Computer Networks Assignment-2

10.03.2025

Birudugadda Srivibhav (22110050), Srivathsa Vamsi Chaturvedula (22110260)
Computer Science and Engineering
IIT Gandhinagar

Task-1

The congestion schemas are: Cubic, Vegas, H-TCP

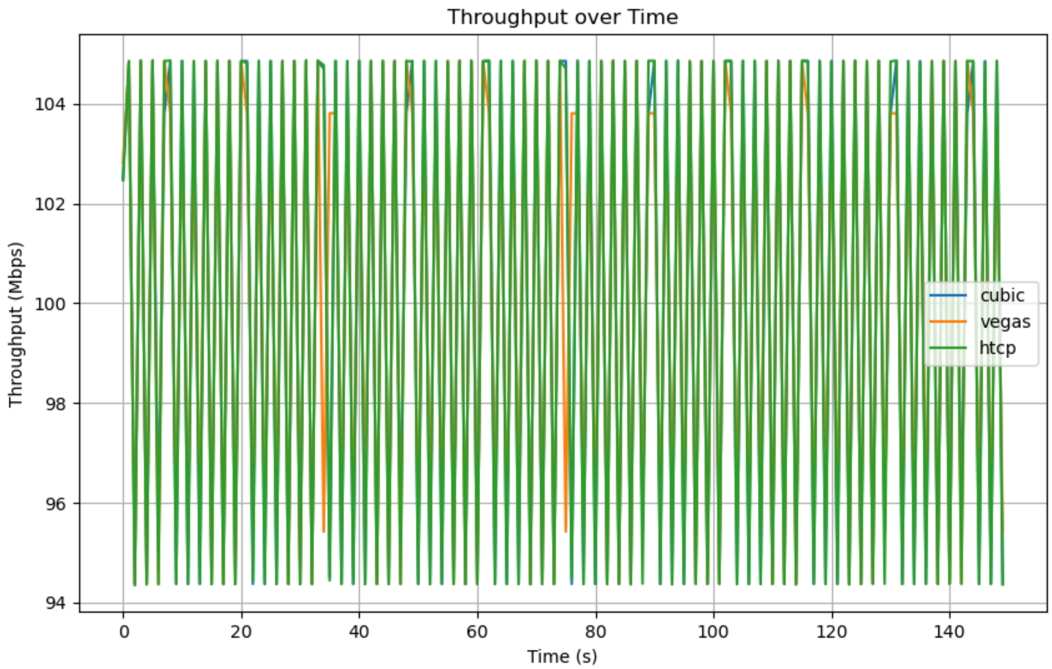
a) Run the client on H1 and the server on H7. Measure the below parameters and summarize the observations for the three congestion schemes as applicable.

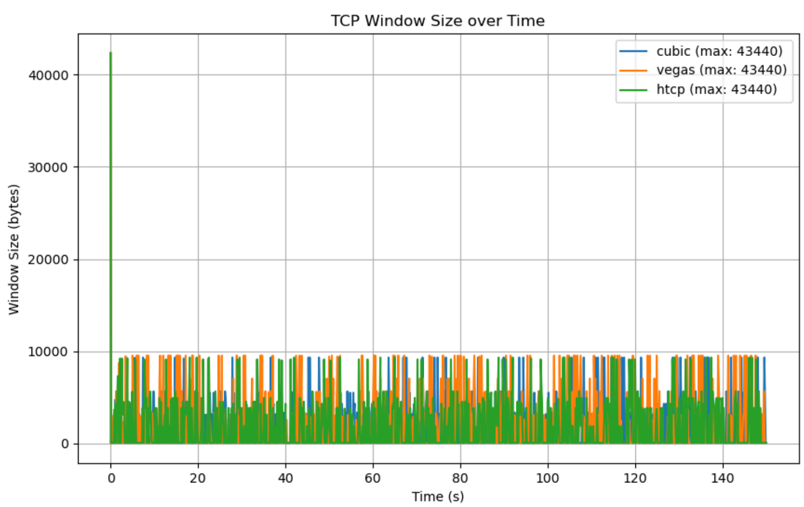
1. Throughput over time (with valid Wireshark I/O graphs)
2. Goodput
3. Packet loss rate
4. Maximum window size achieved (with valid Wireshark I/O graphs).

