westwood

Throughput:

Wireshark IO graph



Throughput:

Statistics

Measurement	Captured	Displayed	Marked
Packets	18833	10631 (56.4%)	—
Time span, s	27.578	19.662	—
Average pps	682.9	540.7	—
Average packet size, B	21167	37441	—
Bytes	398634010	398032074 (99.8%)	0
Average bytes/s	14 M	20 M	—
Average bits/s	115 M	161 M	—

Goodput:

Total goodput bytes for TCP packets sent from 10.0.0.1 (excluding retransmissions): **397033210 bytes**

Packet Loss Rate:

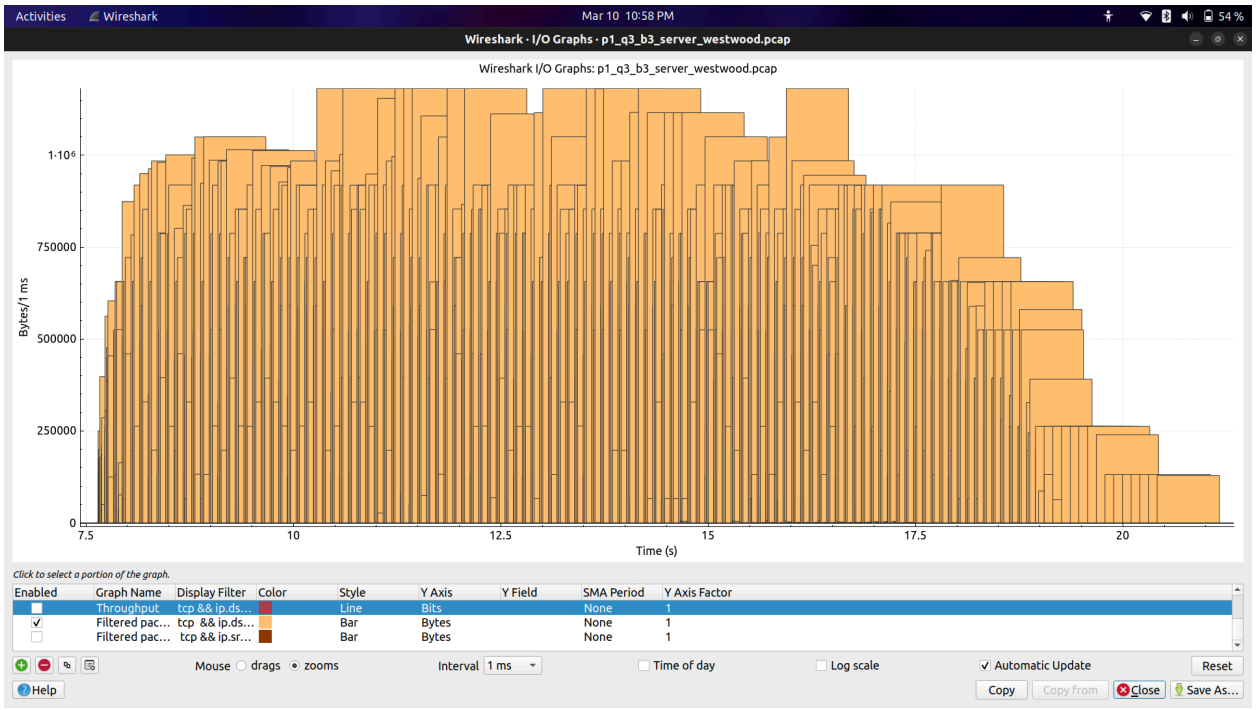
Total Sent: 10631

Distinct packets Sent: 10134

Packets Lost: 497

Packet Loss Rate (Packets lost/Total Packet sent): 4.68%

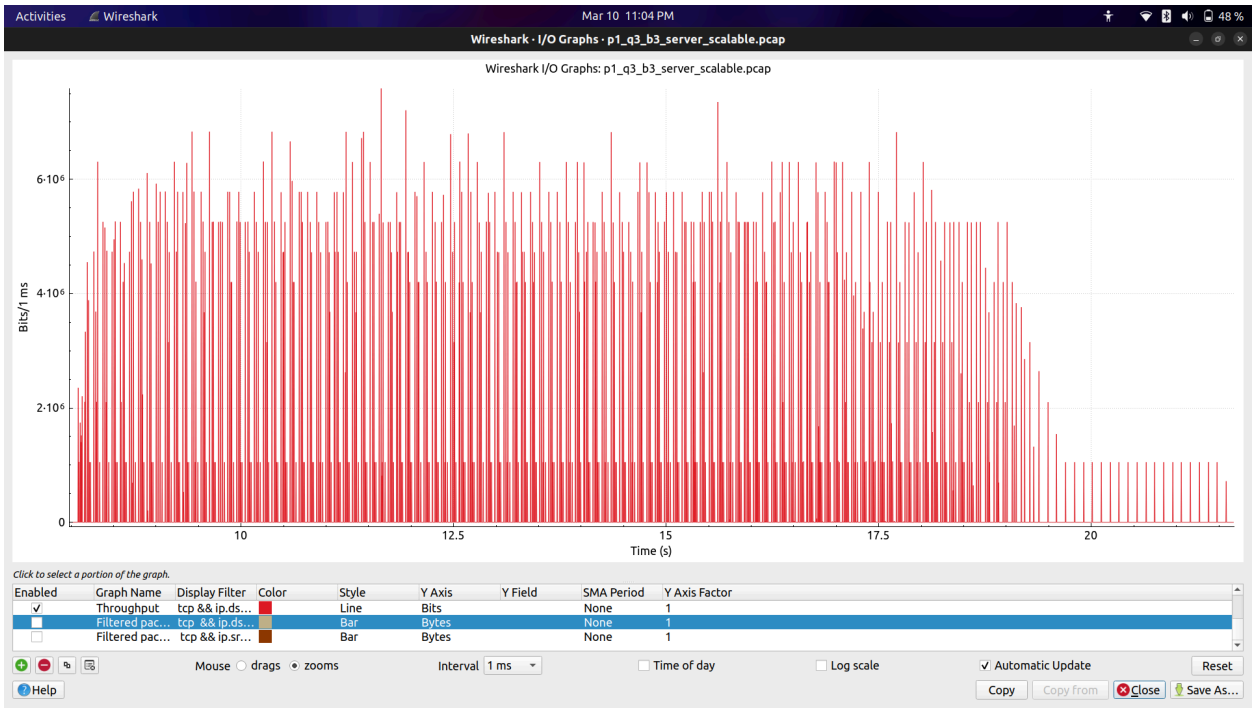
Window size: tcp && ip.dst==10.0.0.7 && tcp.window_size:



Scalable

Throughput:

Wireshark IO graph



Throughput:

Statistics

Measurement	Captured	Displayed	Marked
Packets	18674	10557 (56.5%)	—
Time span, s	28.034	20.054	—
Average pps	666.1	526.4	—
Average packet size, B	20871	36862	—
Bytes	389747907	389147846 (99.8%)	0
Average bytes/s	13 M	19 M	—
Average bits/s	111 M	155 M	—

Goodput:

Total goodput bytes for TCP packets sent from 10.0.0.1 (excluding retransmissions): **388115214 bytes**

