Computer Networks CS-331

Assignment - 2

Team - 10

Shubham Agrawal - 22110249
Vraj Shah - 22110292

Protocol List Assigned

- Protocol 1 - Reno
    - TCP Reno is a traditional congestion control algorithm that follows an AIMD (Additive Increase, Multiplicative Decrease) approach to manage network traffic. It begins with a slow start phase, where the congestion window (cwnd) grows exponentially until a packet loss is detected. Once loss occurs, Reno shifts to congestion avoidance, where cwnd increases linearly to prevent network overload. Reno handles packet loss using Fast Retransmit and Fast Recovery, but it reduces the congestion window by half when loss is detected, leading to a sawtooth pattern in throughput. This aggressive reduction limits its performance in high-bandwidth, high-latency networks, making it less efficient than modern congestion control schemes.

- Protocol 2 - BIC
    - TCP BIC is designed to perform better in high-bandwidth, high-delay networks by addressing Reno's inefficiencies. It uses a binary search approach to adjust the congestion window, rapidly increasing cwnd when bandwidth is underutilized and slowing down as it nears the optimal rate. This results in a more stable congestion control mechanism compared to Reno, as it does not react as aggressively to packet loss. BIC ensures better fairness and higher throughput but can still cause unfair competition with other congestion control protocols like Reno and CUBIC, especially in mixed network environments.

- Protocol 3 - Highspeed
    - TCP HighSpeed is an enhancement of Reno that aims to improve performance in high-speed, long-distance networks where packet loss is rare but has a significant impact on throughput. Unlike Reno, which halves the congestion window upon packet loss, HighSpeed reduces it by a smaller factor, allowing it to recover much faster. It scales better in networks with high bandwidth-delay products (BDP) by using more aggressive window growth while remaining stable under congestion. However, its aggressive behavior can lead to unfair bandwidth sharing with standard TCP variants, making it more suitable for dedicated high-performance networks rather than mixed environments.

Task-1

We designed the required network with seven hosts (H1-H7) and four switches (S1-S4), where H7 acted as the TCP server, and H1-H6 were TCP clients, using the Mininet Topology. The network configuration included:

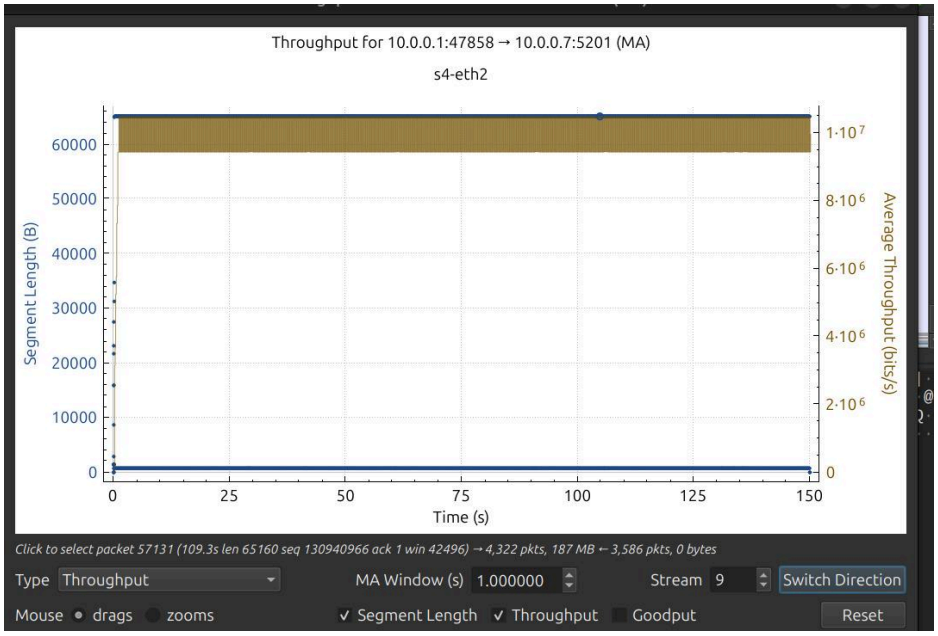
- TCP Server: Hosted on H7, listening on port 5201.
- TCP Clients: H1-H6 generated traffic towards H7 using different congestion control algorithms.
- Traffic Generation: Implemented using iperf3 with the command
- Packet Capture: Wireshark was used to capture the traffic and plot graphs.

Part a

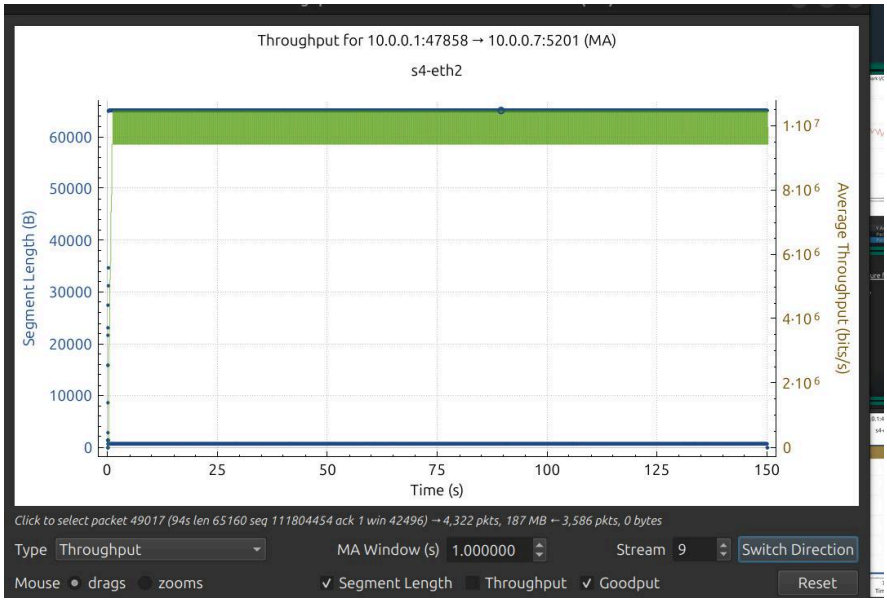
- The first part involved running the iperf3 client on H1 and the server on H7 for 150 seconds.
- Ran code for each congestion control algorithm (Reno, BIC, and HighSpeed), capturing the network traffic for each using wireshark on s4-eth2 interface and getting plots required.

Protocol: RENO

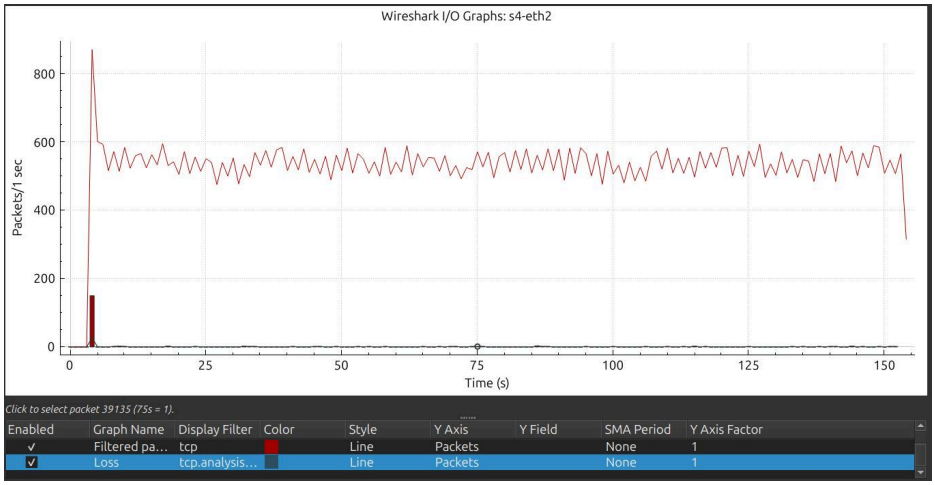
1. Throughput



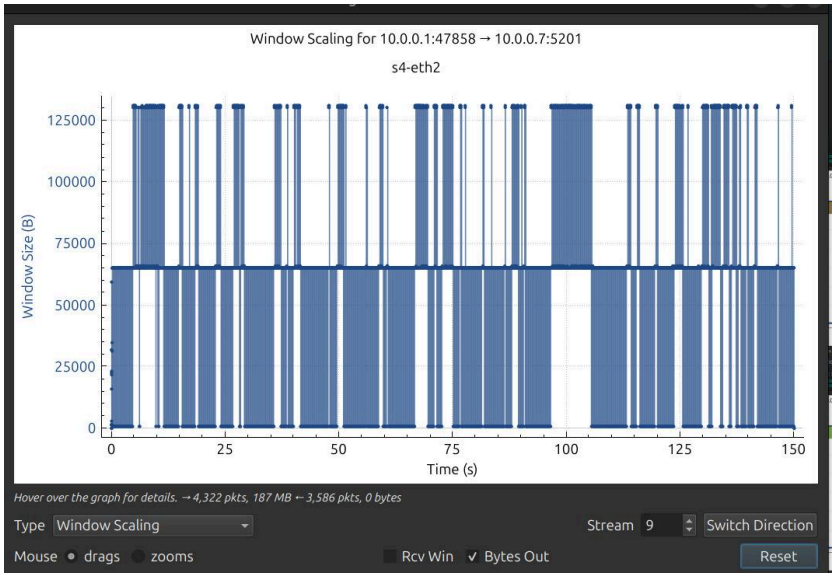
2. Goodput



3. Packet Loss Rate

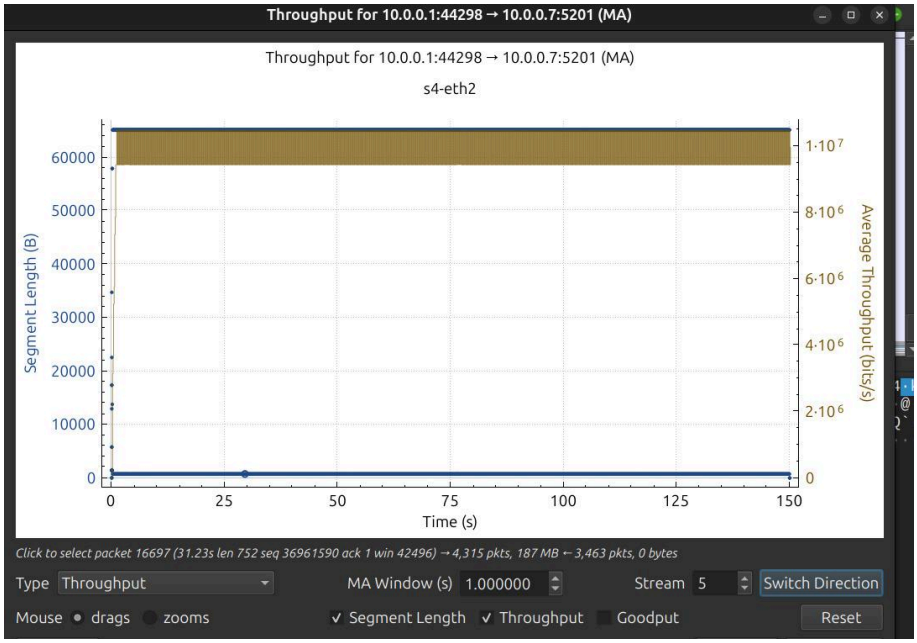


4. Maximum Window size achieved

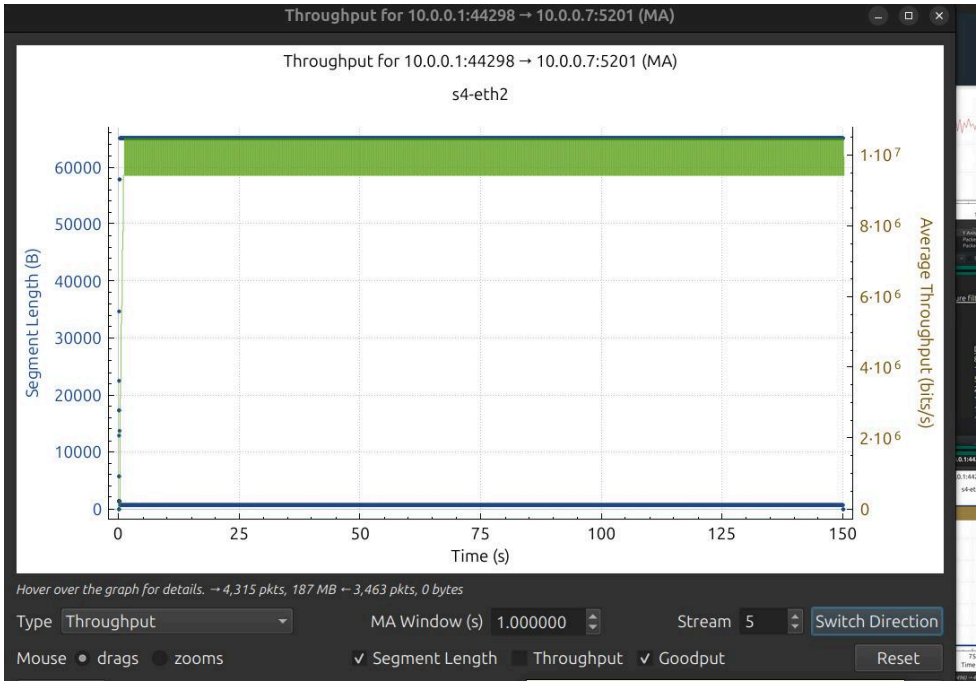


Protocol: BIC

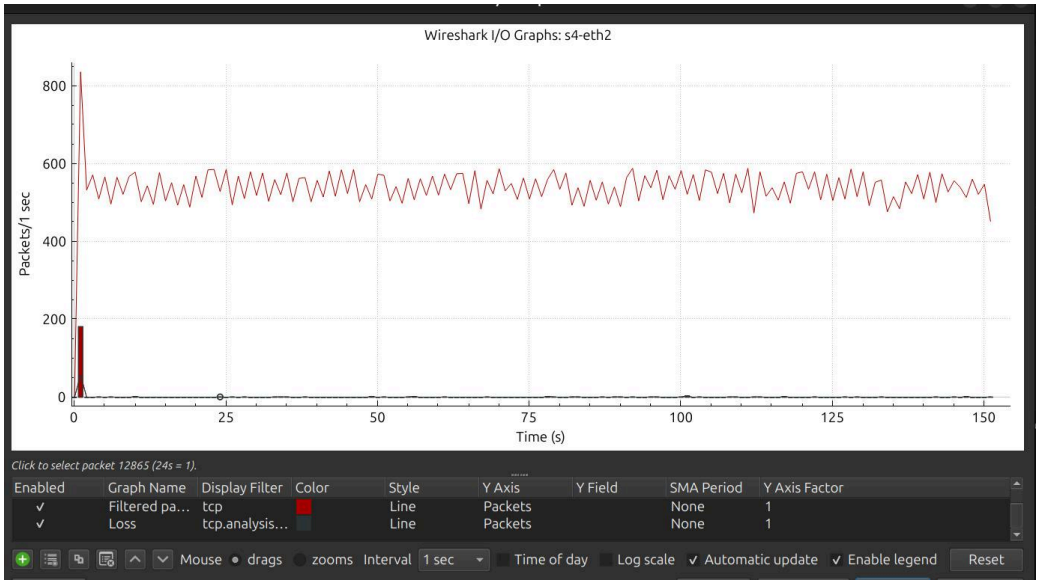
1. Throughput



2. Goodput



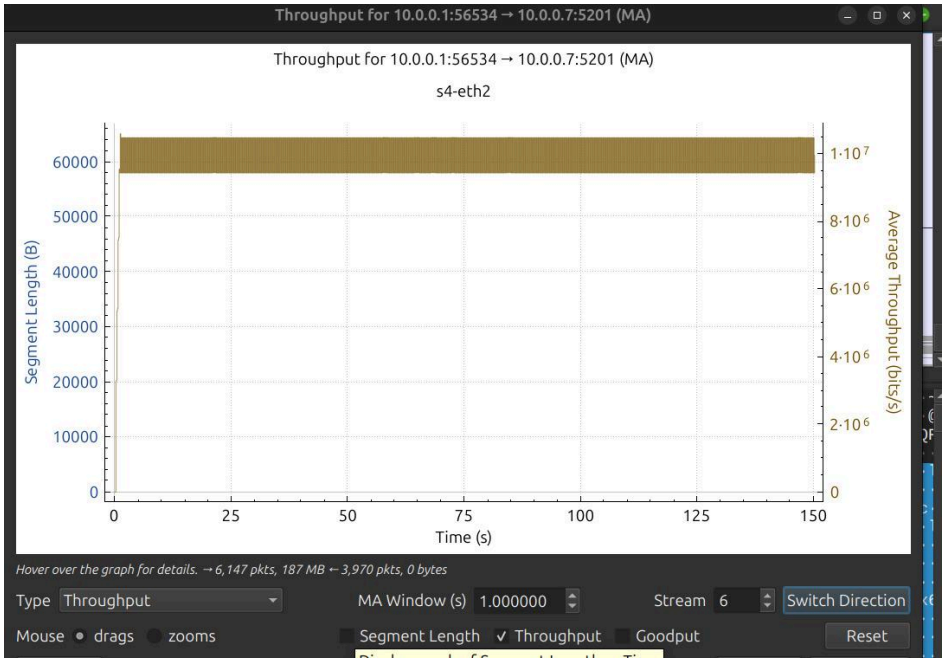
3. Packet Loss Rate



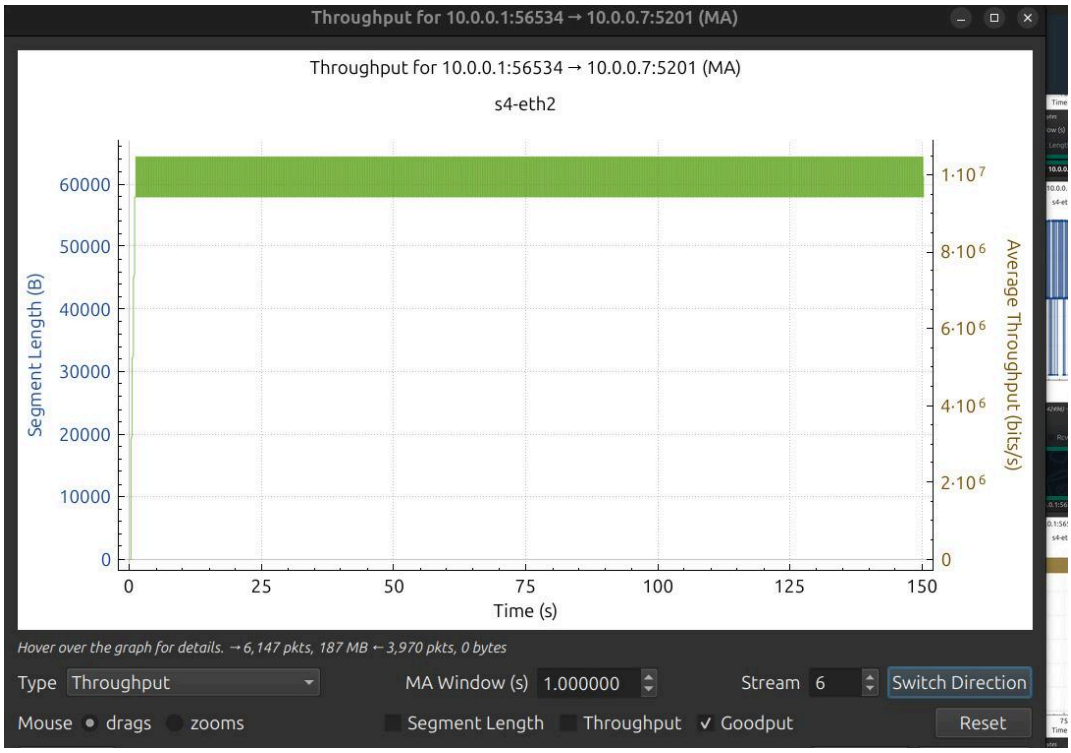
4. Maximum Window size achieved

Protocol: HIGHSPEED

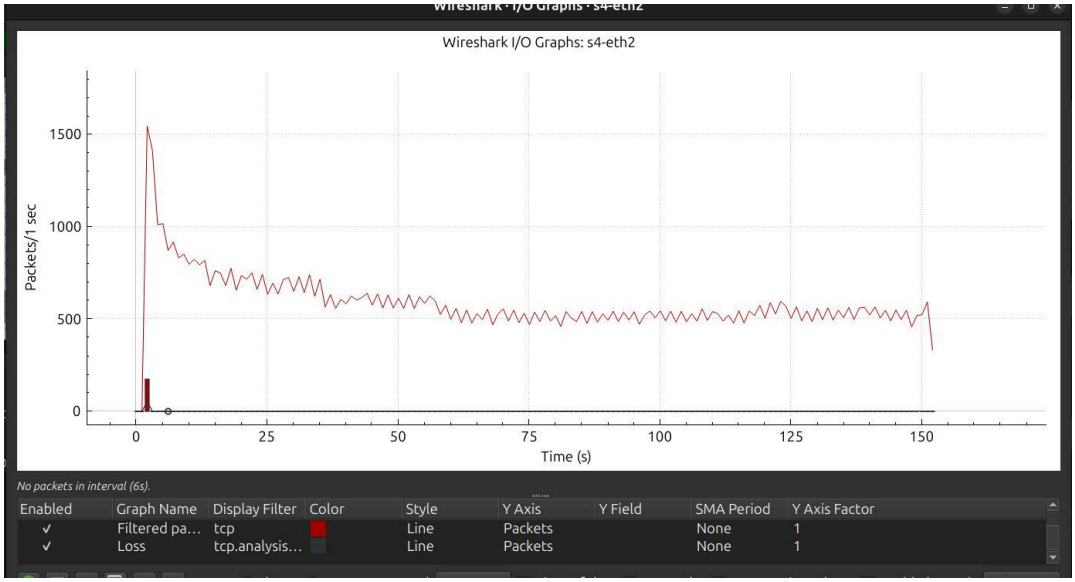
1. Throughput



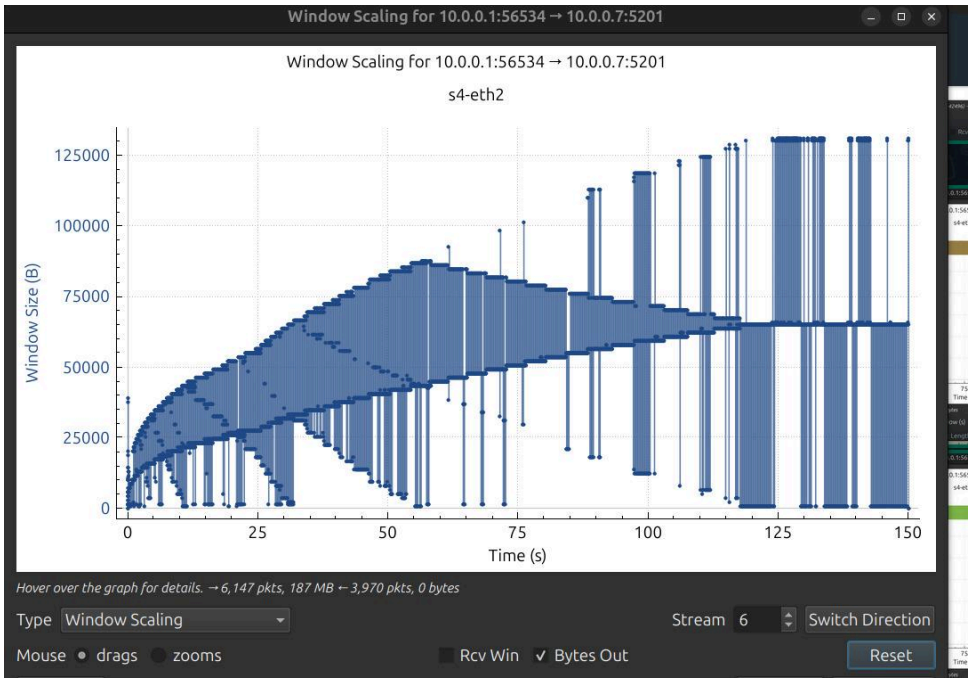
2. Goodput



3. Packet Loss Rate



4. Maximum Window size achieved



Observation

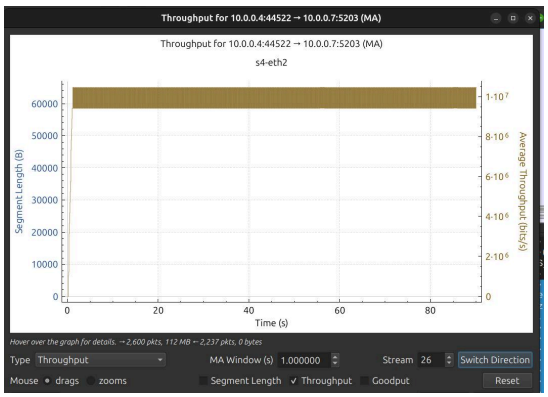
All the graphs from each congestion control protocol appear similar when using a single client connection. This could be because a single client does not generate significant congestion in the network, causing all congestion control protocols to behave similarly with only minor differences.

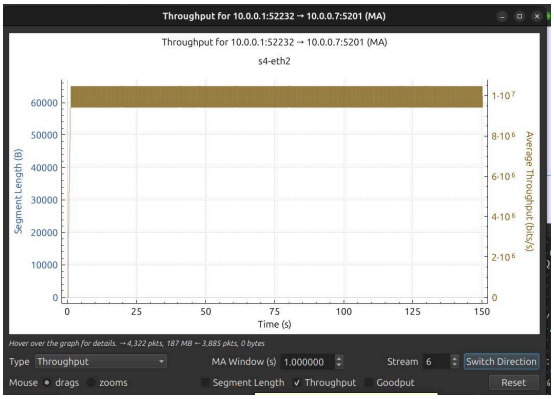
Part b

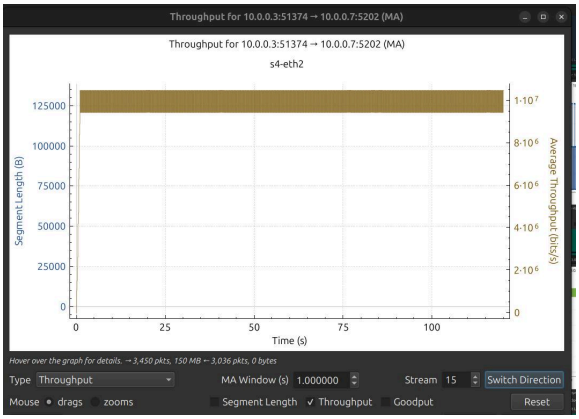
- The second part involved running the iperf3 client on H1, H3 and H4 in a staggered manner(H1 starts at T=0s and runs for 150s, H3 at T=15s and runs for T=120s, H4 at T=30s and runs for 90s) and the server on H7 for 150 seconds.
- Ran code for each congestion control algorithm (Reno, BIC, and HighSpeed), capturing the network traffic for each using wireshark on s4-eth2 interface and getting plots required.

Protocol: RENO

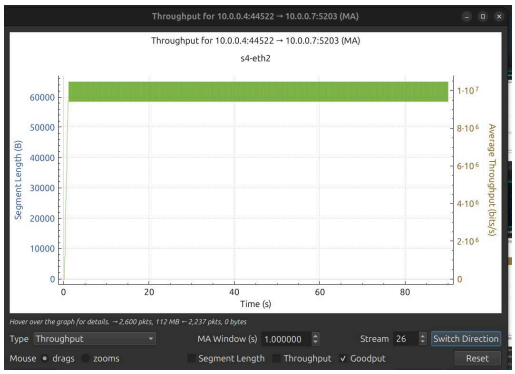
5. Throughput

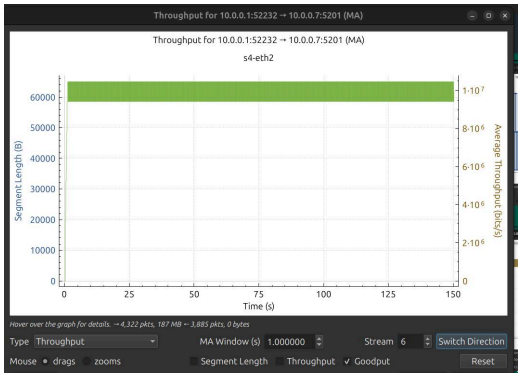


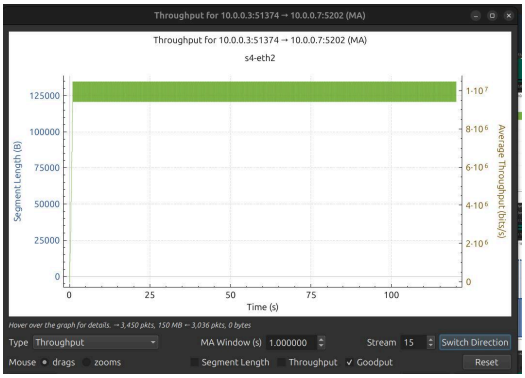




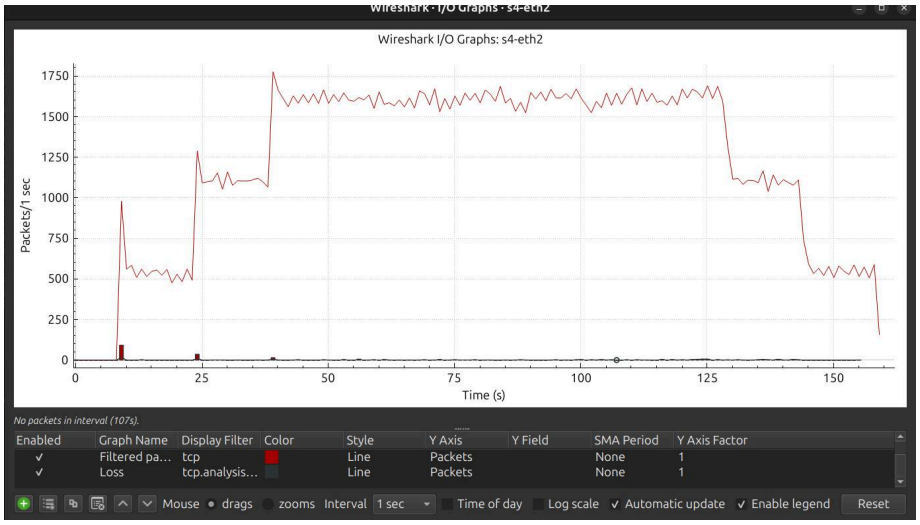
6. Goodput



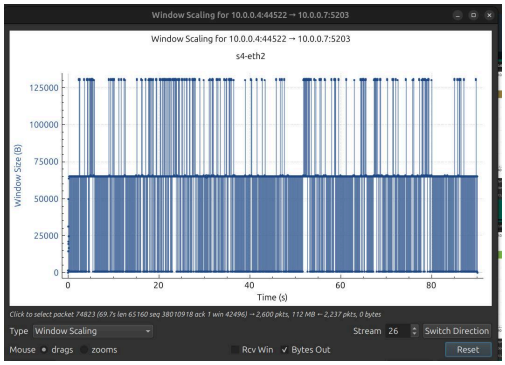




7. Packet Loss Rate

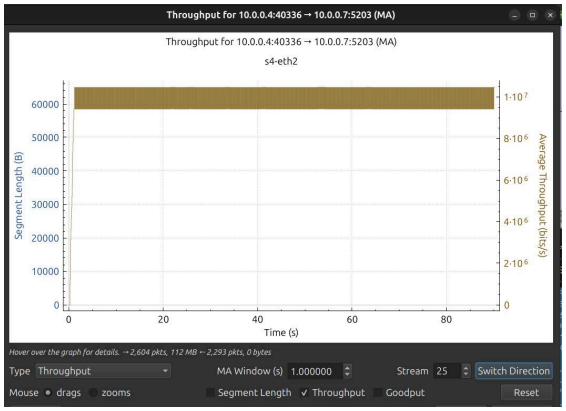


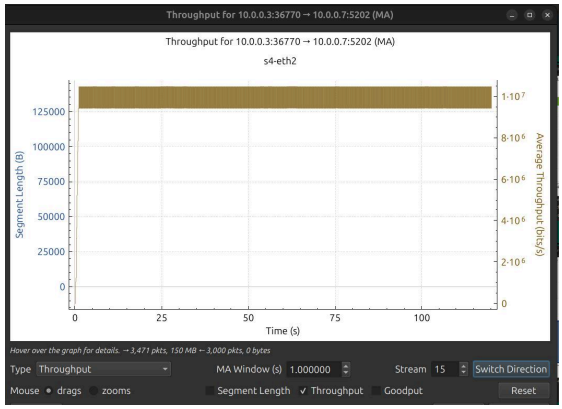
8. Maximum Window size achieved

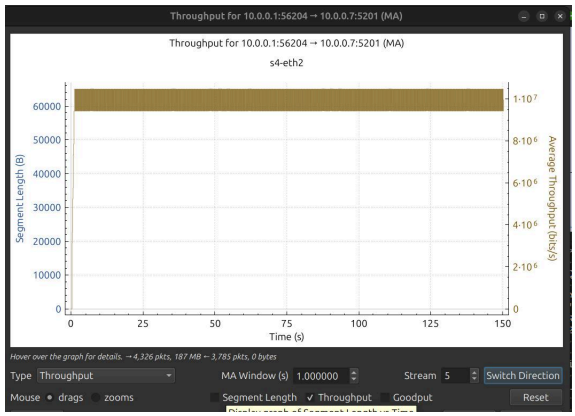


Protocol: BIC

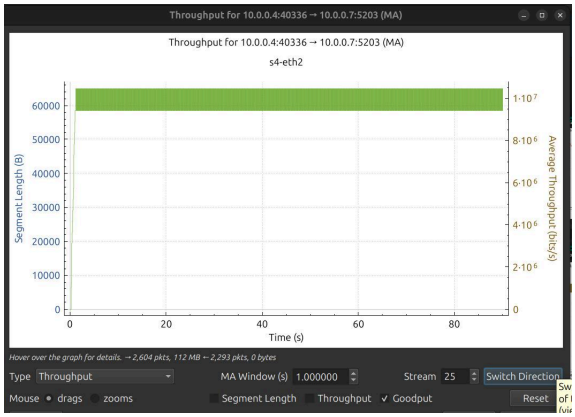
1. Throughput

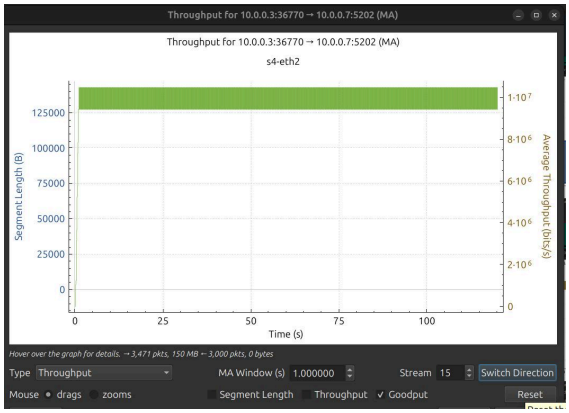


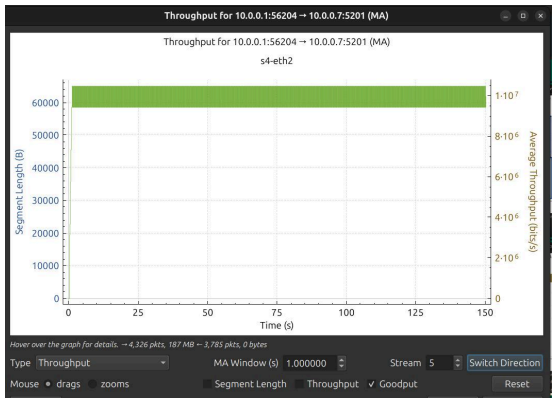




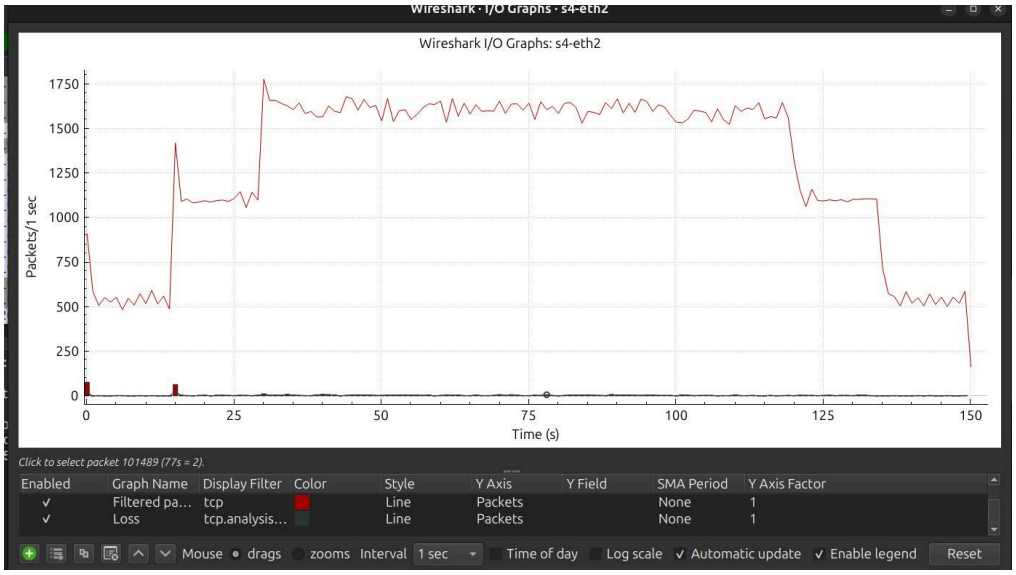
2. Goodput



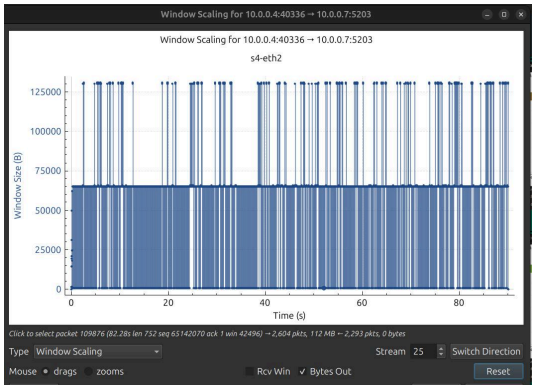


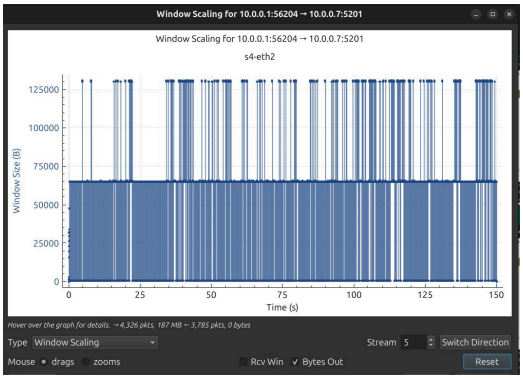


3. Packet Loss Rate



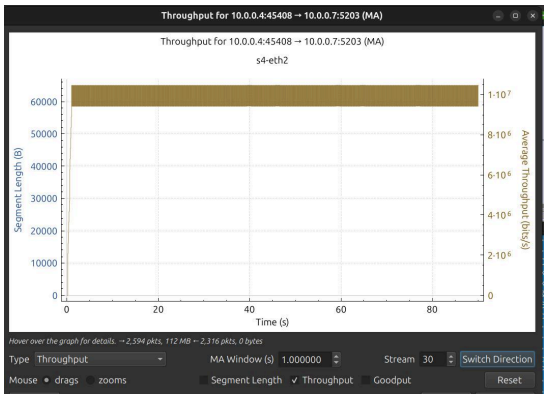
4. Maximum Window size achieved

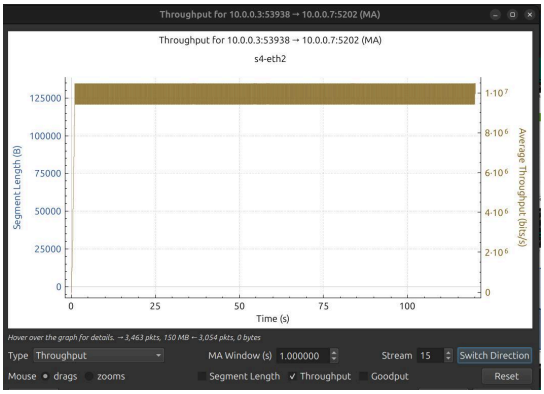


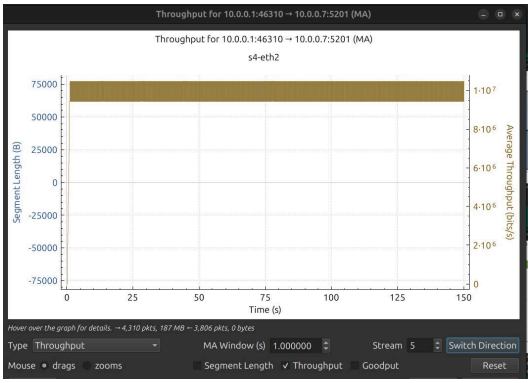


Protocol: HIGHSPEED

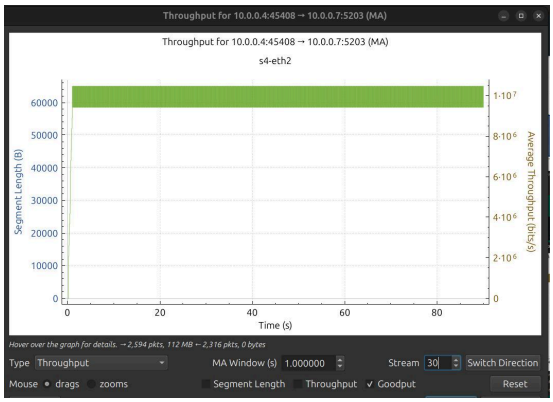
1. Throughput

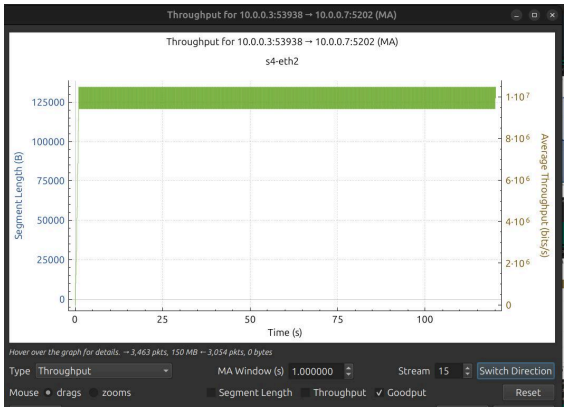

Throughput for 10.0.0.4:45408 → 10.0.0.7:5203 (MA)
Throughput for 10.0.0.4:45408 → 10.0.0.7:5203 (MA)
s4-eth2
Segment Length (B)
Average Throughput (bits/s)
Time (s)
Hover over the graph for details. →2,594 pkts, 112 MB ← 2,316 pkts, 0 bytes
Type Throughput
MA Window (s) 1.000000
Stream 30
Switch Direction
Mouse drags zooms
Segment Length Throughput Goodput
Reset