Code :

```python
def experiment_a(net):
    """Run experiment A: H1 -> H7 with different congestion control algorithms"""
    info('*** Running Experiment A\n')

    h1, h7 = net.get('h1', 'h7')
    server_ip = h7.IP()

    for algo in CONGESTION_ALGOS:
        info(f'*** Starting experiment with {algo}\n')

        # Ensure output directory exists
        os.makedirs('results/experiment_a', exist_ok=True)

        # Start capture
        pcap_file = f'results/experiment_a/h1_h7_{algo}.pcap'
        capture_pid = start_capture(net, h7, pcap_file)

        # Start server
        run_server(h7)

        # Run client
        output_file = run_client(h1, server_ip, cong_ctrl=algo)

        # Stop capture
        stop_capture(h7, capture_pid)

        # Wait for tcpdump to finish writing to the file
        time.sleep(2)

        # Move output file to results directory
        os.system(f'mv {output_file} results/experiment_a/')

        # Clean up
        h7.cmd('pkill -9 iperf3')
        time.sleep(2)  # Wait for cleanup
```

This code defines a function called experiment_a that executes a single-flow network congestion experiment. The experiment involves one client (H1) connecting to a server (H7). For each congestion control algorithm in the CONGESTION_ALGOS variable, the function creates a results directory, starts packet capture on H7, launches an iperf3 server on H7, and then runs a client on H1 to generate network traffic. After the client finishes, the function stops the packet capture, waits briefly for data to be written, moves the output file to the results directory, and cleans up by terminating any remaining iperf3 processes.





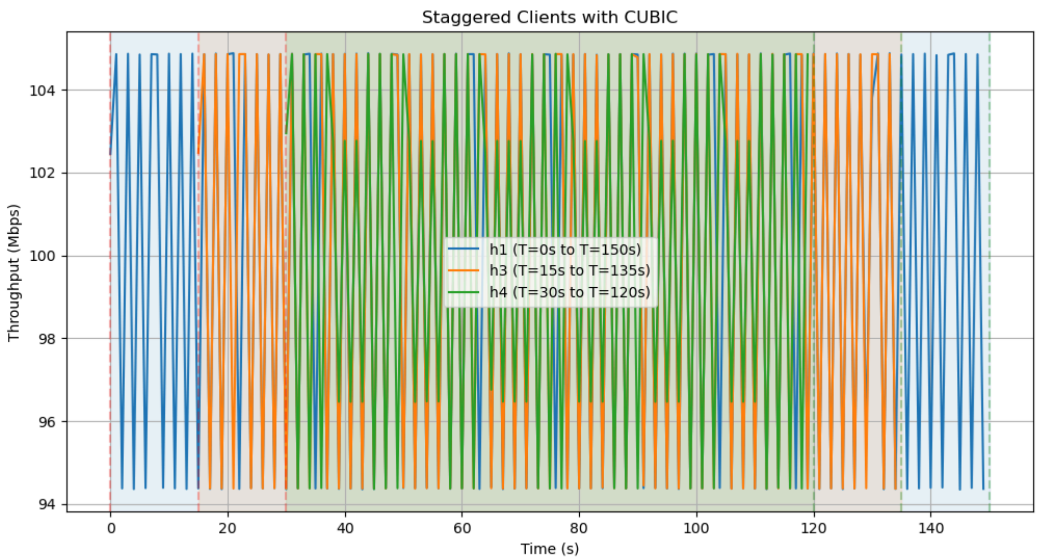
```
Summary of Results:
--------------------------------------------------------------------------------
Algorithm  Goodput (Mbps)  Packet Loss (%) Max Window Size Retransmits
--------------------------------------------------------------------------------
cubic      100.02          0.00            43440           43
vegas      100.02          0.00            43440           56
htcp       100.02          0.00            43440           69
--------------------------------------------------------------------------------
```

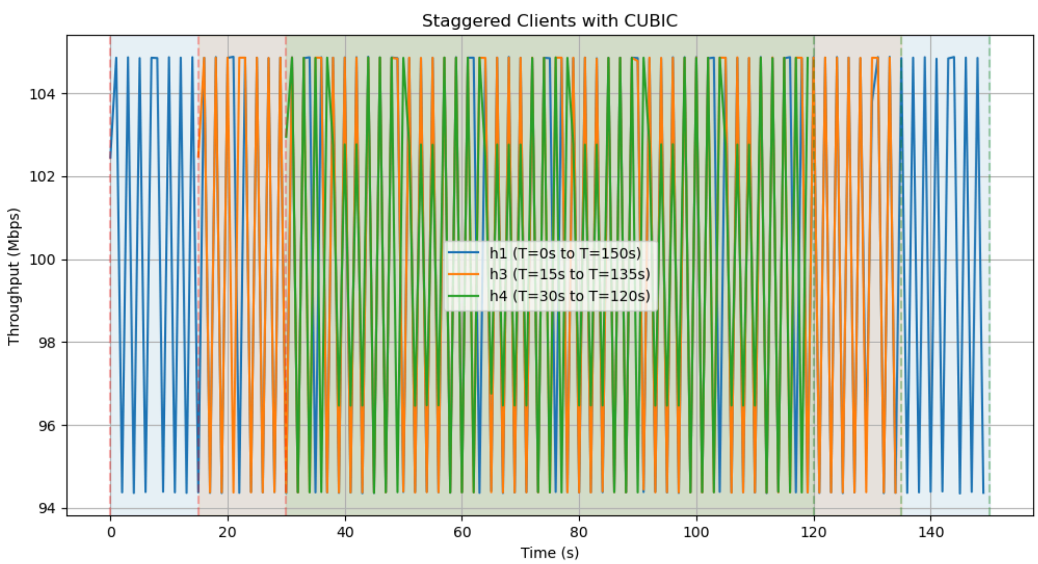
b)Run the clients on H1, H3 and H4 in a staggered manner(H1 starts at T=0s and runs for 150s, H3 at T=15s and runs for T=120s, H4 at T=30s and runs for 90s) and the server on H7. Measure the parameters listed in part (a) and explain the observations, for the 3 congestion schemes for all the three flows.