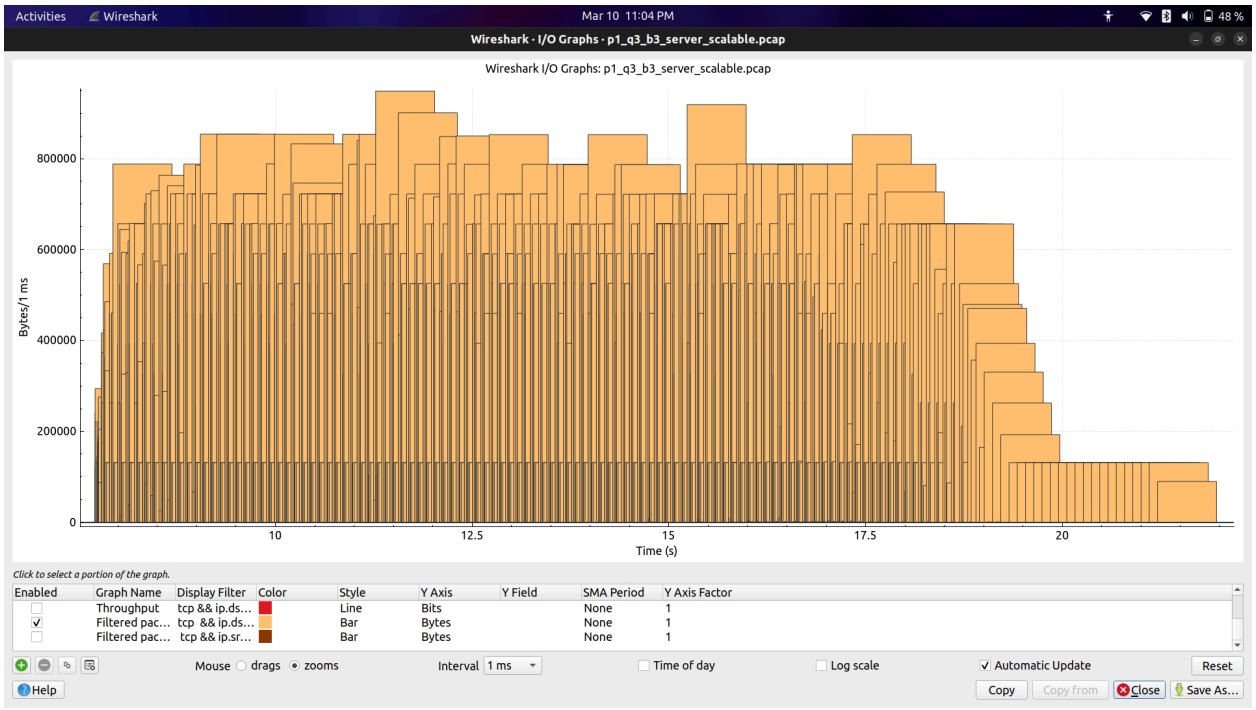
Packet Loss Rate:

Total Sent: 10557

Distinct packets Sent: 10042

Packets Lost: 515
Packet Loss Rate (Packets lost/Total Packet sent): 4.88%

Window size: tcp && ip.dst==10.0.0.7 && tcp.window_size:



Comparisons from the above observations

BBR outperformed both Westwood and Scalable in terms of throughput, goodput, and lower packet loss. BBR had the highest throughput (21M bytes/sec displayed, 172M bits/sec). BBR led with 422.47M bytes of goodput – most efficient in transferring useful data. BBR had the lowest packet loss (4.17%), meaning it was the most reliable.

Westwood had a balanced performance, with good throughput but slightly higher packet loss than BBR. Westwood was slightly behind BBR (20M bytes/sec, 161M bits/sec), about 5-6% lower. Westwood followed with 397.03M bytes, lower than BBR. Westwood had a slightly higher loss (4.68%).

Scalable had the worst performance, with the lowest throughput, highest packet loss, and least efficiency. Scalable had the lowest throughput (19M bytes/sec, 155M bits/sec), ~10% lower than BBR. Scalable had the lowest at 388.11M bytes, 8% less than BBR and 2.2% lower than Westwood. Scalable had the worst loss rate (4.88%).

Use

BBR is the best choice for high-speed, low-loss applications (e.g., streaming, gaming, large data transfers).

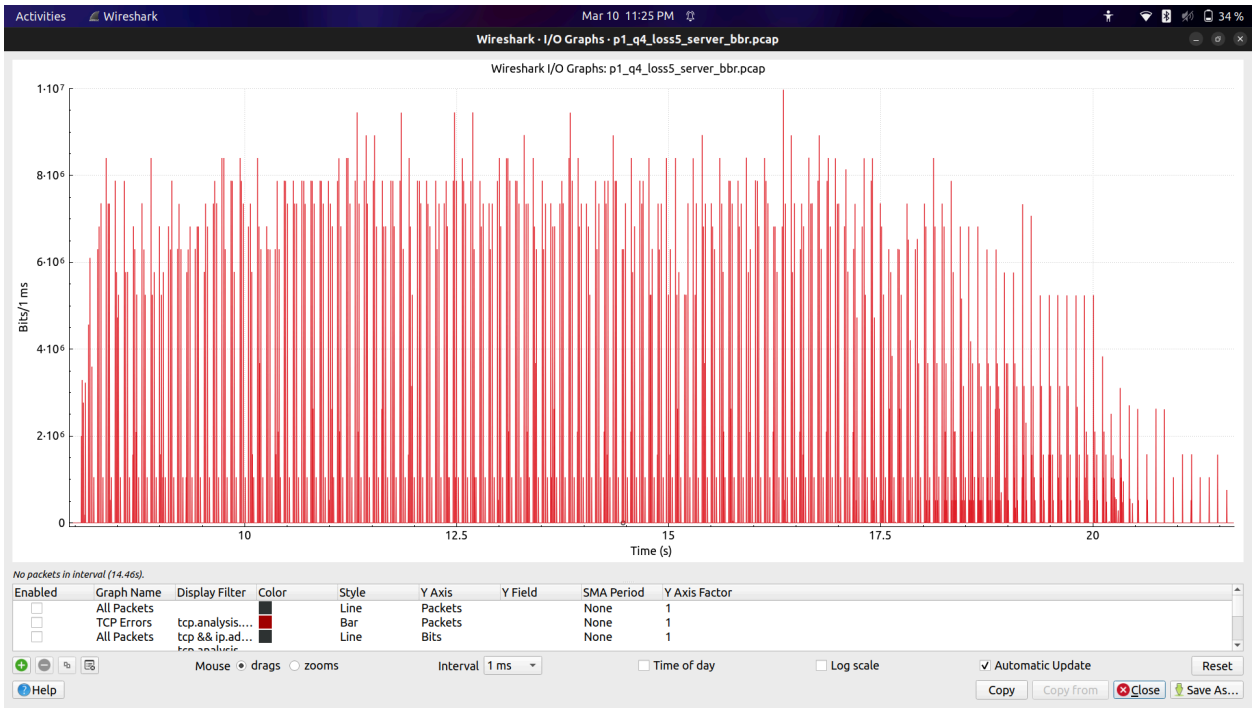
Westwood is a good middle ground, balancing efficiency with slightly higher loss.

Scalable is the weakest performer in this setup—higher loss without throughput benefits.

Part 2 D
Link loss = 5%, Running client on H1, H3 and H4 and server on H7

BBR
Throughput:

Wireshark IO graph



Throughput:

Statistics

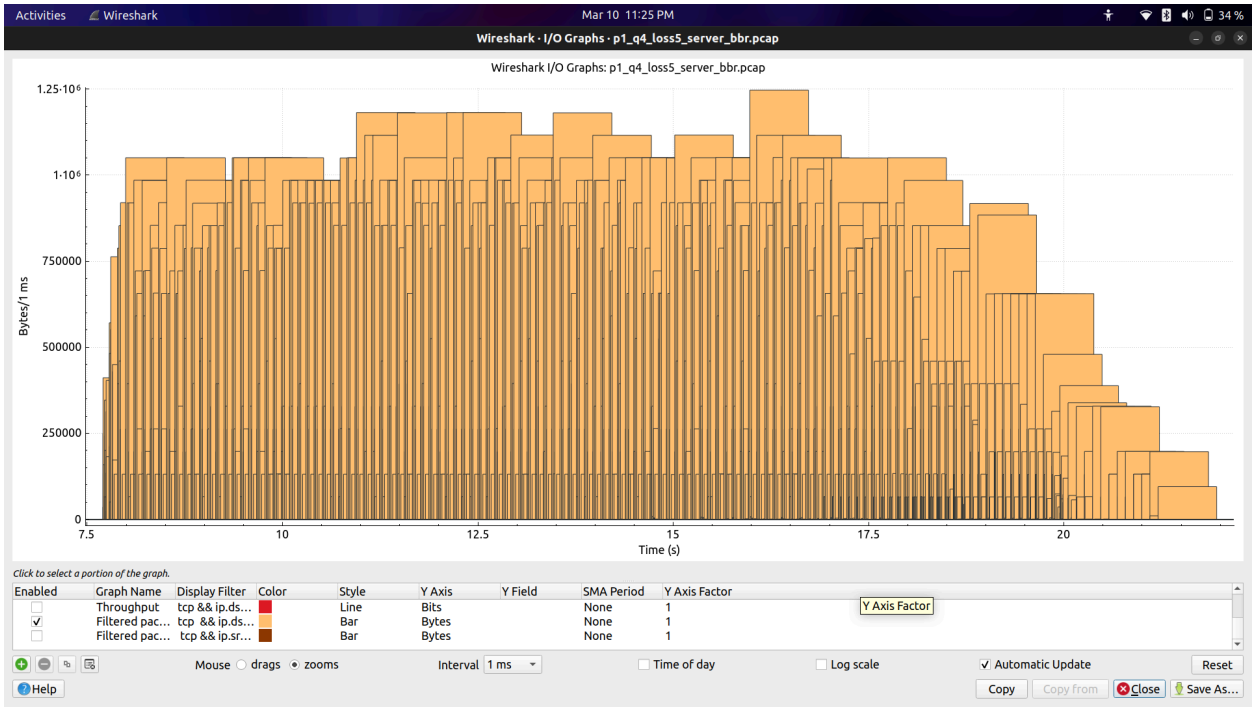
Measurement	Captured	Displayed	Marked
Packets	19039	10796 (56.7%)	—
Time span, s	28.423	20.064	—
Average pps	669.9	538.1	—
Average packet size, B	22379	39409	—
Bytes	426067160	425458961 (99.9%)	0
Average bytes/s	14 M	21 M	—
Average bits/s	119 M	169 M	—