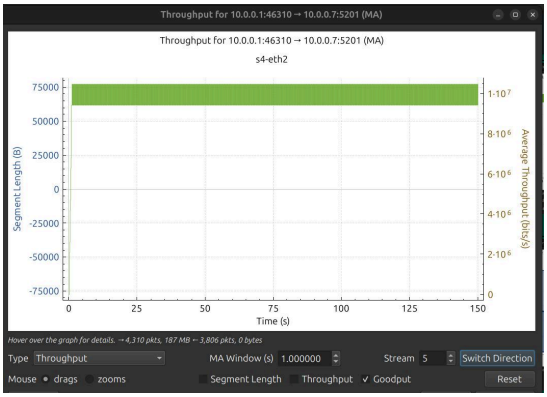

Throughput for 10.0.0.3:53938 → 10.0.0.7:5202 (MA)
Throughput for 10.0.0.3:53938 → 10.0.0.7:5202 (MA)
s4-eth2
Segment Length (B)
Average Throughput (bits/s)
Time (s)
Hover over the graph for details. →3,463 pkts, 150 MB ← 3,054 pkts, 0 bytes
Type Throughput
MA Window (s) 1.000000
Stream 15
Switch Direction
Mouse drags zooms
Segment Length Throughput Goodput
Reset


Throughput for 10.0.0.1:46310 → 10.0.0.7:5201 (MA)
Throughput for 10.0.0.1:46310 → 10.0.0.7:5201 (MA)
s4-eth2
Segment Length (B)
Average Throughput (bits/s)
Time (s)
Hover over the graph for details. →4,310 pkts, 187 MB ← 3,806 pkts, 0 bytes
Type Throughput
MA Window (s) 1.000000
Stream 5
Switch Direction
Mouse drags zooms
Segment Length Throughput Goodput
Reset

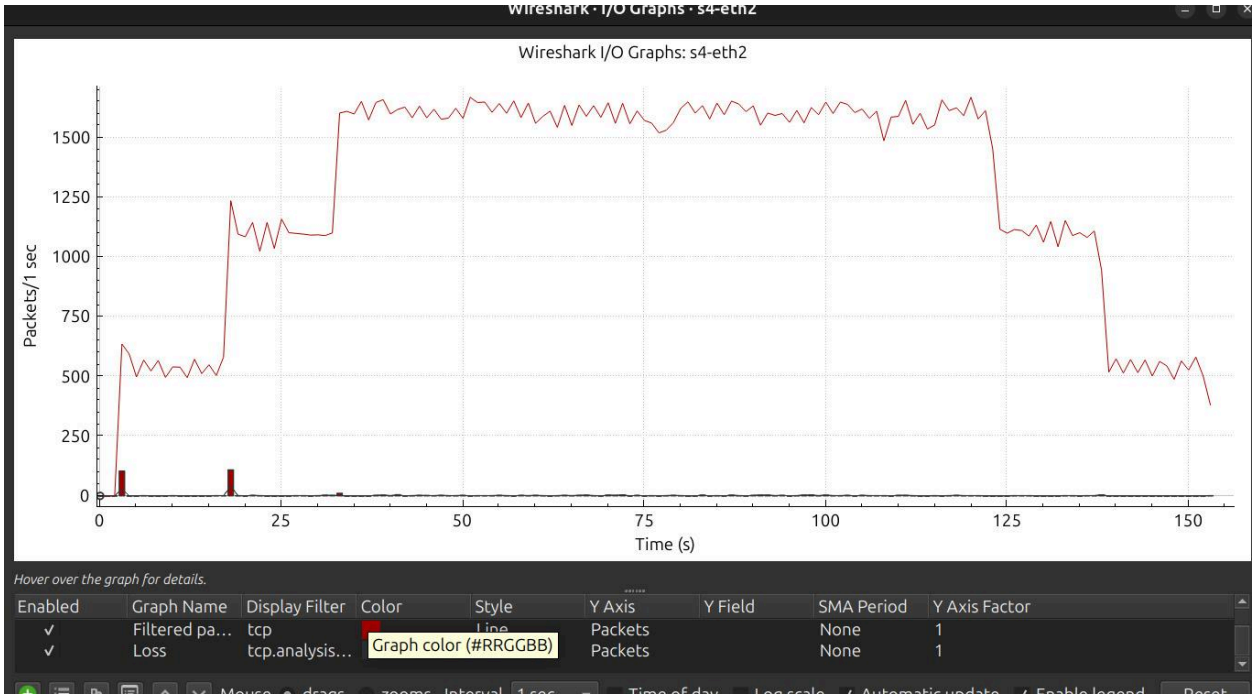
2. Goodput


Throughput for 10.0.0.4:45408 → 10.0.0.7:5203 (MA)
Throughput for 10.0.0.4:45408 → 10.0.0.7:5203 (MA)
s4-eth2
Segment Length (B)
Average Throughput (bits/s)
Time (s)
Hover over the graph for details. →2,594 pkts, 112 MB ← 2,316 pkts, 0 bytes
Type Throughput
MA Window (s) 1.000000
Stream 30
Switch Direction
Mouse drags zooms
Segment Length Throughput Goodput
Reset

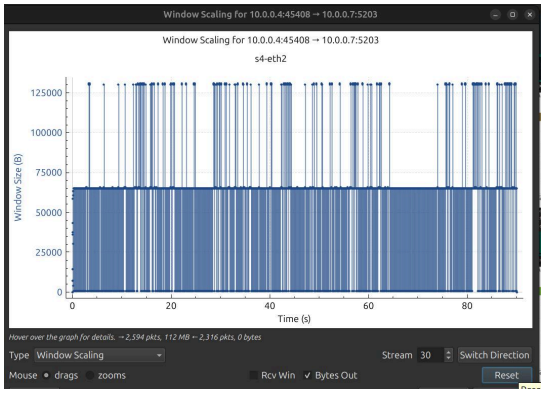

Throughput for 10.0.0.3:53938 → 10.0.0.7:5202 (MA)
Throughput for 10.0.0.3:53938 → 10.0.0.7:5202 (MA)
s4-eth2
Segment Length (B)
Average Throughput (bits/s)
Time (s)
Hover over the graph for details. →3,463 pkts, 150 MB ← 3,054 pkts, 0 bytes
Type Throughput
MA Window (s) 1.000000
Stream 15
Switch Direction
Mouse drags zooms
Segment Length Throughput Goodput
Reset

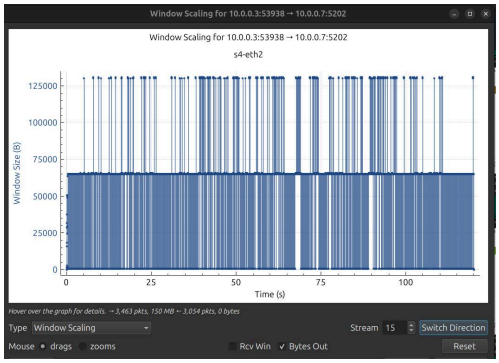

Throughput for 10.0.0.1:46310 → 10.0.0.7:5201 (MA)
Throughput for 10.0.0.1:46310 → 10.0.0.7:5201 (MA)
s4-eth2
Segment Length (B)
Average Throughput (bits/s)
Time (s)
Hover over the graph for details. →4,310 pkts, 187 MB ← 3,806 pkts, 0 bytes
Type Throughput
MA Window (s) 1.000000
Stream 5
Switch Direction
Mouse drags zooms
Segment Length Throughput Goodput
Reset

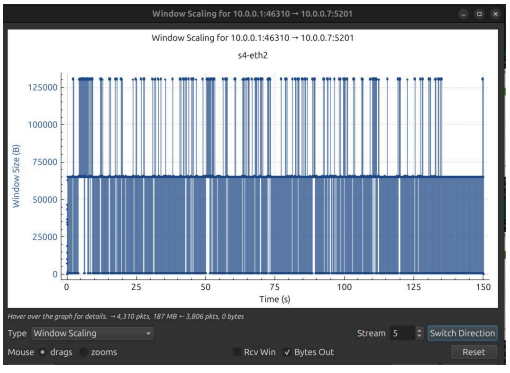
3. Packet Loss Rate



4. Maximum Window size achieved







Observation

With multiple staggered clients, the throughput and goodput graphs show slight gaps when zoomed in. This could be due to the server managing connections in a round-robin manner, where each client experiences some delay while waiting for an acknowledgment from the server

due to multiple active connections. Additionally, there is a slight packet loss. However, the overall behavior remains similar, with no significant difference observed between single and staggered client scenarios

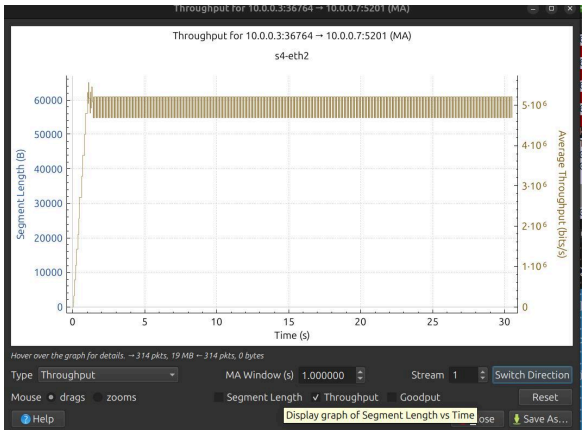
Part c

- The third part involved configuring the links with the following bandwidths:
    - Link S1-S2: 100Mbps
    - Links S2-S3: 50Mbps
    - Links S3-S4: 100Mbps
- Running for four separate scenarios:
    1. Link S2-S4 is active with client on H3 and server on H7.
    2. Link S1-S4 is active with:
        a. Running client on H1 and H2 and server on H7
        b. Running client on H1 and H3 and server on H7
        c. Running client on H1, H3 and H4 and server on H7
- Ran code for each congestion control algorithm (Reno, BIC, and HighSpeed) for all four scenarios, capturing the network traffic for each using wireshark on s4-eth2 interface and getting plots required.

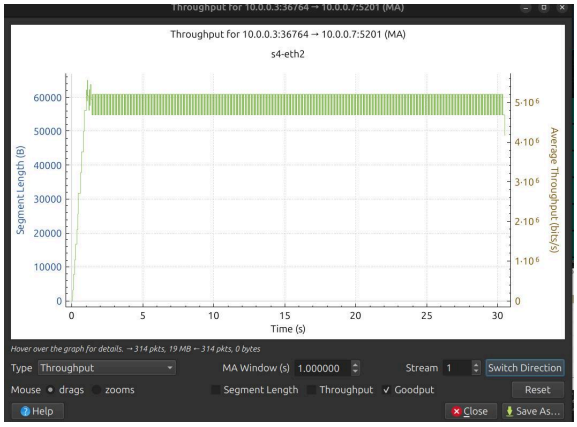
Scenario 1

Protocol: RENO

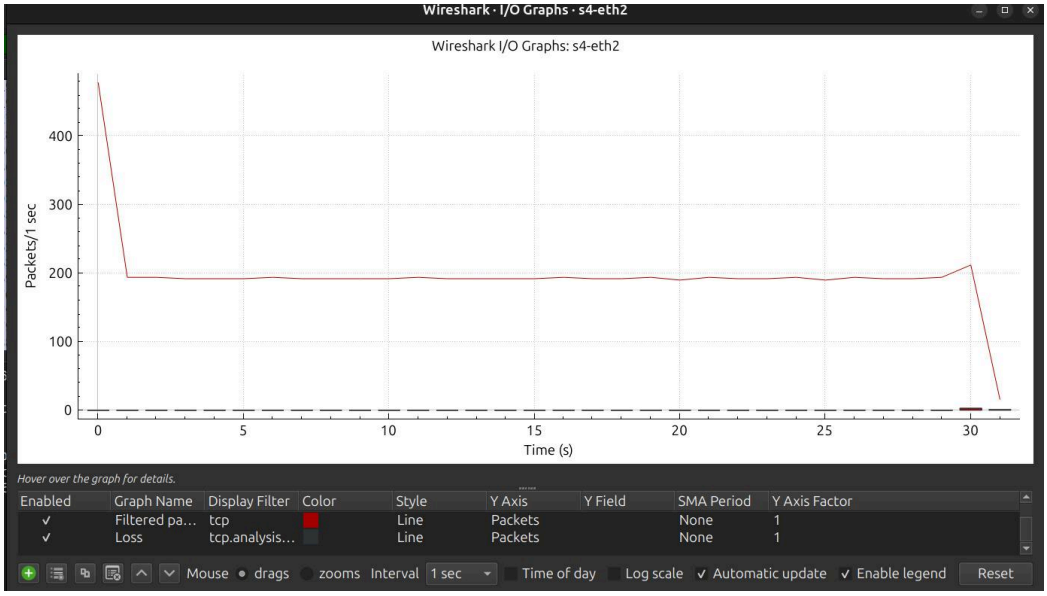
1. Throughput



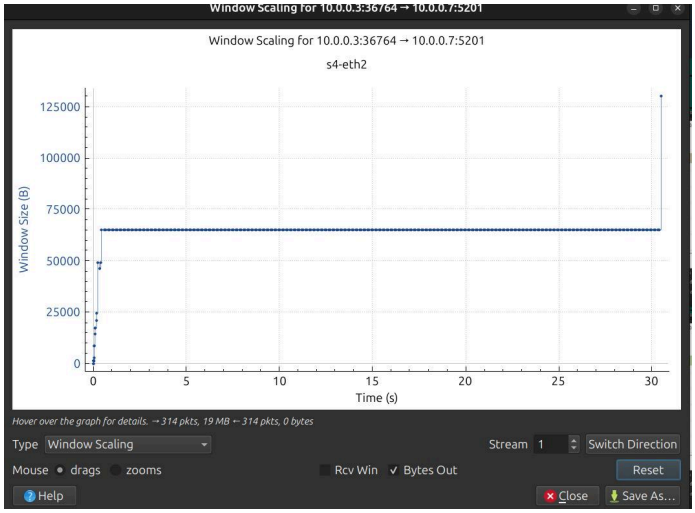
2. Goodput



3. Packet Loss Rate

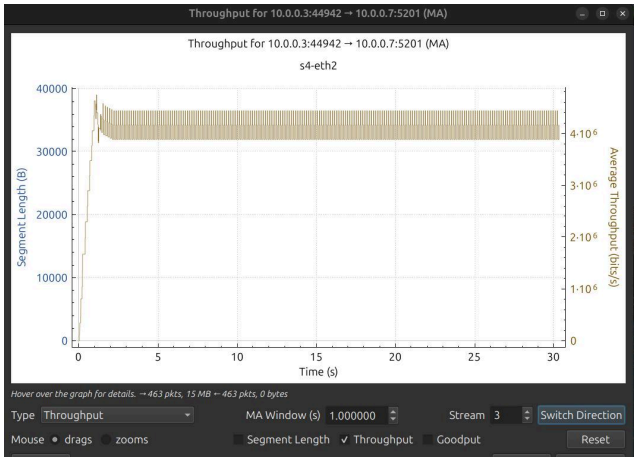


4. Maximum Window size achieved

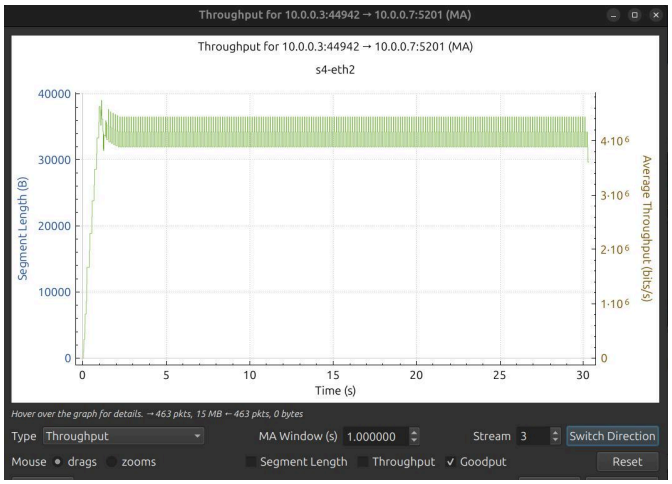


Protocol: BIC

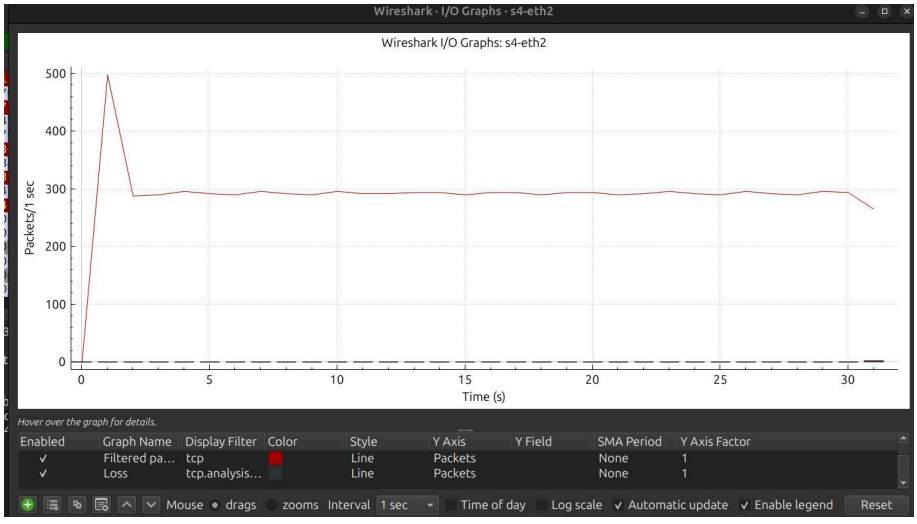
1. Throughput



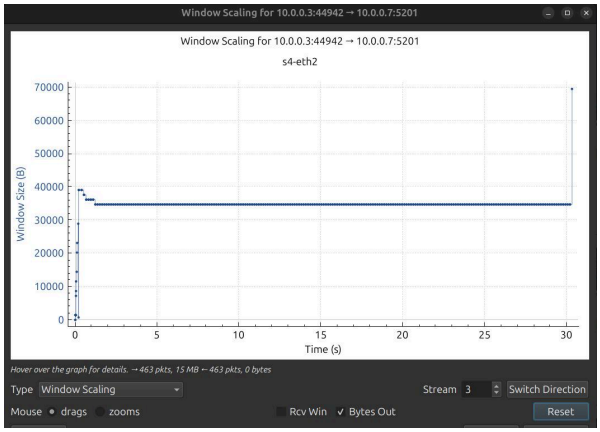
2. Goodput



3. Packet Loss Rate

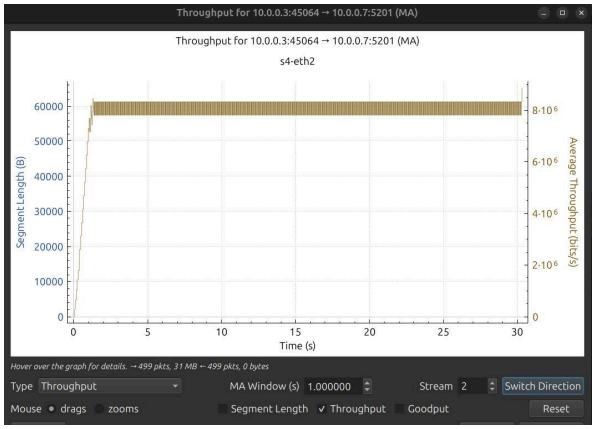


4. Maximum Window size achieved

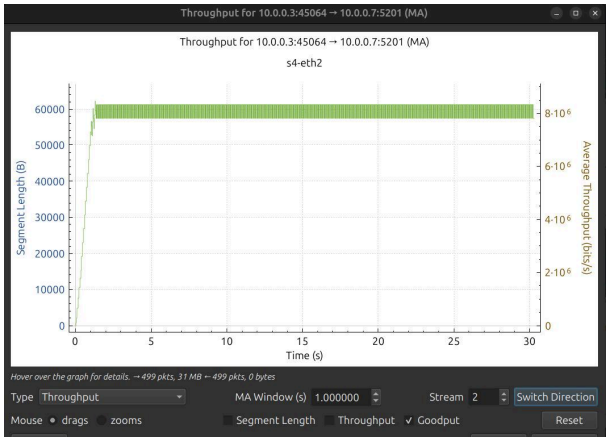


Protocol: HIGHSPEED

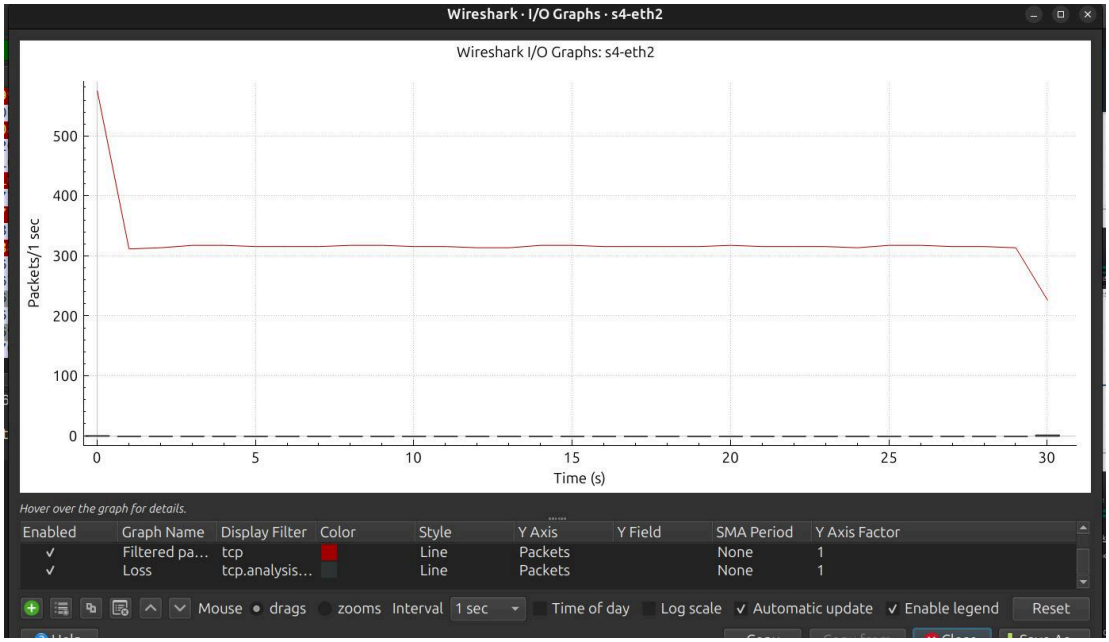
1. Throughput



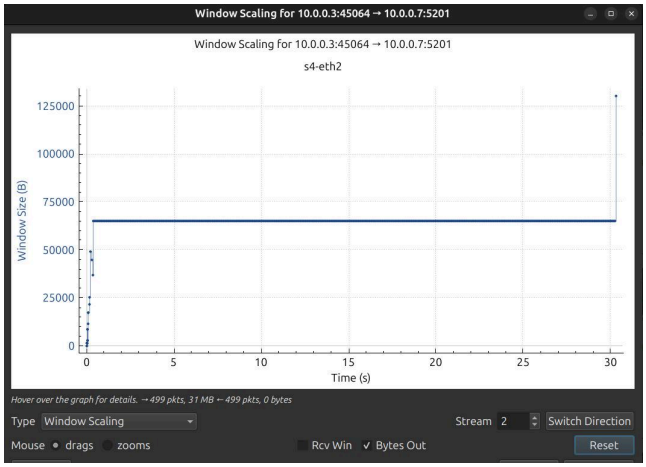
2. Goodput



3. Packet Loss Rate



4. Maximum Window size achieved



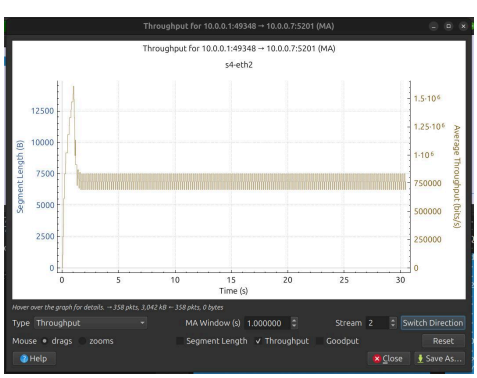
Observation

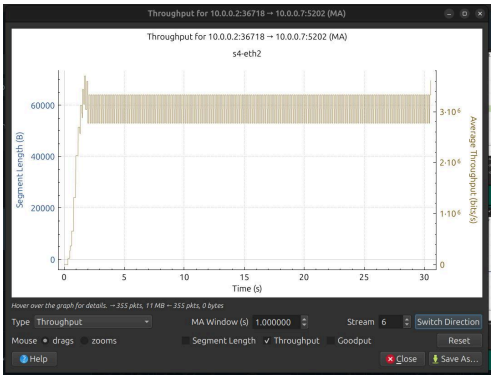
In this scenario, a single client (H3) transmitted data to the server (H7) through a bottleneck link (S2-S3) with a lower bandwidth (50 Mbps), while other links operated at 100 Mbps. Since only one client was active, the throughput remained consistent, limited only by the bottleneck capacity. TCP congestion control mechanisms gradually increased the congestion window (cwnd) until the bottleneck link reached full utilization. The throughput graph showed a steady increase during the slow start phase, followed by a plateau at around 50 Mbps. With no competing traffic, packet loss was minimal, and the goodput closely followed throughput with a slight overhead from TCP headers.

Scenario 2a

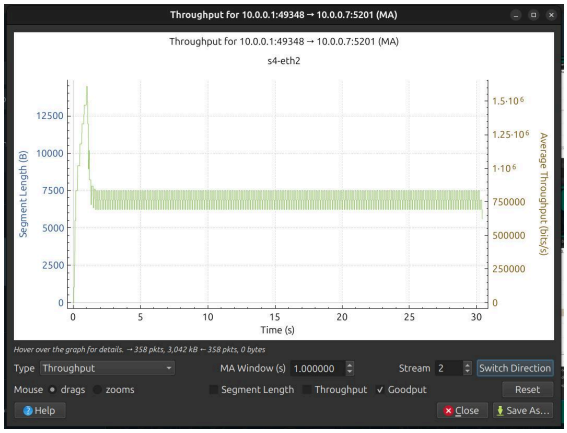
Protocol: RENO

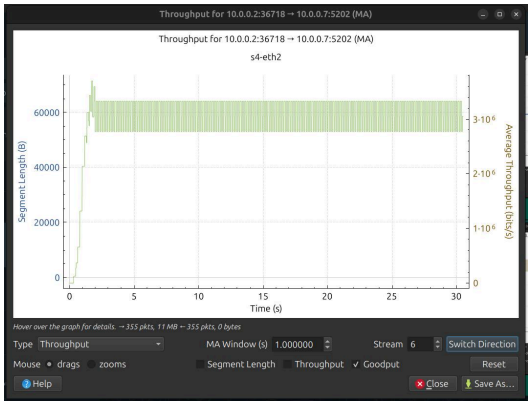
1. Throughput



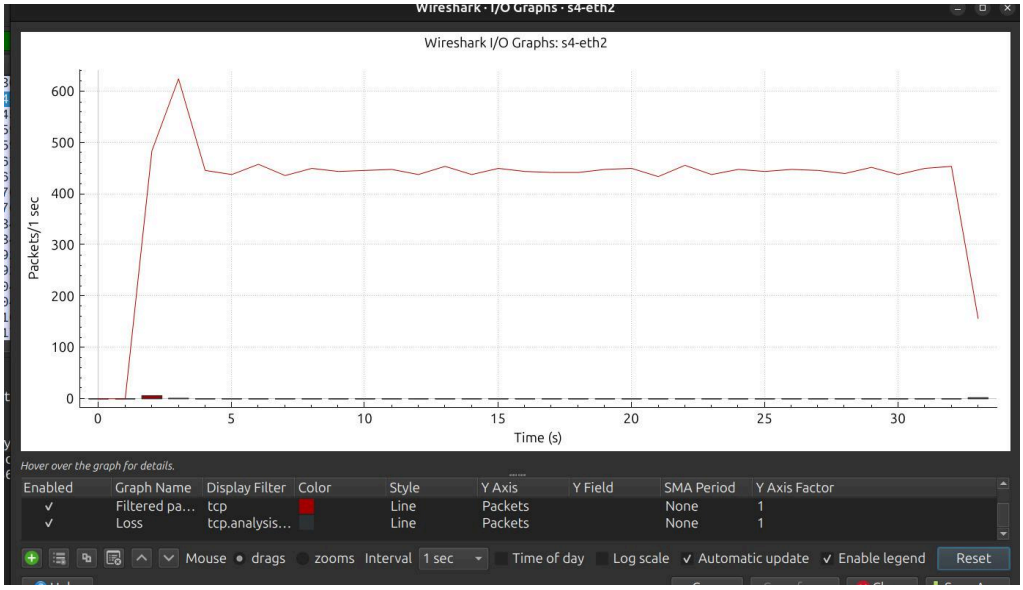


2. Goodput

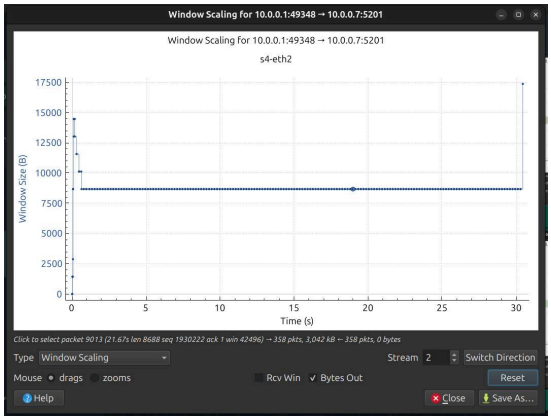


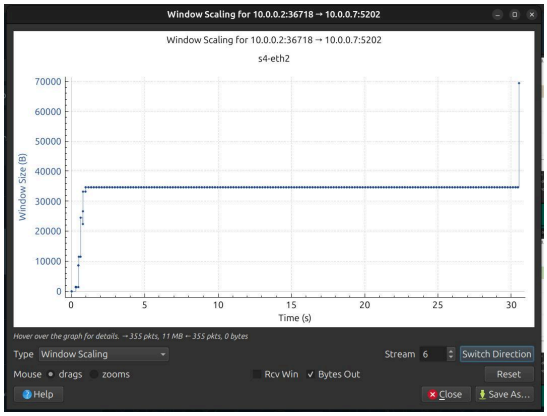


3. Packet Loss Rate



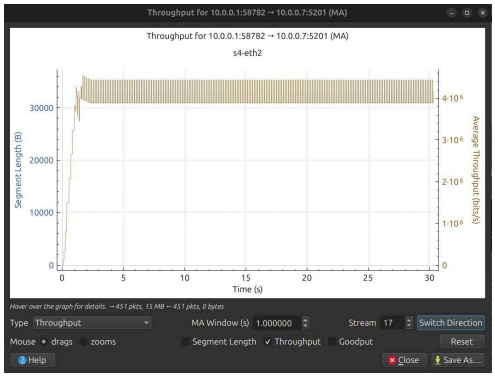
4. Maximum Window size achieved

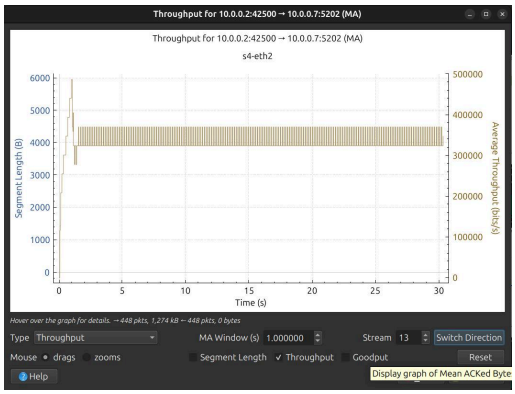




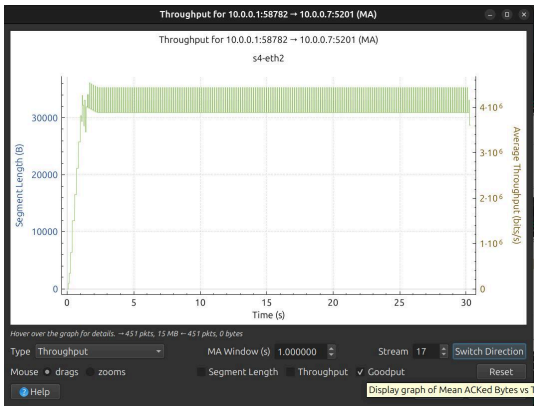
Protocol: BIC

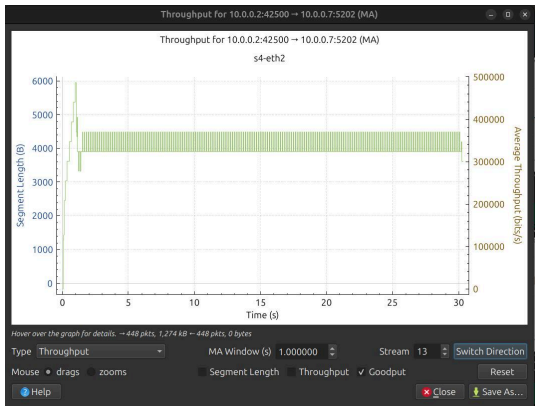
1. Throughput



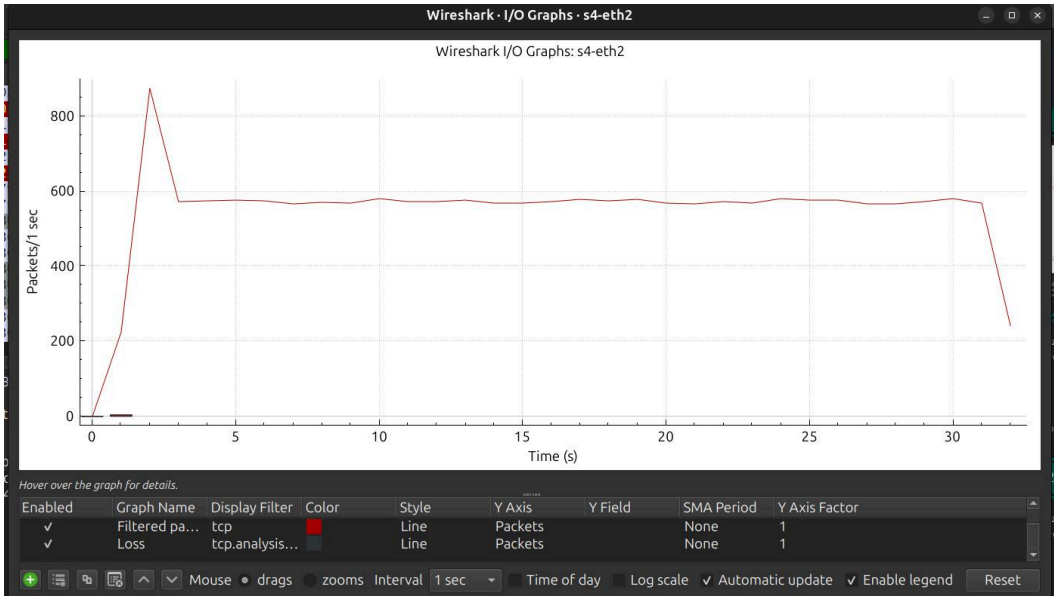


2. Goodput

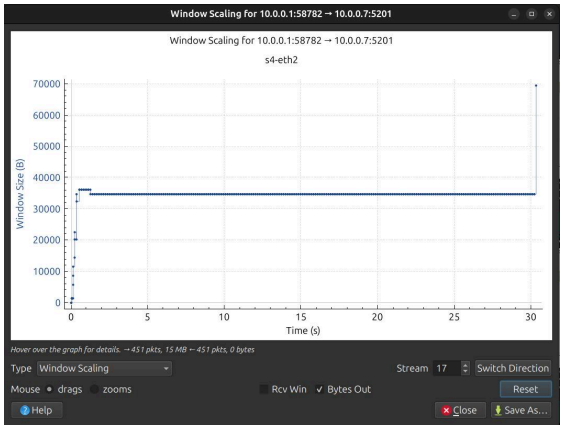


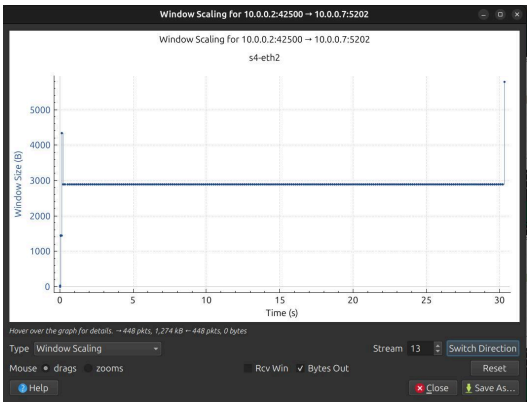


3. Packet Loss Rate



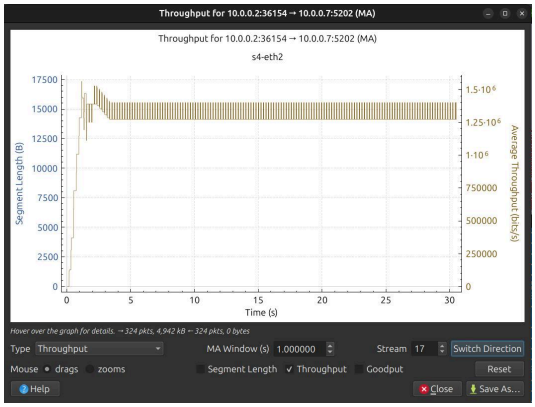
4. Maximum Window size achieved

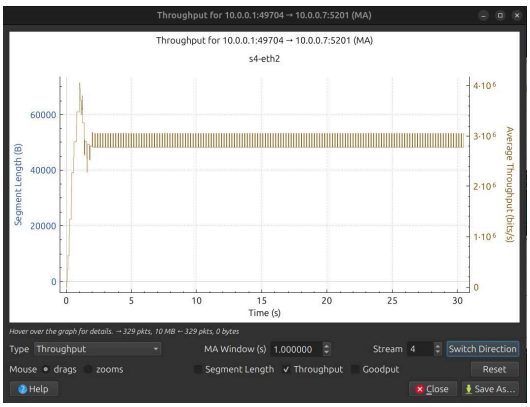




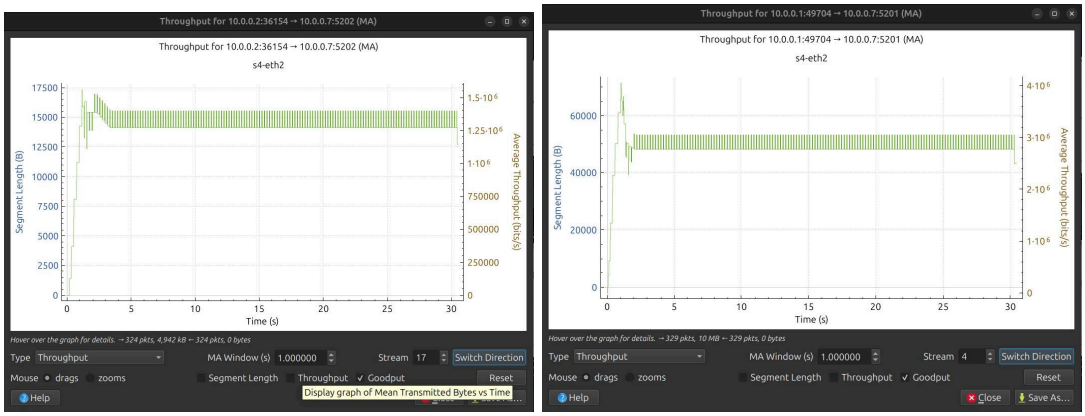
Protocol: HIGHSPEED

1. Throughput

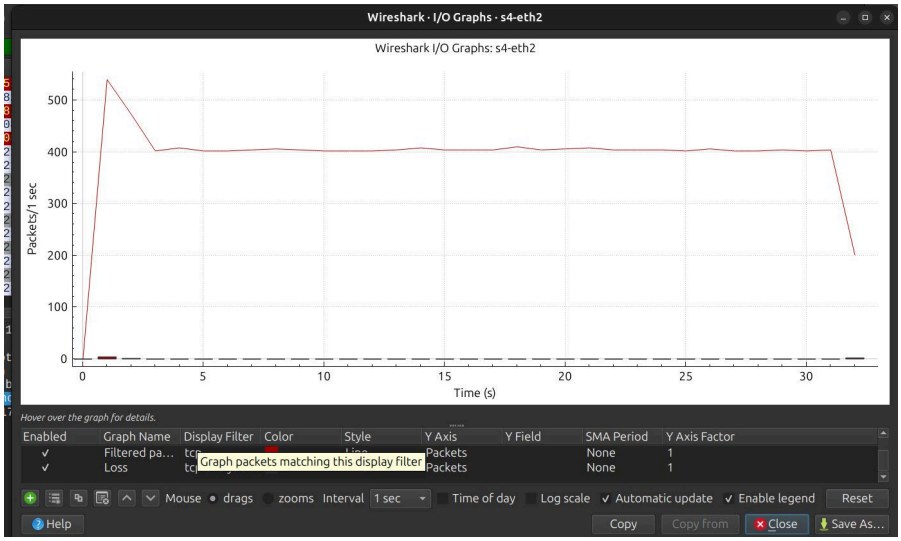




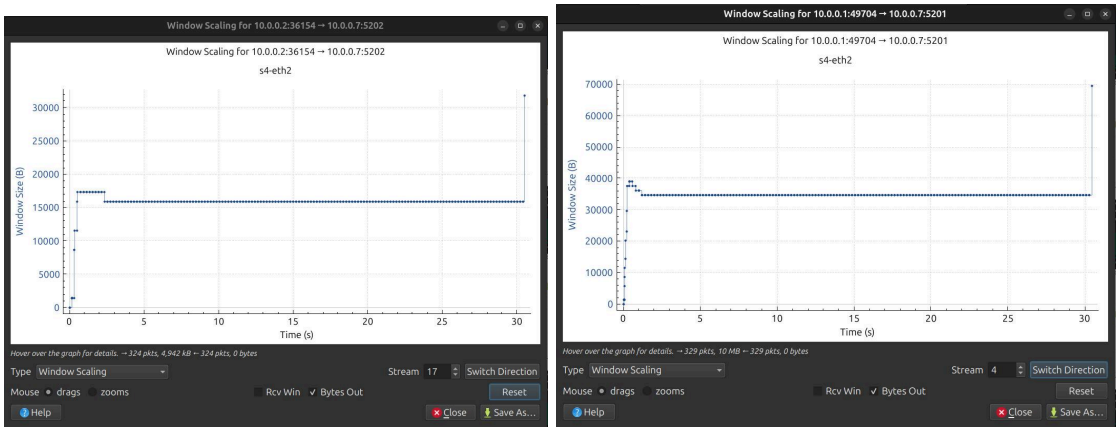
2. Goodput



3. Packet Loss Rate



4. Maximum Window size achieved



Observation

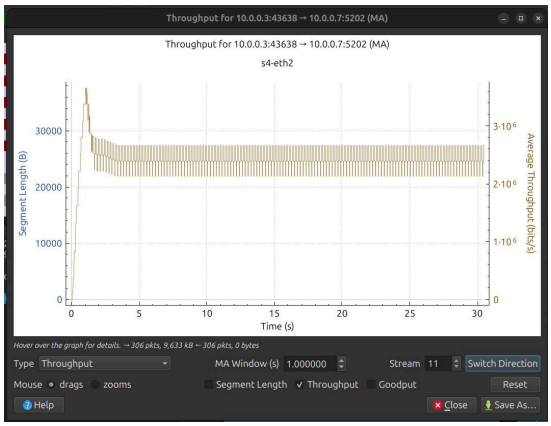
In this case, two clients (H1 and H2) transmitted data simultaneously to H7 via different paths, both passing through S1-S4. Since the network topology allowed both flows to have equal access to available bandwidth, the results showed fair bandwidth sharing between the two clients. TCP Reno struggled to maintain fairness, as it aggressively reduced its congestion

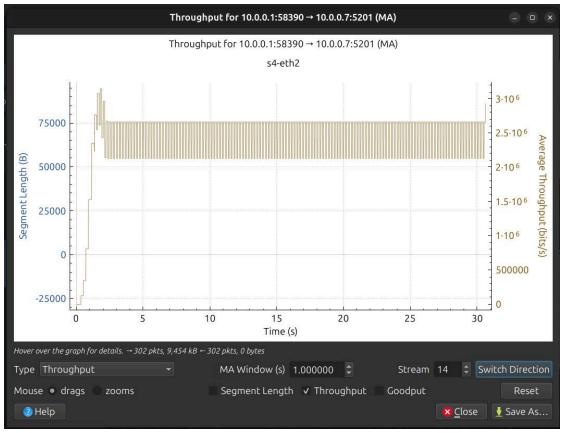
window upon packet loss, which allowed one flow to dominate the available bandwidth at times. BIC and HighSpeed demonstrated more adaptive behavior, resulting in more balanced throughput graphs. However, HighSpeed exhibited aggressive window growth, leading to noticeable fluctuations in throughput. Goodput reflected these trends, with packet loss being slightly higher than in Scenario 1 due to increased congestion. The throughput graphs displayed an initial increase, followed by fluctuations as both clients competed for bandwidth.