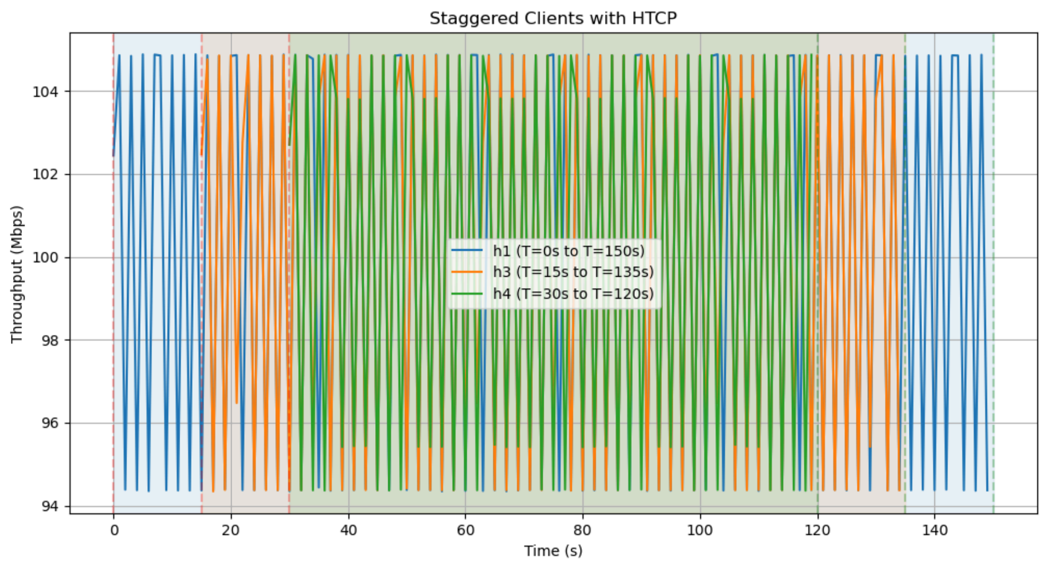
Code used

```python
def experiment_b(net):
    """Run experiment B: Staggered clients H1, H3, H4 -> H7"""
    info('*** Running Experiment B\n')

    h1, h3, h4, h7 = net.get('h1', 'h3', 'h4', 'h7')
    server_ip = h7.IP()

    for algo in CONGESTION_ALGOS:
        info(f'*** Starting experiment with {algo}\n')

        # Ensure output directory exists
        os.makedirs('results/experiment_b', exist_ok=True)

        # Start capture
        pcap_file = f'results/experiment_b/staggered_{algo}.pcap'
        capture_pid = start_capture(net, h7, pcap_file)

        # Start multiple server instances on different ports
        run_server(h7, port=5201)  # For H1
        run_server(h7, port=5202)  # For H3
        run_server(h7, port=5203)  # For H4

        # Start H1 client at T=0s for 150s
        h1.cmd(f'iperf3 -c {server_ip} -p 5201 -b 10M -P 10 -t 150 -C {algo} -J > results/experiment_b/h1_staggered_{algo}.json &')

        # Start H3 client at T=15s for 120s
        time.sleep(15)
        h3.cmd(f'iperf3 -c {server_ip} -p 5202 -b 10M -P 10 -t 120 -C {algo} -J > results/experiment_b/h3_staggered_{algo}.json &')

        # Start H4 client at T=30s for 90s
        time.sleep(15)
        h4.cmd(f'iperf3 -c {server_ip} -p 5203 -b 10M -P 10 -t 90 -C {algo} -J > results/experiment_b/h4_staggered_{algo}.json &')

        # Wait for all experiments to finish
        time.sleep(120)  # Total wait = 15 + 15 + 120 = 150s

        # Stop capture
        stop_capture(h7, capture_pid)

        # Clean up all iperf3 instances
        h7.cmd('pkill -9 iperf3')
        time.sleep(2)  # Wait for cleanup
```