Goodput:
Total goodput bytes for TCP packets sent from 10.0.0.1 (excluding retransmissions): **424483249 bytes**

Packet Loss Rate:
Total Sent: 10796
Distinct packets Sent: 10350

Packets Lost: 446
Packet Loss Rate (Packets lost/Total Packet sent): 4.13%

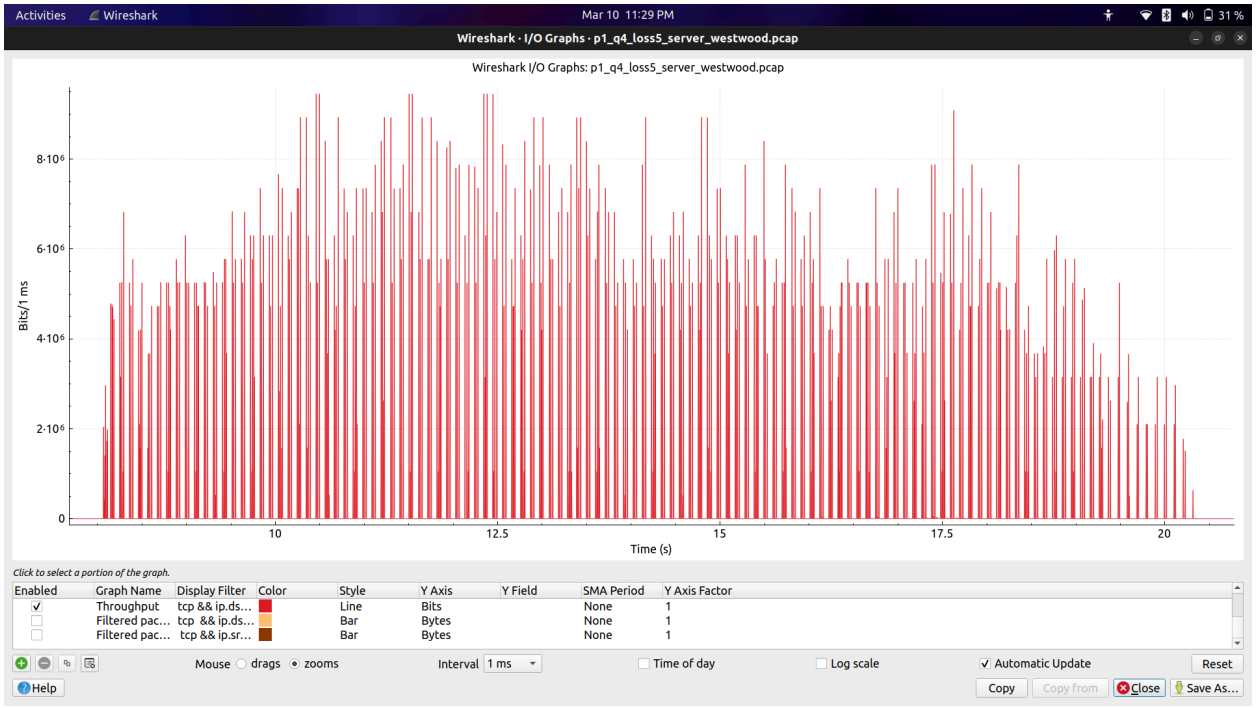
Window size: tcp && ip.dst==10.0.0.7 && tcp.window_size:



Westwood

Throughput:

Wireshark IO graph



Throughput:

Statistics

Measurement	Captured	Displayed	Marked
Packets	18615	10584 (56.9%)	—
Time span, s	28.025	20.060	—
Average pps	664.2	527.6	—
Average packet size, B	22064	38751	—
Bytes	410729464	410136717 (99.9%)	0
Average bytes/s	14 M	20 M	—
Average bits/s	117 M	163 M	—

Goodput:

Total goodput bytes for TCP packets sent from 10.0.0.1 (excluding retransmissions): **409148477 bytes**

Packet Loss Rate:

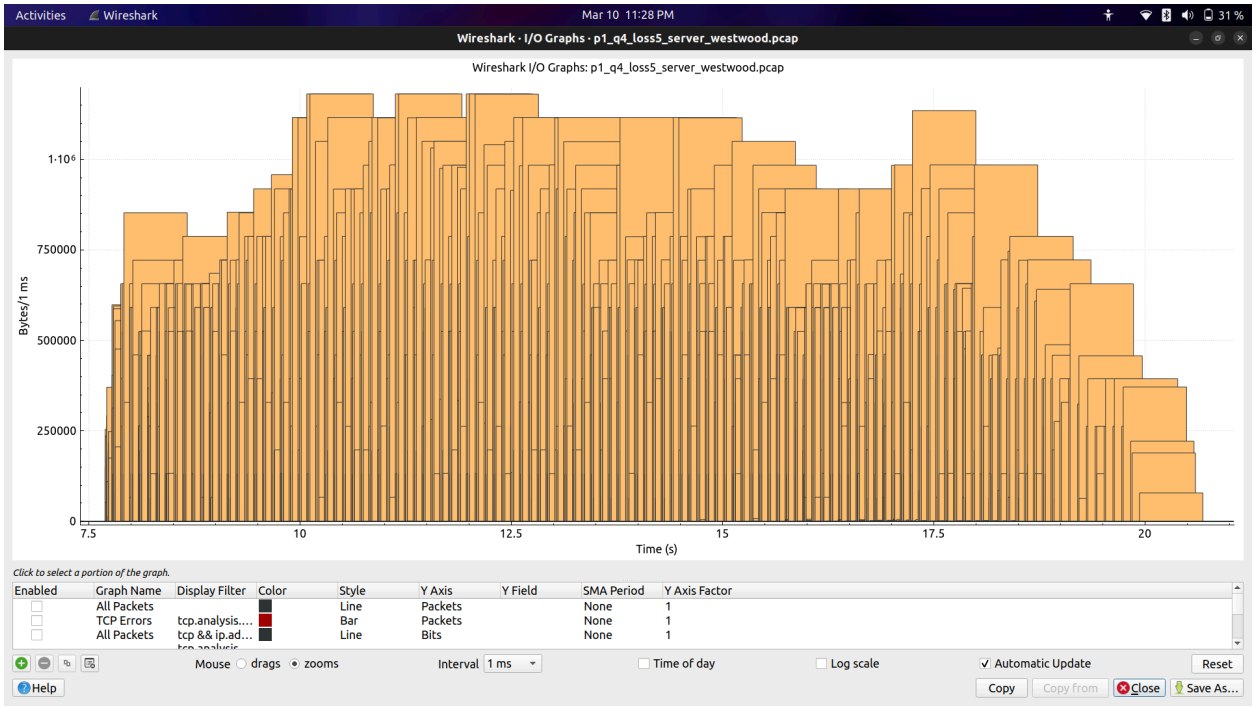
Total Sent: 10584

Distinct packets Sent: 10098

Packets Lost: 486

Packet Loss Rate (Packets lost/Total Packet sent): 4.59%

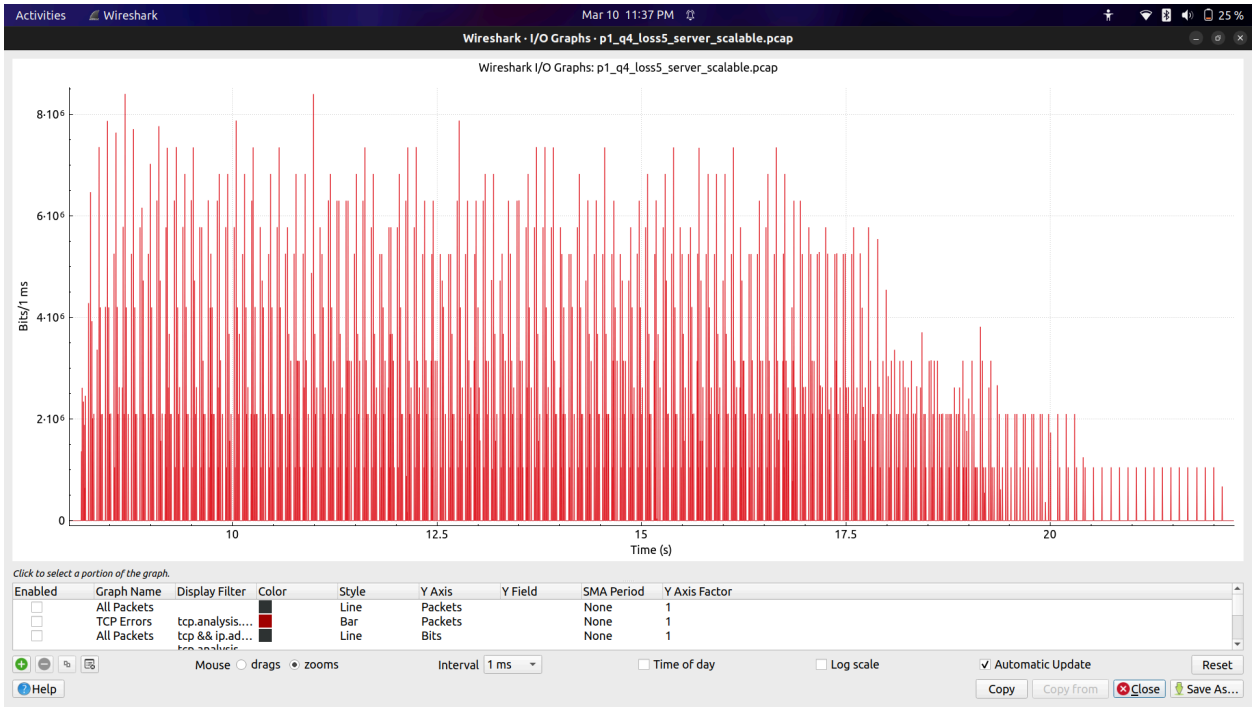
Window size: tcp && ip.dst==10.0.0.7 && tcp.window_size:



Scalable:

Throughput:

Wireshark IO graph



Throughput:

Statistics

Measurement	Captured	Displayed	Marked
Packets	18237	10331 (56.6%)	—
Time span, s	28.046	20.050	—
Average pps	650.3	515.3	—
Average packet size, B	21567	38015	—
Bytes	393316780	392732445 (99.9%)	0
Average bytes/s	14 M	19 M	—
Average bits/s	112 M	156 M	—

Goodput:

Total goodput bytes for TCP packets sent from 10.0.0.1 (excluding retransmissions): **391766181 bytes**

Packet Loss Rate:

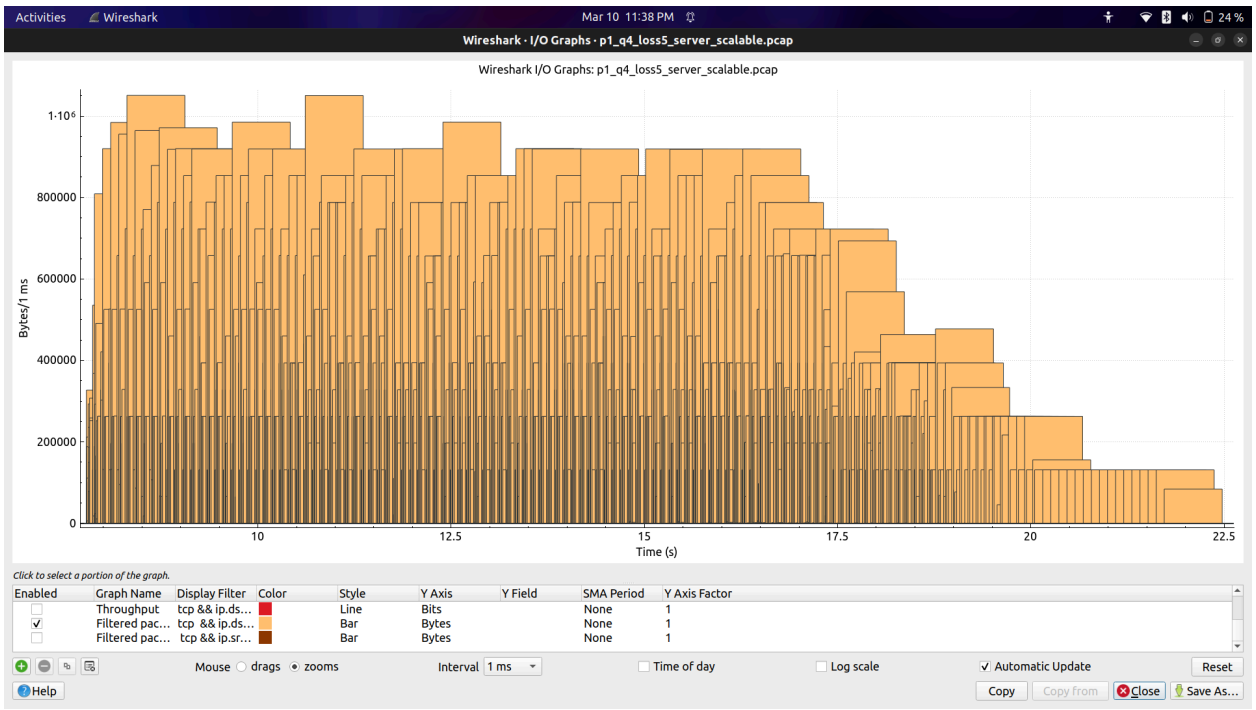
Total Sent: 10331

Distinct packets Sent: 9849

Packets Lost: 482

Packet Loss Rate (Packets lost/Total Packet sent): 4.67%

Window size: tcp && ip.dst==10.0.0.7 && tcp.window_size:



Comparisons from the above observations

As there is a specified loss rate of 5%, the loss should be higher than the case in part 2 (where there was no loss).

On comparing the respective congestion control schemes with the no loss case in part 2, the number of packets lost is higher. Number of retransmissions is higher. The throughput is higher, this is because there are more retransmissions due to more packets being lost. The goodput is lower. The maximum congestion window size is higher than the no loss case in part 2, this may be because as there are more packets to be sent (original + retransmission), the client tries to increase the congestion window size more and more. This window size increases till a packet loss is observed.

Task 2: Implementation and mitigation of SYN flood attack

Perform this task in a virtual/isolated environment. You can implement any client-server program that generates tcp packets. Preferably use two VM's or two host machines

A)
We used Minimet for the purposes of emulating SYN-flood attack.