Scenario 2b

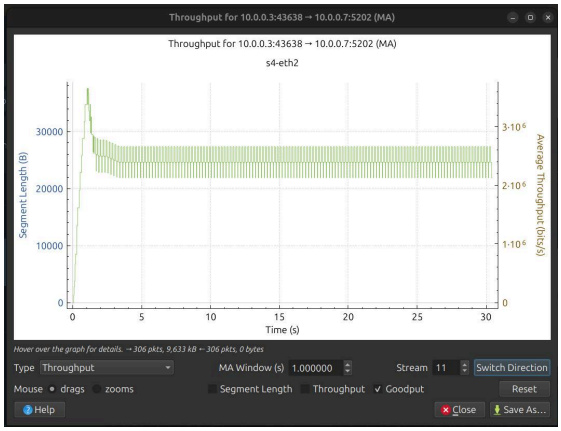
Protocol: RENO

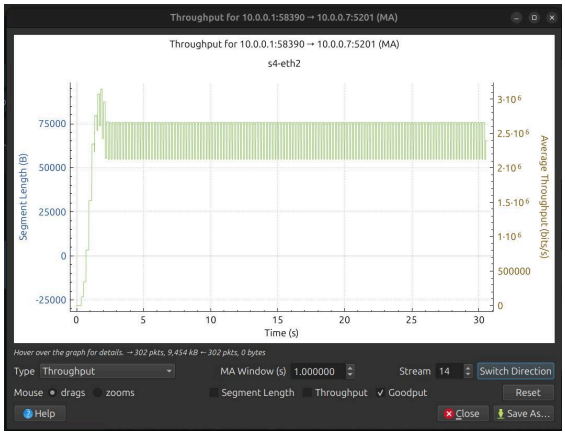
1. Throughput



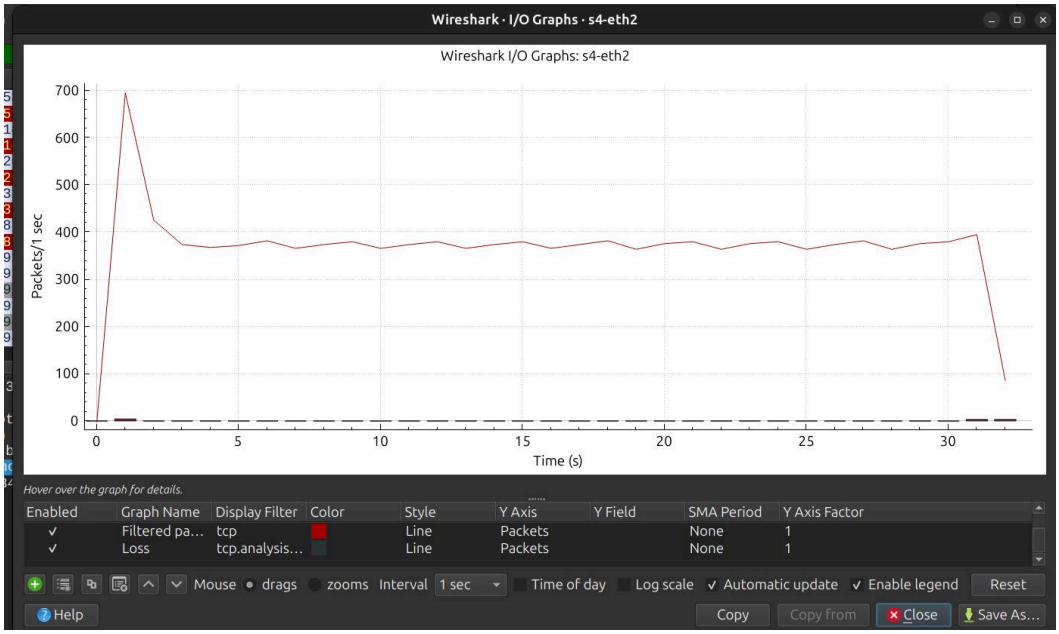


2. Goodput

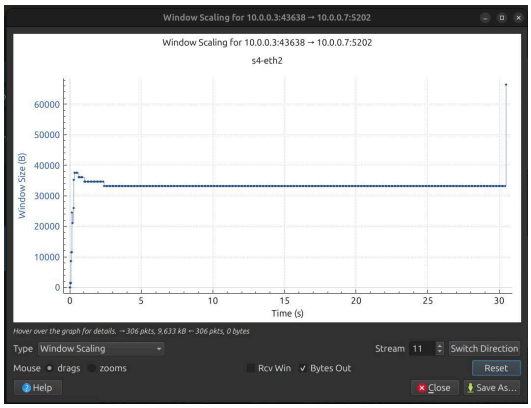


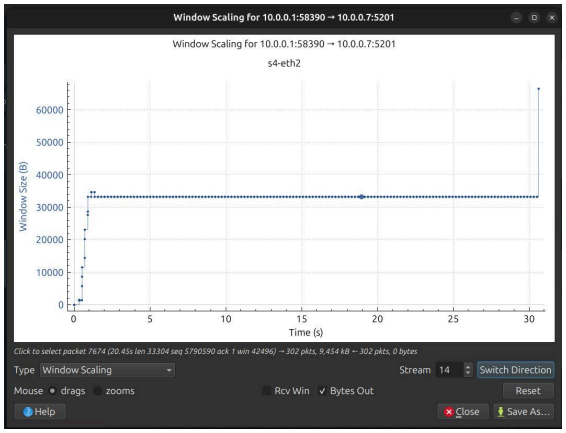


3. Packet Loss Rate



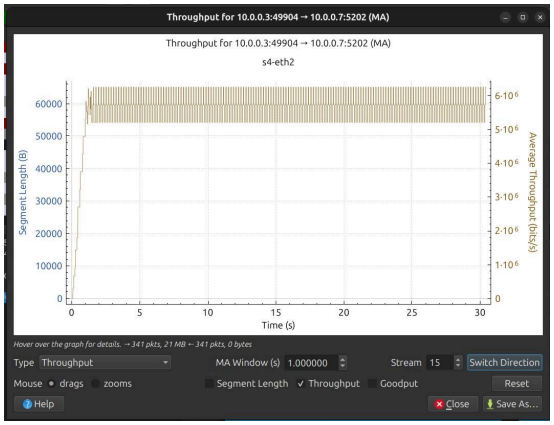
4. Maximum Window size achieved

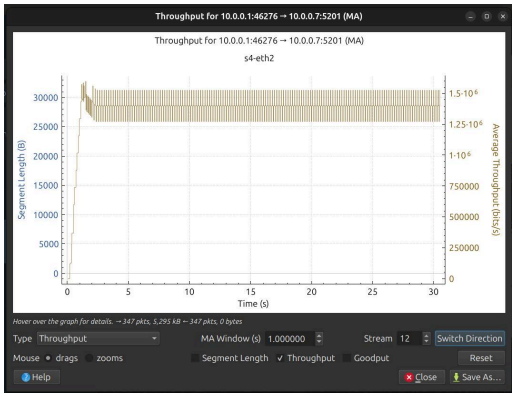




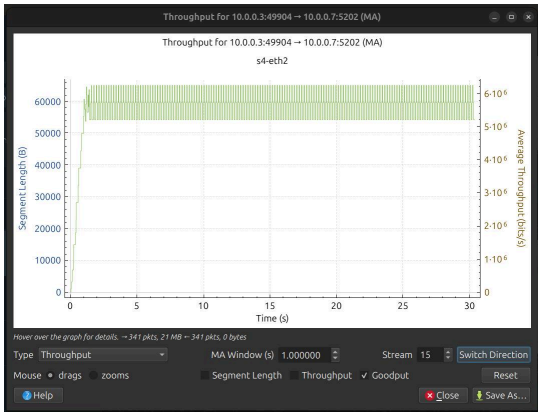
Protocol: BIC

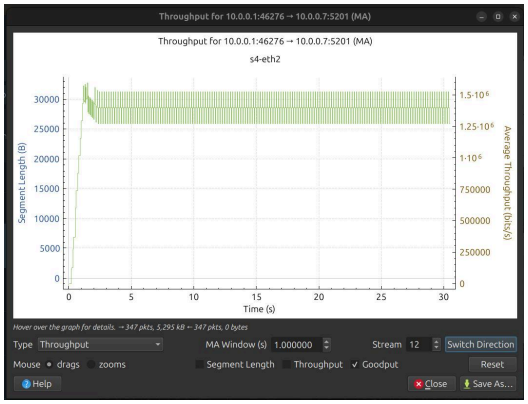
1. Throughput



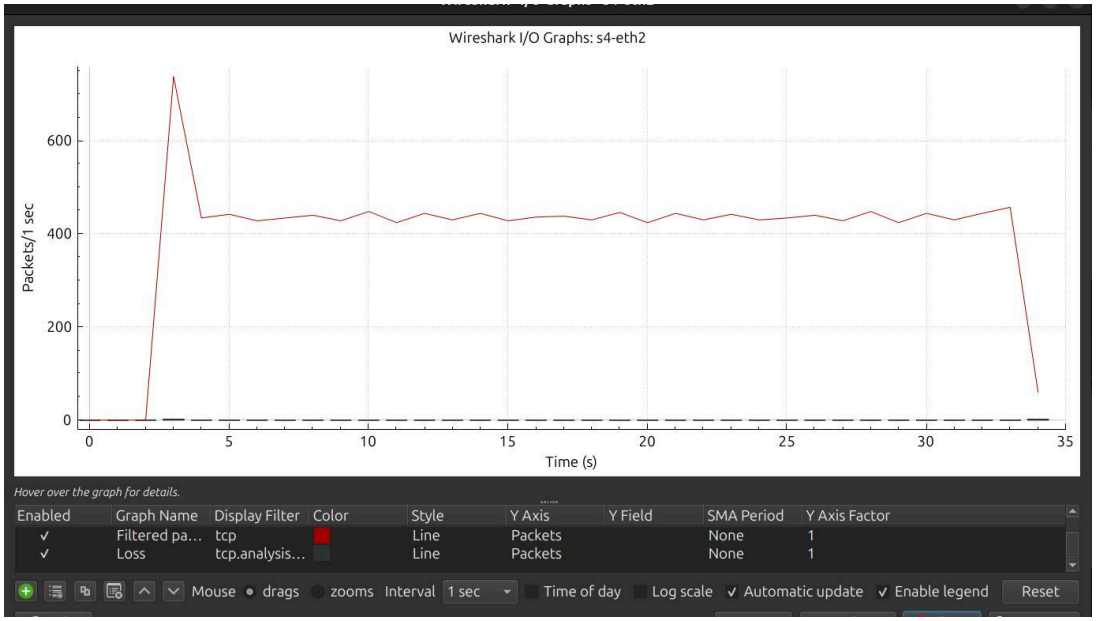


2. Goodput

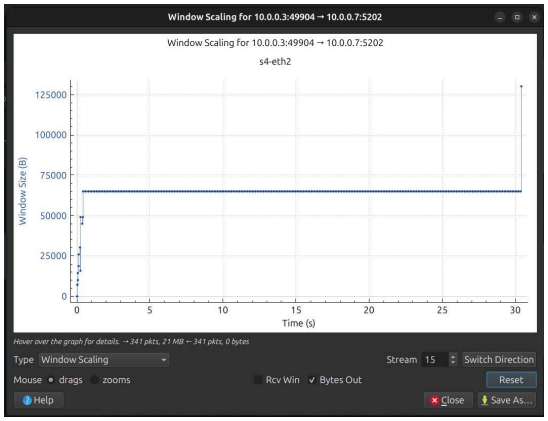


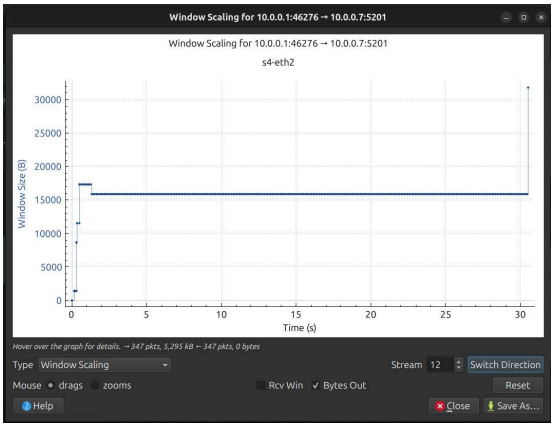


3. Packet Loss Rate



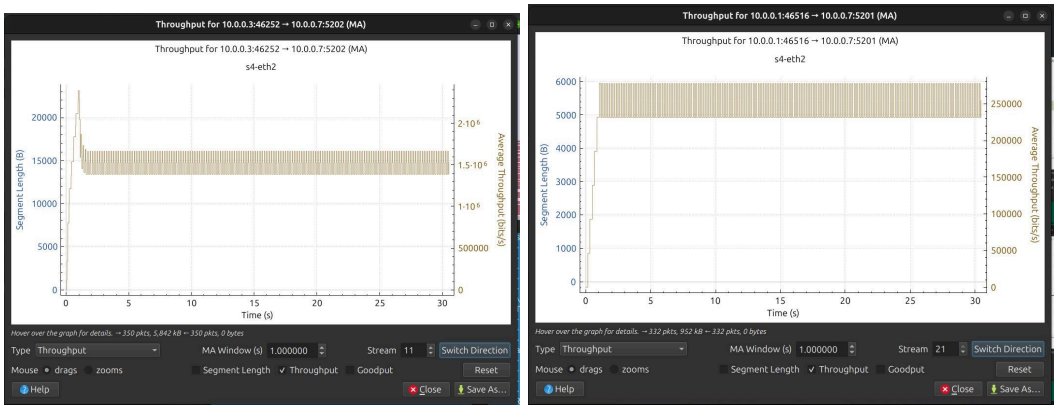
4. Maximum Window size achieved



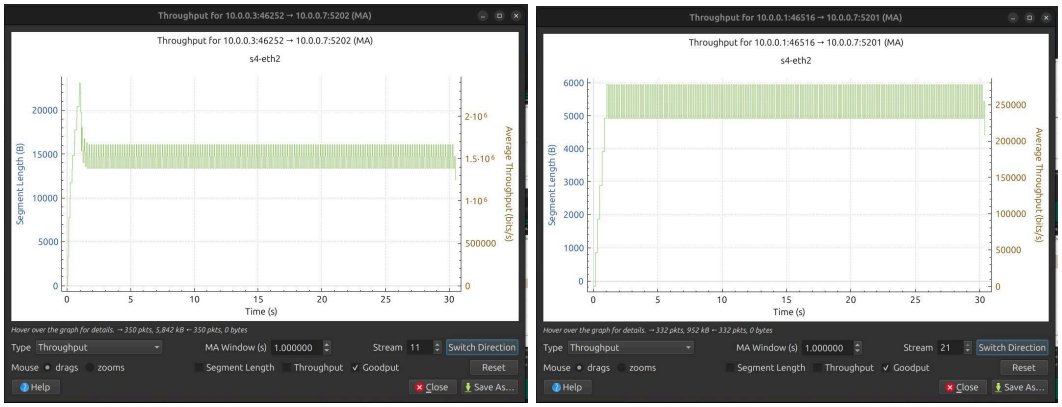


Protocol: HIGHSPEED

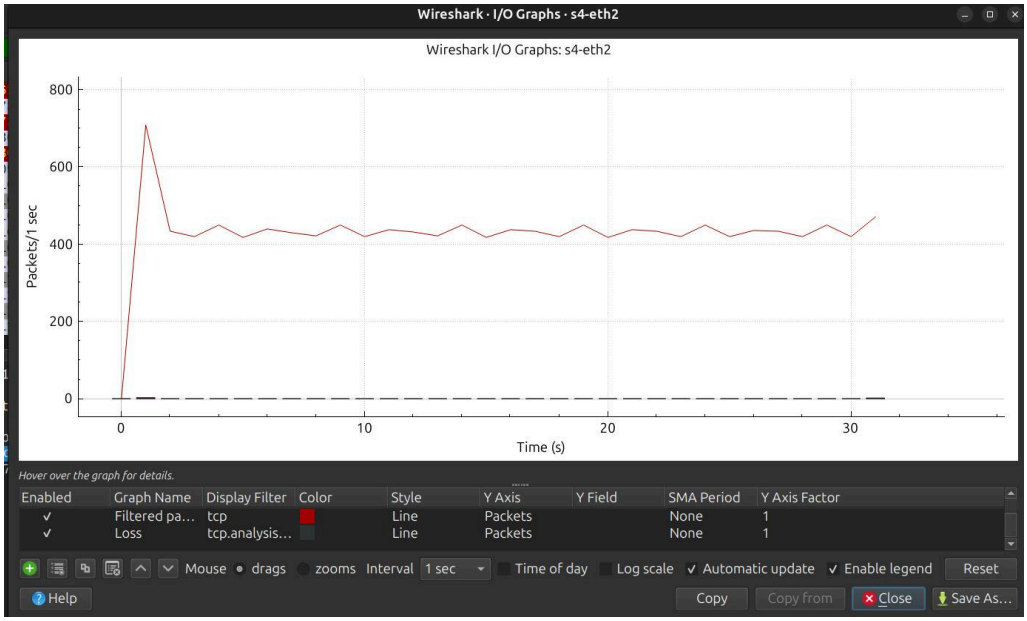
1. Throughput



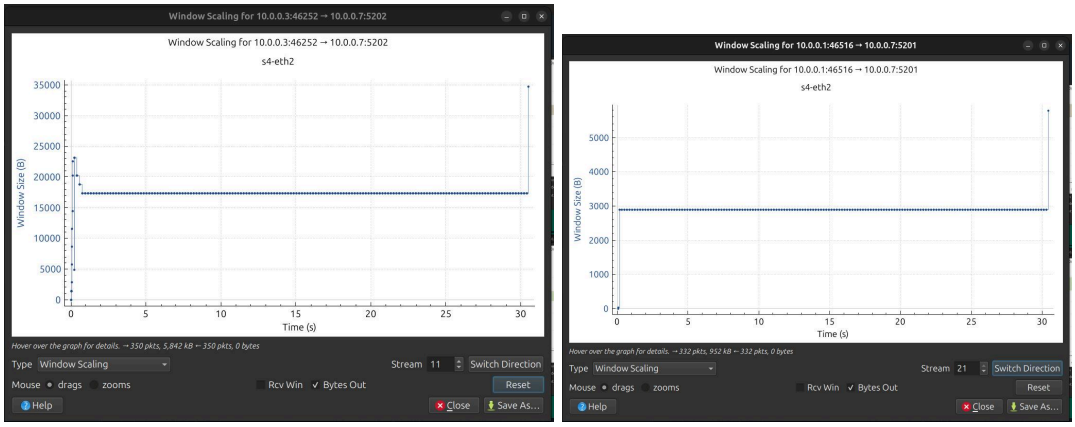
2. Goodput



3. Packet Loss Rate



4. Maximum Window size achieved



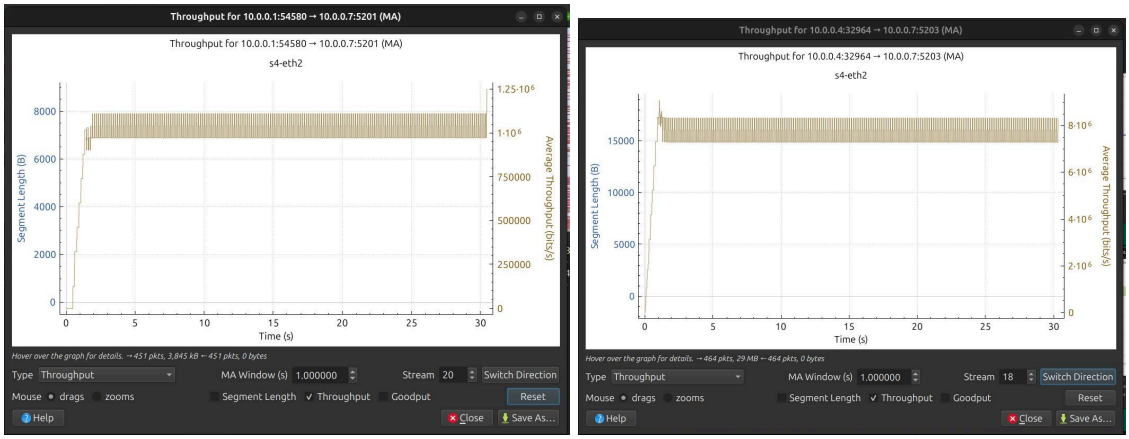
Observation

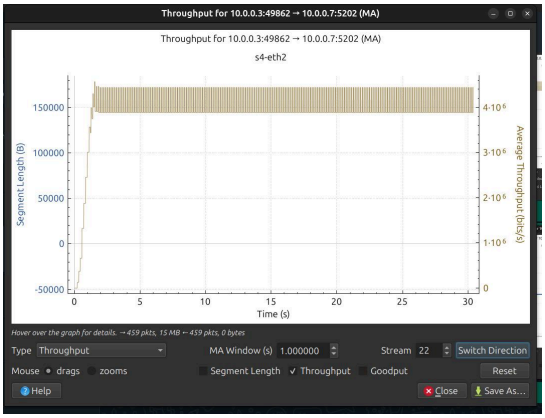
In this case, where H1 and H3 share a bottleneck link (S2-S3 with 50 Mbps), both clients competed to utilize the limited bandwidth, leading to either fair sharing or unfair competition depending on the congestion control protocol. TCP Reno struggled, resulting in unstable throughput with frequent drops, and one connection occasionally dominated due to congestion collapse. TCP BIC demonstrated fairer bandwidth distribution, dynamically adjusting the congestion window size to ensure smoother sharing. TCP HighSpeed initially achieved higher throughput but became unstable due to its aggressive window growth. The throughput graph showed initially high rates for both clients, but as congestion built, fluctuations and fairness issues emerged. The goodput graph remained fairly constant with respect to the throughput graph.

Scenario 2c

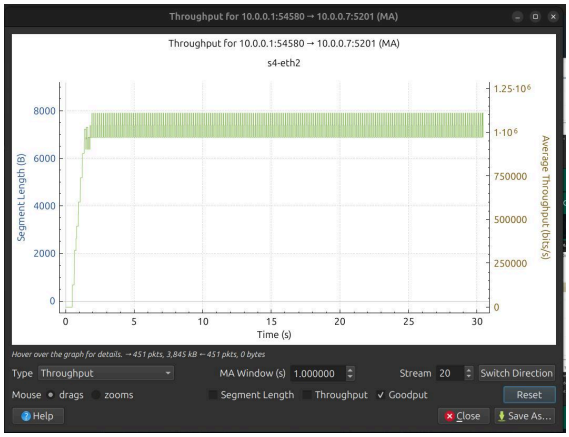
Protocol: RENO

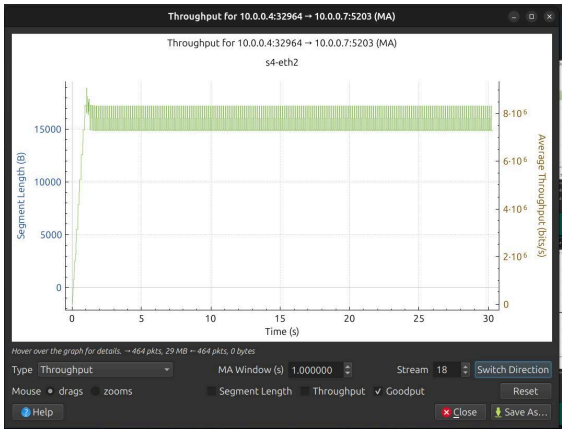
1. Throughput

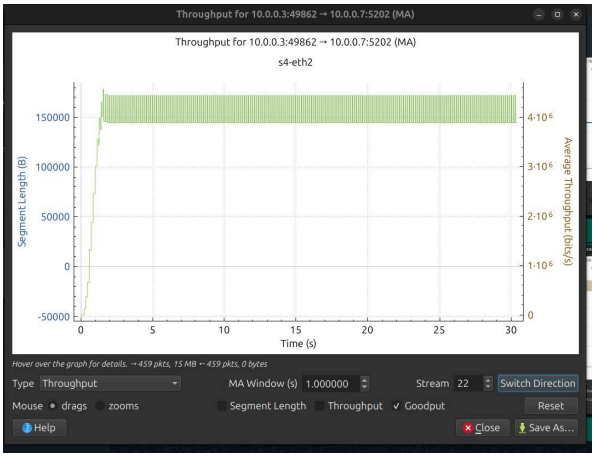




2. Goodput

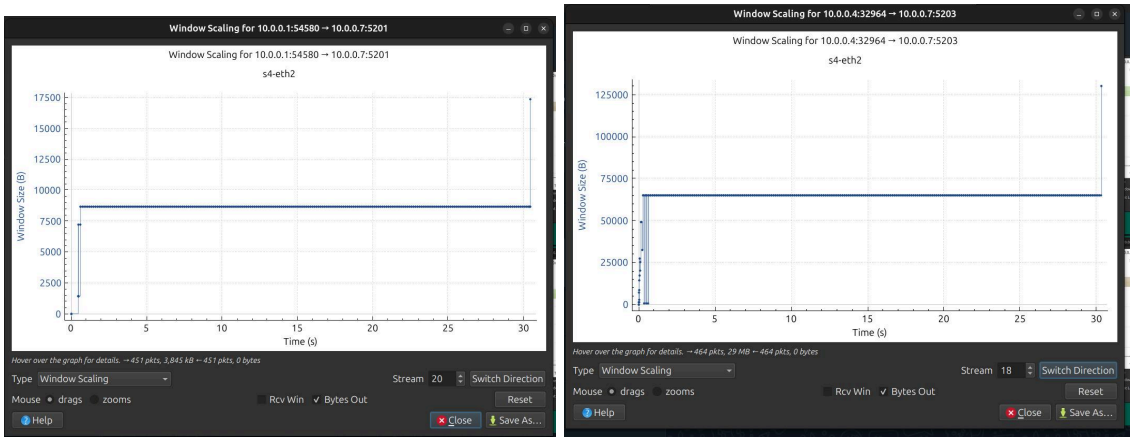


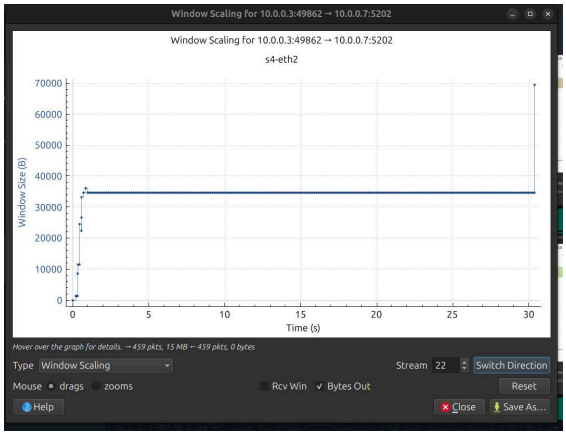




3. Packet Loss Rate

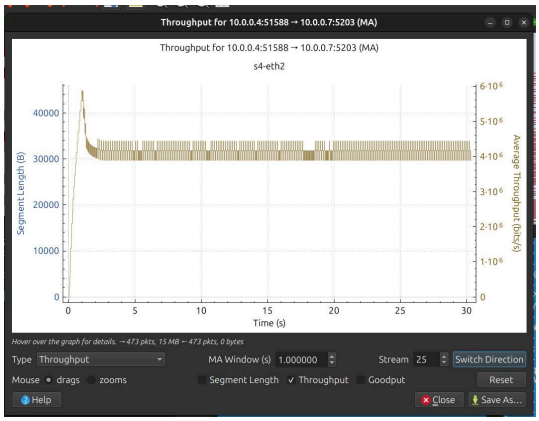
4. Maximum Window size achieved

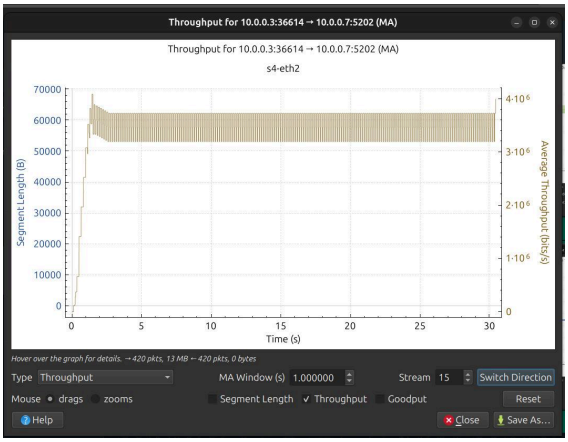


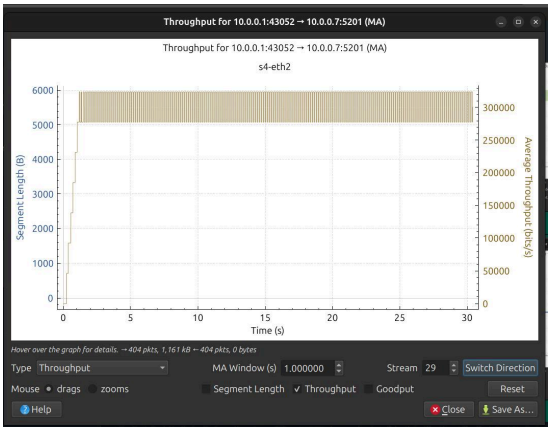


Protocol: BIC

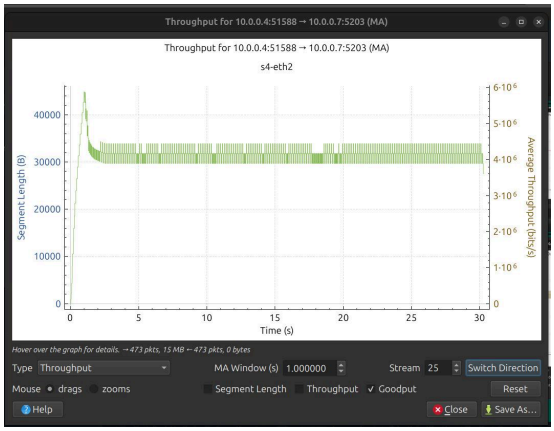
1. Throughput

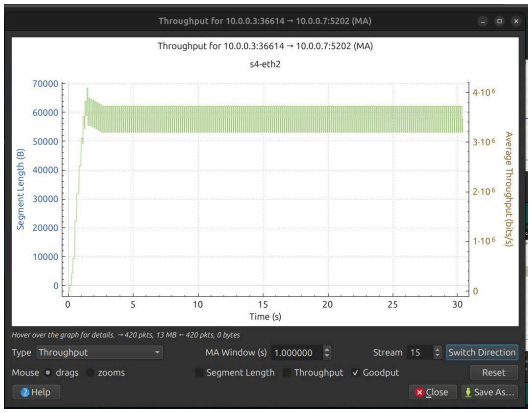

Throughput for 10.0.0.4:51588 → 10.0.0.7:5203 (MA)
Throughput for 10.0.0.4:51588 → 10.0.0.7:5203 (MA)
s4-eth2
Segment Length (B)
Average Throughput (bits/s)
Time (s)
Hover over the graph for details. →473 pkts, 15 MB ← 473 pkts, 0 bytes
Type Throughput
MA Window (s) 1.000000
Stream 25
Switch Direction
Mouse drags zooms
Segment Length Throughput Goodput
Reset
Help
Close
Save As...

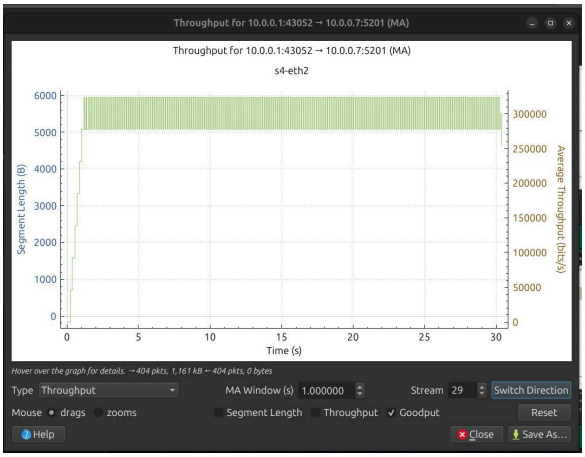

Throughput for 10.0.0.3:36614 → 10.0.0.7:5202 (MA)
Throughput for 10.0.0.3:36614 → 10.0.0.7:5202 (MA)
s4-eth2
Segment Length (B)
Average Throughput (bits/s)
Time (s)
Hover over the graph for details. →420 pkts, 13 MB ← 420 pkts, 0 bytes
Type Throughput
MA Window (s) 1.000000
Stream 15
Switch Direction
Mouse drags zooms
Segment Length Throughput Goodput
Reset
Help
Close
Save As...


Throughput for 10.0.0.1:43052 → 10.0.0.7:5201 (MA)
Throughput for 10.0.0.1:43052 → 10.0.0.7:5201 (MA)
s4-eth2
Segment Length (B)
Average Throughput (bits/s)
Time (s)
Hover over the graph for details. →404 pkts, 1,161 kB ← 404 pkts, 0 bytes
Type Throughput
MA Window (s) 1.000000
Stream 29
Switch Direction
Mouse drags zooms
Segment Length Throughput Goodput
Reset
Help
Close
Save As...

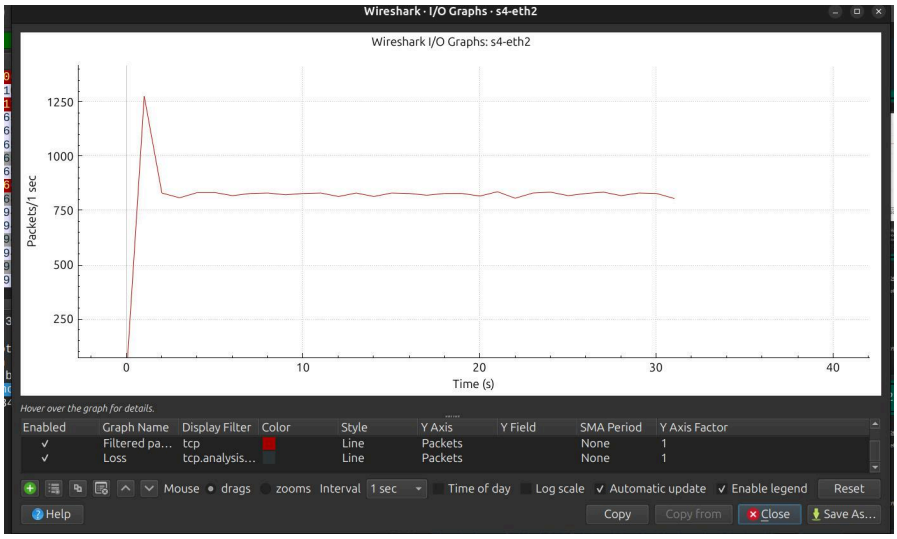
2. Goodput


Throughput for 10.0.0.4:51588 → 10.0.0.7:5203 (MA)
Throughput for 10.0.0.4:51588 → 10.0.0.7:5203 (MA)
s4-eth2
Segment Length (B)
Average Throughput (bits/s)
Time (s)
Hover over the graph for details. →473 pkts, 15 MB ← 473 pkts, 0 bytes
Type Throughput
MA Window (s) 1.000000
Stream 25
Switch Direction
Mouse drags zooms
Segment Length Throughput Goodput
Reset
Help
Close
Save As...

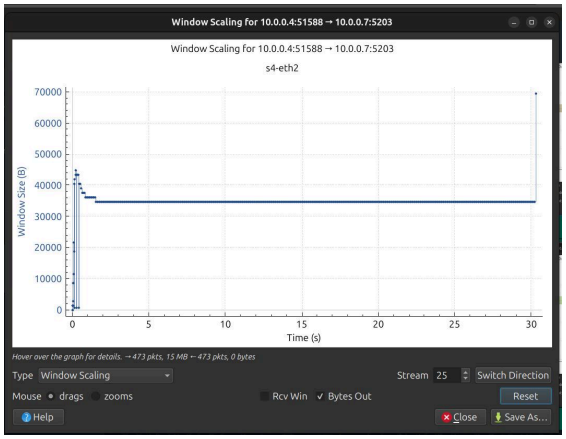

Throughput for 10.0.0.3:36614 → 10.0.0.7:5202 (MA)
Throughput for 10.0.0.3:36614 → 10.0.0.7:5202 (MA)
s4-eth2
Segment Length (B)
Average Throughput (bits/s)
Time (s)
Hover over the graph for details. →420 pkts, 13 MB ← 420 pkts, 0 bytes
Type Throughput
MA Window (s) 1.000000
Stream 15
Switch Direction
Mouse drags zooms
Segment Length Throughput Goodput
Reset
Help
Close
Save As...

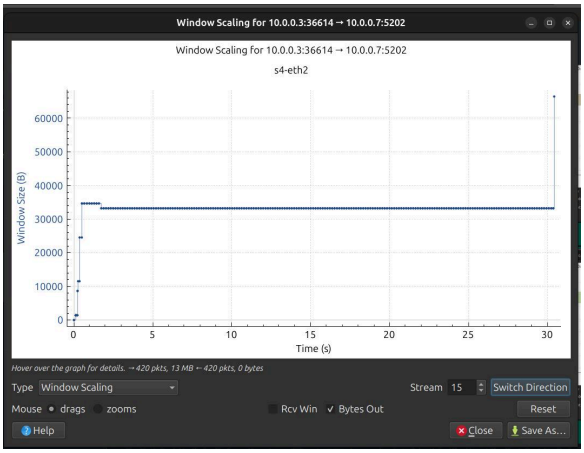

Throughput for 10.0.0.1:43052 → 10.0.0.7:5201 (MA)
Throughput for 10.0.0.1:43052 → 10.0.0.7:5201 (MA)
s4-eth2
Segment Length (B)
Average Throughput (bits/s)
Time (s)
Hover over the graph for details. →404 pkts, 1,161 kB ← 404 pkts, 0 bytes
Type Throughput
MA Window (s) 1.000000
Stream 29
Switch Direction
Mouse drags zooms
Segment Length Throughput Goodput
Reset
Help
Close
Save As...