This code defines a function called `experiment_b` that executes a network congestion experiment with staggered client connections. The experiment involves three clients (H1, H3, and H4) connecting to a server (H7) at different time intervals. H1 starts first and runs for 150 seconds, then H3 joins 15 seconds later and runs for 120 seconds, and finally H4 joins 30 seconds after the start and runs for 90 seconds. Each client uses iperf3 to generate network traffic at 10 Mbps with 10 parallel connections. The experiment captures packet data in a pcap file and saves performance results in JSON format for each client. The experiment is repeated for different congestion control algorithms defined in the CONGESTION_ALGOS variable.








```
Staggered Clients Experiment Summary:
--------------------------------------------------------------------------------
Algorithm  Client Goodput (Mbps)  Retransmits  Packet Loss (%)
--------------------------------------------------------------------------------
cubic      h1     100.02          30           0.00
cubic      h3     100.03          50           0.00
cubic      h4     100.06          27           0.00
vegas      h1     100.02          42           0.00
vegas      h3     100.03          29           0.00
vegas      h4     100.05          59           0.00
htcp       h1     100.02          54           0.00
htcp       h3     100.03          72           0.00
htcp       h4     100.06          23           0.00
--------------------------------------------------------------------------------
```

Task-2

In this section, we implemented a simple client-server architecture where the client sends the server a message, *"This is a TCP packet,"* every second. This is considered regular or legitimate traffic. We conducted the experiment on two different Ubuntu Linux versions (20.04 and 24.04) and ensured that both Wi-Fi and Bluetooth were disabled during the data transfer.

Experimental Setup

- **Client (Ubuntu 20.04)**: Sends TCP packets.
- **Server (Ubuntu 24.04)**: Receives TCP packets.

Implementation of SYN flood attack

To optimize our SYN attack experiment, we configured the server with the following network parameters:

- **`net.ipv4.tcp_max_syn_backlog`** – set to **1024**, which is relatively low and favors the success of a SYN flood attack.
- **`net.ipv4.tcp_syncookies`** – Disabled (**set to 0**) to prevent the system from mitigating the attack using SYN cookies.
- **`net.ipv4.tcp_synack_retries`** – Reduced to **2** to limit the number of SYN-ACK retries, making it easier to exhaust server resources.

```
(cn) birud_ubuntu_24.04@chinnu:~/CN/Assignment2$ python server_side.py
net.ipv4.tcp_max_syn_backlog = 1024
net.ipv4.tcp_syncookies = 0
net.ipv4.tcp_synack_retries = 2
Server listening on 0.0.0.0:8080
```

On the client side, we implemented a simple mechanism to send TCP packets every second, which is considered legitimate traffic.

```python
def send_legitimate_traffic(self):
    while True:
        client = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
        client.connect((self.host, self.port))

        message = "This is a TCP packet"
        client.sendall(message.encode())
        print(f"Sent message: {message}")

        try:
            response = client.recv(1024).decode()
            print(f"Received from server: {response}")
        except socket.timeout:
            print("No response received")

        client.close()
        time.sleep(1)
```

```
(cn2) birud_ubuntu_20.04@chinnu:~/CN/Assignment2$ python client_side.py
Sent message: This is a TCP packet
Received from server: This is a TCP packet
Sent message: This is a TCP packet
Received from server: This is a TCP packet
Sent message: This is a TCP packet
Received from server: This is a TCP packet
```

Before starting the client, we also initiated traffic capture on the client side using the ***tcpdump*** command.

```
(base) birud_ubuntu_20.04@chinnu:~/CN/Assignment2$ sudo tcpdump -i lo -n host 172.23.198.251 -s 0 -w client_traffic.pcap
tcpdump: listening on lo, link-type EN10MB (Ethernet), capture size 262144 bytes
```

We waited for 20 seconds before initiating the SYN attack to capture legitimate traffic. After this period, we launched the SYN attack using the `hping3` command.

```
(base) birud_ubuntu_20.04@chinnu:~$ sudo timeout 100 hping3 -S --rand-source -p 8080 -c 1000000000 -i u10000 1
72.23.198.251
HPING 172.23.198.251 (eth0 172.23.198.251): S set, 40 headers + 0 data bytes
len=44 ip=172.23.198.251 ttl=64 DF id=0 sport=8080 flags=SA seq=1 win=64240 rtt=9.5 ms
len=44 ip=172.23.198.251 ttl=64 DF id=0 sport=8080 flags=SA seq=3 win=64240 rtt=9.2 ms
len=44 ip=172.23.198.251 ttl=64 DF id=0 sport=8080 flags=SA seq=5 win=64240 rtt=8.8 ms
len=44 ip=172.23.198.251 ttl=64 DF id=0 sport=8080 flags=SA seq=7 win=64240 rtt=1009.4 ms
len=44 ip=172.23.198.251 ttl=64 DF id=0 sport=8080 flags=SA seq=9 win=64240 rtt=8.4 ms
len=44 ip=172.23.198.251 ttl=64 DF id=0 sport=8080 flags=SA seq=11 win=64240 rtt=7.8 ms
len=44 ip=172.23.198.251 ttl=64 DF id=0 sport=8080 flags=SA seq=13 win=64240 rtt=7.0 ms
len=44 ip=172.23.198.251 ttl=64 DF id=0 sport=8080 flags=SA seq=15 win=64240 rtt=6.2 ms
len=44 ip=172.23.198.251 ttl=64 DF id=0 sport=8080 flags=SA seq=17 win=64240 rtt=5.7 ms
len=44 ip=172.23.198.251 ttl=64 DF id=0 sport=8080 flags=SA seq=19 win=64240 rtt=4.8 ms
len=44 ip=172.23.198.251 ttl=64 DF id=0 sport=8080 flags=SA seq=21 win=64240 rtt=4.4 ms
len=44 ip=172.23.198.251 ttl=64 DF id=0 sport=8080 flags=SA seq=23 win=64240 rtt=4.1 ms
```