Implementation of SYN flood attack
We modified the receiver's (server) Linux kernel in ways to implement the TCP/IP SYN flood attack as follows:

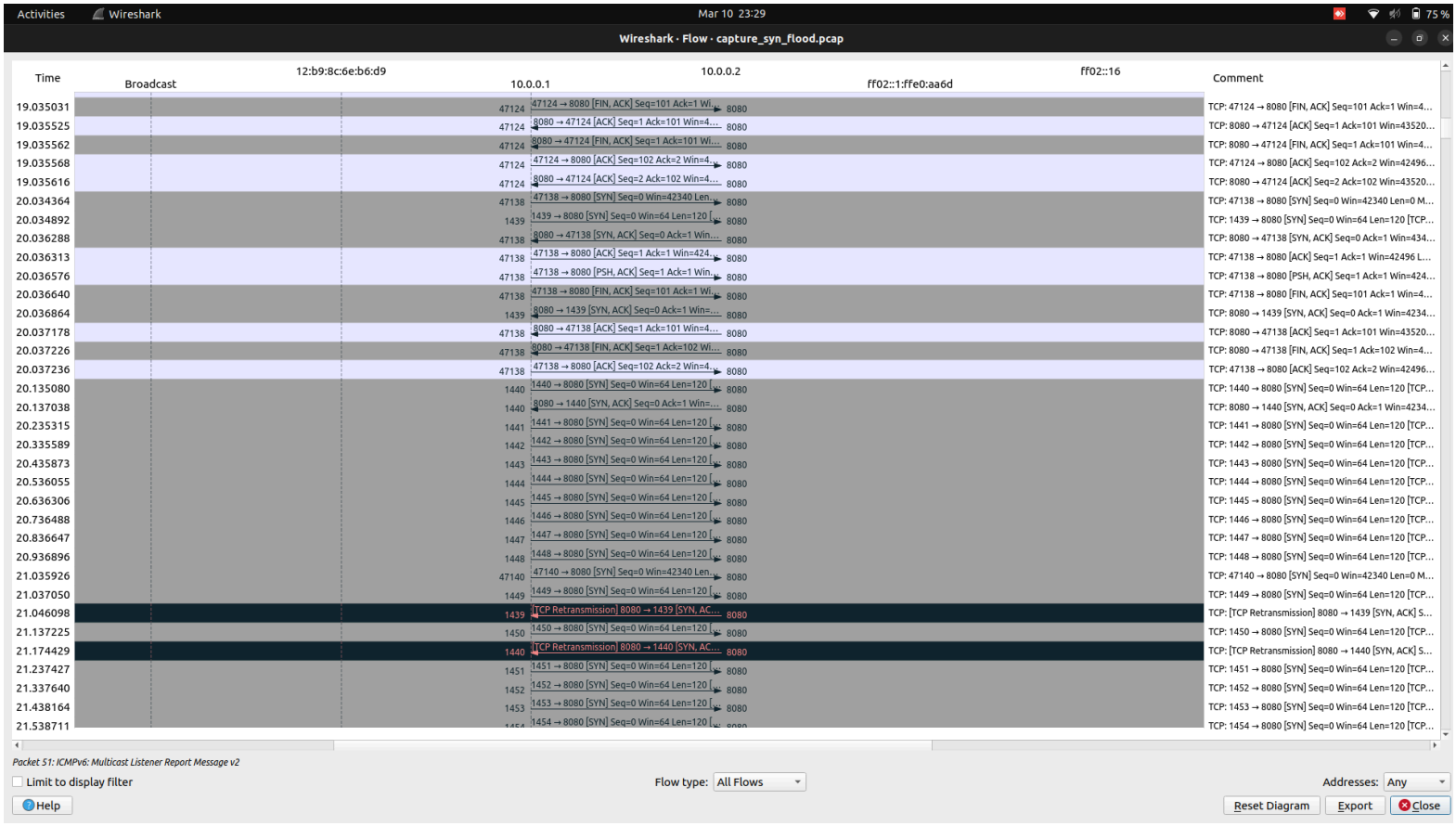
1. **net.ipv4.tcp_max_syn_backlog** – Increases the maximum number of pending SYN requests in the backlog. We set it to 1000 when the default value was 512. (we later realized this value had little to no effect)
2. **net.ipv4.tcp_syncookies** – Disables SYN cookies (set to 0) to prevent the system from mitigating the SYN flood attack. We set this to 0 for allowing SYN flood attack and set it back to 1 for mitigation.
3. **net.ipv4.tcp_synack_retries** – Reduces the number of SYN-ACK retries, making it easier to exhaust resources. We reduced the default 'safe' value of 5 to 1 to allow for SYN flood attack.

A common **TCP server** (with Linux Kernel parameters modified as above) with only 1 port was set up with its **listen() API with argument 2**. **Legitimate traffic** was emulated by allowing for TCP clients from the same IP host to periodically connect to the server via different source ports, in intervals of **1s**. The **Attack traffic** was emulated to establish half-open connections with the TCP server's only port via different source ports on the same IP host as the Legitimate traffic. The Attack traffic would send **10 SYN packets per second** to the TCP server via **hping3** tool but not respond to the SYN-ACK segment received from the server, thus emulating a SYN flood attack.

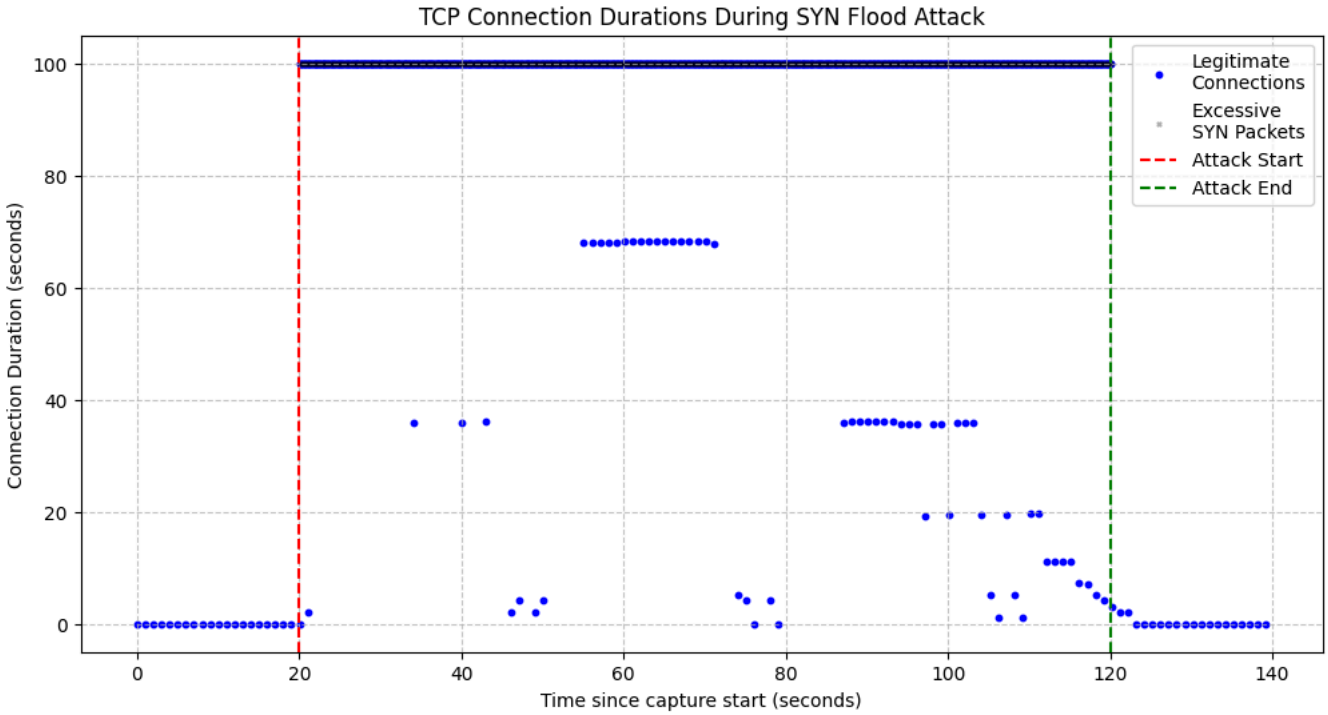
Additionally, in order to **bypass the Linux Kernel on the client side** from detecting rogue SYN and SYN-ACK packets, its response to SYN-ACK packets, which was a **RST connection**, **was intentionally dropped** using **iptables tool** to allow for prevalence of half-open connections.