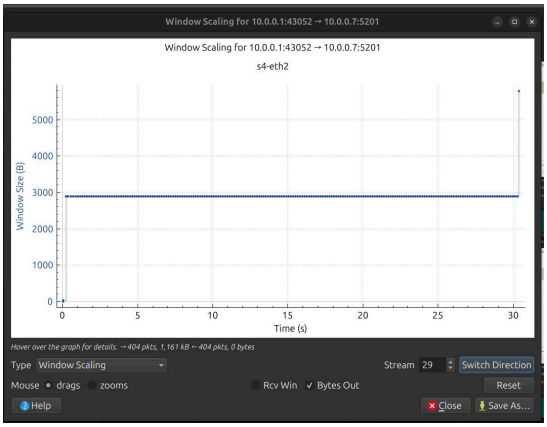
3. Packet Loss Rate



4. Maximum Window size achieved

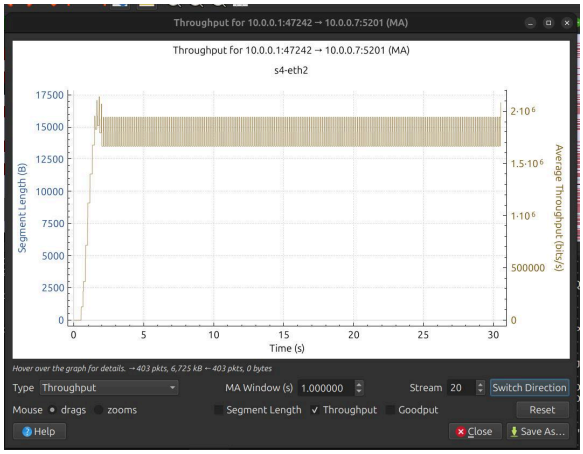


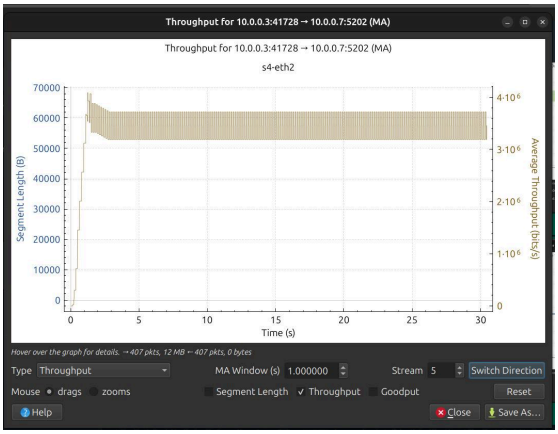


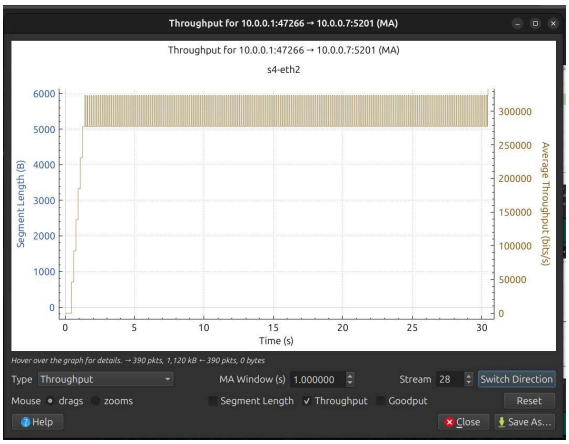


Protocol: HIGHSPEED

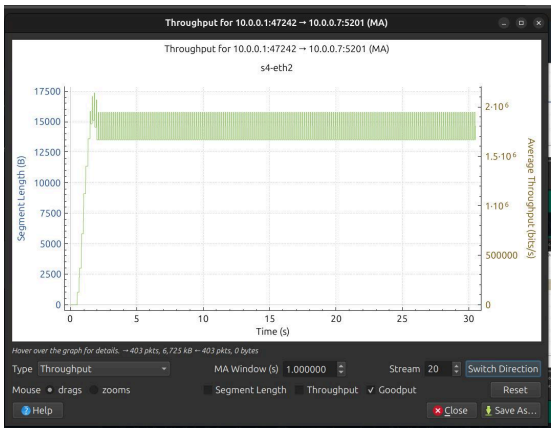
1. Throughput

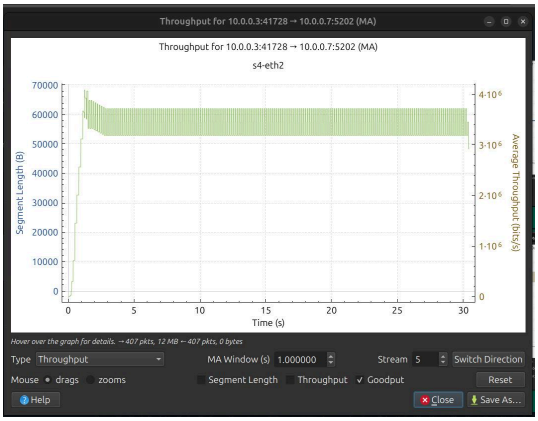

Throughput for 10.0.0.1:47242 → 10.0.0.7:5201 (MA)
Throughput for 10.0.0.1:47242 → 10.0.0.7:5201 (MA)
s4-eth2
Segment Length (B)
Average Throughput (bits/s)
Time (s)
Hover over the graph for details. → 403 pkts, 6,725 kB ← 403 pkts, 0 bytes
Type Throughput
MA Window (s) 1.000000
Stream 20
Switch Direction
Mouse drags zooms
Segment Length Throughput Goodput
Reset
Help
Close
Save As…

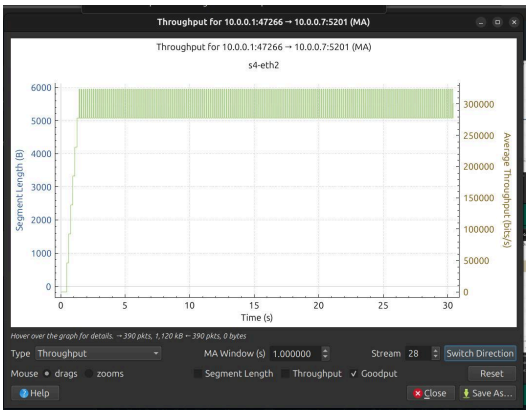

Throughput for 10.0.0.3:41728 → 10.0.0.7:5202 (MA)
Throughput for 10.0.0.3:41728 → 10.0.0.7:5202 (MA)
s4-eth2
Segment Length (B)
Average Throughput (bits/s)
Time (s)
Hover over the graph for details. → 407 pkts, 12 MB ← 407 pkts, 0 bytes
Type Throughput
MA Window (s) 1.000000
Stream 5
Switch Direction
Mouse drags zooms
Segment Length Throughput Goodput
Reset
Help
Close
Save As…


Throughput for 10.0.0.1:47266 → 10.0.0.7:5201 (MA)
Throughput for 10.0.0.1:47266 → 10.0.0.7:5201 (MA)
s4-eth2
Segment Length (B)
Average Throughput (bits/s)
Time (s)
Hover over the graph for details. → 390 pkts, 1,120 kB ← 390 pkts, 0 bytes
Type Throughput
MA Window (s) 1.000000
Stream 28
Switch Direction
Mouse drags zooms
Segment Length Throughput Goodput
Reset
Help
Close
Save As…

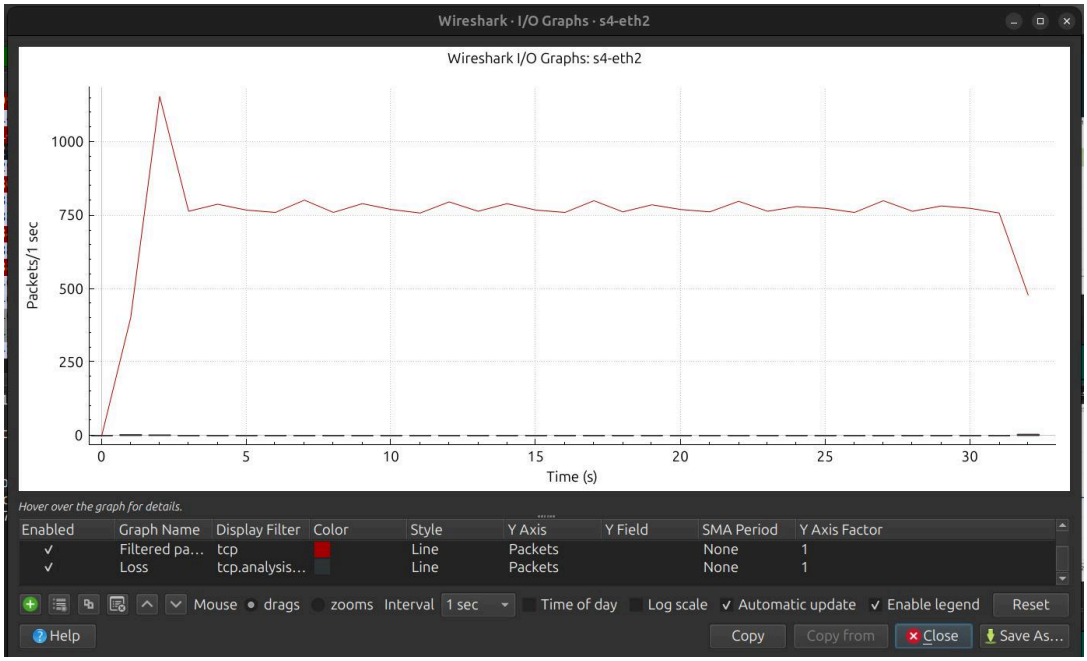
2. Goodput


Throughput for 10.0.0.1:47242 → 10.0.0.7:5201 (MA)
Throughput for 10.0.0.1:47242 → 10.0.0.7:5201 (MA)
s4-eth2
Segment Length (B)
Average Throughput (bits/s)
Time (s)
Hover over the graph for details. → 403 pkts, 6,725 kB ← 403 pkts, 0 bytes
Type Throughput
MA Window (s) 1.000000
Stream 20
Switch Direction
Mouse drags zooms
Segment Length Throughput Goodput
Reset
Help
Close
Save As…

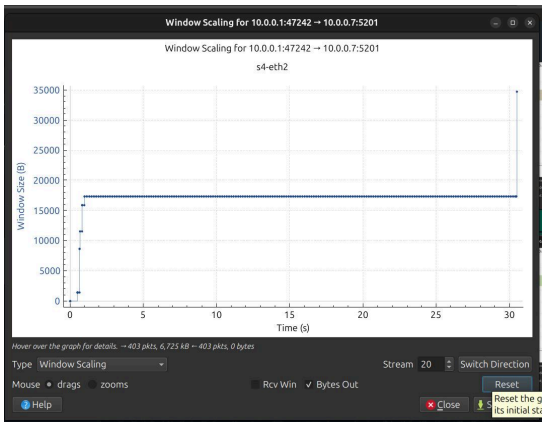

Throughput for 10.0.0.3:41728 → 10.0.0.7:5202 (MA)
Throughput for 10.0.0.3:41728 → 10.0.0.7:5202 (MA)
s4-eth2
Segment Length (B)
Average Throughput (bits/s)
Time (s)
Hover over the graph for details. → 407 pkts, 12 MB ← 407 pkts, 0 bytes
Type Throughput
MA Window (s) 1.000000
Stream 5
Switch Direction
Mouse drags zooms
Segment Length Throughput Goodput
Reset
Help
Close
Save As…

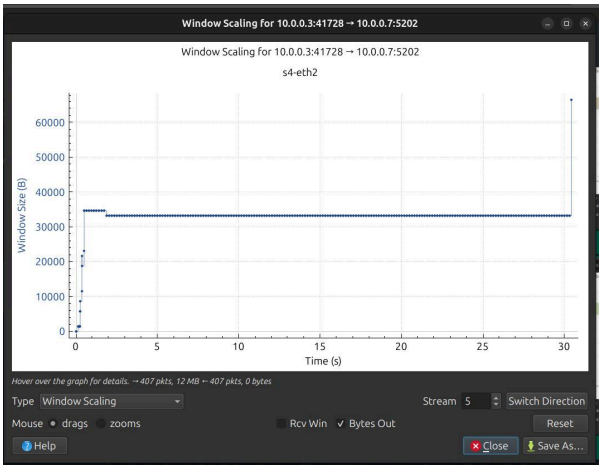

Throughput for 10.0.0.1:47266 → 10.0.0.7:5201 (MA)
Throughput for 10.0.0.1:47266 → 10.0.0.7:5201 (MA)
s4-eth2
Segment Length (B)
Average Throughput (bits/s)
Time (s)
Hover over the graph for details. → 390 pkts, 1,120 kB ← 390 pkts, 0 bytes
Type Throughput
MA Window (s) 1.000000
Stream 28
Switch Direction
Mouse drags zooms
Segment Length Throughput Goodput
Reset
Help
Close
Save As…

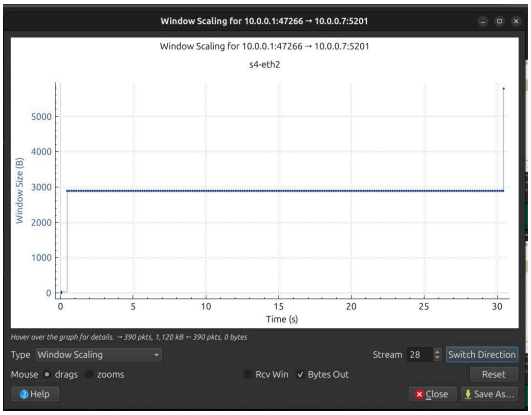
3. Packet Loss Rate



4. Maximum Window size achieved







Observation

In this scenario, three clients (H1, H3, and H4) were introduced at staggered intervals, leading to congestion that gradually increased over time. The results demonstrated progressive bandwidth contention, where the first client initially achieved high throughput. However, as new clients joined, bandwidth was redistributed among all active connections. The throughput graphs showed a gradual decline in individual client throughput as more clients joined, but the total network throughput remained near capacity. Goodput graphs displayed slight gaps, indicating that each client experienced delays due to the server handling multiple connections in a round-robin manner.

Part d

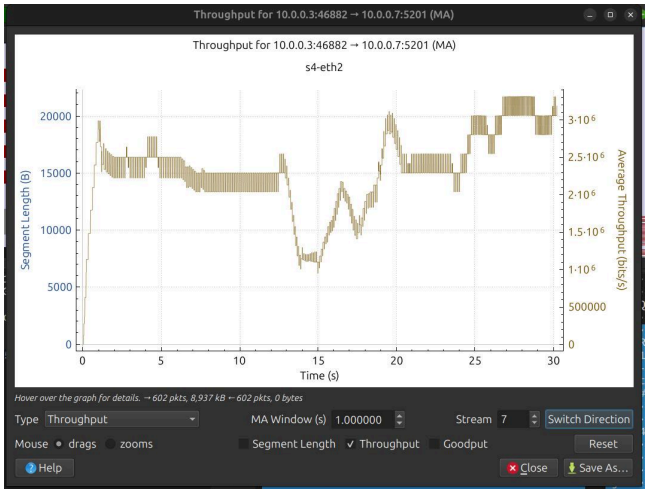
- The fourth part involved the link loss parameter of the link S2-S3 to 1% and 5% and repeating the part c for each.
- Ran code for each congestion control algorithm (Reno, BIC, and HighSpeed) for all four scenarios, capturing the network traffic for each using wireshark on s4-eth2 interface and getting plots required.