Breakdown of the command:

- `timeout 100`: Ensures that the command runs for 100 seconds before terminating automatically.
- `-S`: Sends SYN packets, initiating a TCP handshake but never completing it, simulating a SYN flood attack.
- `--rand-source`: Spoofs the source IP addresses, making it harder for the target server to identify and block the attack.
- `-p 8080`: Specifies port `8080` as the target port, where the attack packets will be sent.
- `-c 1000000000`: Sets the maximum number of packets to send, though the `timeout` command will likely terminate it before reaching this limit.
- `-i u10000`: Controls the packet sending rate, where `u10000` specifies an interval of 10,000 microseconds (or 10 milliseconds) between packets.
- `172.23.198.251`: The target IP address of the server under attack.

This command continuously sends spoofed SYN packets to the server's port 8080 for 100 seconds, aiming to overwhelm the server's connection backlog and disrupt legitimate traffic.

After running for 100 seconds, the SYN attack is automatically stopped due to the `timeout 100` parameter in the command. This ensures that the attack does not run indefinitely and allows us to observe the server's response once the attack ceases.

```
len=44 ip=172.23.198.251 ttl=64 DF id=0 sport=8080 flags=SA seq=9357 win=64240 rtt=3.8 ms
len=44 ip=172.23.198.251 ttl=64 DF id=0 sport=8080 flags=SA seq=9361 win=64240 rtt=3.1 ms
len=44 ip=172.23.198.251 ttl=64 DF id=0 sport=8080 flags=SA seq=9548 win=64240 rtt=1010.0 ms
len=44 ip=172.23.198.251 ttl=64 DF id=0 sport=8080 flags=SA seq=9564 win=64240 rtt=7.5 ms
len=44 ip=172.23.198.251 ttl=64 DF id=0 sport=8080 flags=SA seq=9672 win=64240 rtt=8.4 ms

--- 172.23.198.251 hping statistic ---
9769 packets transmitted, 889 packets received, 91% packet loss
round-trip min/avg/max = 0.1/18.9/1010.0 ms
```

After running the legitimate traffic for an additional 20 seconds post-attack, we stop capturing network traffic.

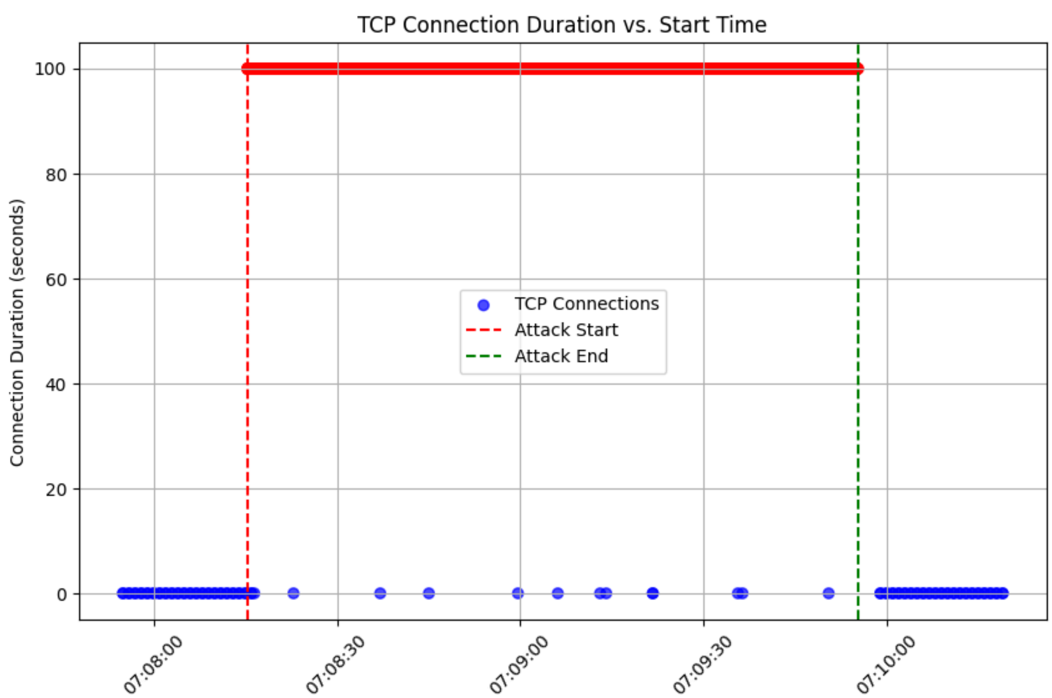
```
(base) birud_ubuntu_20.04@chinnu:~/CN/Assignment2$ sudo tcpdump -i lo -n host 172.23.198.251 -s 0 -w client_tr
affic.pcap
tcpdump: listening on lo, link-type EN10MB (Ethernet), capture size 262144 bytes
^C10243 packets captured
20486 packets received by filter
0 packets dropped by kernel
```