For the mitigation emulation, the modifications made to the TCP Server's Linux Kernel Parameters and the alterations made to the client side TCP connection regarding RSTing connections were reverted to default, thus, **RSTing any stray, malicious connections** and **allowing for legitimate traffic to operate normally**.

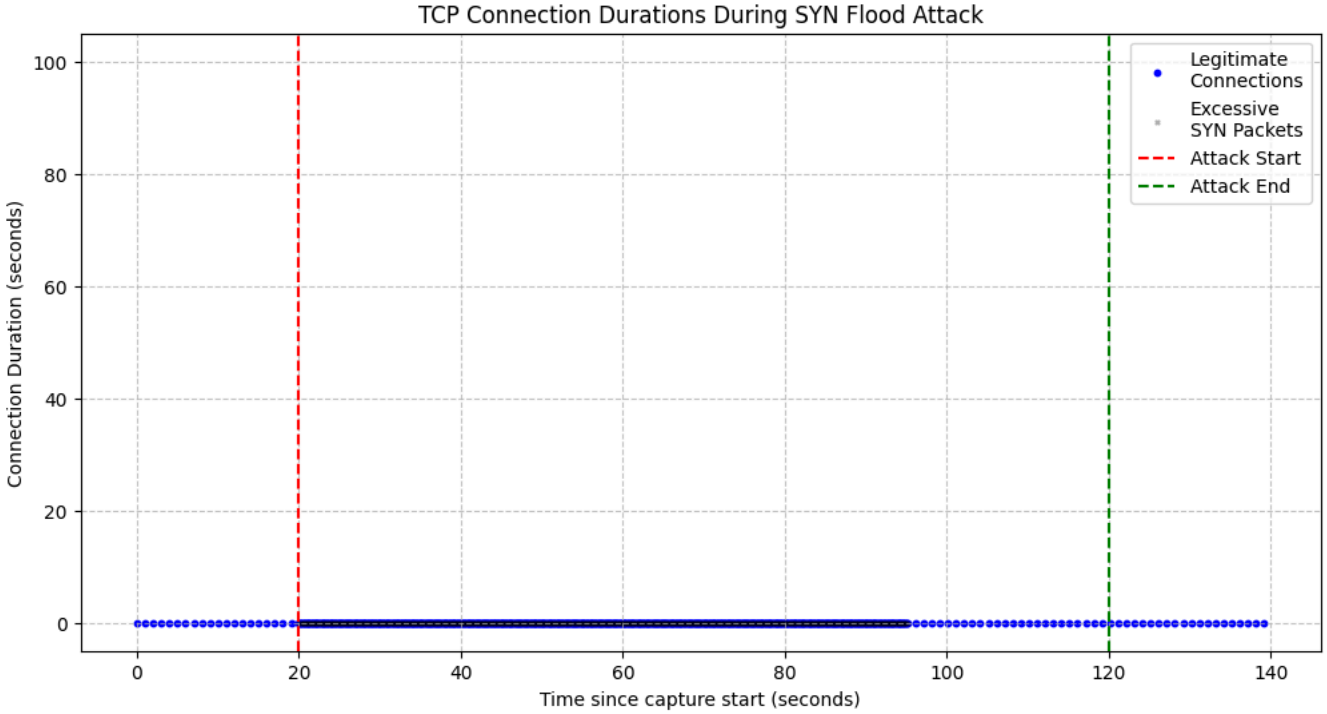
The following flow graph is a snapshot of the traffic just before the SYN flood attack starts. Notice how legitimate traffic periodically connects to the server almost every 1s until 20s, when SYN flood attack starts and when there is an excess of SYN packets moving in the network as a part of Attack traffic.



The following graph demonstrates the **duration of TCP connections of across time**. Notice how because of the Attack traffic, the connection duration of the legitimate traffic also increases substantially, thus, depicting Denial of Service



The following graph depicts the **mitigation of SYN flood attack** by RSTing the half-open connections and ensuring (ideally) no overhead on connection durations of legitimate traffic.



Task 3: Analyze the effect of Nagle's Algorithm on TCP/IP performance

You will transmit a 4 **KB file** over a TCP connection for a duration of **~2 minutes** with a transfer rate of 40 bytes/second. Perform the following four combinations by enabling/disabling Nagle's Algorithm and Delayed-ACK on both the client and server sides:

1. Nagle's Algorithm enabled, Delayed-ACK enabled.
2. Nagle's Algorithm enabled, Delayed-ACK disabled.
3. Nagle's Algorithm disabled, Delayed-ACK enabled.
4. Nagle's Algorithm disabled, Delayed-ACK disabled.

For each configuration measure the following performance metrics and compare. Explain your Observations:

1. Throughput
2. Goodput
3. Packet loss rate
4. Maximum packet size achieved.

A)

Nagle's Algorithm

Nagle's Algorithm is a TCP optimization technique designed to improve network efficiency by reducing the number of small packets transmitted. It works on the sender's side by buffering small data segments and delaying their transmission until either:

1. A full-sized segment (typically the Maximum Segment Size, MSS) can be sent, or
2. An acknowledgment (ACK) for previously sent data is received. This reduces network overhead, especially on high-latency networks, by minimizing the number of packets with small payloads. However, it can introduce latency in applications requiring immediate data transmission, such as real-time systems.

To **disable Nagle's Algorithm** meant to set **TCP_NODELAY hook to 1**, thus enabling no-delay transmission. Conversely, to enable Nagle's Algorithm, one has to set the TCP_NODELAY hook in the **setsockopt()** API to be 0. This was done to both the client and server.

Delayed-ACK

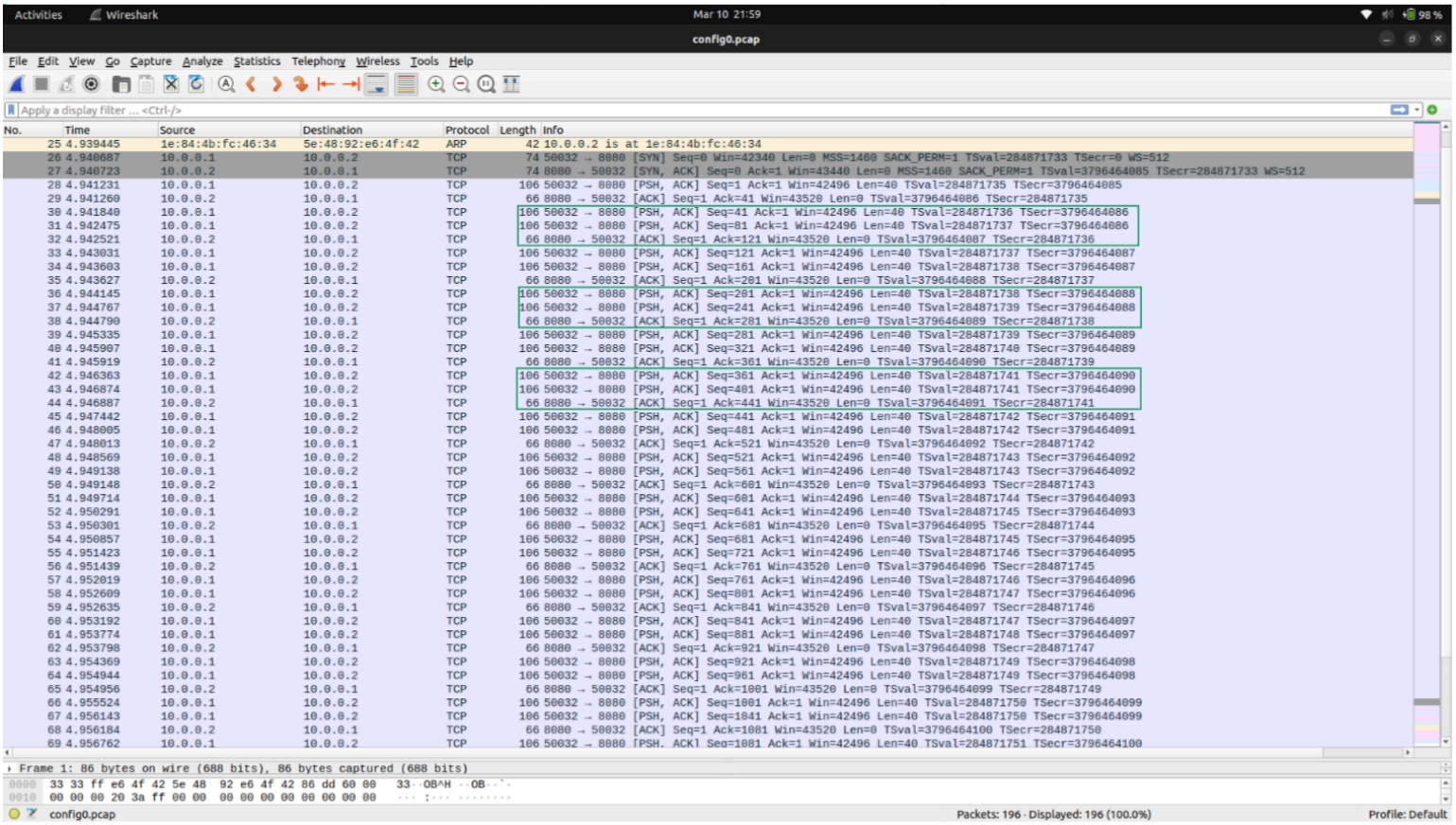
Delayed-ACK is a TCP mechanism implemented on the receiver's side to enhance efficiency by reducing the number of standalone acknowledgment packets. Instead of sending an ACK immediately after receiving a segment, the receiver waits (typically up to 200ms) to see if it can piggyback the ACK with outgoing data or combine ACKs for multiple received segments. This reduces network traffic but can increase latency if the sender is waiting for an ACK before sending more data.