Loss- 1%

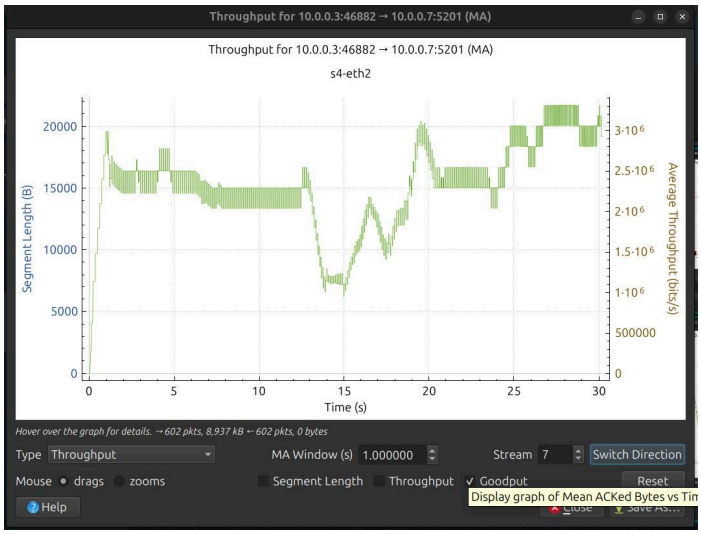
Scenario 1

Protocol: RENO

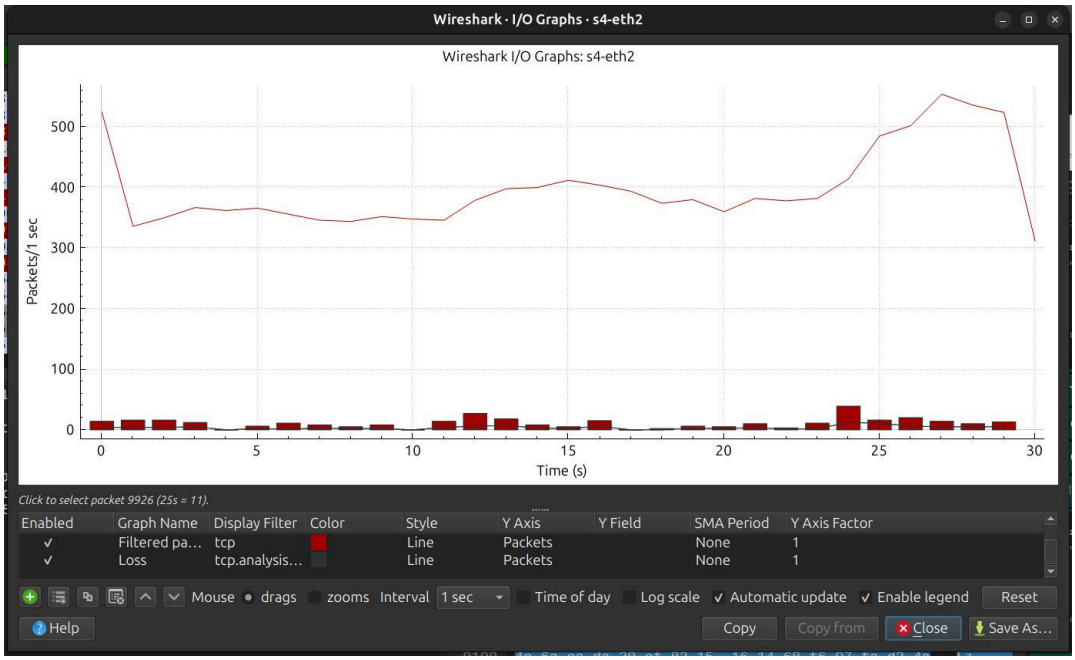
1. Throughput



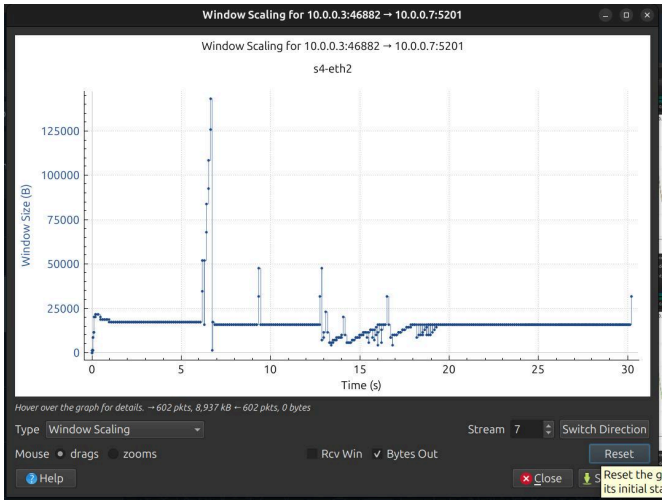
2. Goodput



3. Packet Loss Rate

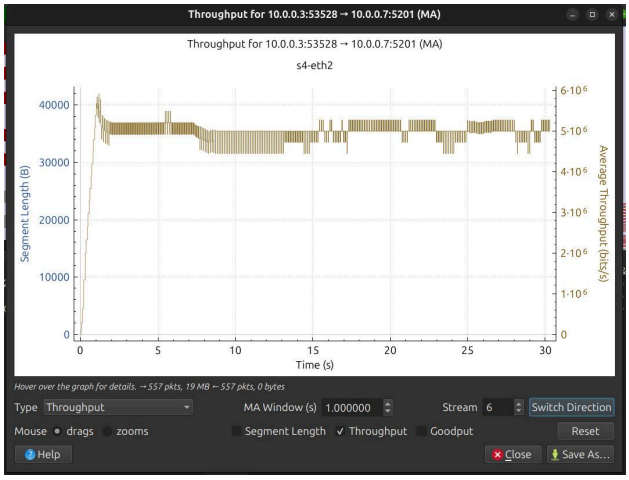


4. Maximum Window size achieved

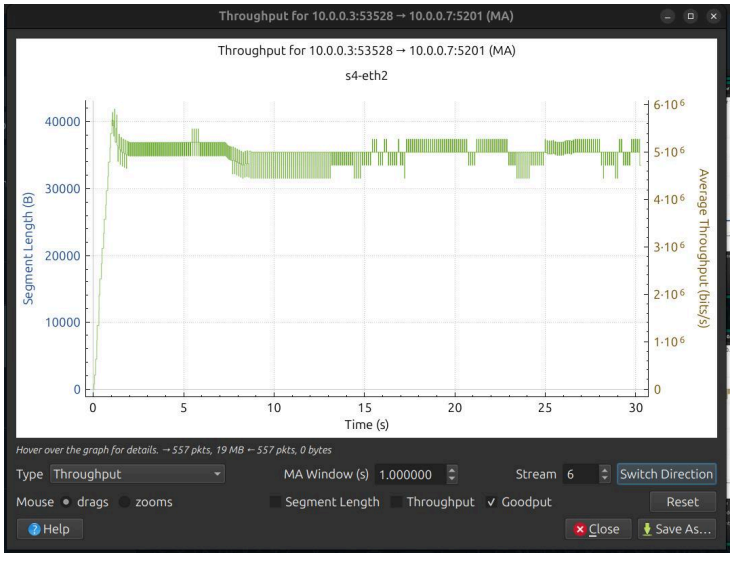


Protocol: BIC

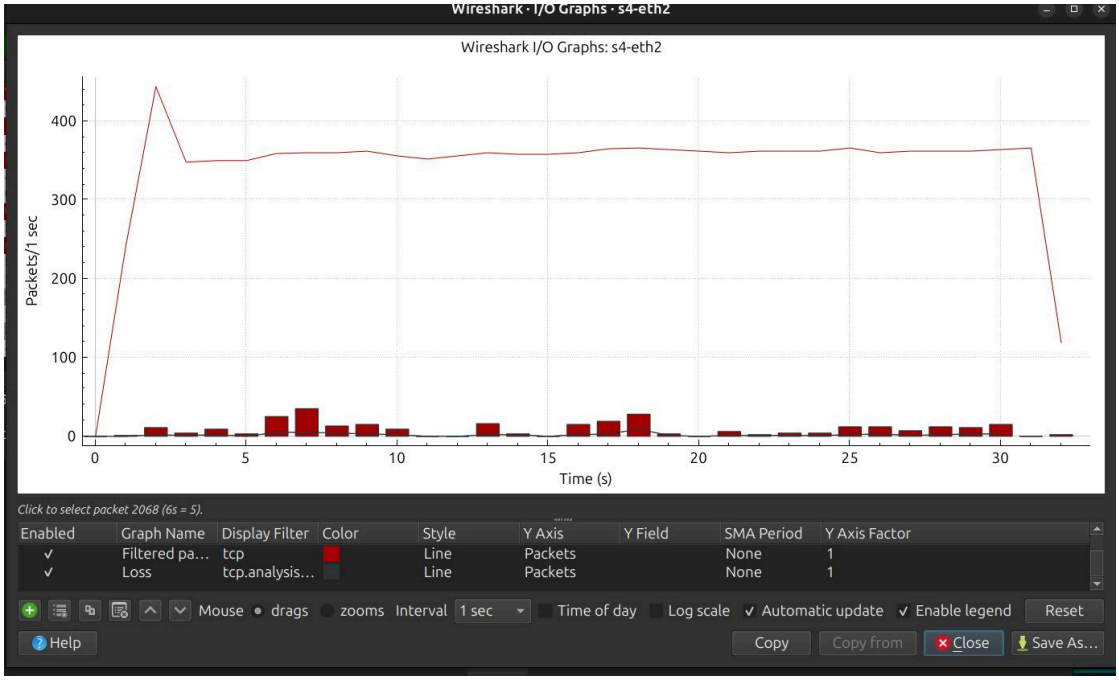
1. Throughput



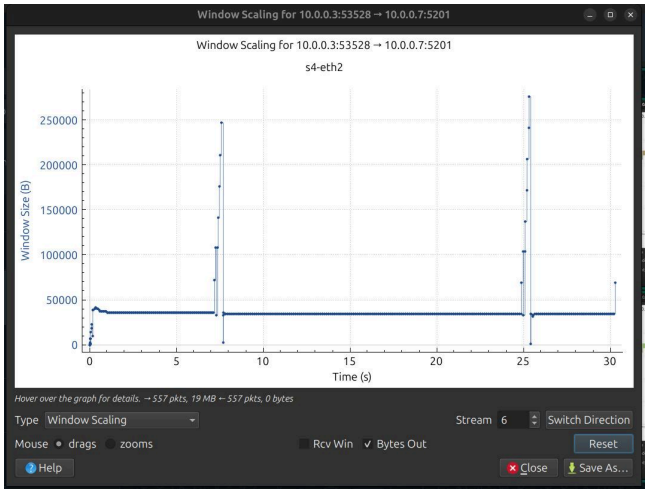
2. Goodput



3. Packet Loss Rate

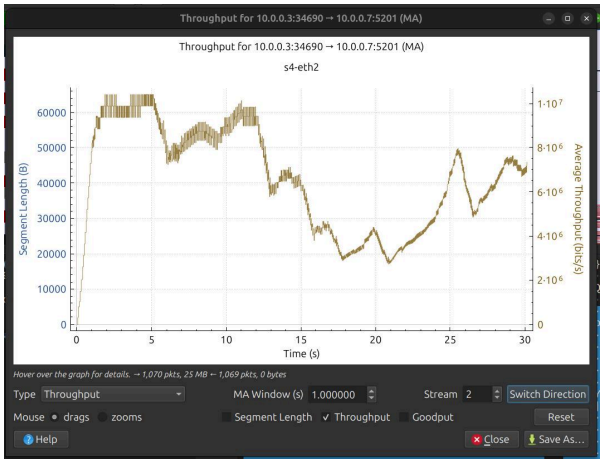


4. Maximum Window size achieved

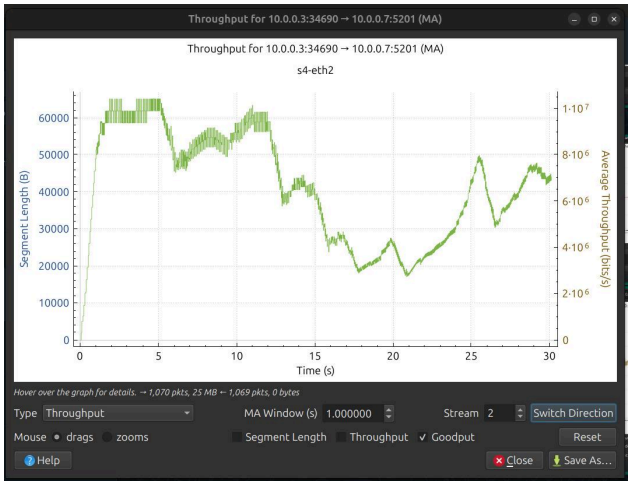


Protocol: HIGHSPEED

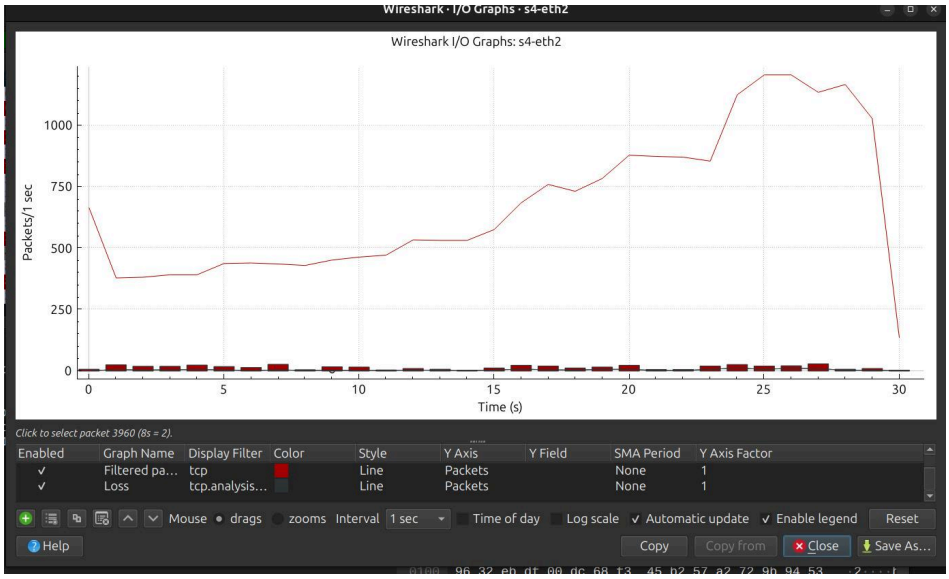
1. Throughput



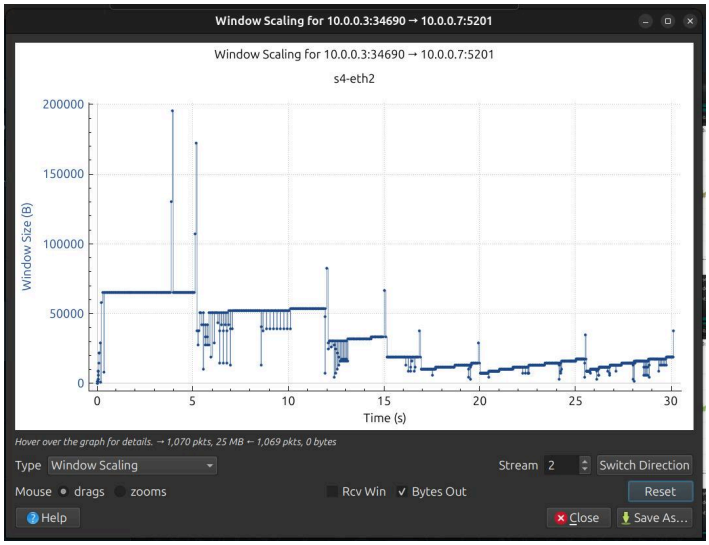
2. Goodput



3. Packet Loss Rate



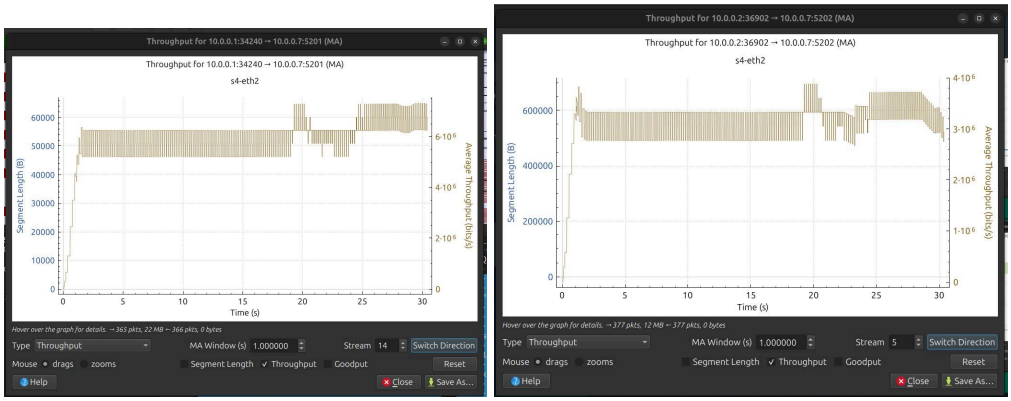
4. Maximum Window size achieved



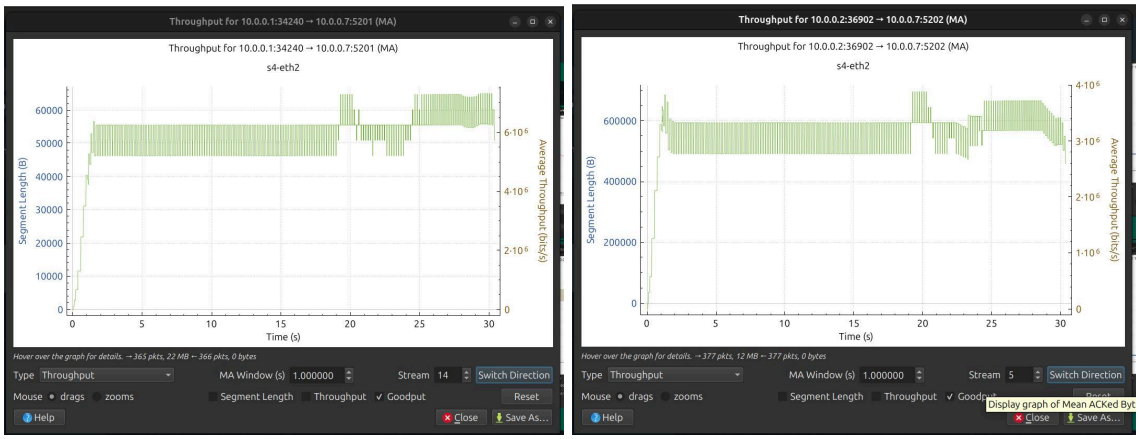
Scenario 2a

Protocol: RENO

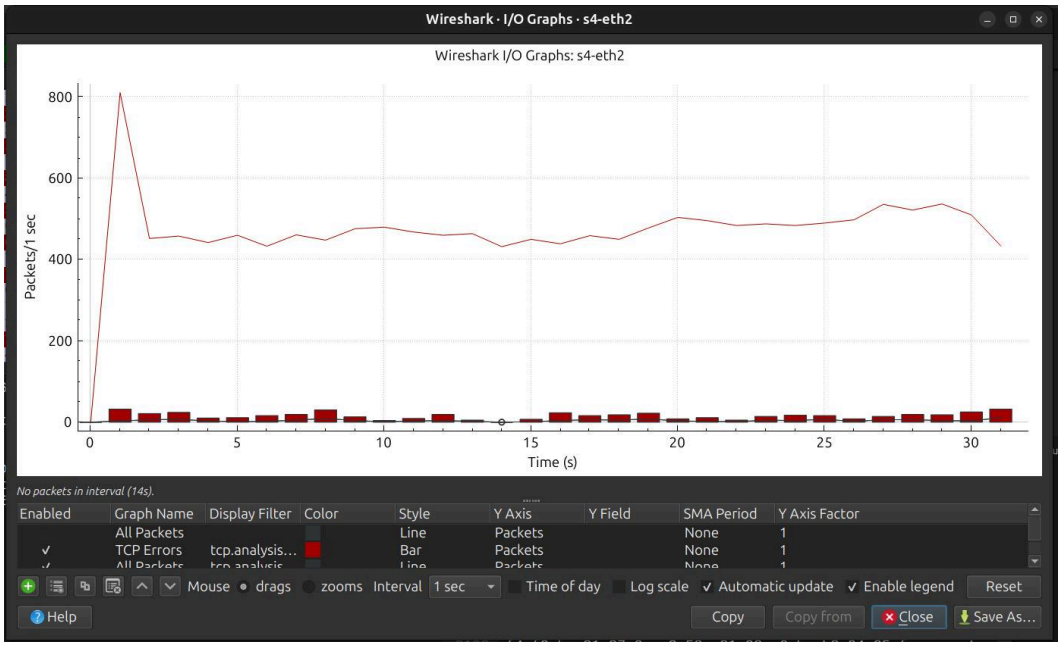
1. Throughput



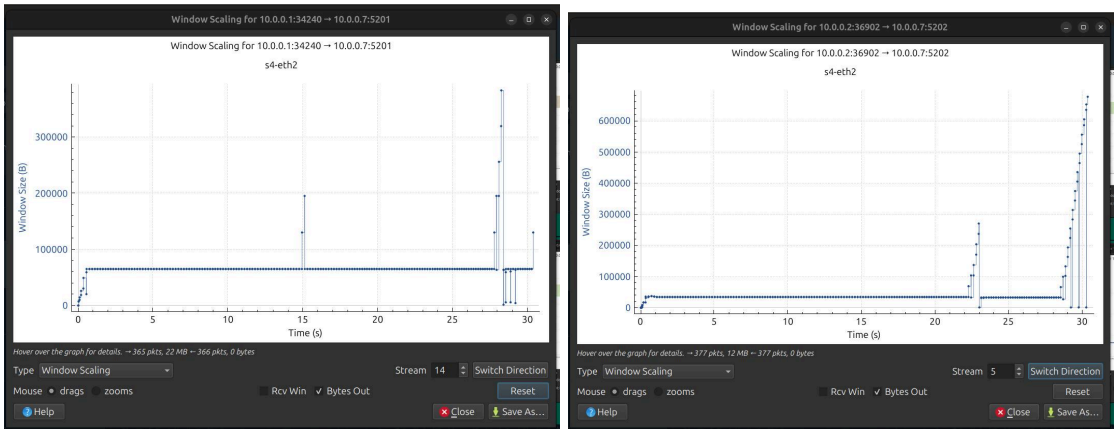
2. Goodput



3. Packet Loss Rate

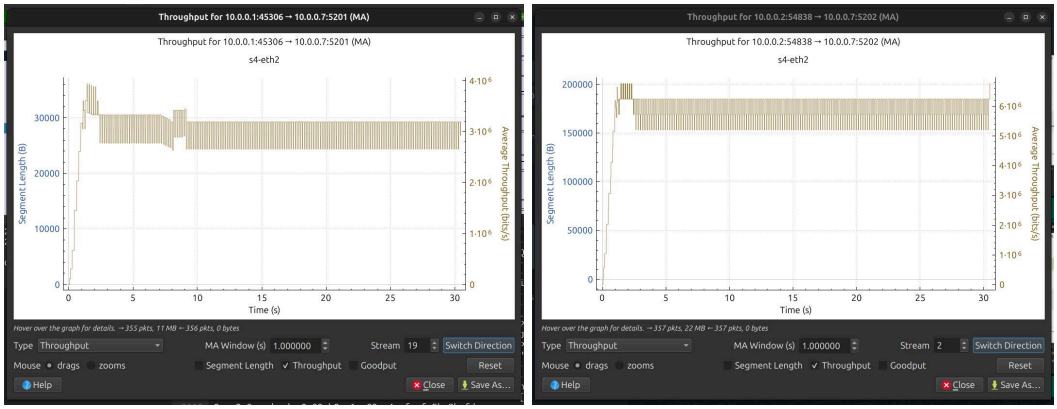


4. Maximum Window size achieved

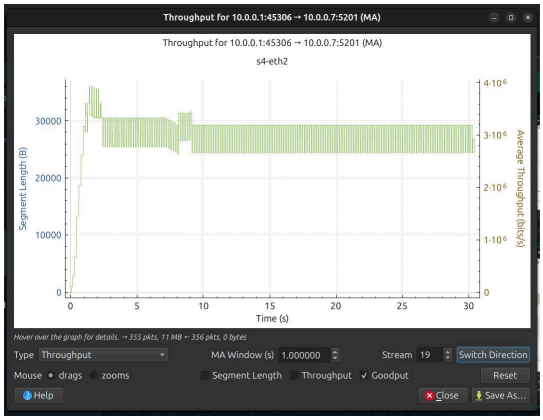


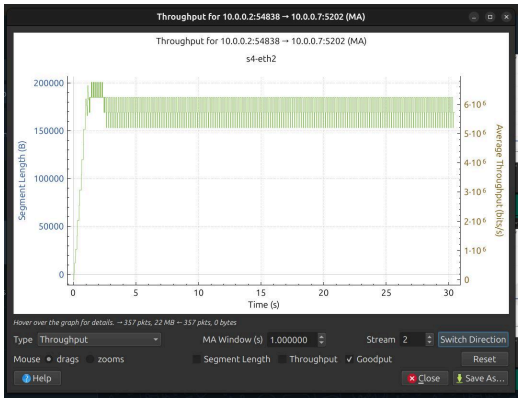
Protocol: BIC

1. Throughput

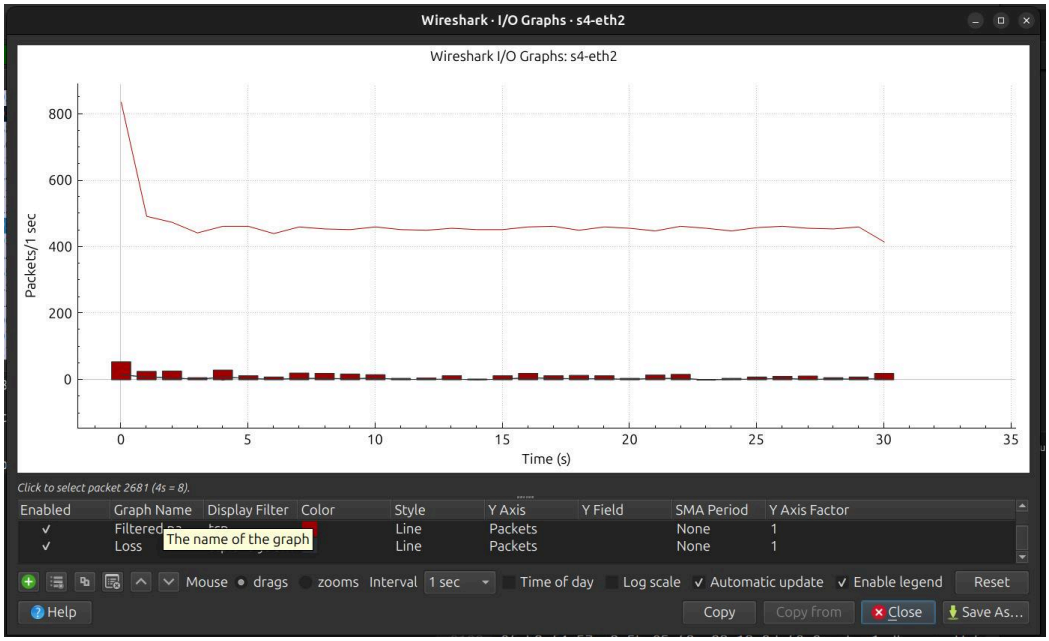


2. Goodput

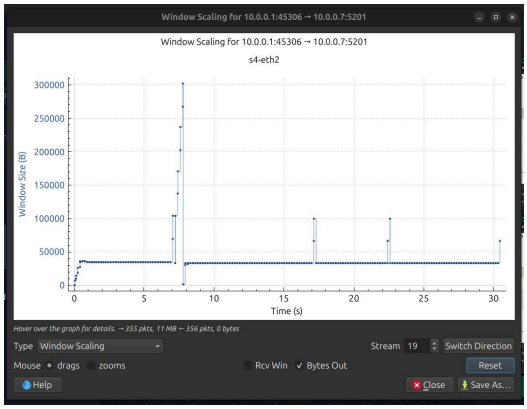


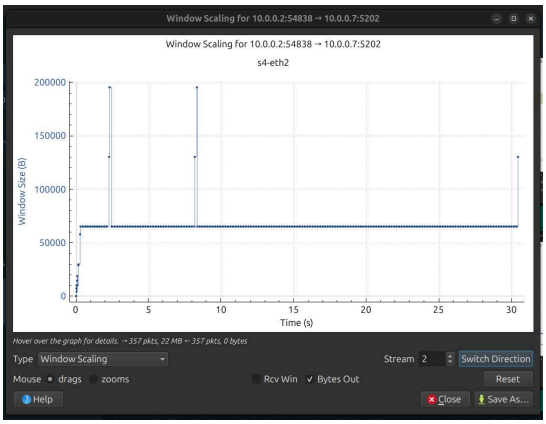


3. Packet Loss Rate



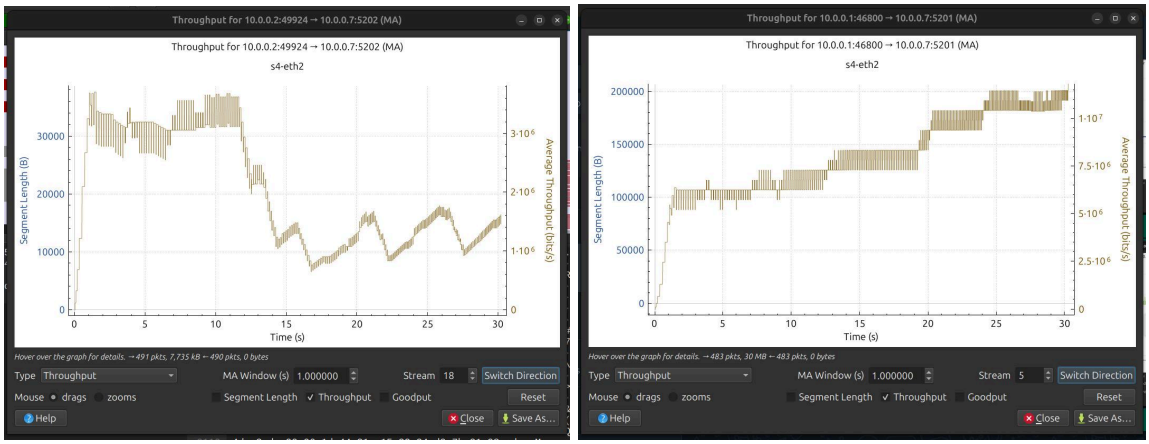
4. Maximum Window size achieved



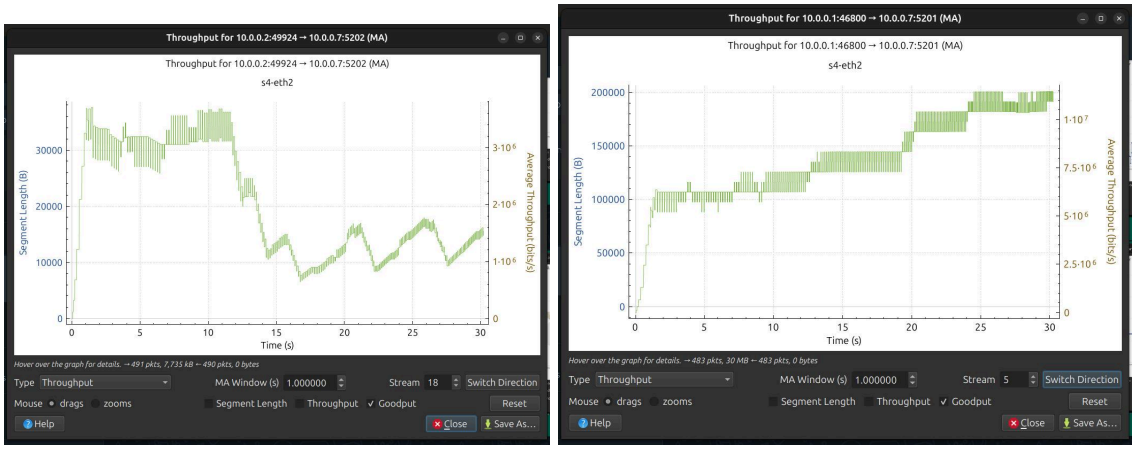


Protocol: HIGHSPEED

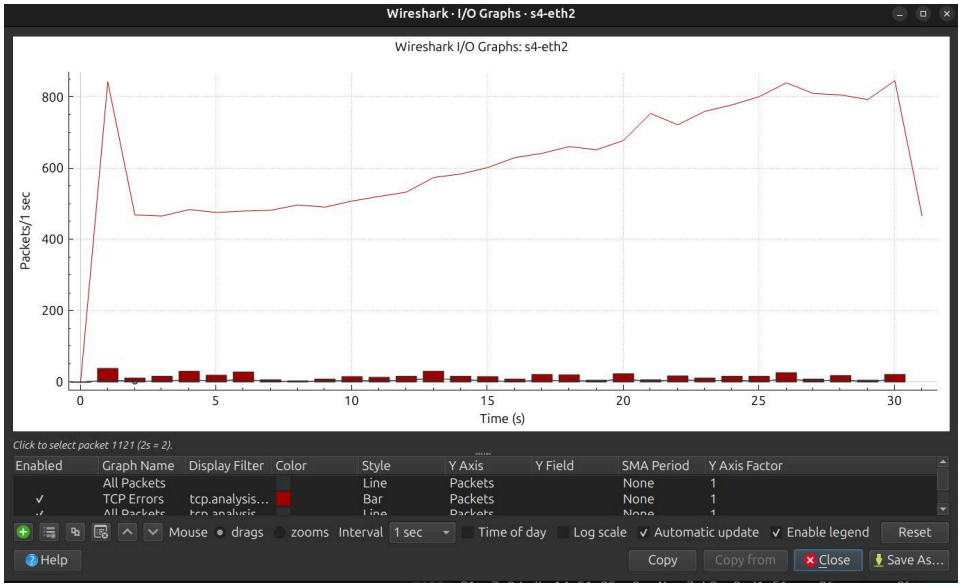
1. Throughput



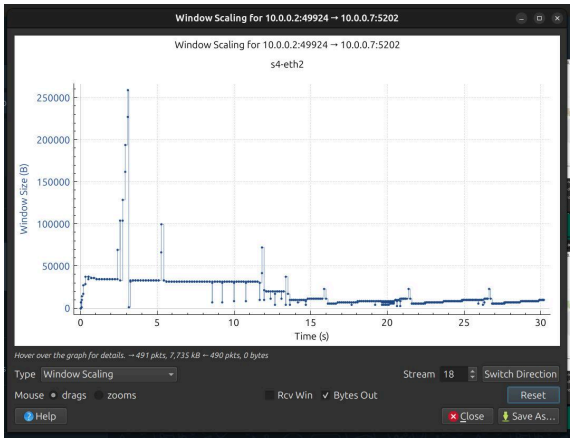
2. Goodput

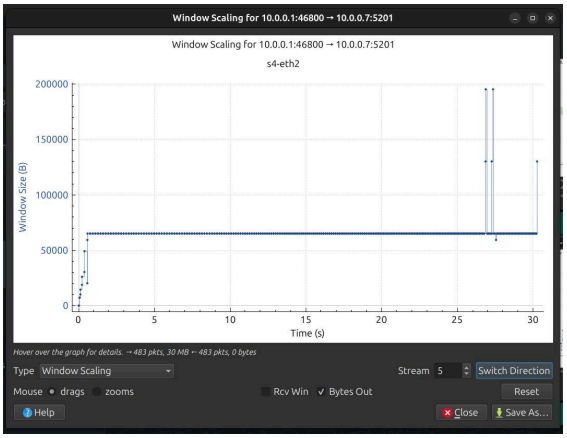


3. Packet Loss Rate



4. Maximum Window size achieved

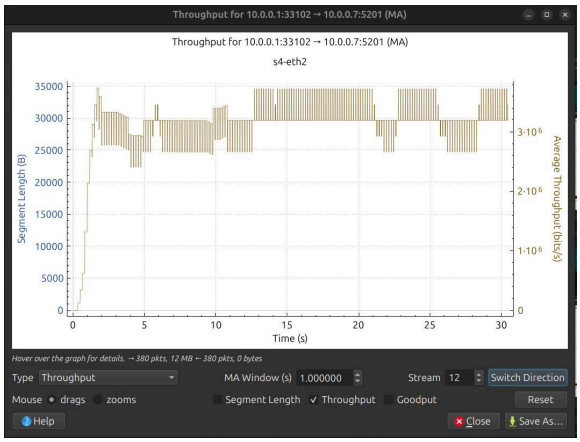


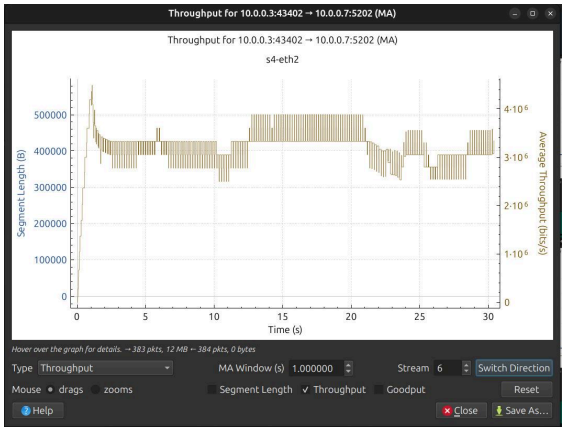


Scenario 2b

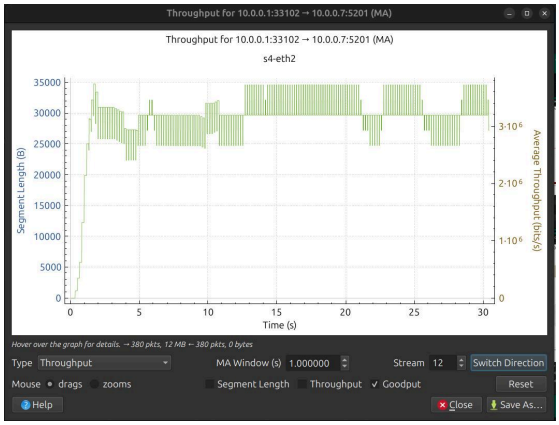
Protocol: RENO

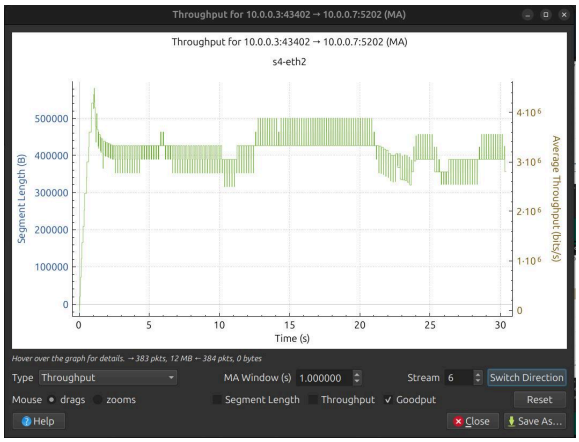
1. Throughput



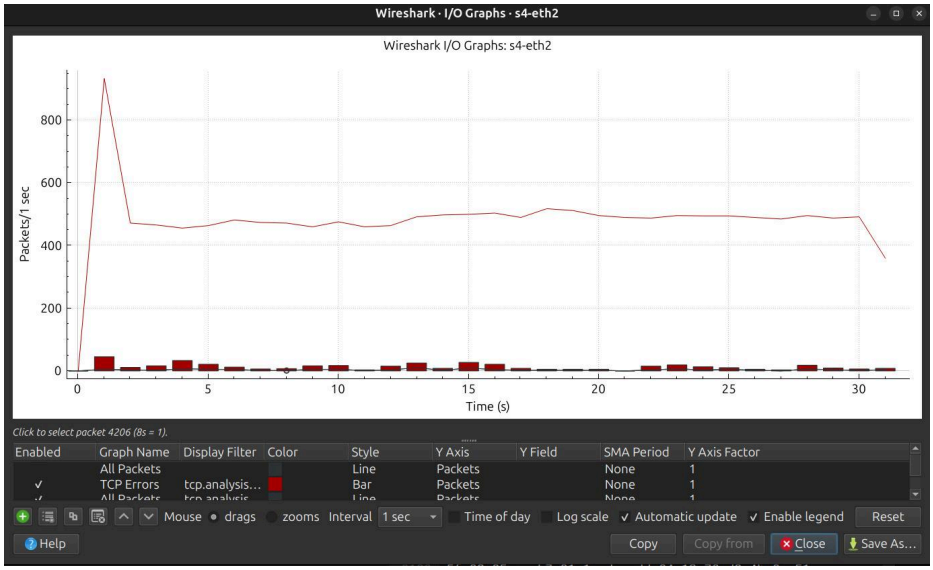


2. Goodput

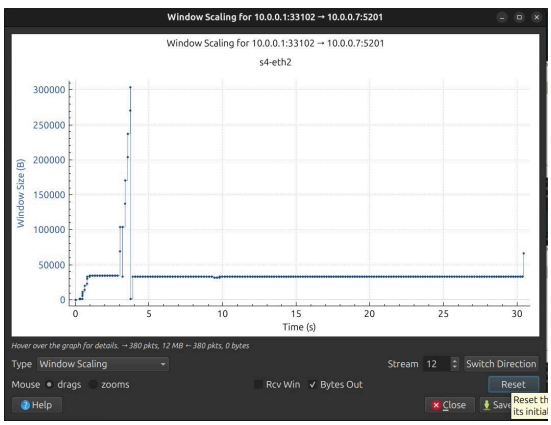


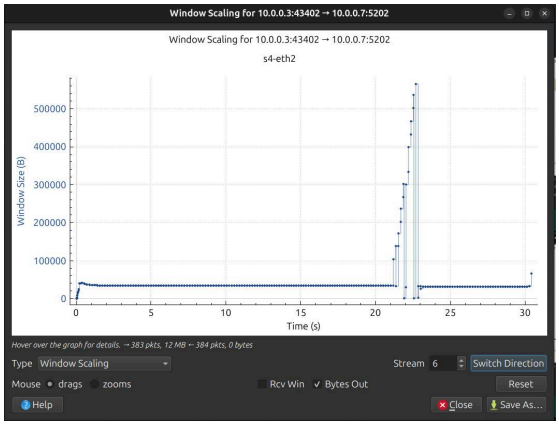


3. Packet Loss Rate



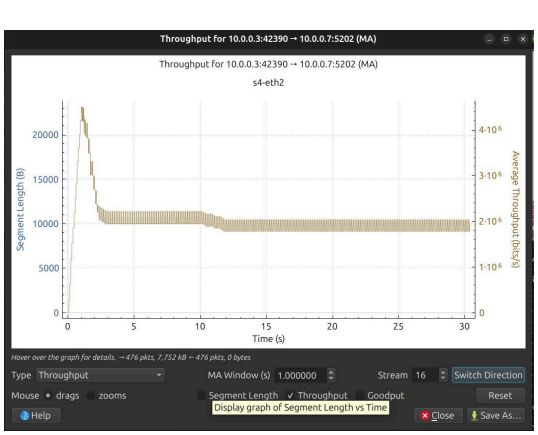
4. Maximum Window size achieved

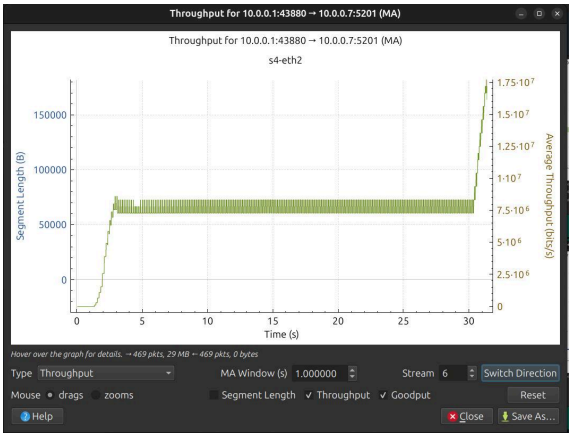




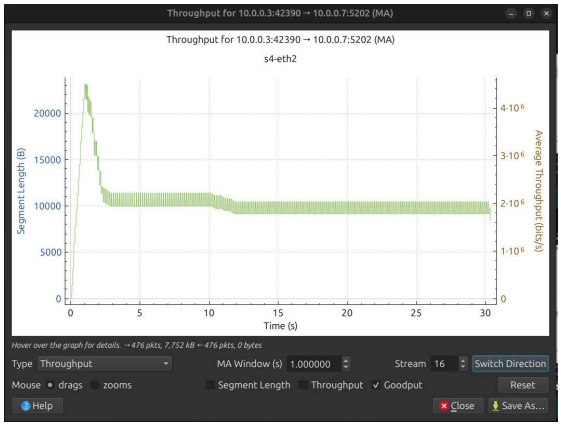
Protocol: BIC

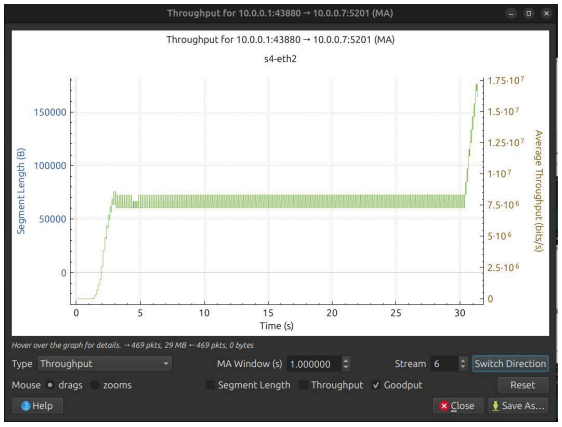
1. Throughput



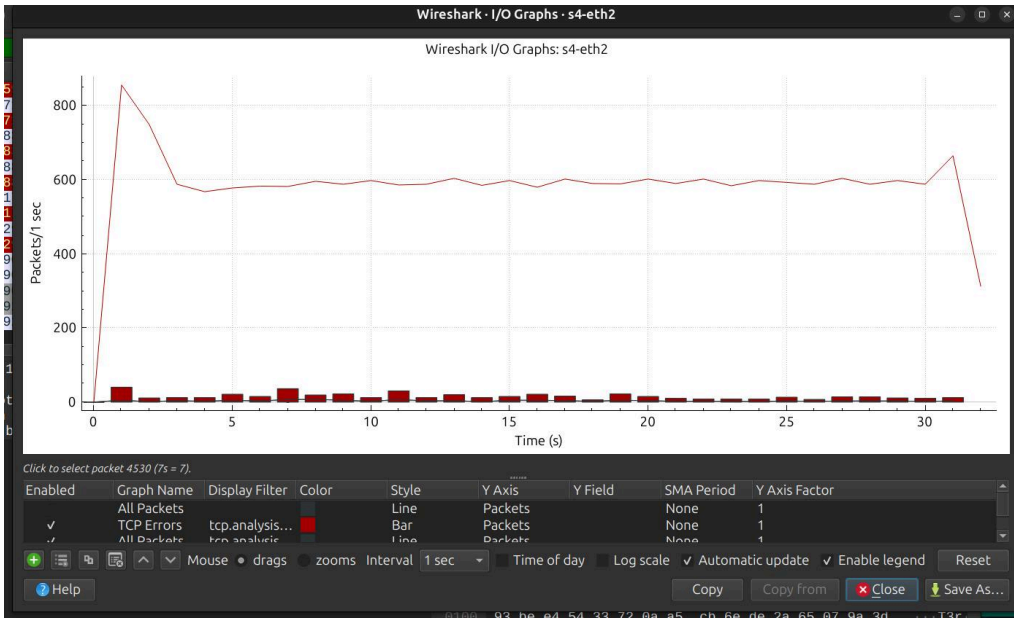


2. Goodput

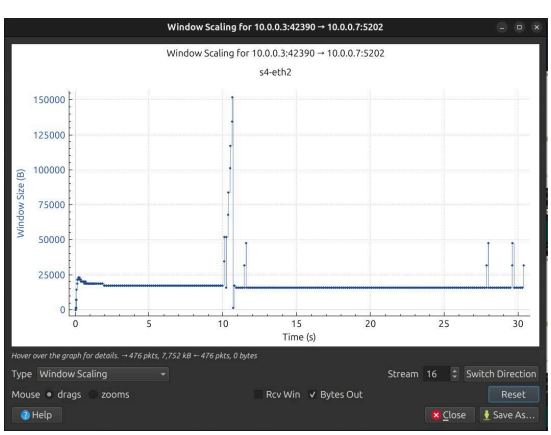


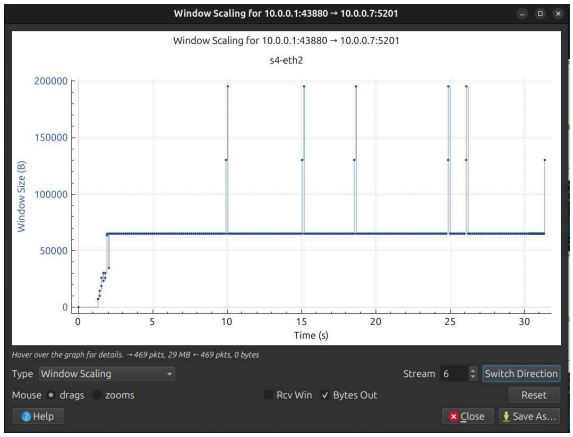


3. Packet Loss Rate



4. Maximum Window size achieved



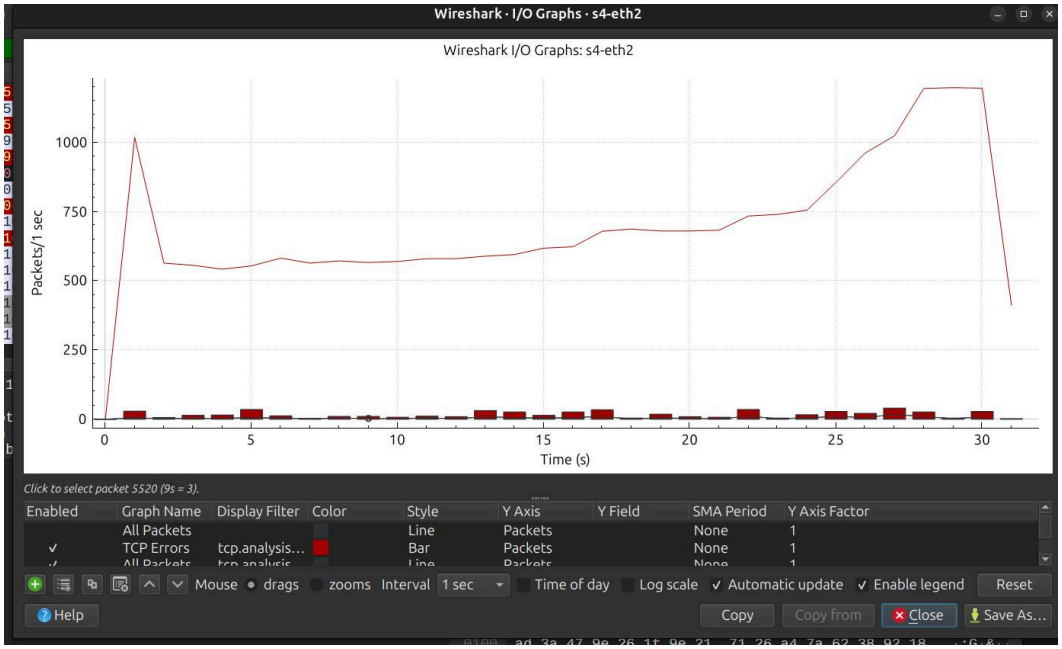


Protocol: HIGHSPEED

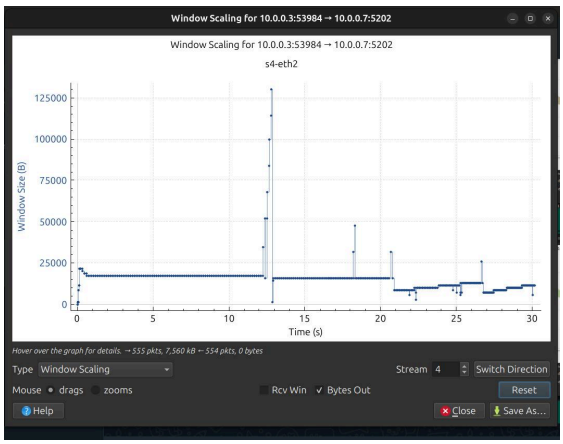
1. Throughput

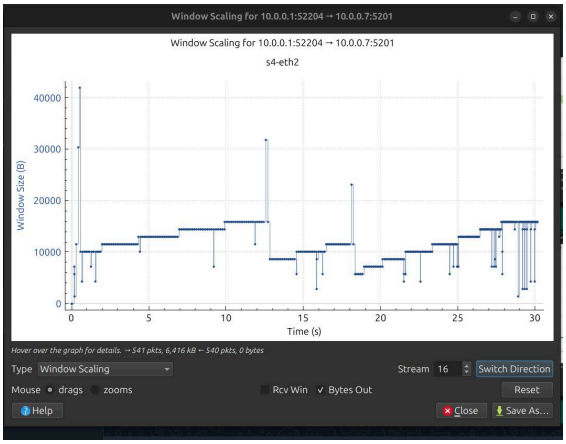
2. Goodput

3. Packet Loss Rate



4. Maximum Window size achieved

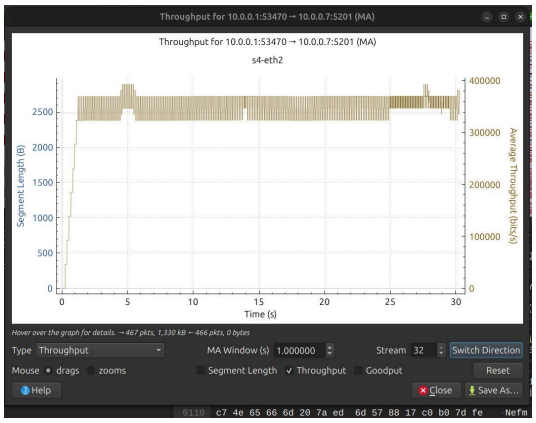


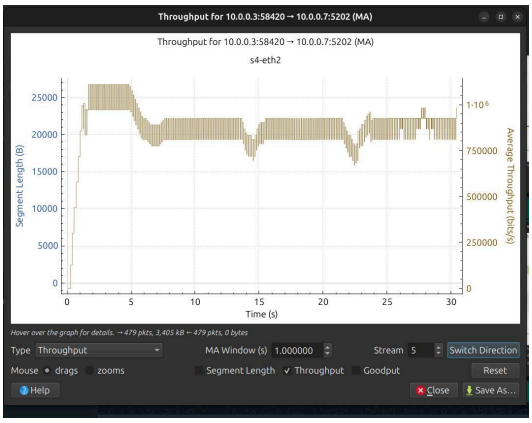


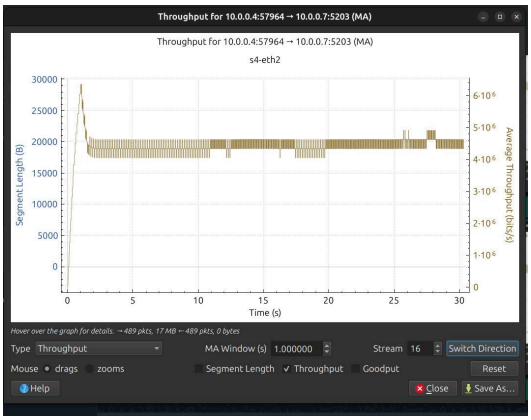
Scenario 2c

Protocol: RENO

1. Throughput

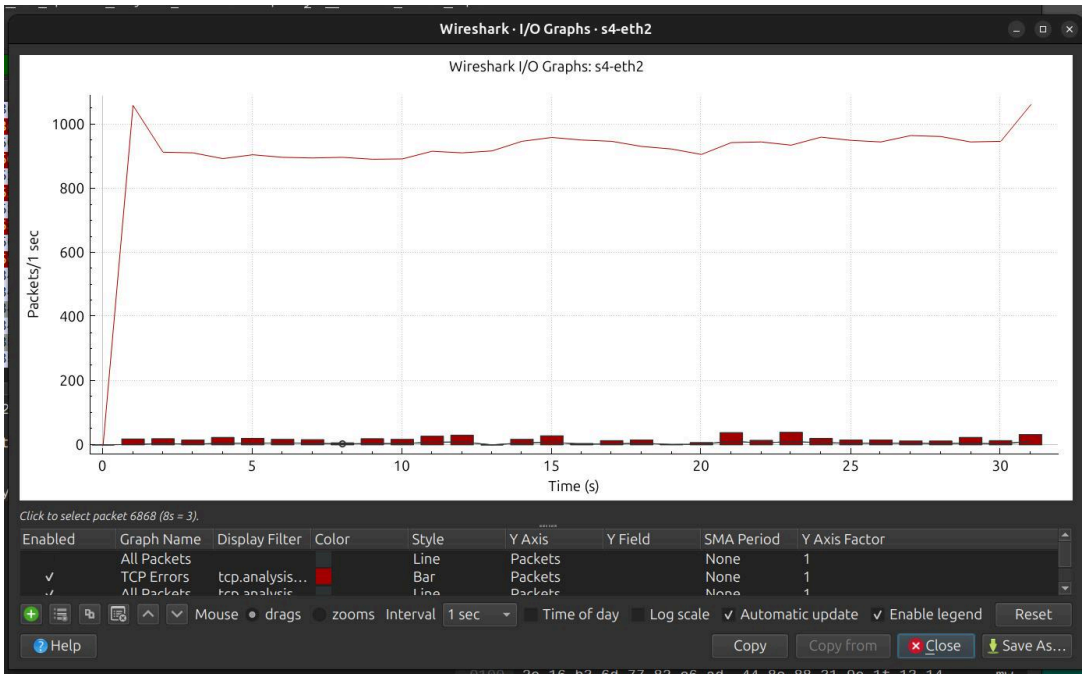




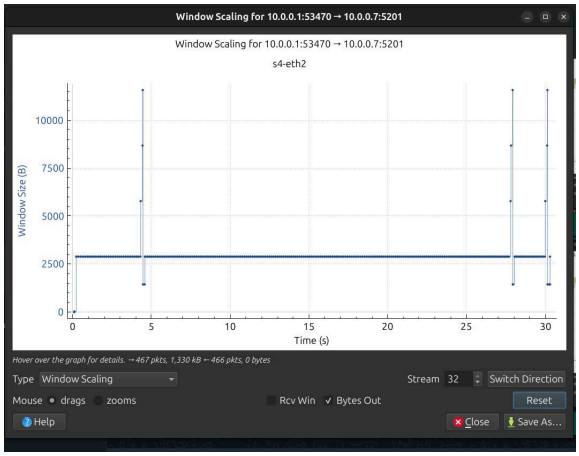


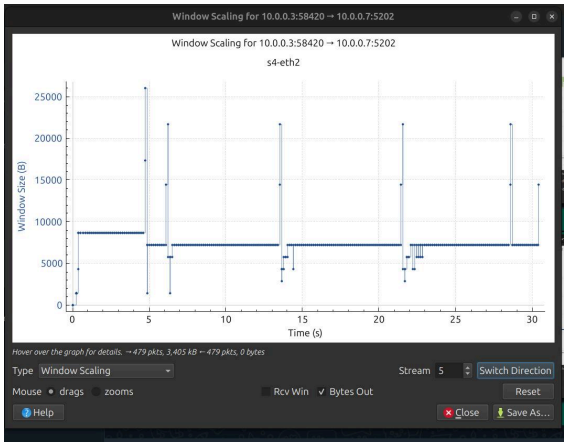
2. Goodput

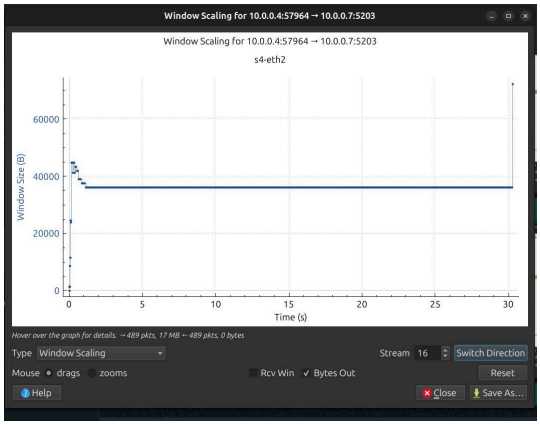
3. Packet Loss Rate



4. Maximum Window size achieved

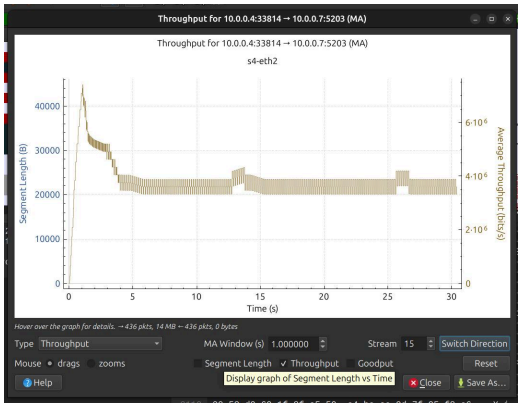


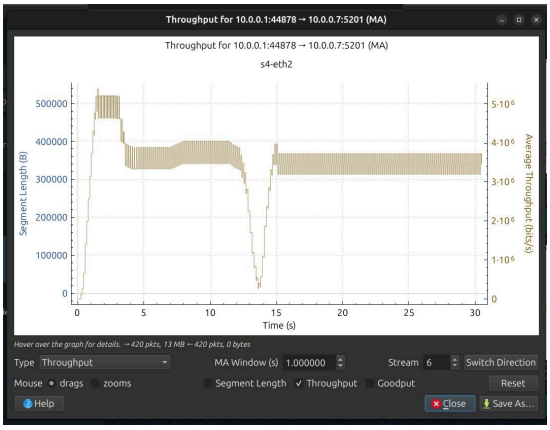


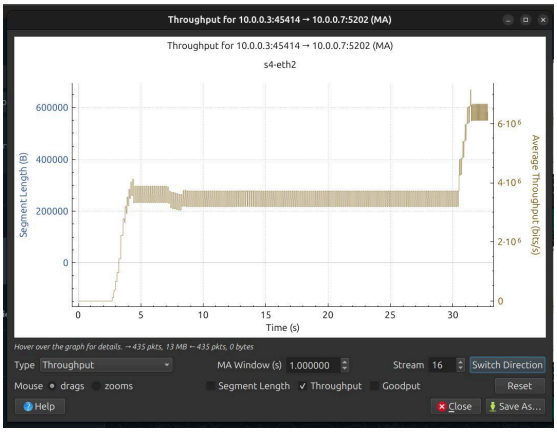


Protocol: BIC

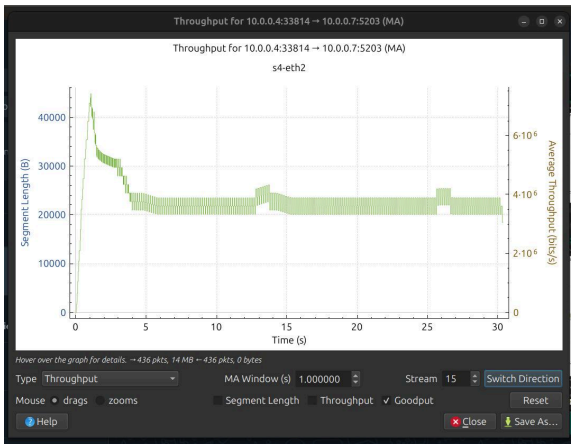
1. Throughput

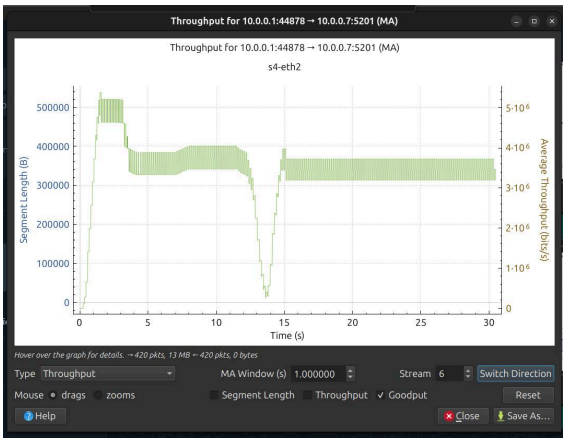


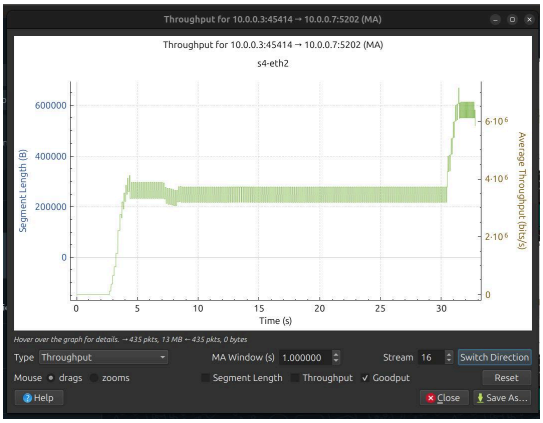




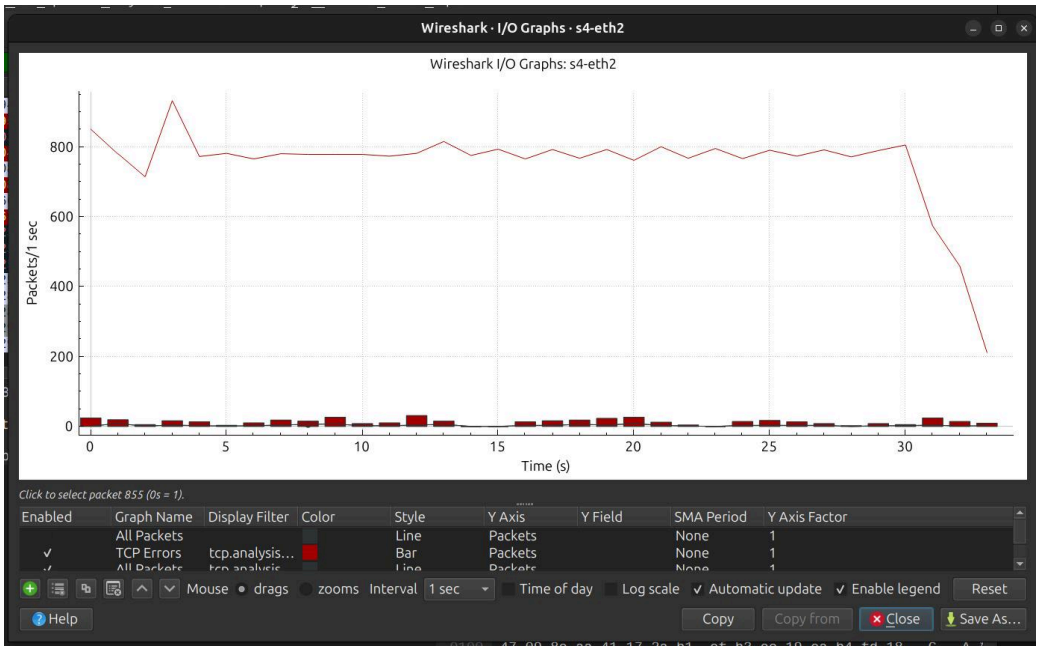
2. Goodput



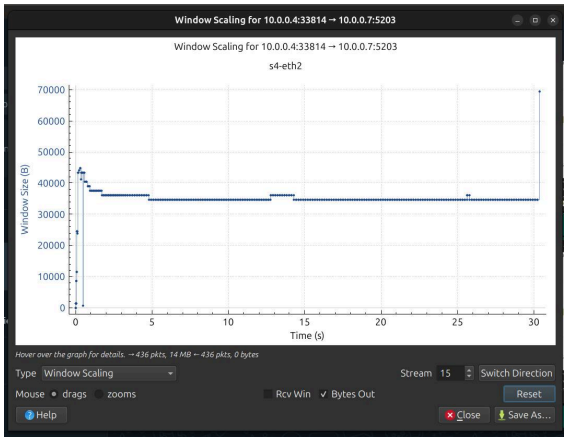


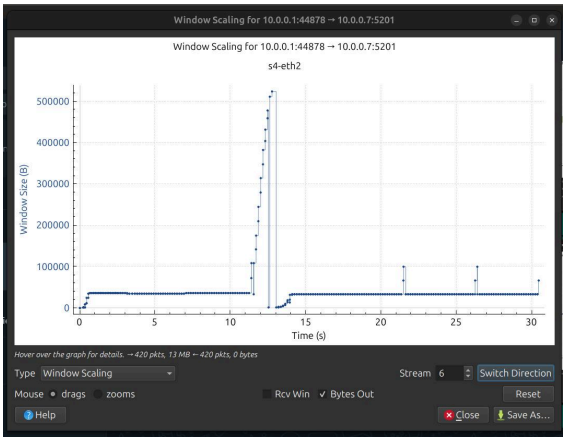


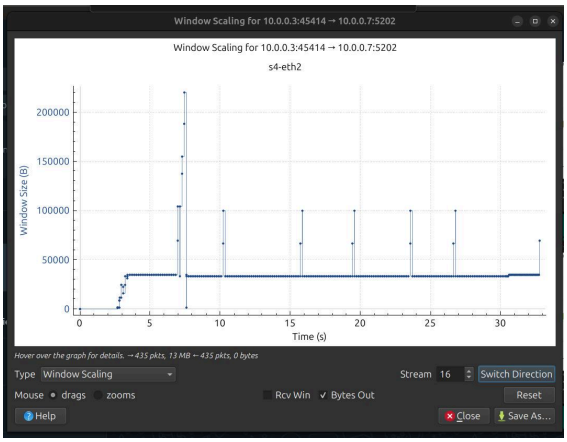
3. Packet Loss Rate



4. Maximum Window size achieved

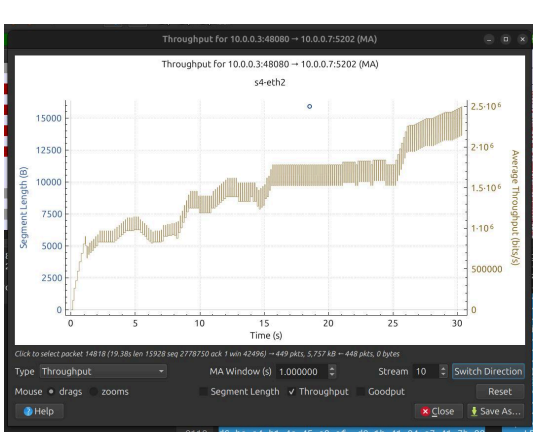


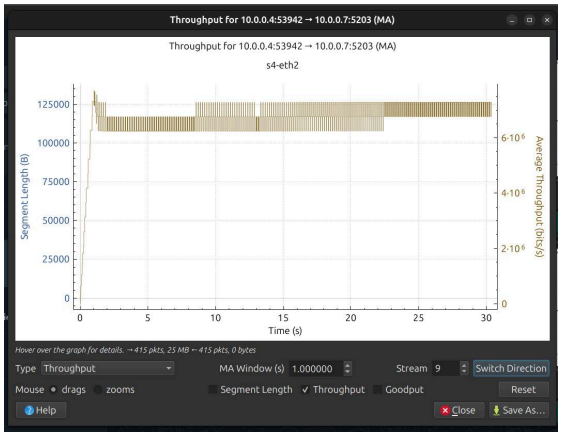


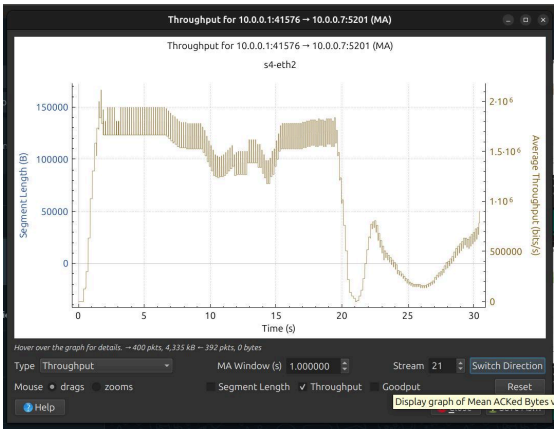


Protocol: HIGHSPEED

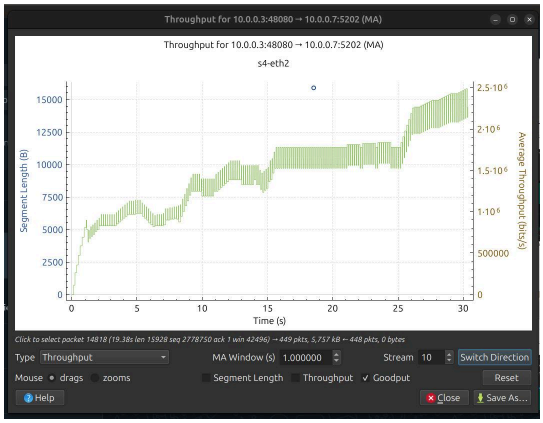
1. Throughput

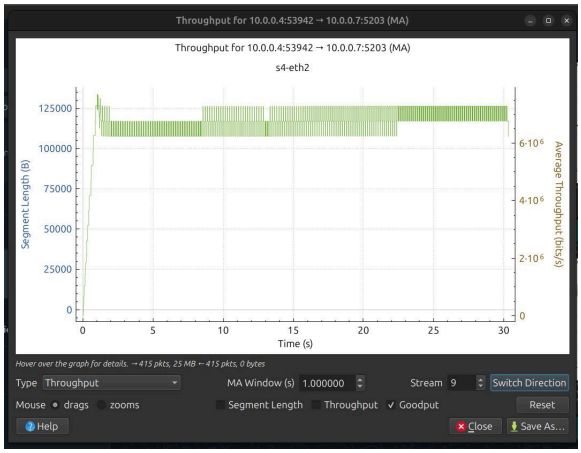

Throughput for 10.0.0.3:48080 → 10.0.0.7:5202 (MA)
Throughput for 10.0.0.3:48080 → 10.0.0.7:5202 (MA)
s4-eth2
Segment Length (B)
Average Throughput (bits/s)
Time (s)
Type Throughput
MA Window (s) 1.000000
Stream 10
Switch Direction
Mouse drags zooms
Segment Length Throughput Goodput
Reset
Help
Close
Save As...

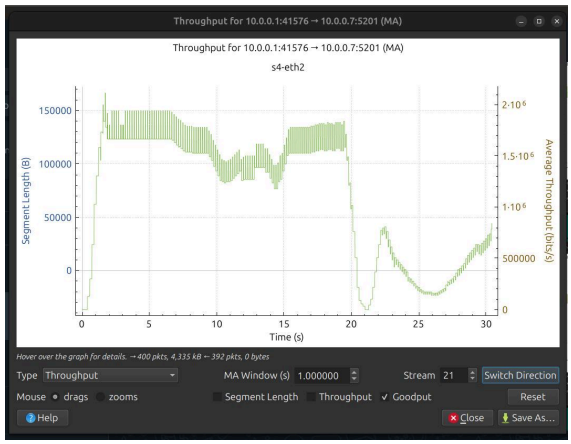

Throughput for 10.0.0.4:53942 → 10.0.0.7:5203 (MA)
Throughput for 10.0.0.4:53942 → 10.0.0.7:5203 (MA)
s4-eth2
Segment Length (B)
Average Throughput (bits/s)
Time (s)
Type Throughput
MA Window (s) 1.000000
Stream 9
Switch Direction
Mouse drags zooms
Segment Length Throughput Goodput
Reset
Help
Close
Save As...