Now we plot our results.

To analyze the impact of the SYN attack, we processed the output PCAP file to calculate the connection duration for each recorded TCP connection. The connection duration was determined as the time difference between the first SYN packet and either the ACK following a FIN-ACK or the first RESET packet. If a connection did not properly terminate, a default duration of 100 seconds was assigned.

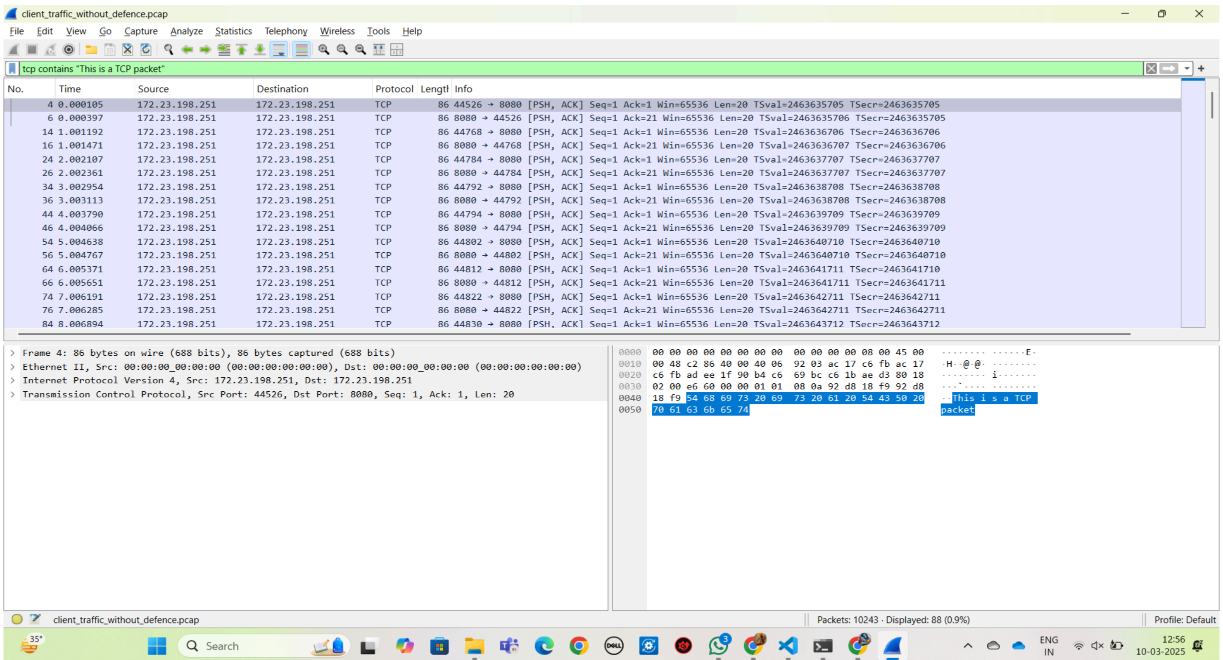
Using the extracted connection start times and durations, we plotted **Connection Duration vs. Connection Start Time** to visualize the effect of the attack. Each connection was represented as a data point, with **legitimate traffic marked in blue** and **SYN flood attack connections (incomplete connections) marked in red**.