To **disable Delayed-ACK** meant to set **TCP_QUICKACK hook to 1**, thus enabling no-delay ACK transmissions. Conversely, to enable Delayed-ACK, one has to set the TCP_QUICKACK hook in the **setsockopt()** API to be 0. Again, this was done to both the client and server.

NOTE: In order to properly appreciate the performance of the algorithms, the transmission rate was changed from the originally proposed 40Bps to 80KBps. Effects of these algorithms were only observed around these rates.

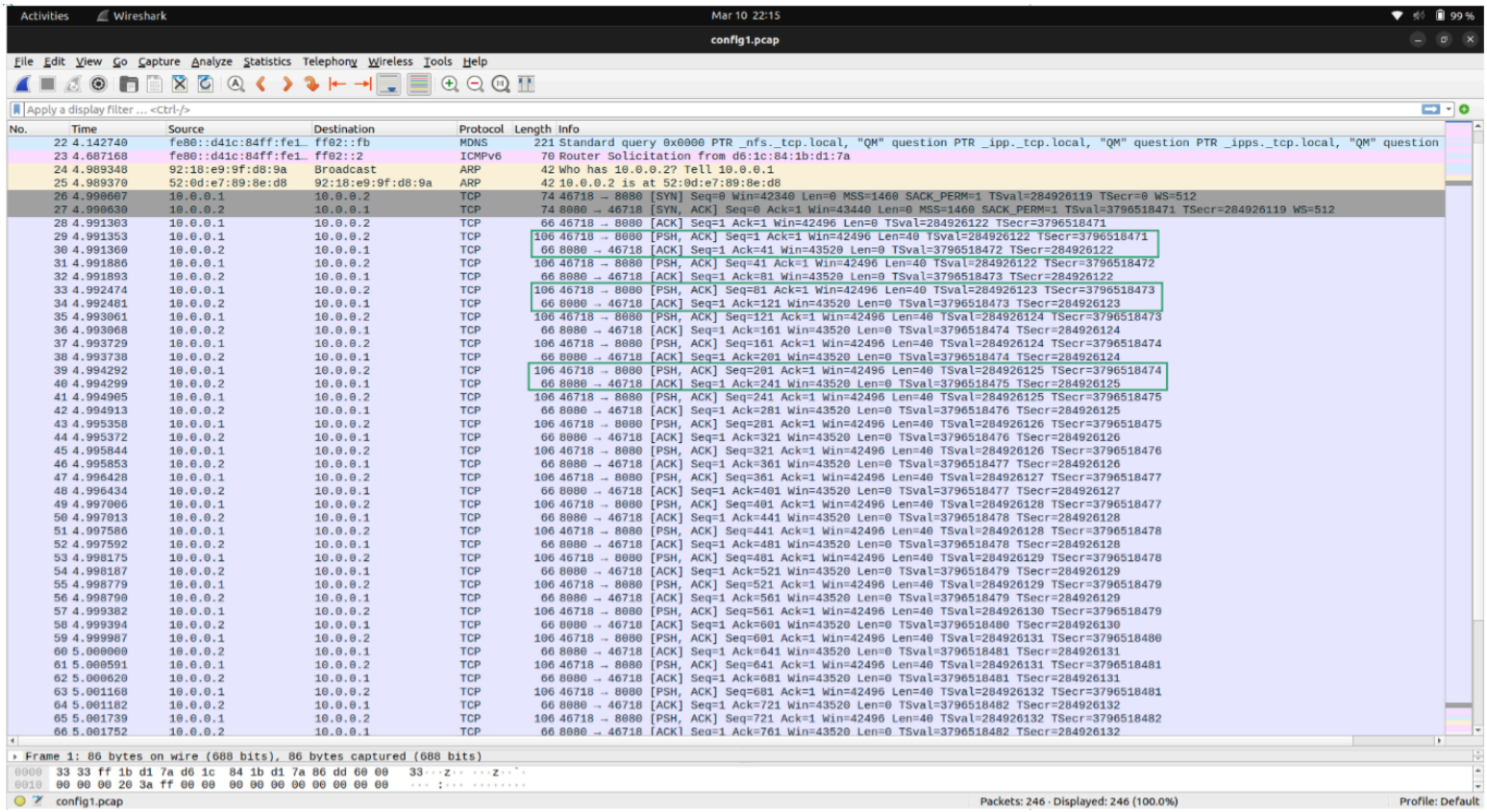
For the four different configurations, the following are some interesting observations captured:

config0 - 0, 0 => Nagle Dis, Delayed Ack (config_no = 0)