Throughput for 10.0.0.1:41576 → 10.0.0.7:5201 (MA)
Throughput for 10.0.0.1:41576 → 10.0.0.7:5201 (MA)
s4-eth2
Segment Length (B)
Average Throughput (bits/s)
Time (s)
Type Throughput
MA Window (s) 1.000000
Stream 21
Switch Direction
Mouse drags zooms
Segment Length Throughput Goodput
Reset
Help
Display graph of Mean ACKed Bytes

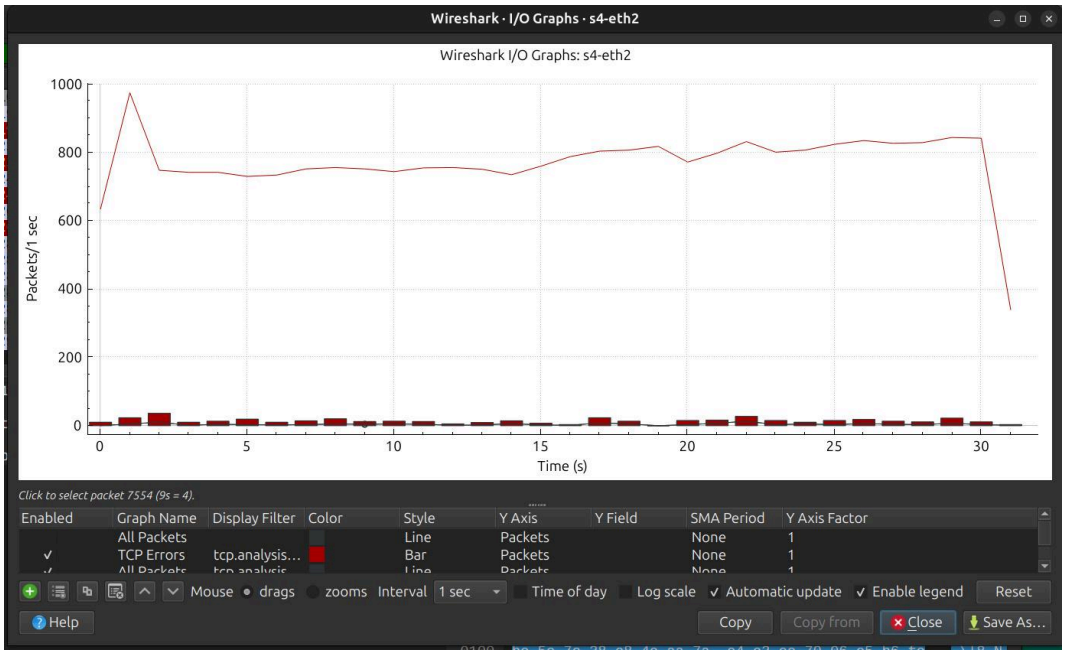
2. Goodput


Throughput for 10.0.0.3:48080 → 10.0.0.7:5202 (MA)
Throughput for 10.0.0.3:48080 → 10.0.0.7:5202 (MA)
s4-eth2
Segment Length (B)
Average Throughput (bits/s)
Time (s)
Type Throughput
MA Window (s) 1.000000
Stream 10
Switch Direction
Mouse drags zooms
Segment Length Throughput Goodput
Reset
Help
Close
Save As...

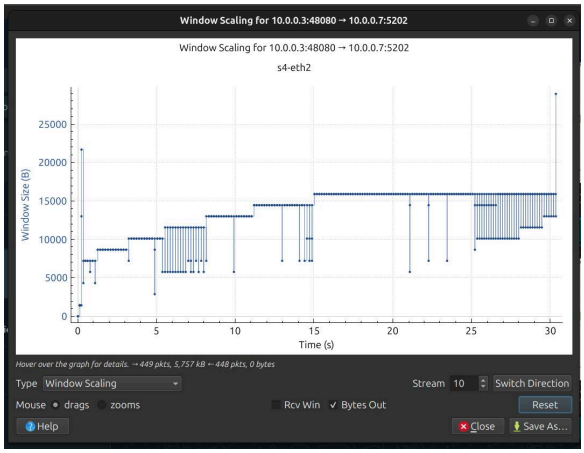

Throughput for 10.0.0.4:53942 → 10.0.0.7:5203 (MA)
Throughput for 10.0.0.4:53942 → 10.0.0.7:5203 (MA)
s4-eth2
Segment Length (B)
Average Throughput (bits/s)
Time (s)
Type Throughput
MA Window (s) 1.000000
Stream 9
Switch Direction
Mouse drags zooms
Segment Length Throughput Goodput
Reset
Help
Close
Save As...

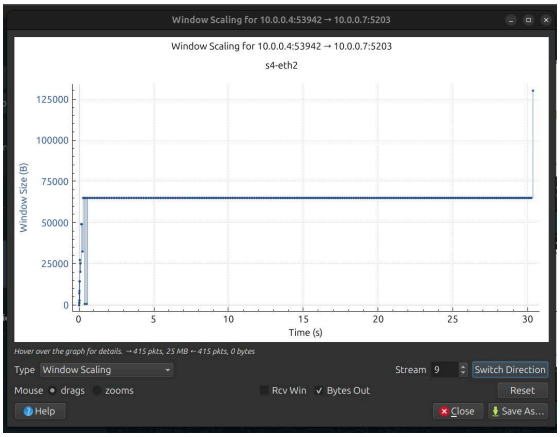

Throughput for 10.0.0.1:41576 → 10.0.0.7:5201 (MA)
Throughput for 10.0.0.1:41576 → 10.0.0.7:5201 (MA)
s4-eth2
Segment Length (B)
Average Throughput (bits/s)
Time (s)
Type Throughput
MA Window (s) 1.000000
Stream 21
Switch Direction
Mouse drags zooms
Segment Length Throughput Goodput
Reset
Help
Close
Save As...

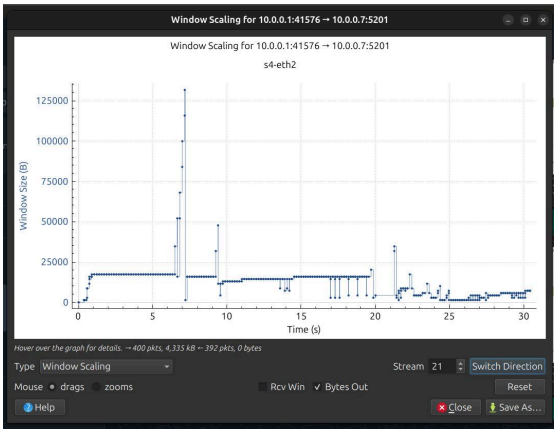
3. Packet Loss Rate



4. Maximum Window size achieved







Observations

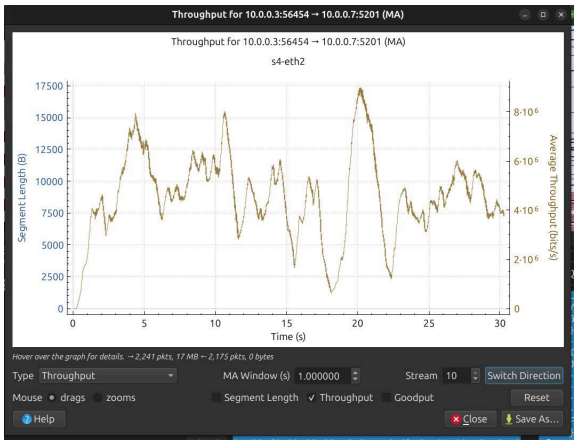
With 1% packet loss, the overall throughput decreased slightly compared to the no-loss scenario, as TCP congestion control mechanisms detected packet drops and adjusted the congestion window (cwnd). For a single client, Reno experienced minor fluctuations in throughput due to its multiplicative decrease response to loss, but BIC and HighSpeed recovered faster, maintaining a more stable transmission rate. In the two-client case, the competition for bandwidth resulted in more noticeable variations, with Reno suffering more than BIC due to its conservative approach, while HighSpeed occasionally overshot its optimal rate before stabilizing. For multiple clients, the effects of loss became more prominent as more flows competed for limited bandwidth, leading to slight fairness issues—TCP Reno connections struggled to maintain equal throughput, while BIC adapted better, and HighSpeed exhibited small bursts of instability. Goodput was slightly lower than throughput due to retransmissions, but since loss was minimal, it remained relatively close. Packet loss rates showed a small increase in retransmissions, but congestion control protocols still operated effectively without severe degradation.