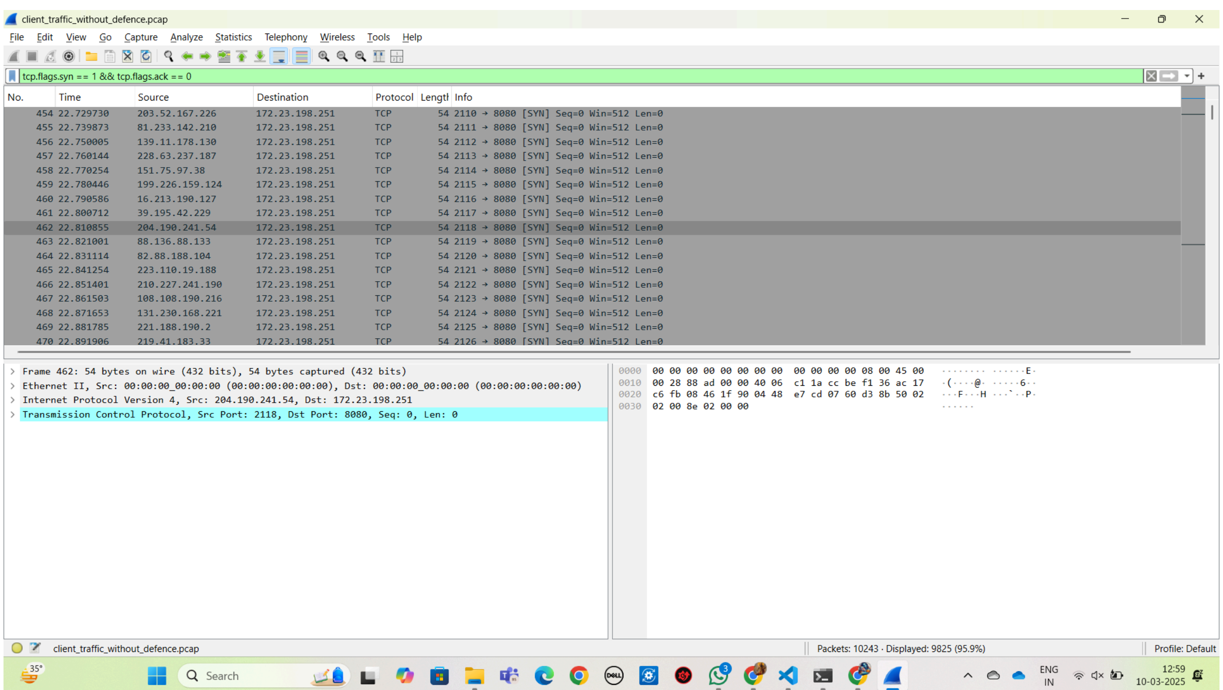
Additionally, we marked the **start and end of the attack** with vertical dashed lines to highlight its impact. The plot helps in identifying anomalies caused by the SYN flood, such as prolonged or incomplete connections. Wireshark was also used to verify the correctness of the extracted connection details, and screenshots of the same are attached below.



The graph clearly illustrates that during the SYN flood attack, although legitimate clients continue to send requests, most of these requests fail to establish a connection. As a result, the volume of successfully established legitimate traffic is significantly lower compared to periods without the attack.

Validation Using Wireshark:





Since our legitimate traffic contains the message "***This is a TCP packet,***" we use Wireshark to search for it using the query: ***tcp contains "This is a TCP packet"***.

Now, we use the query `tcp.flags.syn == 1 && tcp.flags.ack == 0` to filter and identify the SYN flood. In the screenshot below, it is clearly visible that out of **10,243 packets**, **9,825 (95.9%)** are SYN packets without proper closure. This indicates that a large number of connections remain incomplete, confirming the successful execution of the SYN flood attack.