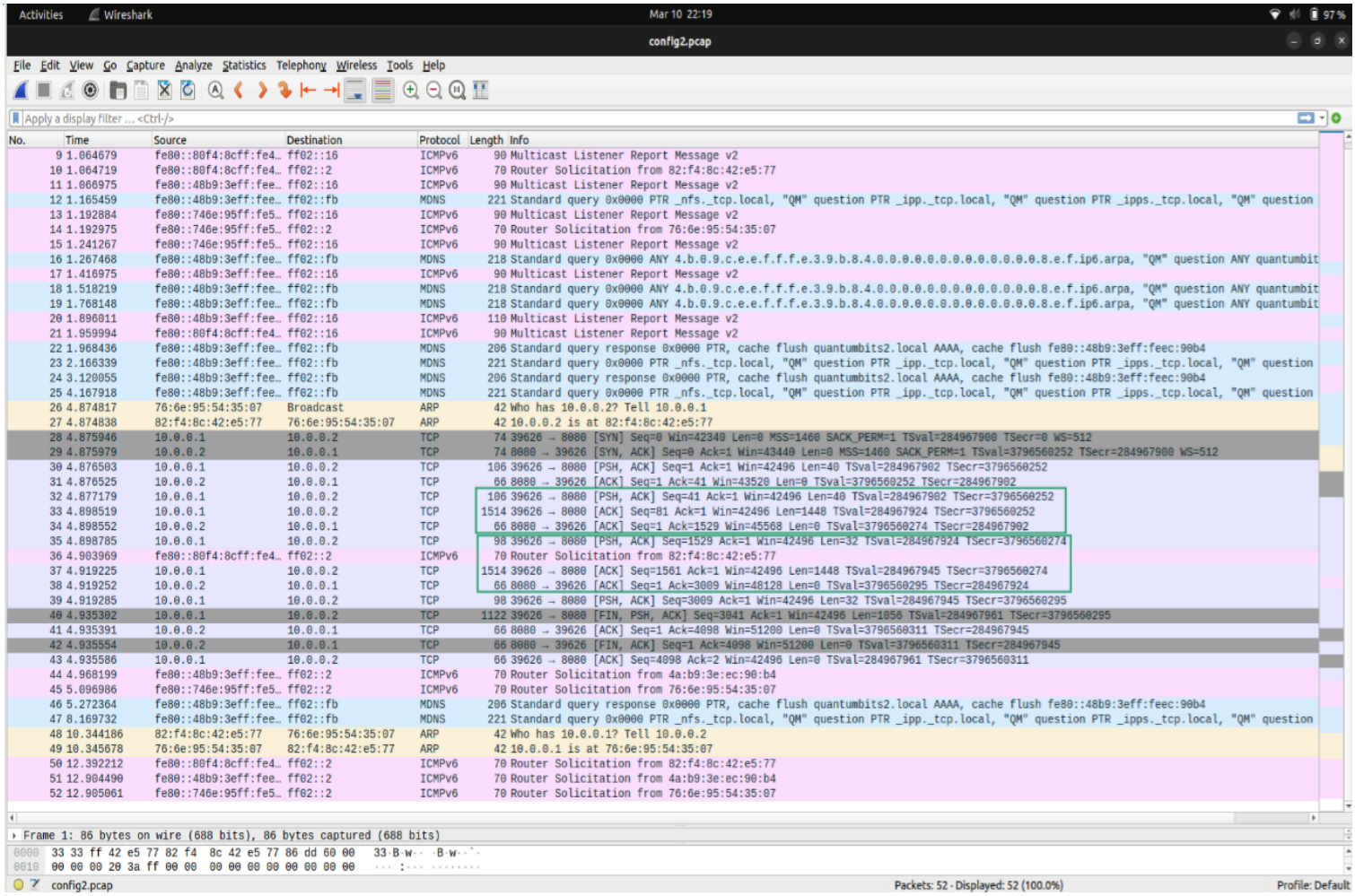
As can be seen from the above snapshot of traffic, with Nagle Disabled, the packets of data size 40B were being sent very quickly without accumulating into a larger packet and Delayed ACKs were observed, ACKing the latest segment received by the server.

config1 - 0, 1 => Nagle Dis, No Delayed Ack (config_no = 1)



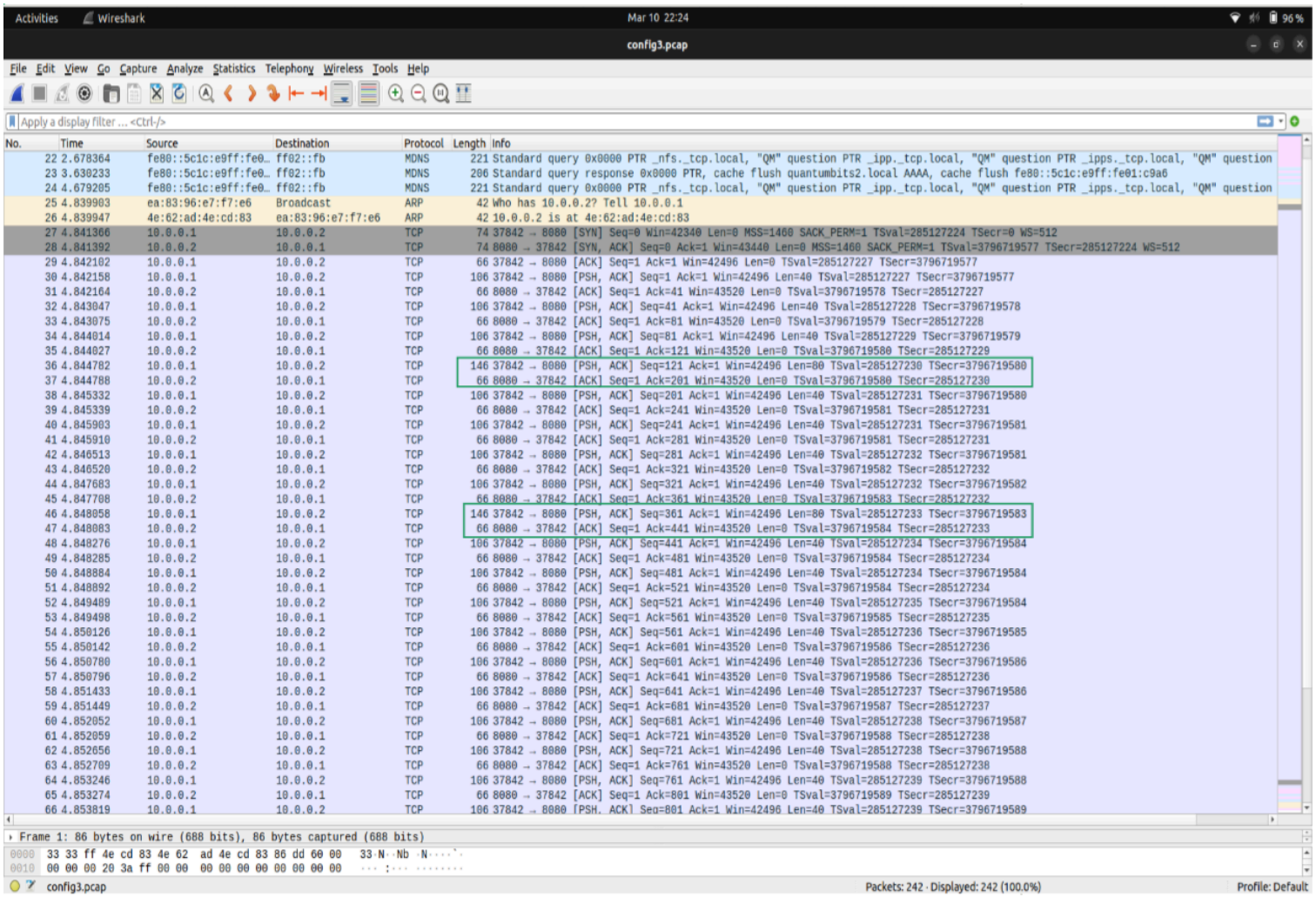
As can be seen from the above snapshot of traffic, with Nagle Disabled, the packets of data size 40B were being sent very quickly without accumulating into a larger packet. Additionally, with no Delayed ACKs, segments received by the server were immediately ACKed without much delay.

config2 - 1, 0 => Nagle En, Delayed Ack (config_no = 2)



As can be seen from the above snapshot of traffic, with Nagle Enabled, the accumulation of data into larger packets is observed, indiciating delay in transmitting the data. Additionally, Delayed ACKs can also be observed.

config3 - 1, 1 => Nagle En, No Delayed Ack (config_no = 3)



As can be seen from the above snapshot of traffic, with Nagle Enabled, the accumulation of data into slightly larger packets (of data size 80B) is observed. Additionally, no Delayed ACKs are observed, as the server ACKs each segment almost immediately as it receives it.

After analyzing the generated pcap files, the following metrics related to the transmission were acquired for different configurations:

1) **Throughput:** (Total TCP bytes transferred / Duration of connection)
2) **Goodput:** (Total useful data bytes / Duration of Connection)
3) **Packet Loss Rate: (#**expected_packets - #ack_numbers)/(#expected_packets) *100
4) **Maximum Packet Size (bytes):** Max size of packets in bytes

Config.No.	Throughput (in Bps)	Goodput (in Bps)	Packet Loss Rate (in %)	Maximum Packet Size (in bytes)
0	235358.28	65705.25	47.22	106
1	299804.59	67167.88	0.00	106
2	85639.94	68753.67	27.27	1514
3	297031.35	67531.34	0.00	146

1. Nagle Dis + No Delayed Ack achieved max Throughput as expected
2. However, Nagle En + Delayed Ack achieved maximum Goodput
3. Packet Loss Rate was zero for No Delayed ACKs scenarios
4. Maximum Packet Size was observed for Nagle En + Delayed ACK configuration

Acknowledgements

Some codes are inspired by GitHub repositories, StackOverflow and ChatGPT.