Loss- 5%

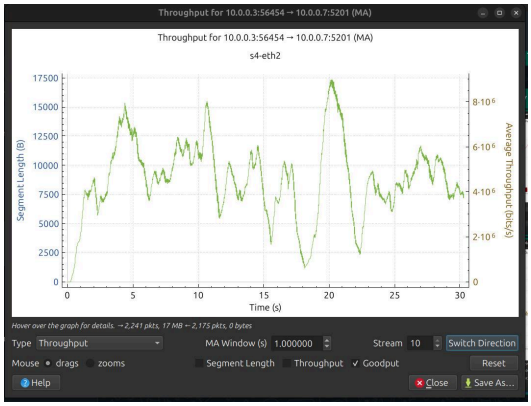
Scenario 1

Protocol: RENO

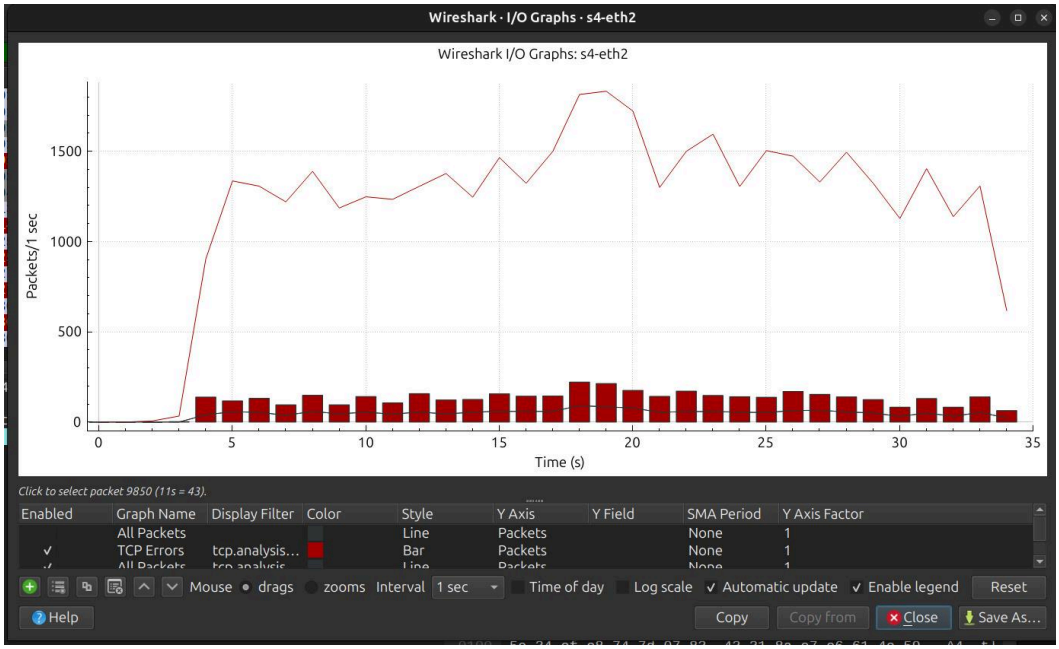
1. Throughput



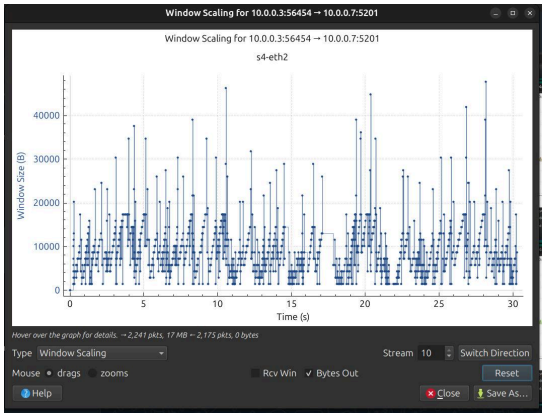
2. Goodput



3. Packet Loss Rate

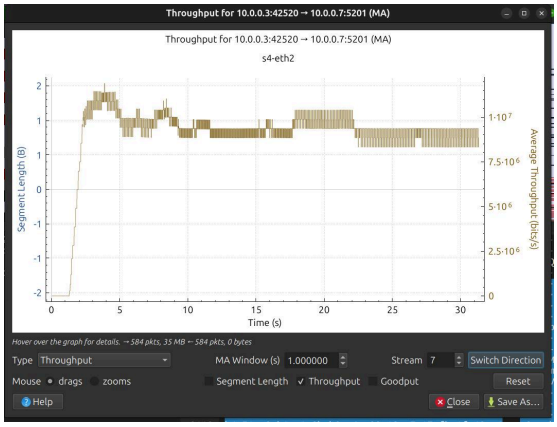


4. Maximum Window size achieved

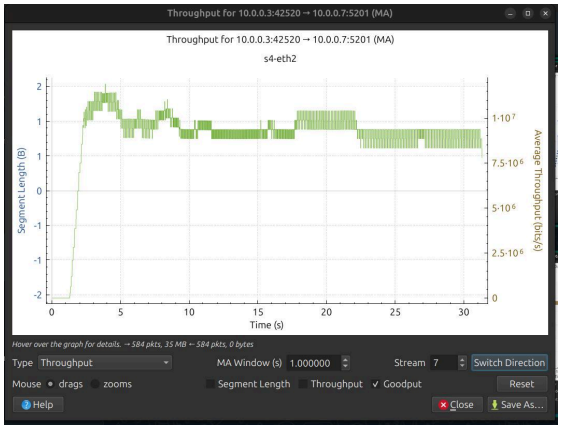


Protocol: BIC

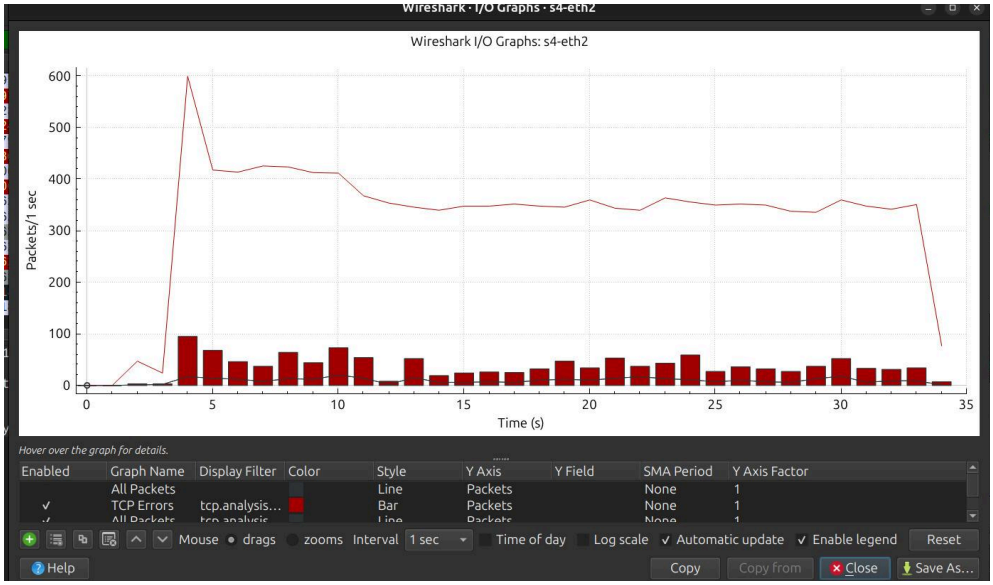
1. Throughput



2. Goodput



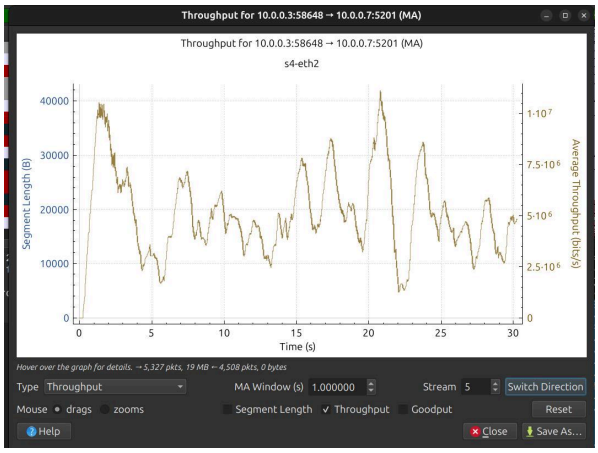
3. Packet Loss Rate



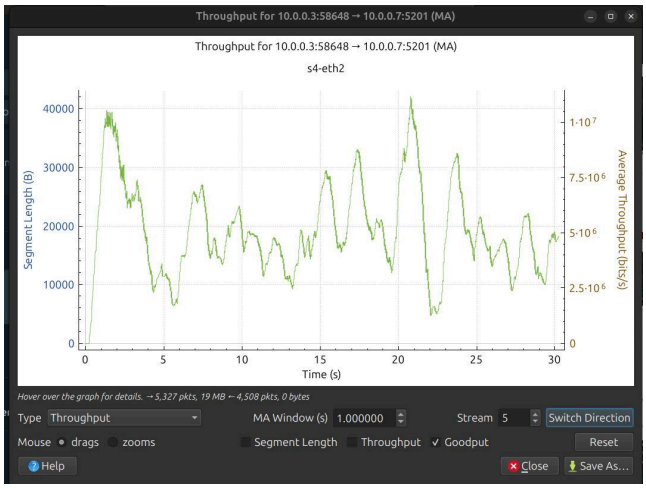
4. Maximum Window size achieved

Protocol: HIGHSPEED

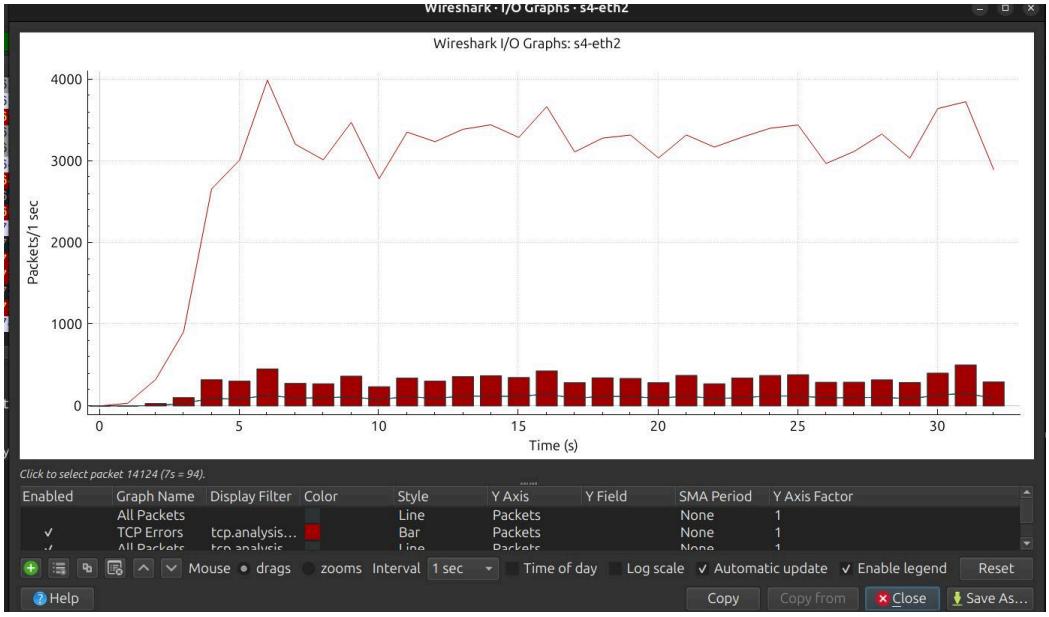
1. Throughput



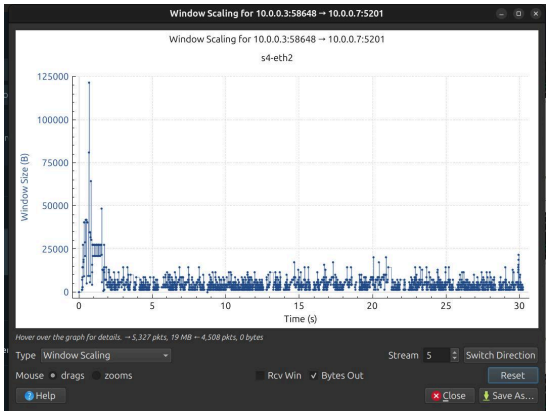
2. Goodput



3. Packet Loss Rate



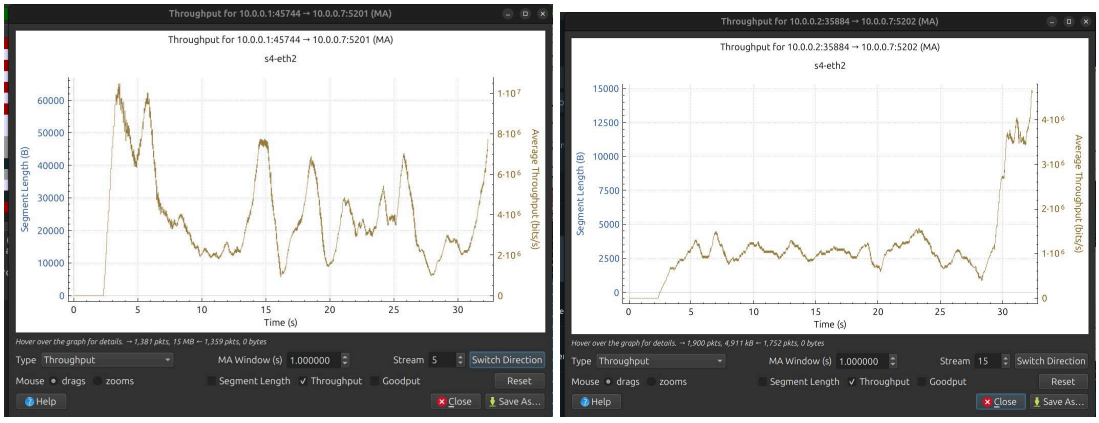
4. Maximum Window size achieved



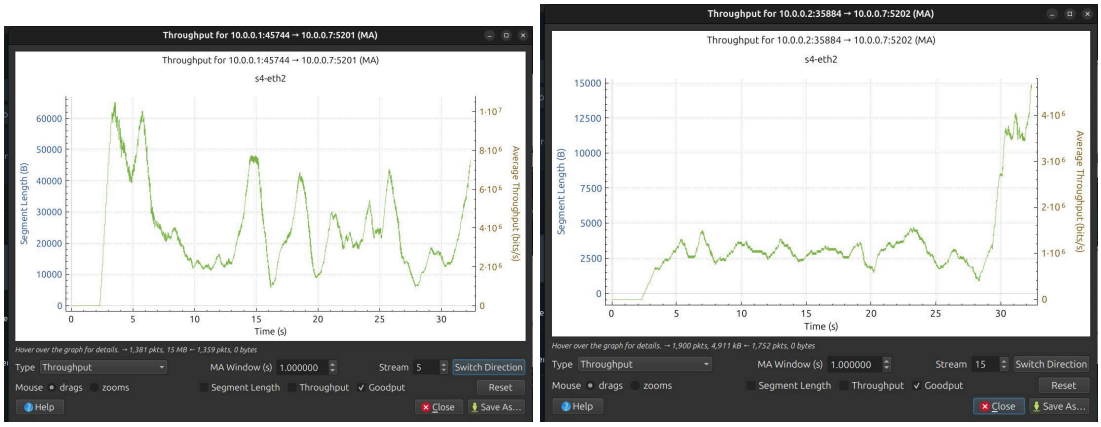
Scenario 2a

Protocol: RENO

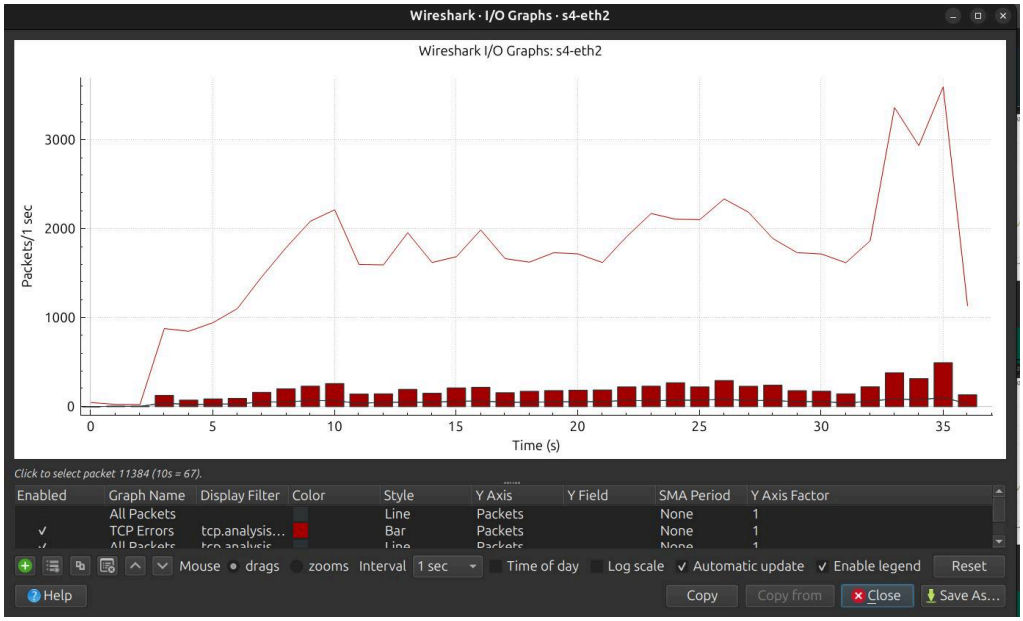
1. Throughput



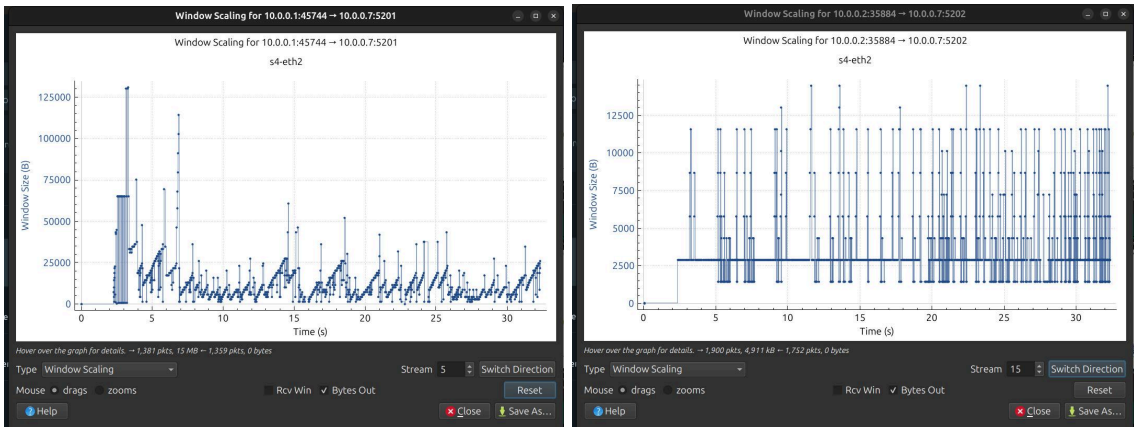
2. Goodput



3. Packet Loss Rate

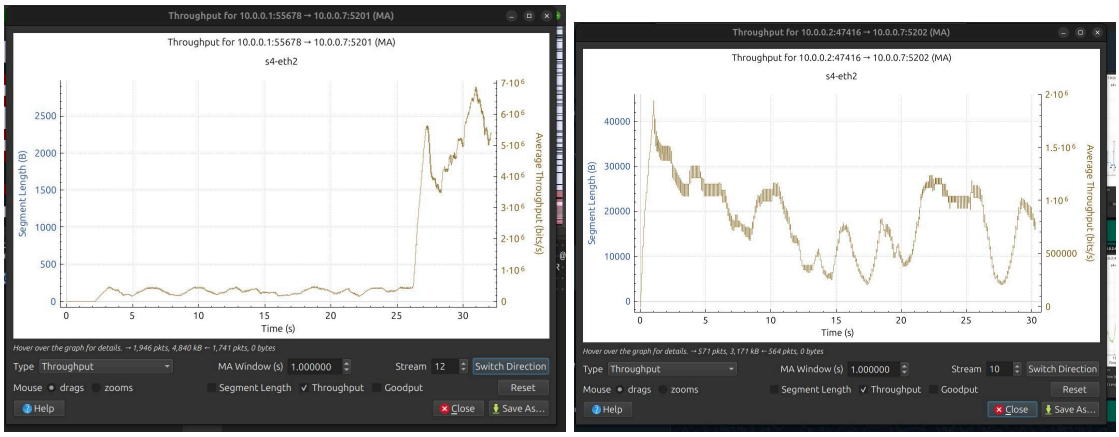


4. Maximum Window size achieved

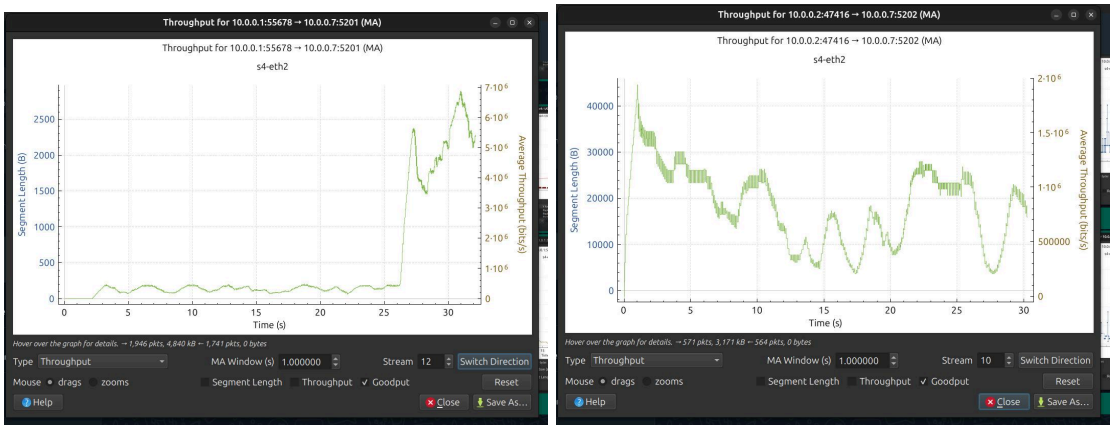


Protocol: BIC

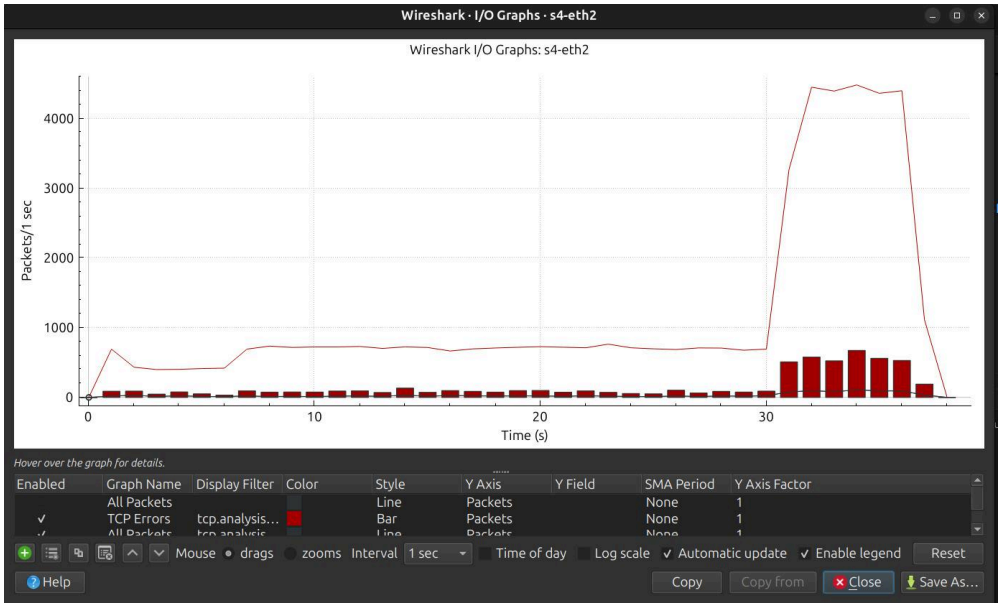
1. Throughput



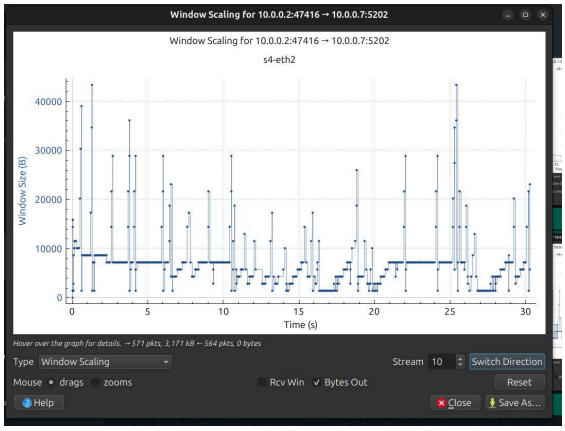
2. Goodput



3. Packet Loss Rate

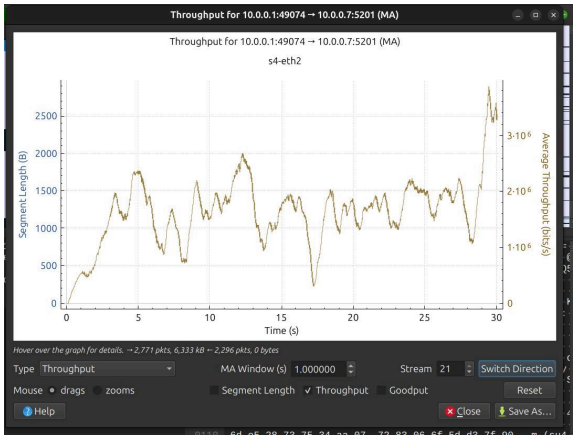


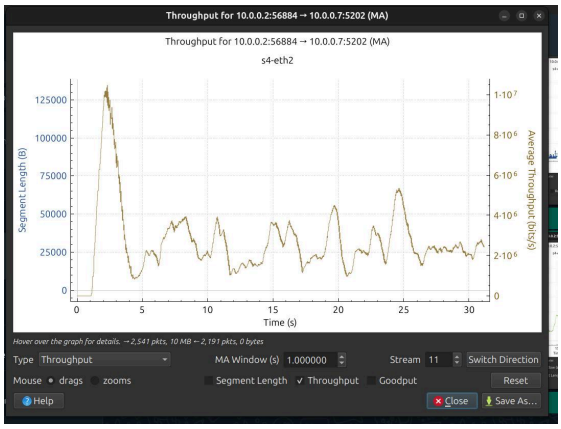
4. Maximum Window size achieved



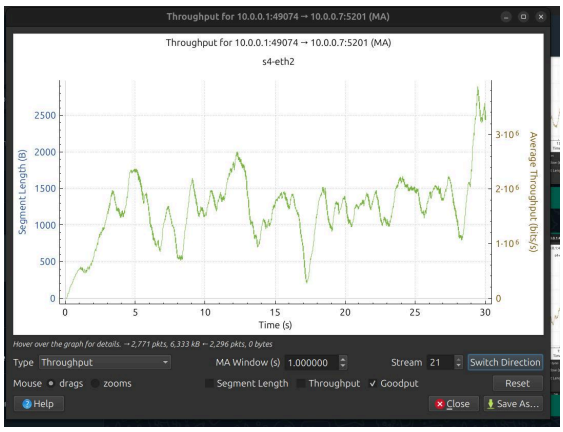
Protocol: HIGHSPEED

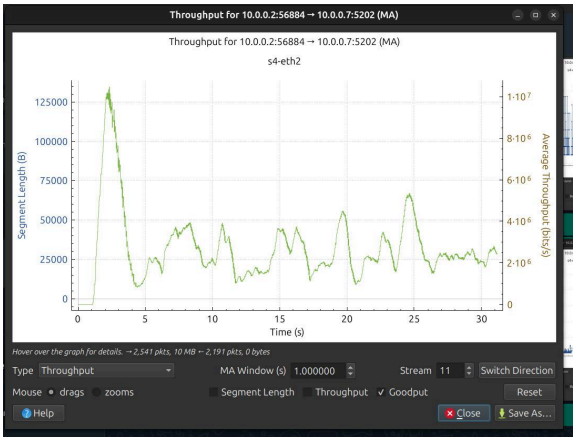
1. Throughput



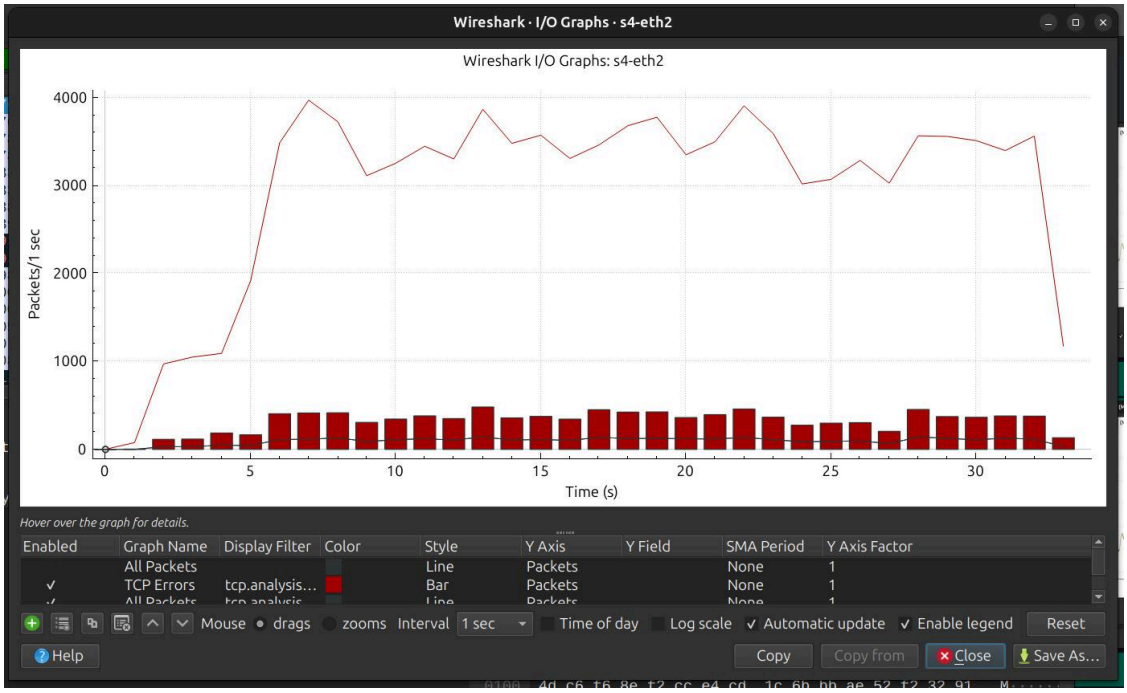


2. Goodput

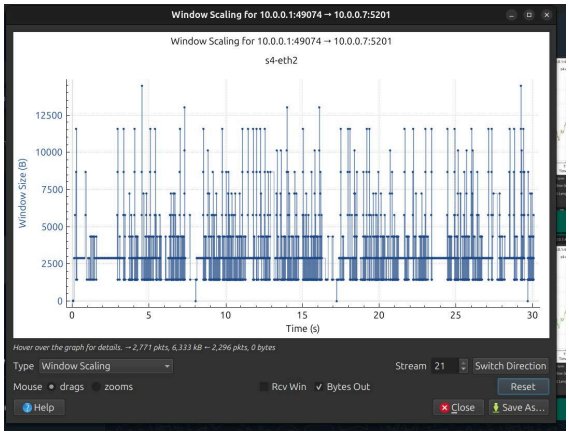


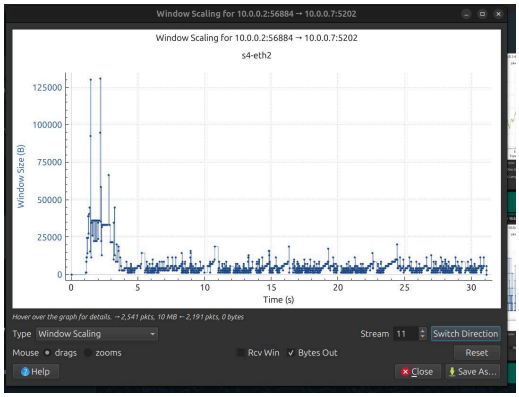


3. Packet Loss Rate



4. Maximum Window size achieved

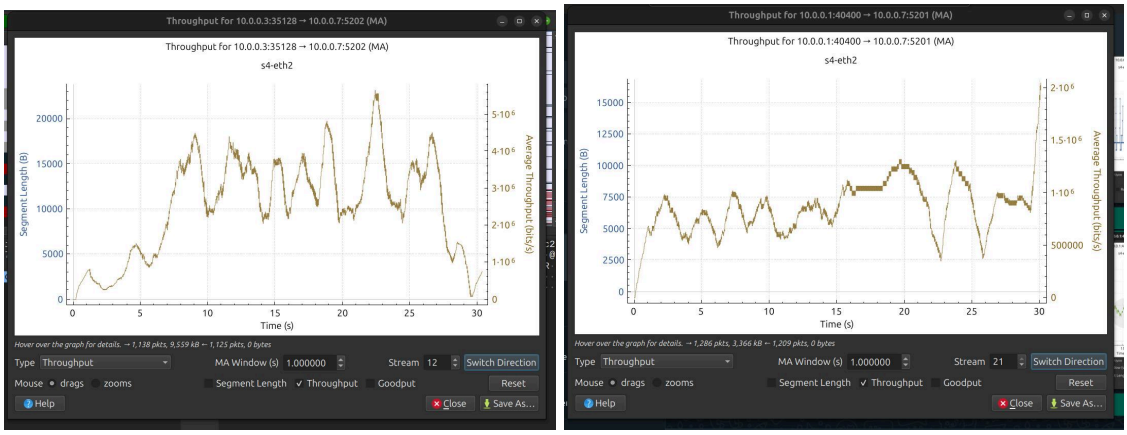




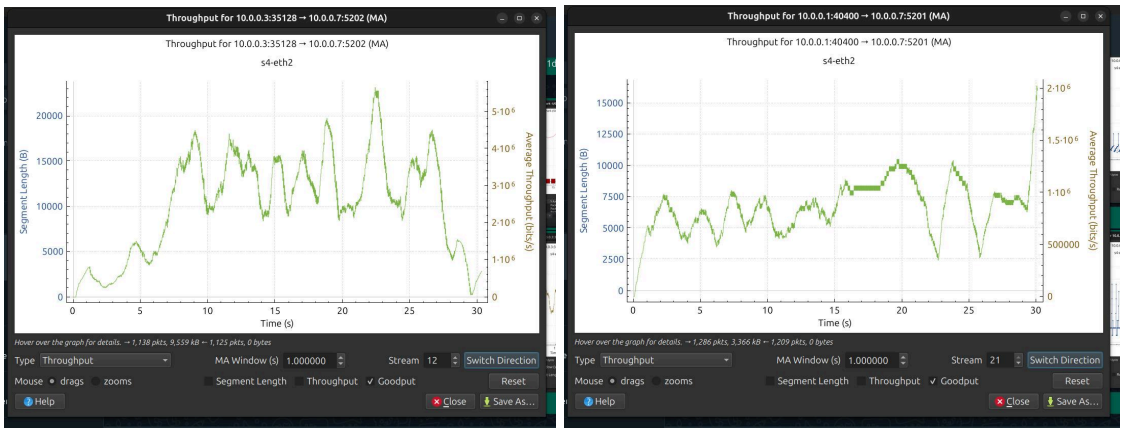
Scenario 2b

Protocol: RENO

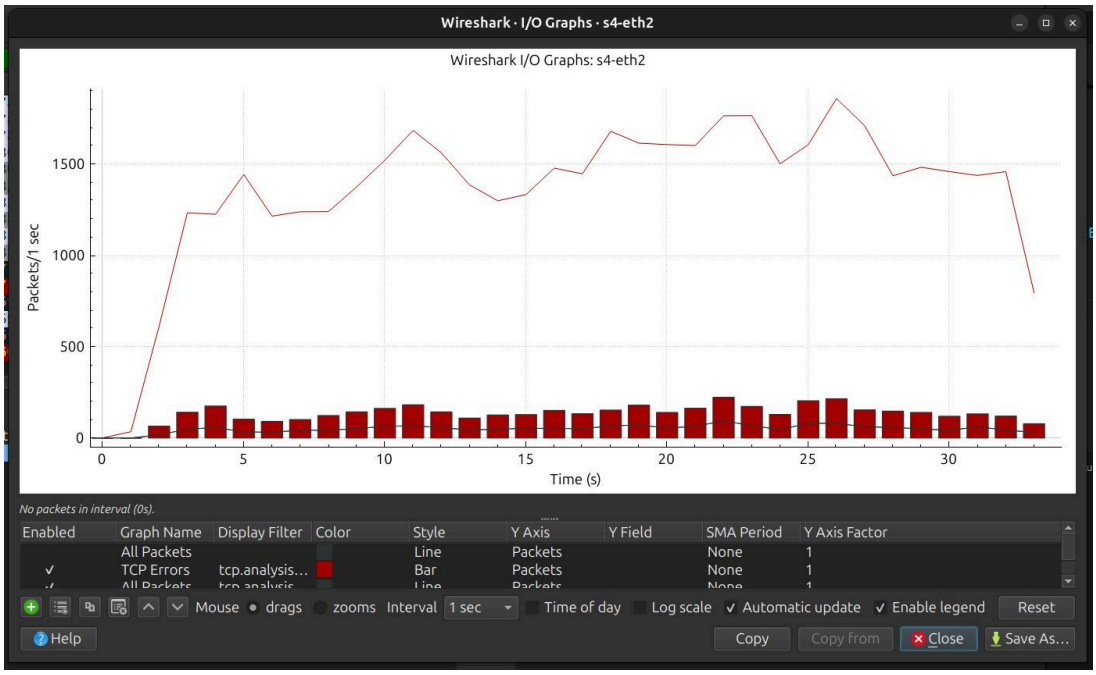
1. Throughput



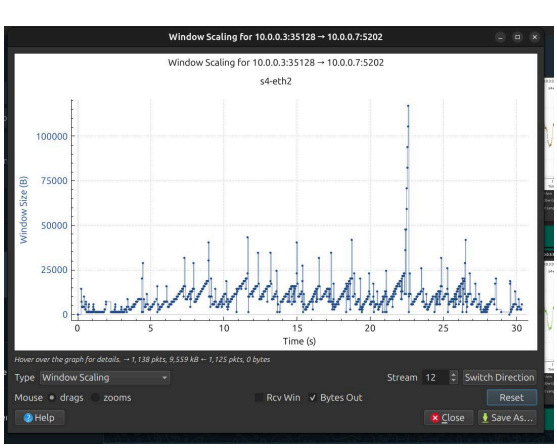
2. Goodput

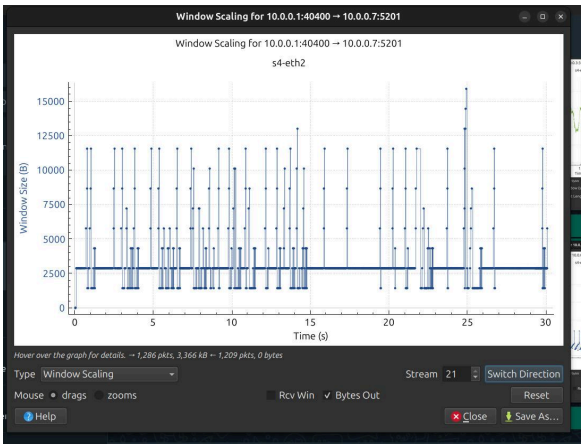


3. Packet Loss Rate



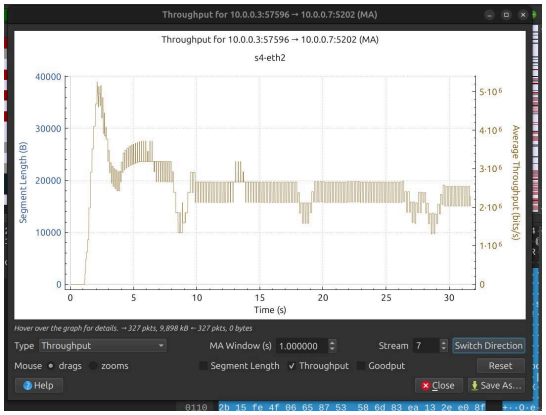
4. Maximum Window size achieved

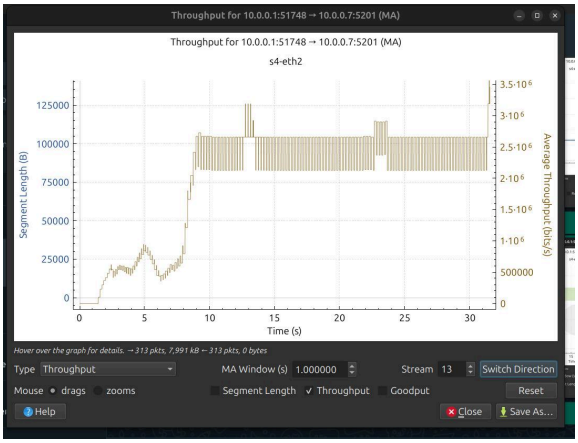




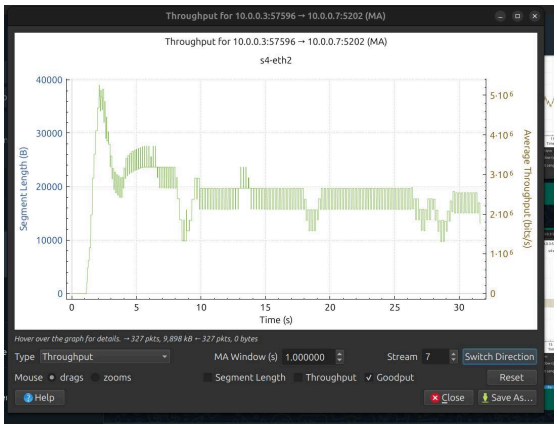
Protocol: BIC

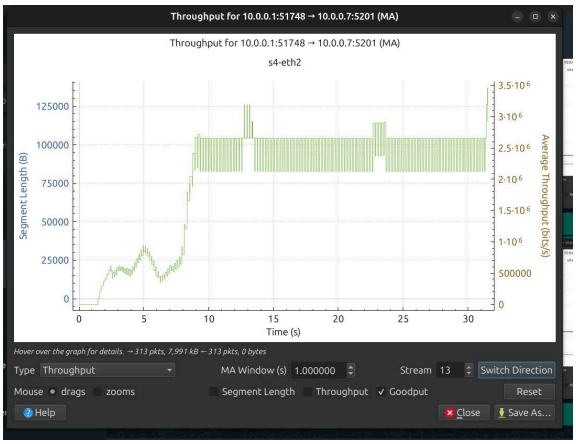
1. Throughput



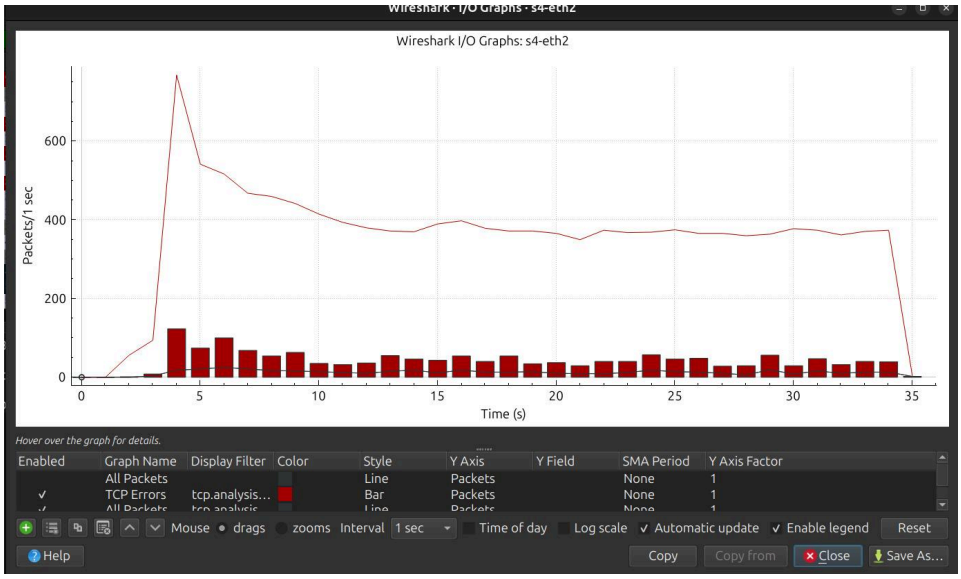


2. Goodput

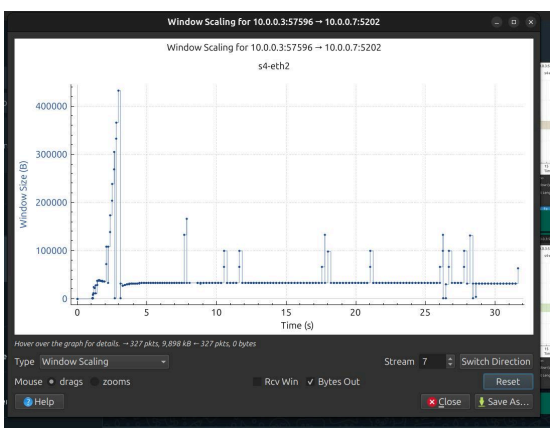


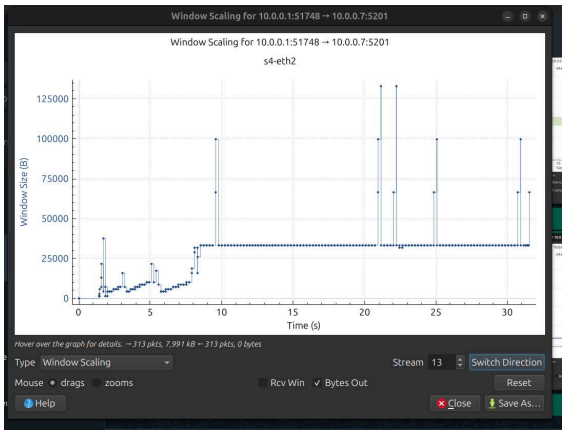


3. Packet Loss Rate



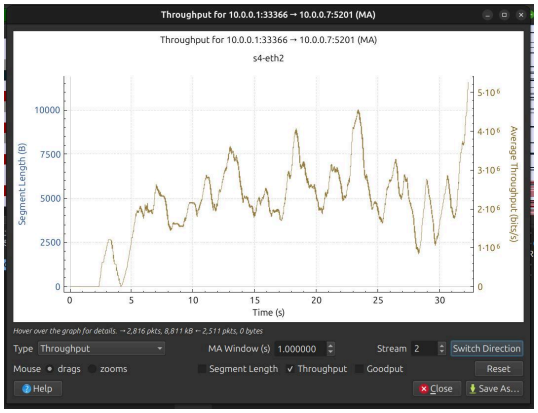
4. Maximum Window size achieved

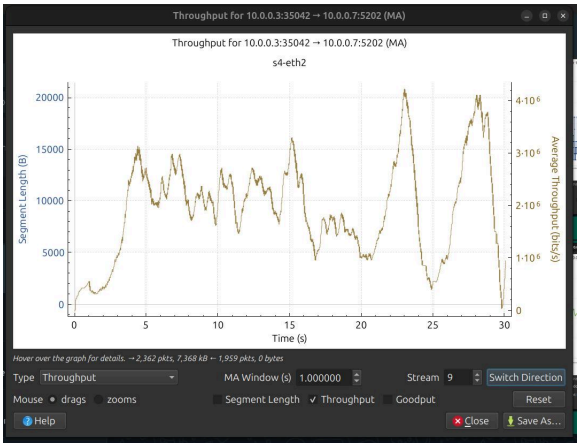




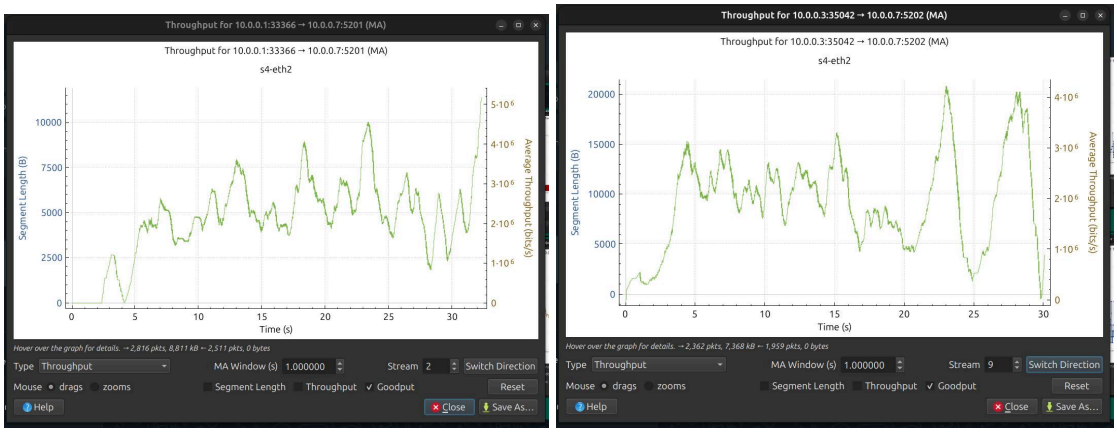
Protocol: HIGHSPEED

1. Throughput

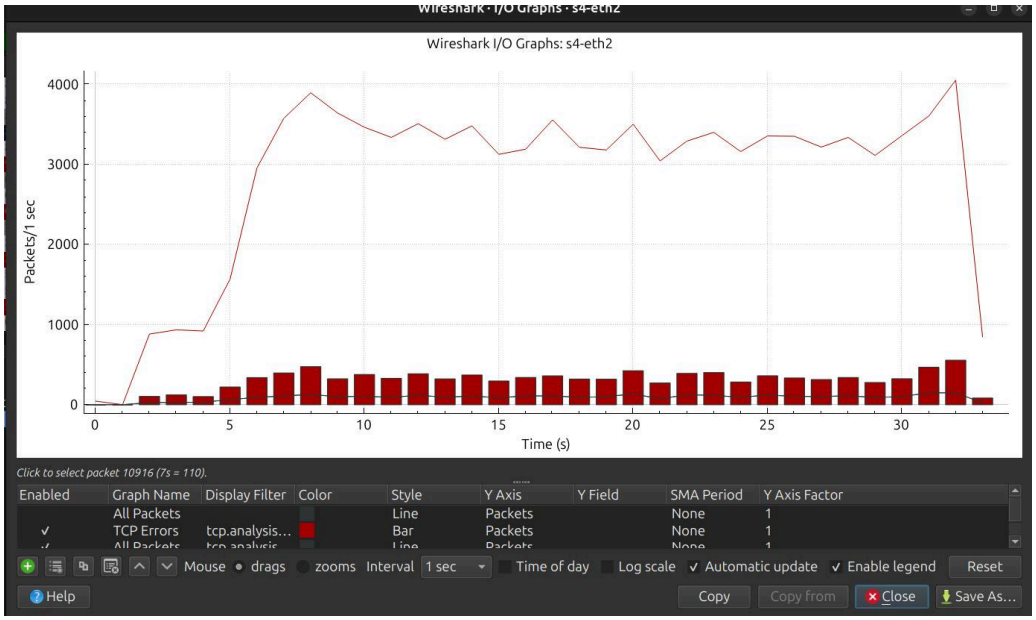




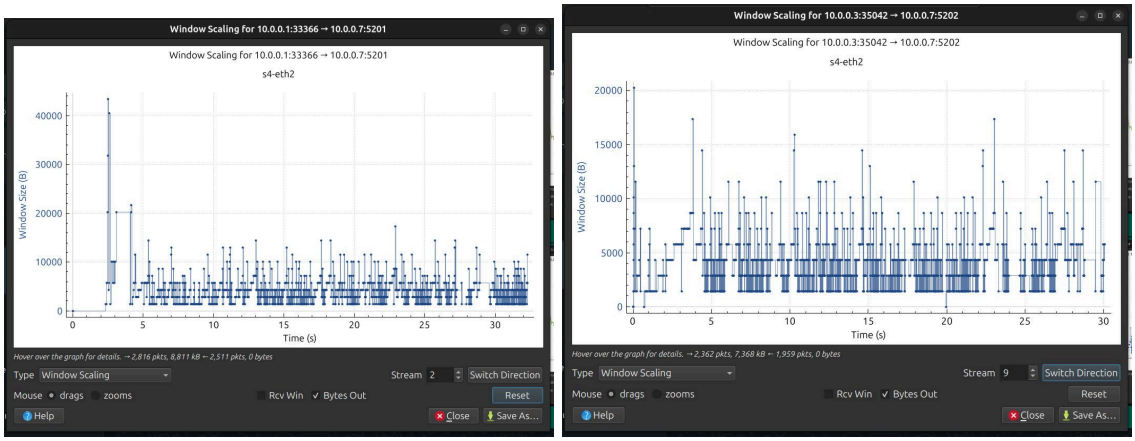
2. Goodput



3. Packet Loss Rate



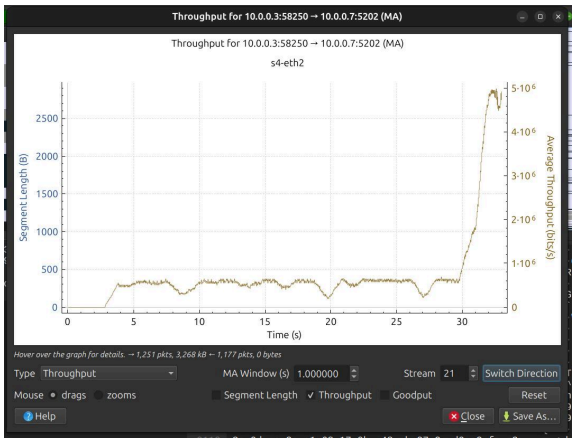
4. Maximum Window size achieved

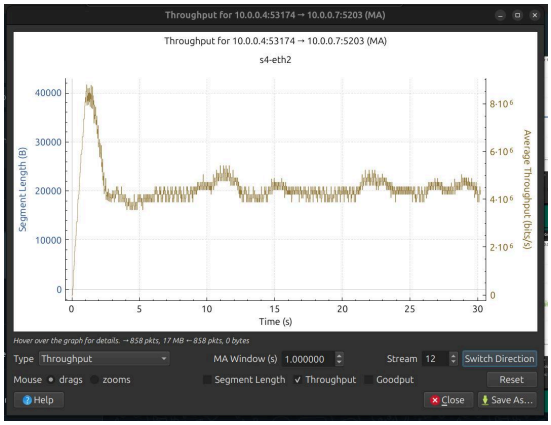


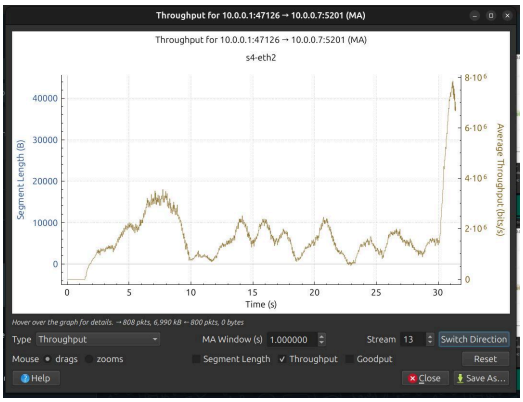
Scenario 2c

Protocol: RENO

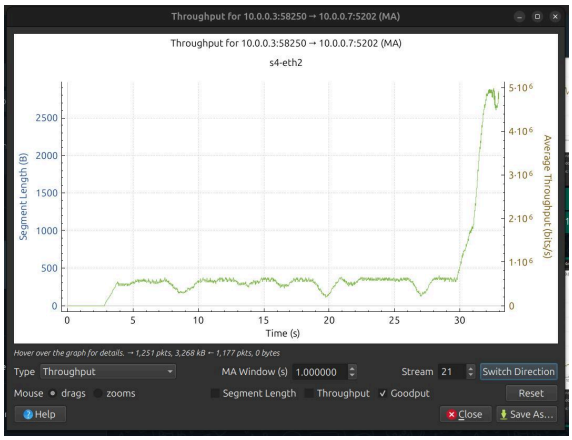
1. Throughput

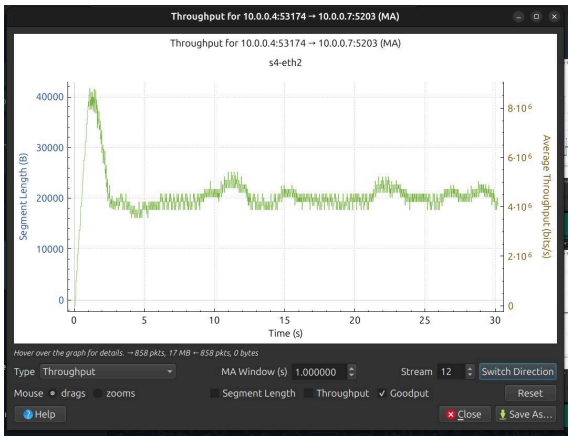


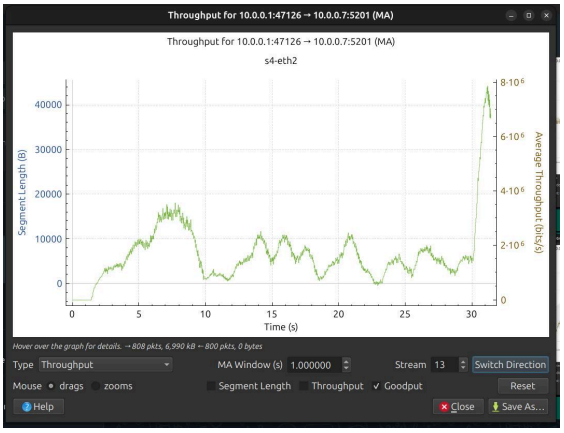




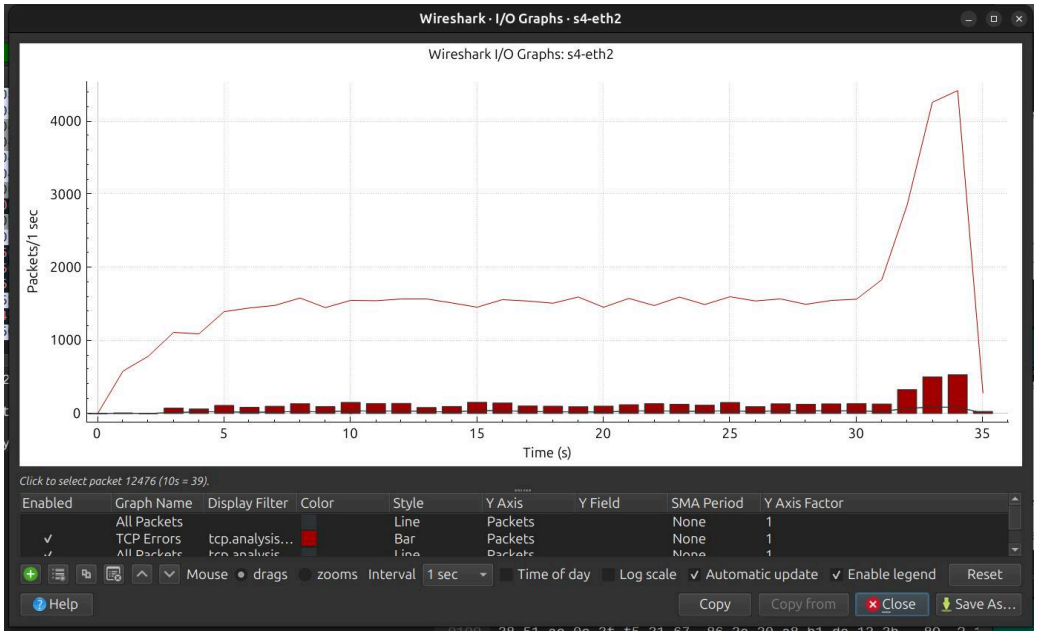
2. Goodput



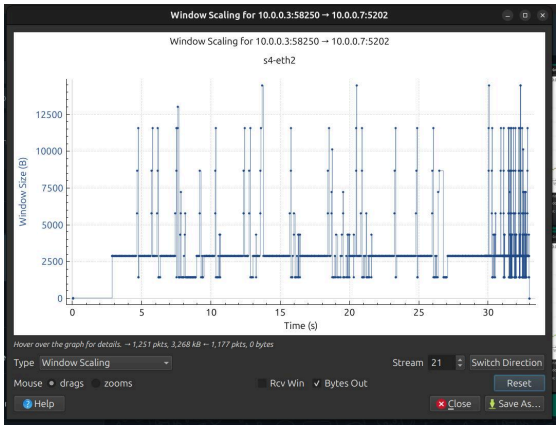


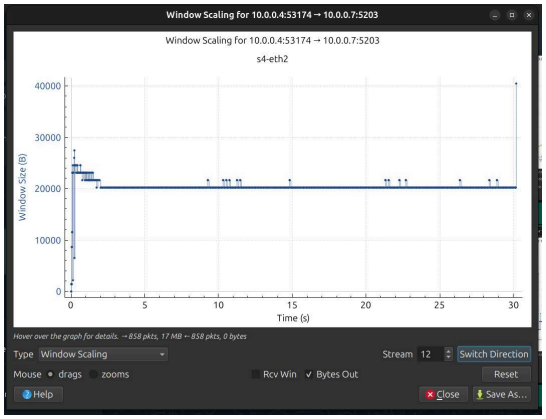


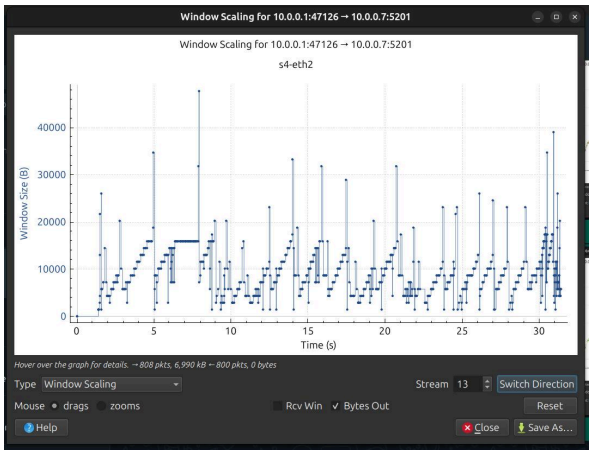
3. Packet Loss Rate



4. Maximum Window size achieved

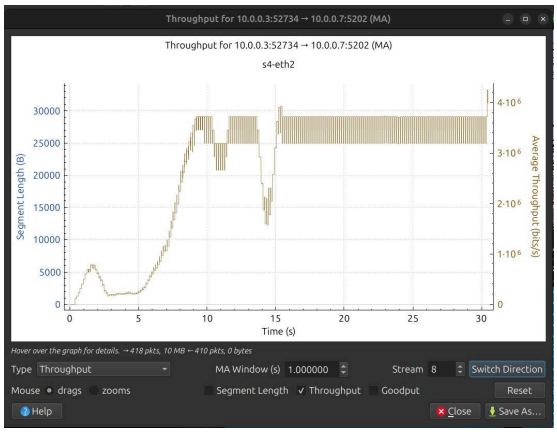


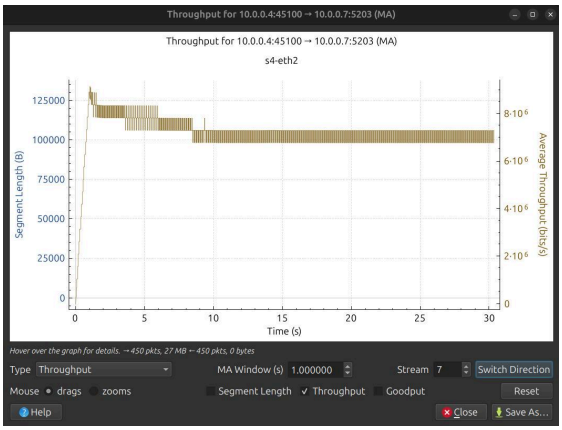


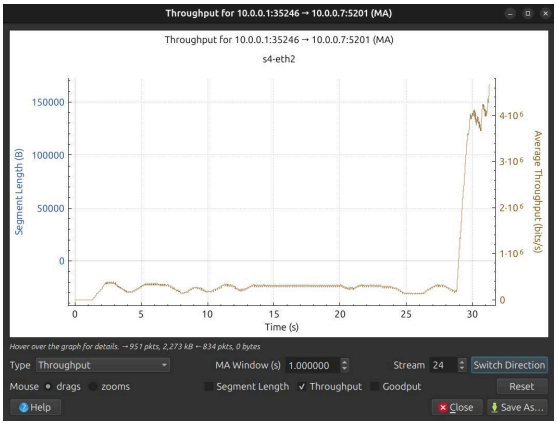


Protocol: BIC

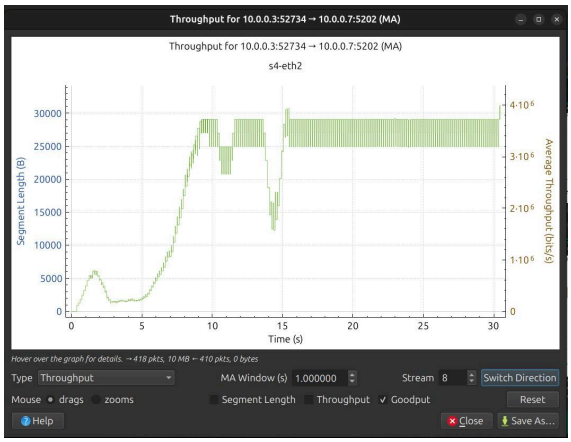
1. Throughput

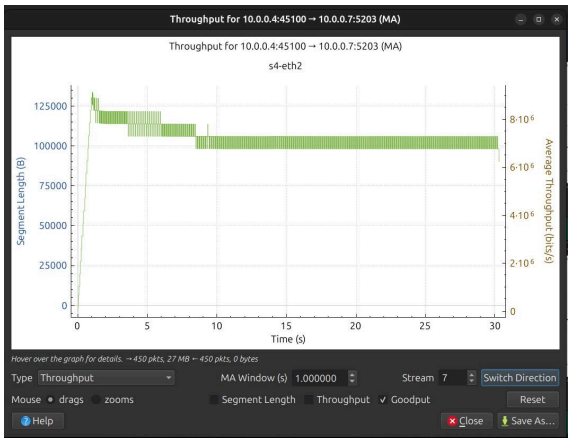


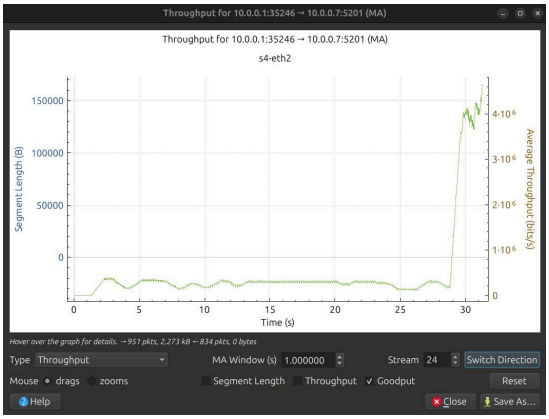




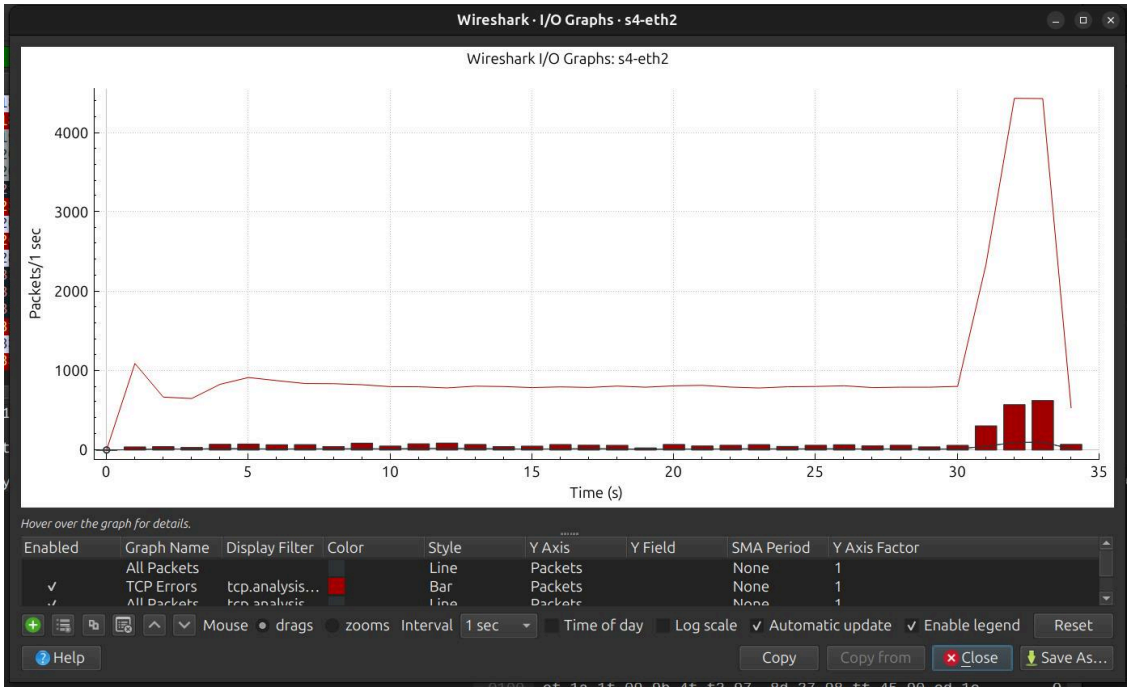
2. Goodput



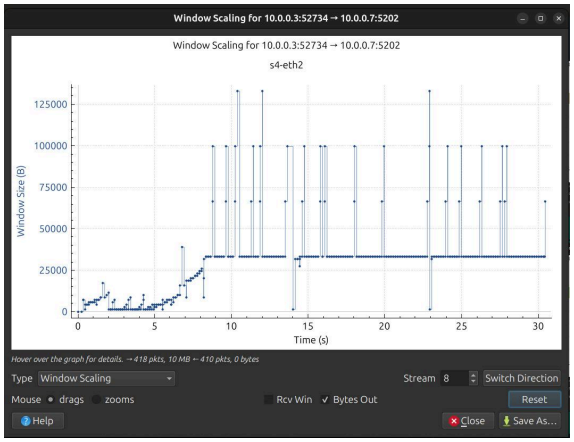


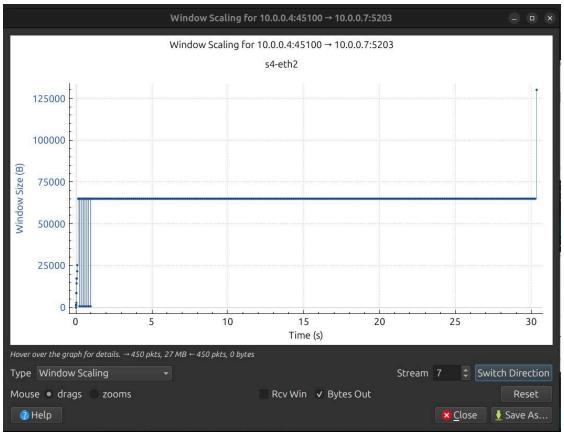


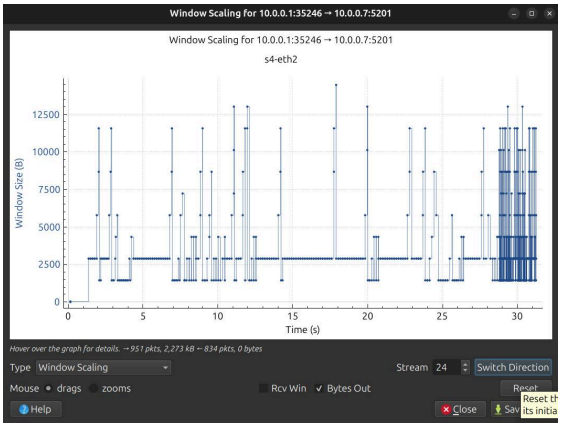
3. Packet Loss Rate



4. Maximum Window size achieved

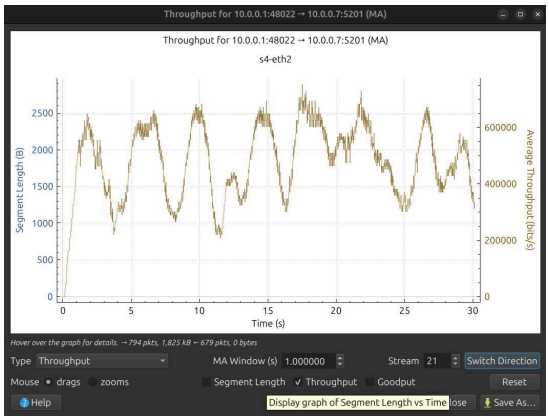


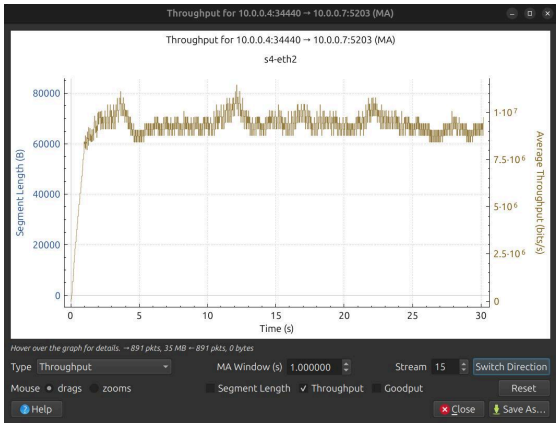


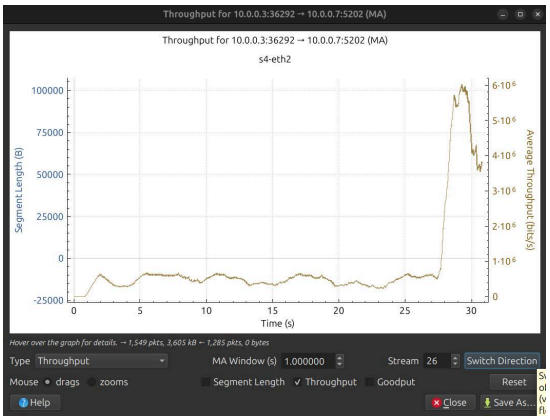


Protocol: HIGHSPEED

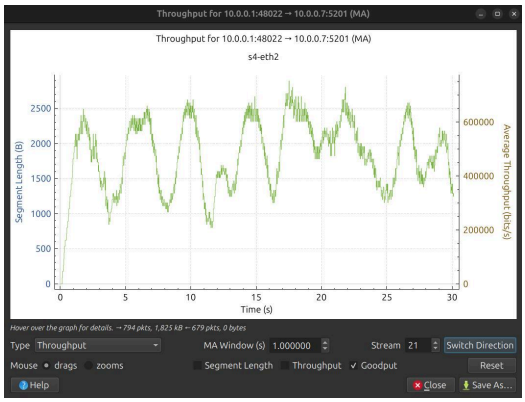
1. Throughput

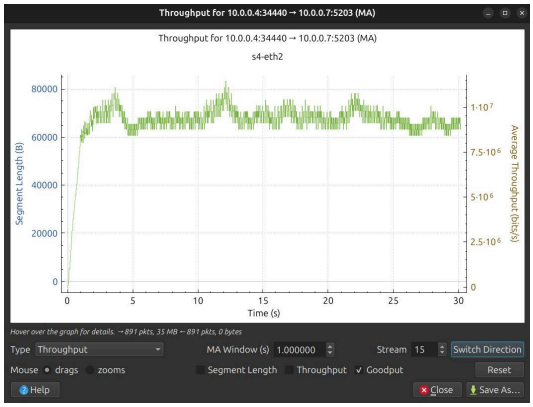


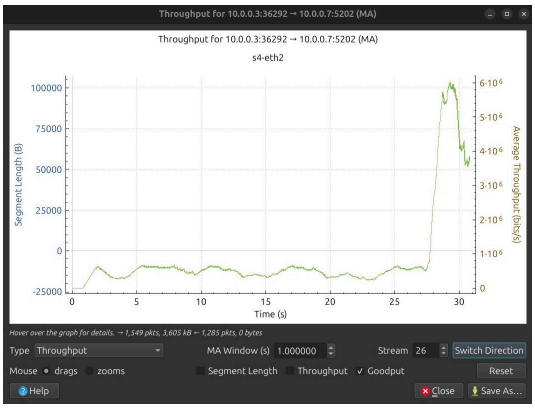




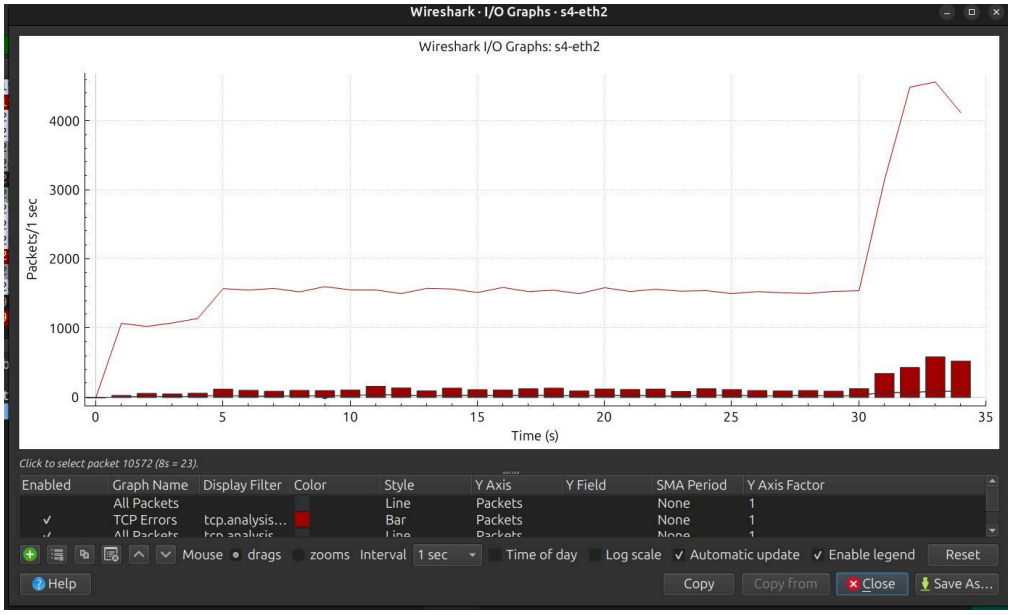
2. Goodput



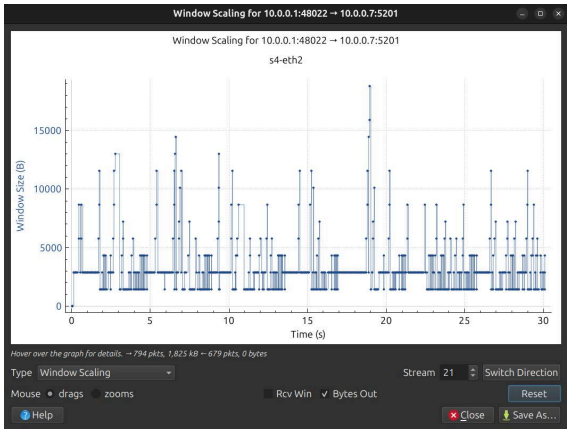


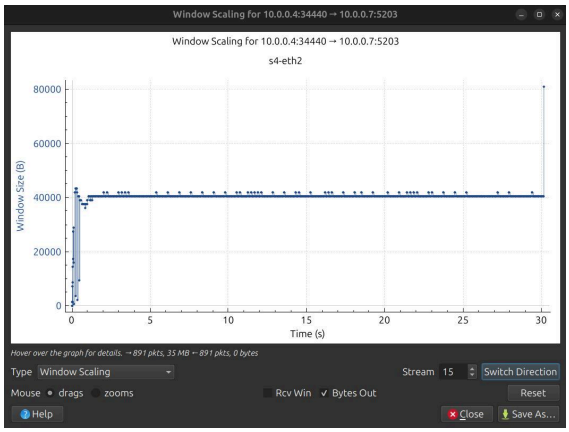


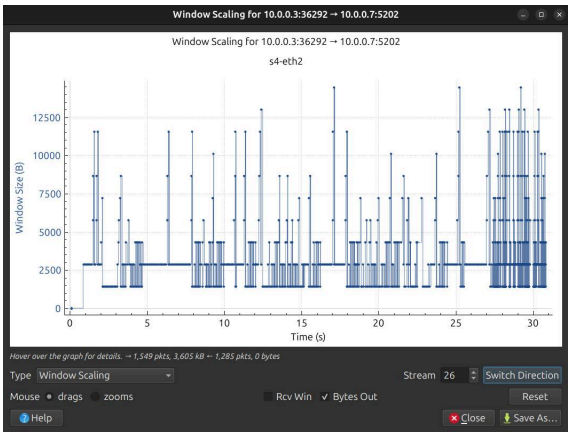
3. Packet Loss Rate



4. Maximum Window size achieved







Observations

At 5% packet loss, the network experienced significant degradation as throughput dropped sharply across all congestion control protocols, with Reno suffering the most due to its aggressive reduction in cwnd. For a single client, TCP Reno failed to fully utilize available bandwidth, resulting in low and highly unstable throughput, while BIC maintained a better window growth strategy but still suffered performance dips. HighSpeed, which relied on rapid

congestion window expansion, became highly unstable, frequently experiencing large drops in throughput followed by bursts of excessive transmission. With two clients, fairness was significantly impacted—connections running Reno suffered congestion collapse, while BIC and HighSpeed occasionally monopolized bandwidth. In a multi-client scenario, congestion effects compounded, causing severe packet loss, high retransmission rates, and a dramatic drop in goodput as most packets were lost or resent multiple times. Network efficiency was severely impacted, with TCP struggling to maintain stable flows. Retransmissions became excessive, leading to increased latency, reduced fairness, and lower overall network utilization.

Task-2

In this task, we implemented and analysed a SYN Flood attack and its mitigation techniques in a controlled network environment.

- A SYN Flood is a denial-of-service (DoS) attack where an attacker sends a large number of TCP SYN packets to a server but never completes the three-way handshake, exhausting the server's resources and making it unresponsive.

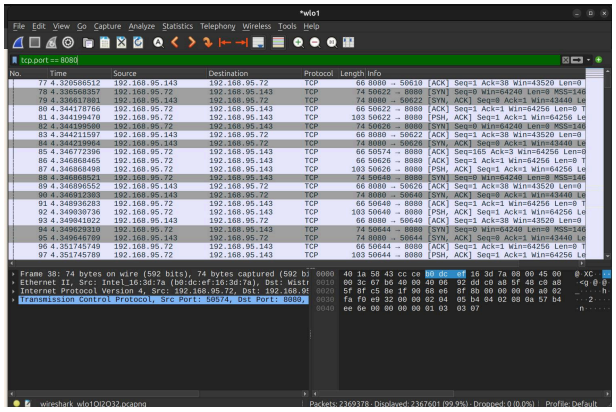
We conducted this task on two machines connected via a mobile hotspot:

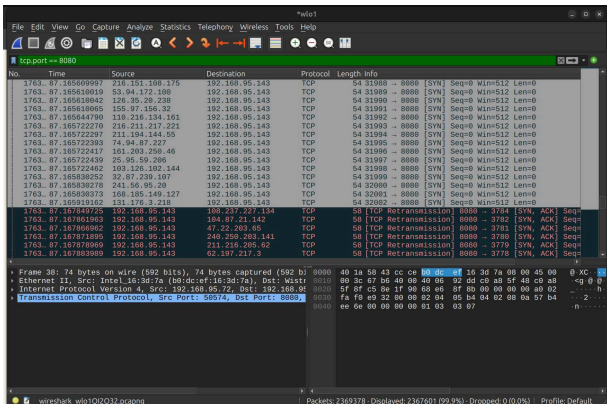
- Attacker Machine: Sent SYN Flood traffic to the server.
- Victim Machine (Server): Hosted a TCP server to observe the effects of the attack.

Part a

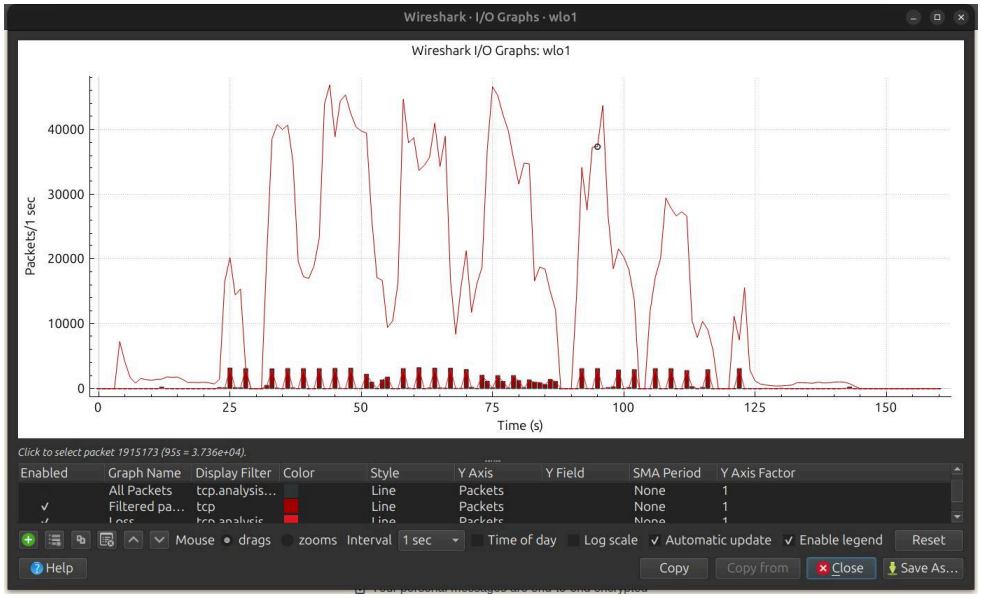
SYN Flood Attack Implementation

- A SYN flood attack was implemented by modifying the Victim's Linux kernel parameters:
  - sudo sysctl -w net.ipv4.tcp_syncookies=0.
  - sudo sysctl -w net.ipv4.tcp_max_syn_backlog=4096.
  - sudo sysctl -w net.ipv4.tcp_synack_retries=1.
- To monitor the traffic, we used wireshark.
- Started the legitimate traffic from the attacker machine using iperf3 command
- After 20 seconds, we started the attack using hping3 --flood command
- After 100 seconds, the attack was stopped
- After 20 seconds, we stopped the legitimate traffic

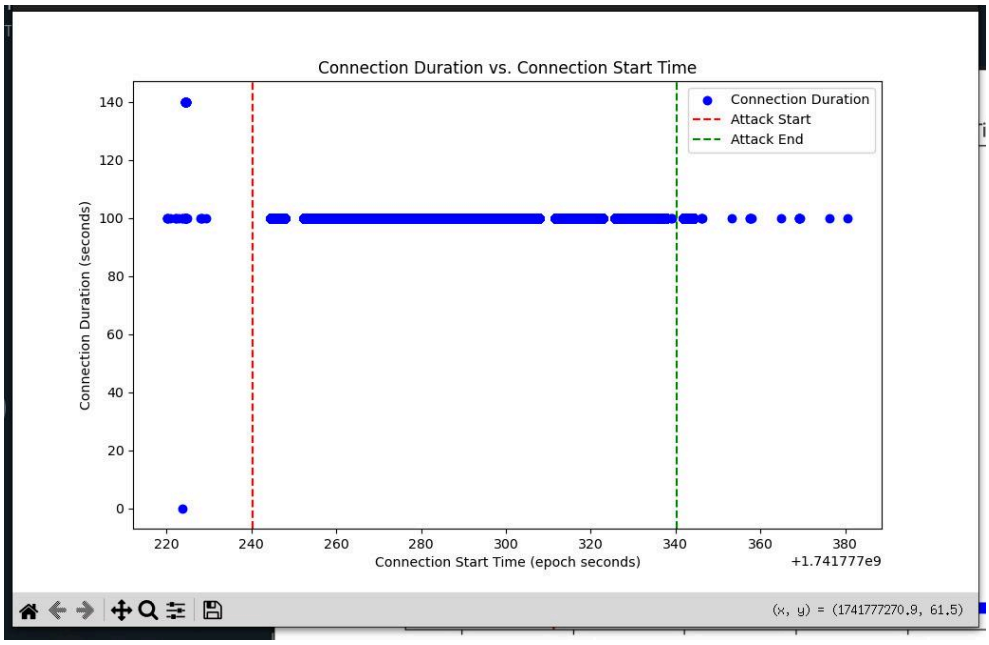




- The .pcap file captured during the experiment was processed to calculate TCP connection durations based on the time difference between:
  - First SYN packet (start of the connection).
  - ACK after FIN-ACK (graceful termination) or RESET (RST) packet (forced termination).
  - If a connection does not end properly, a default duration of 100 seconds is assigned.
- Finally, we plotted the graph between connection duration and connection start time.
- Wireshark Plots:



- Plot between connection duration and connection start time:



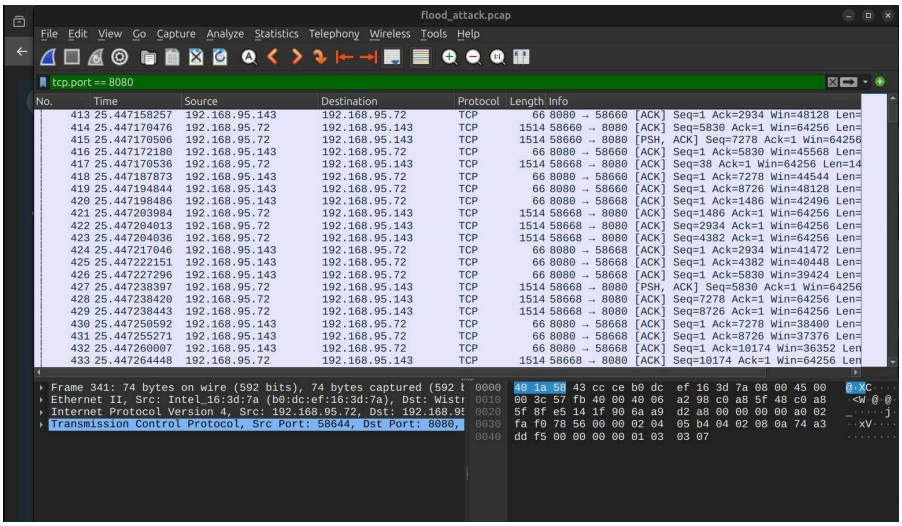
Observation

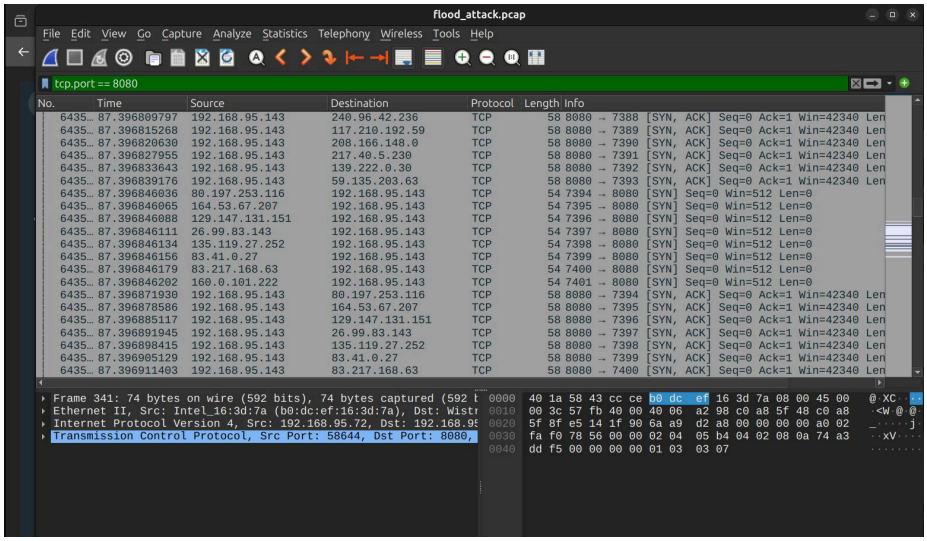
The server became unresponsive shortly after the attack started, as running `hping3 --flood` command revealed thousands of half-open connections in the SYN_RECV state, preventing legitimate clients from establishing new connections. CPU and memory usage on the server spiked significantly due to the excessive handling of these incomplete connections. A Wireshark capture confirmed the attack, showing an overwhelming number of incoming SYN packets without corresponding ACKs from clients. Throughput dropped to near zero as the server was unable to process legitimate requests, and packet retransmissions increased due to the attacker flooding the network with SYN requests that were never completed.

Part b

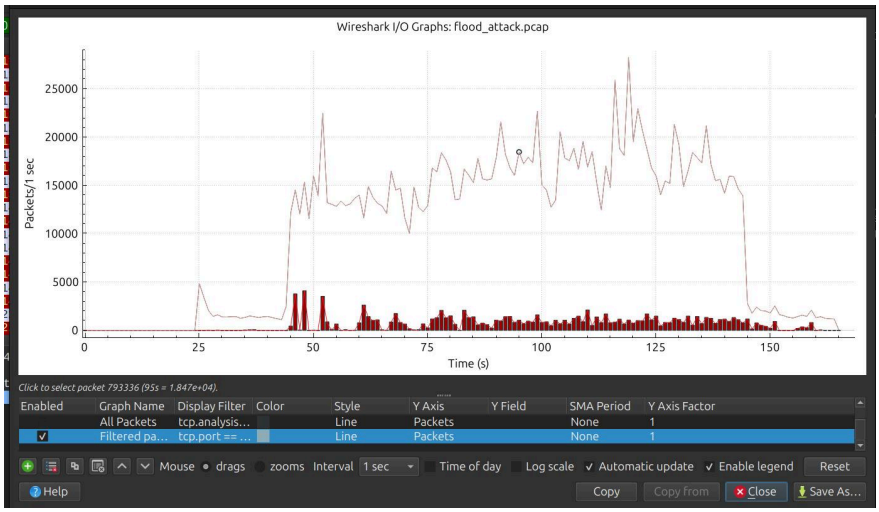
SYN Flood Attack Mitigation Implementation

- A SYN flood attack was implemented by modifying the Victim's Linux kernel parameters:
  - sudo sysctl -w net.ipv4.tcp_syncookies=1.
  - sudo sysctl -w net.ipv4.tcp_max_syn_backlog=64.
  - sudo sysctl -w net.ipv4.tcp_synack_retries=5.
- Then we repeat the same process as in part a.

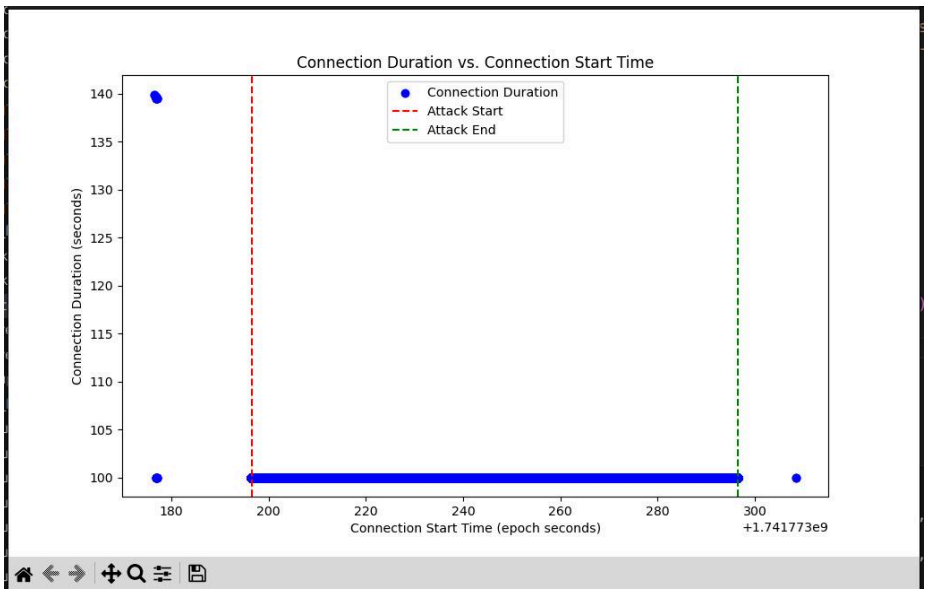




- Wireshark Plots:



- Plot between connection duration and connection start time:



Observation

After implementing SYN flood mitigation techniques, the server's responsiveness significantly improved. Unlike before, when the attack caused the server to become unresponsive and legitimate connections to fail, enabling SYN cookies ensured that the system could handle excessive SYN requests without exhausting resources. Lowering tcp_synack_retries reduced the time half-open connections remained in the queue, allowing faster recovery. As a result, legitimate clients could establish connections, though slight delays and minor throughput fluctuations were still observed under heavy attack conditions. Packet captures showed a drastic reduction in SYN_RECV states, indicating that the server successfully mitigated the attack and maintained normal operations.

Task-3

In this task, we analysed the effects of Nagle's Algorithm and Delayed-ACK over a TCP connection between client and server.

- Nagle's Algorithm reduces network congestion by buffering small packets and sending them only when there is enough data or an ACK is received, improving efficiency but increasing latency for small messages.
- Delayed ACK optimizes TCP performance by waiting before sending an ACK, allowing multiple ACKs to be combined, reducing overhead but potentially delaying packet transmission.

The task was conducted on a directly connected network setup consisting of:

- Client Machine: Initiating TCP connections.
- Server Machine: Listening for TCP connections on port 8080.
- Both machines connected via ethernet cable.

To analyze the impact of Nagle's Algorithm and Delayed ACK, we tested four different configurations, and a 4KB file was sent over the network from client to server at the transfer rate of 40bits/1ms:

Test Case	Nagle's Algorithm	Delayed ACK
Test 1	Enabled	Enabled
Test 2	Enabled	Disabled
Test 3	Disabled	Enabled
Test 4	Disabled	Disabled

Change in condition:

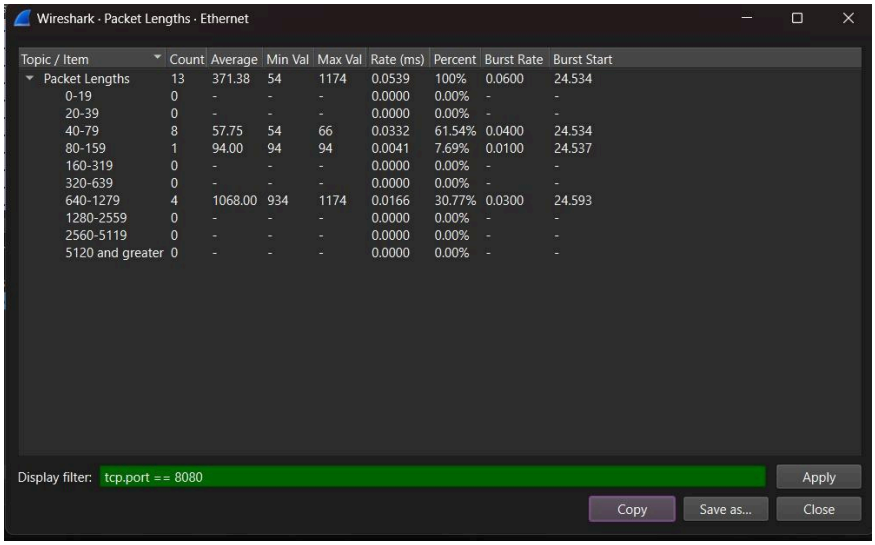
We reduced the packet transfer interval from 1 second to 1 millisecond because a 1-second gap was too long. This rate neutralized the effect of delayed ack and nagle's algo , effectively keeping the number of transmissions, throughput and other parameters constant for all the test cases. By shortening the interval to 1 millisecond, we ensured more frequent transmissions, allowing us to better observe the effects of these mechanisms on network performance.

Results:

Test 1:

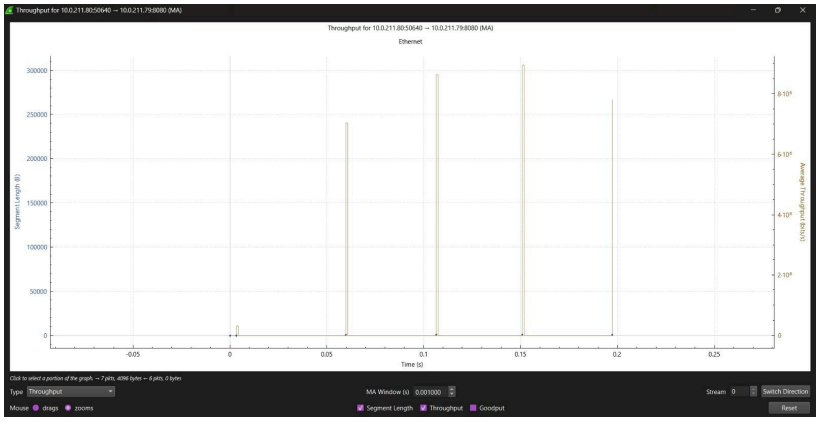
Nagle’s Algorithm	Delayed ACK
Enabled	Enabled

Packet Summary:

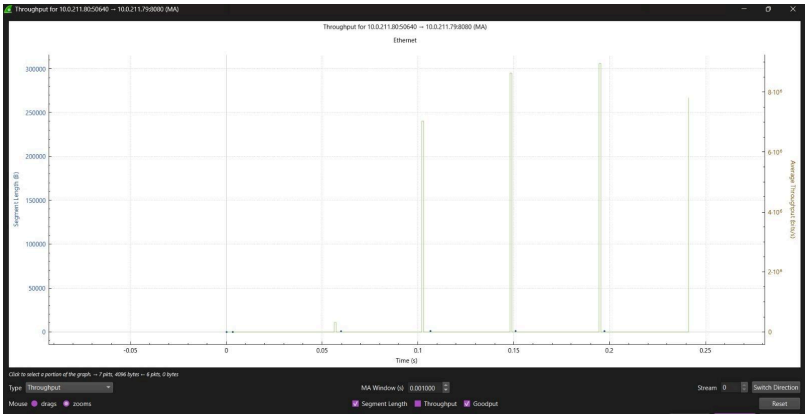


Packet Summary (Max packet size = 1174 B)

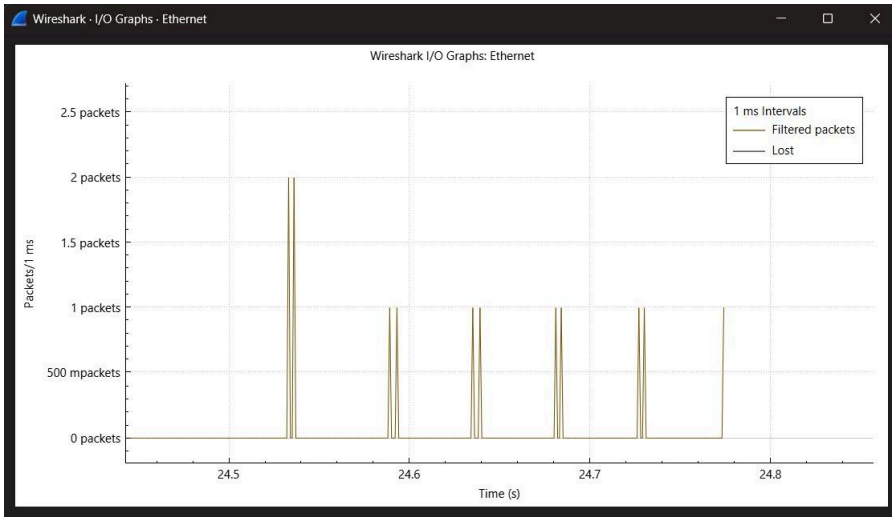
Throughput:



Goodput:



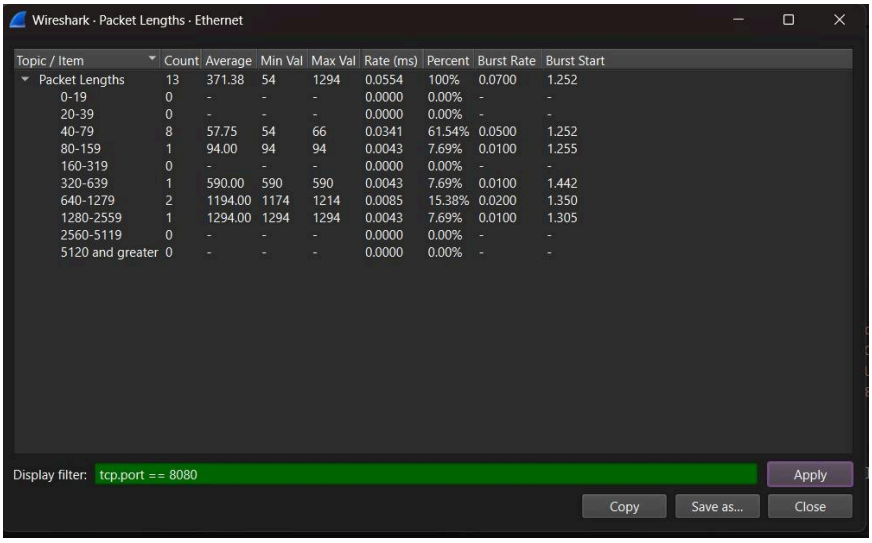
Packet Loss:



Test 2:

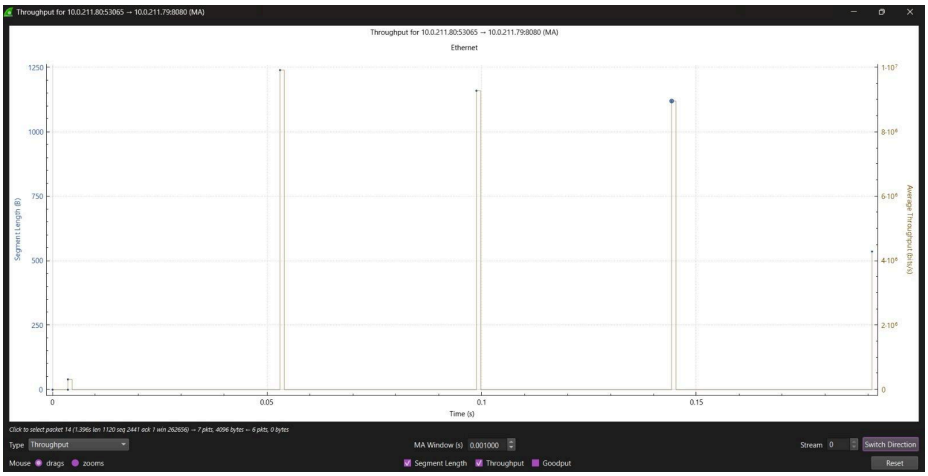
Nagle’s Algorithm	Delayed ACK
Enabled	Disabled

Packet Summary:

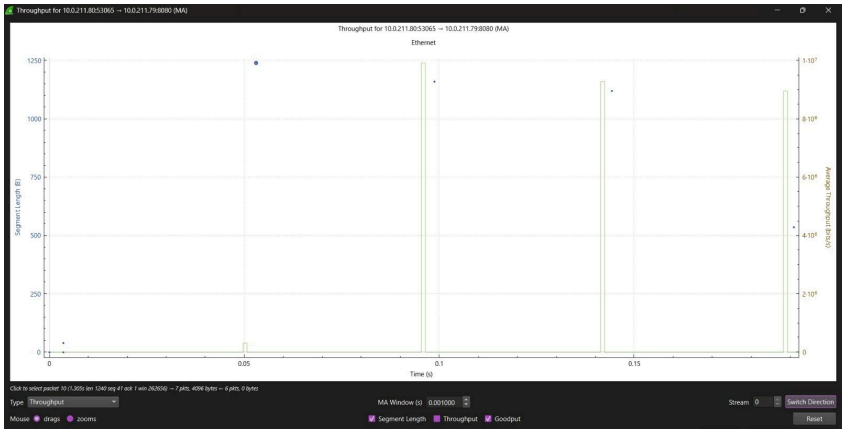


Packet Summary (Max packet size = 1294 B)

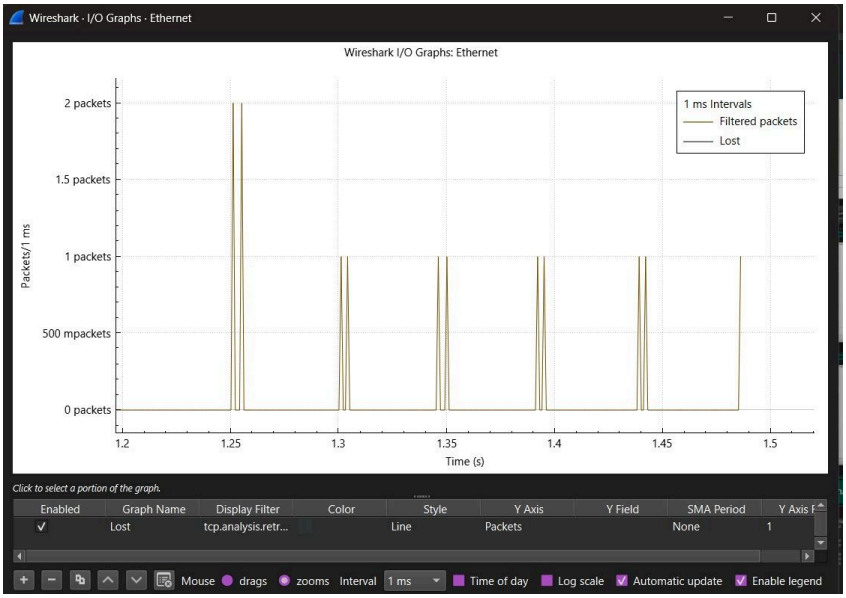
Throughput:



Goodput:



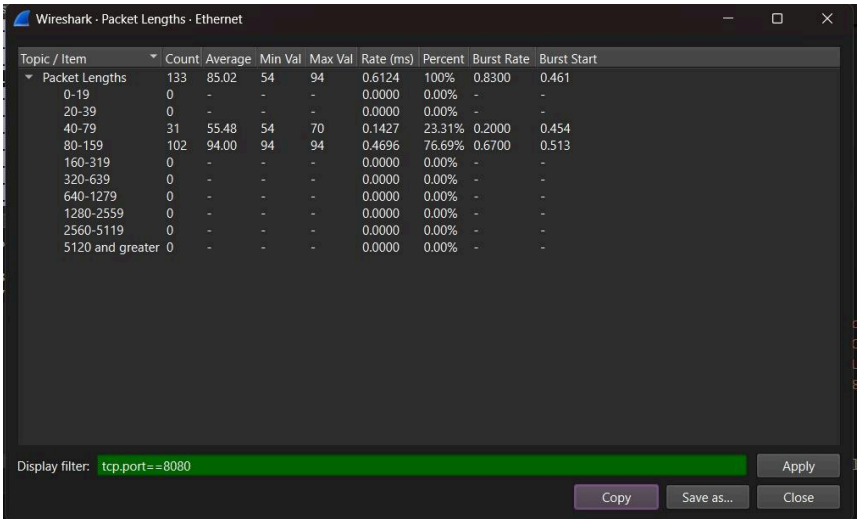
Packet Loss:



Test 3:

Nagle's Algorithm	Delayed ACK
Disabled	Enabled

Packet Summary:

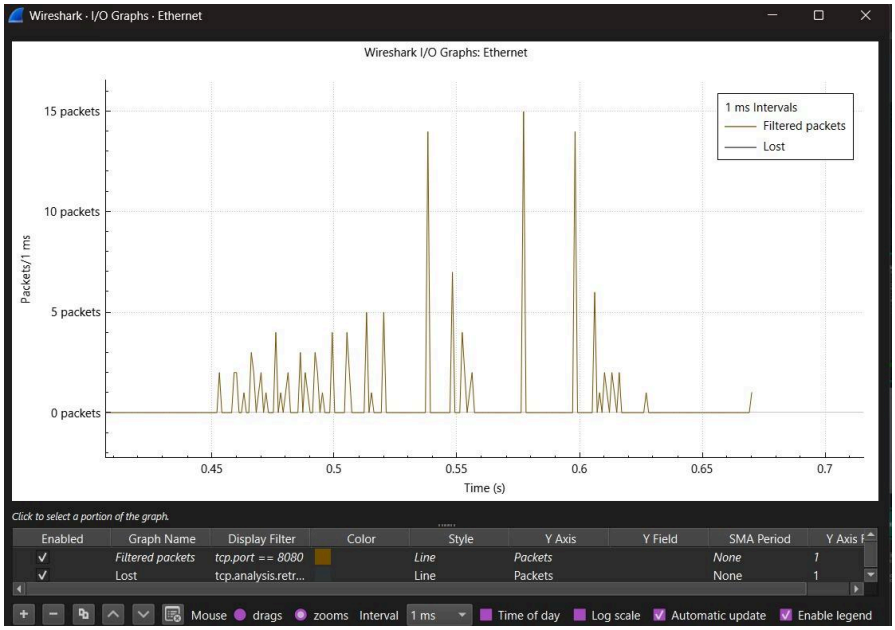


Packet Summary (Max packet size = 94 B)

Throughput:

Goodput:

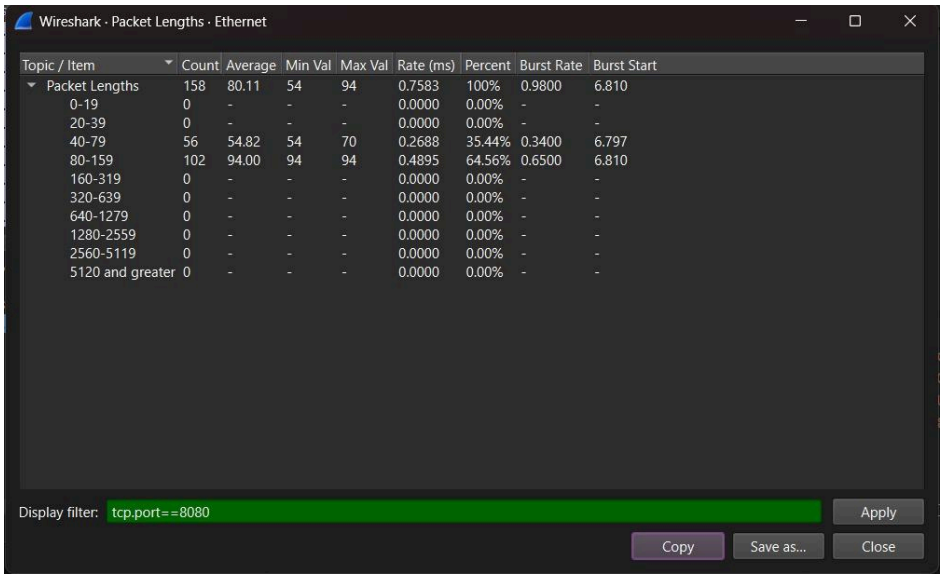
Packet Loss:



Test 4:

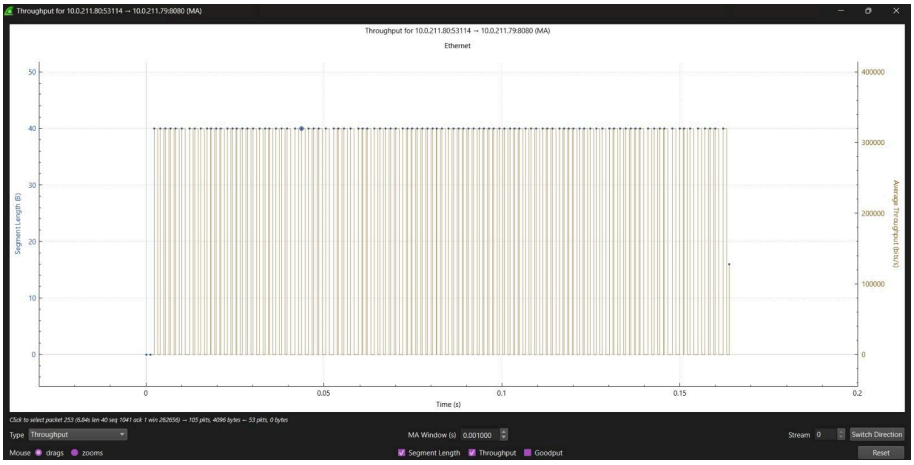
Nagle's Algorithm	Delayed ACK
Disabled	Disabled

Packet Summary:

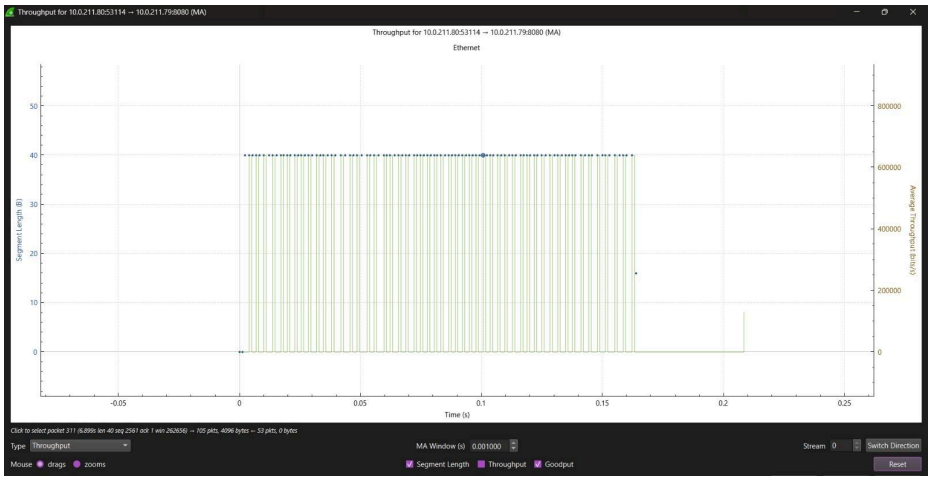


Packet Summary (Max packet size = 1174 B)

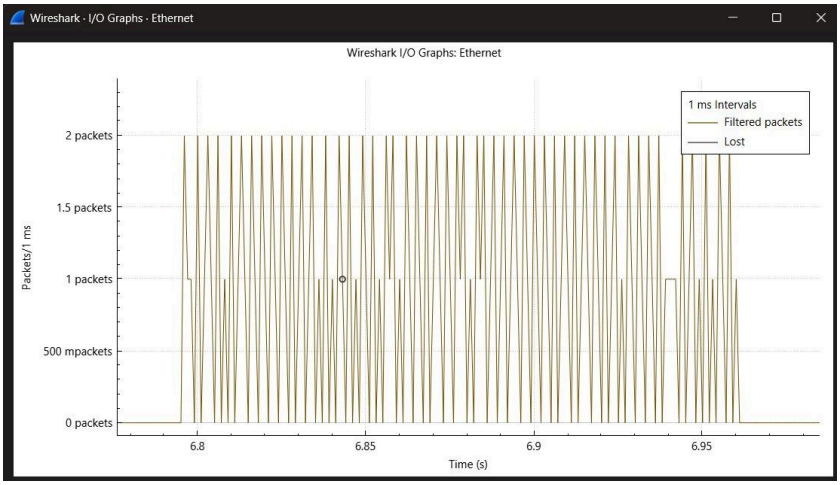
Throughput:



Goodput:



Packet Loss:



Observations

The four configurations, each with different combinations of Nagle's Algorithm and Delayed ACK settings, showed unique behaviors in terms of throughput, latency, packet overhead, and efficiency. When both Nagle's Algorithm and Delayed ACK were turned on, the network achieved higher throughput with fewer packets sent, but this came at the expense of increased latency, particularly for small messages. This setup worked well for bulk data transfers, where batching packets was advantageous. When Nagle's Algorithm was turned off but Delayed ACK stayed on, latency improved a bit, though smaller transmissions still faced some delays due to the receiver's delayed acknowledgment mechanism. Despite this, overall transmission efficiency remained fairly high.

Turning off Delayed ACK while keeping Nagle's Algorithm enabled led to a moderate boost in responsiveness, as the receiver sent acknowledgments immediately, allowing the sender to continue transmitting sooner. However, since Nagle's Algorithm still batched small packets, latency remained noticeable for interactive applications. Finally, when both Nagle's Algorithm and Delayed ACK were disabled, the system became highly responsive with minimal latency, but this came at the cost of increased packet overhead. Small segments were sent rapidly, leading to higher packet rates and reduced throughput efficiency, especially in bandwidth-constrained scenarios.

Collaboration

The task two and task three has been done in collaboration with the Team-6 [Dewansh Singh Chandel and Pranjal Gaur] due to only two linux systems available with the both of the teams. Hence, the plots and images for these two tasks are the same, but the observations have been made individually by each group.

Repository

The complete source code and scripts for this assignment are available at the following github repository:
https://github.com/shubham-agrawal04/CN-Assignmnet-2