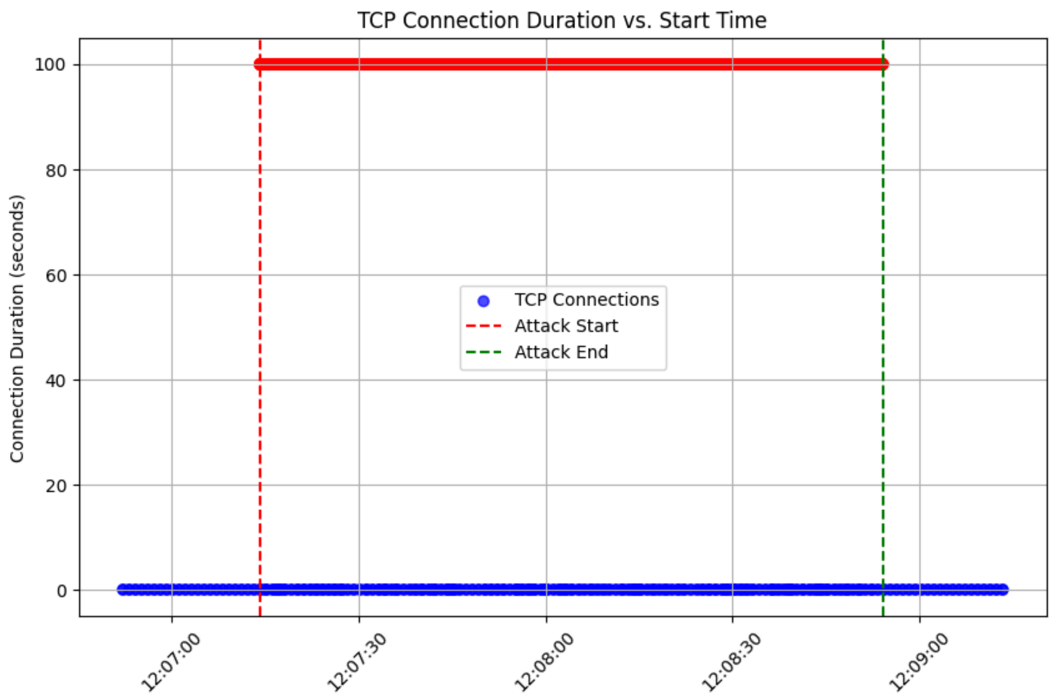
Mitigation of SYN flood attack

To mitigate the impact of the SYN flood attack, we adjusted key TCP parameters on the server to enhance its ability to handle excessive SYN requests efficiently. The following system configurations were applied using the ***sysctl*** command:

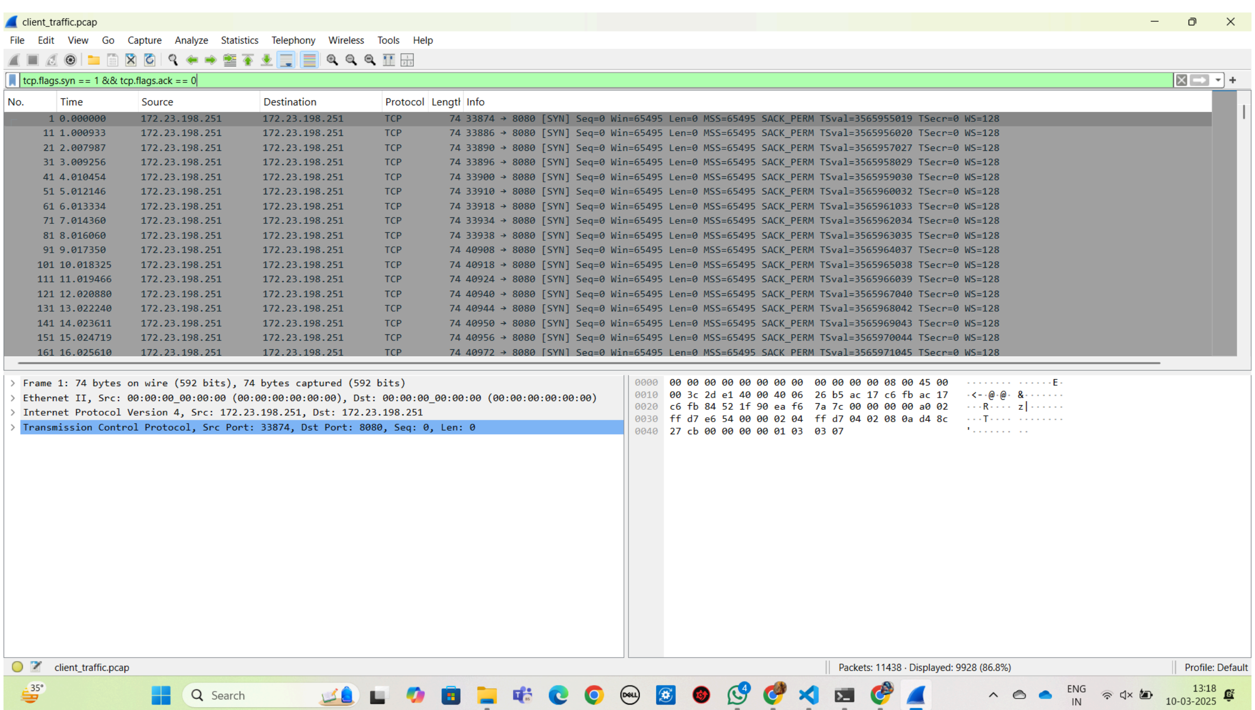
1) **Enable SYN Cookies** (`net.ipv4.tcp_syncookies=1`):
    a) SYN cookies were enabled to prevent resource allocation for half-open connections until the handshake was completed.
    b) This technique ensures that malicious SYN requests do not consume server resources unnecessarily.
2) **Increase Backlog Queue Size** (`net.ipv4.tcp_max_syn_backlog=2048`):
    a) The backlog queue size was increased from 1024 to 2048 to accommodate more simultaneous connection requests during high traffic or an attack.
    b) This helps ensure that legitimate traffic is not blocked when under attack.

```
(cn) birud_ubuntu_24.04@chinnu:~/CN/Assignment2$ python server_side.py
net.ipv4.tcp_max_syn_backlog = 2048
net.ipv4.tcp_syncookies = 1
net.ipv4.tcp_synack_retries = 2
Server listening on 0.0.0.0:8080
```

After setting the parameters, we repeated the same experiments. This time, even during the SYN flood attack, the legitimate traffic was successfully captured and was not lost, as clearly demonstrated in the graph below. This result indicates that the adjustments made to the system have improved its ability to differentiate between attack traffic and legitimate traffic, ensuring the latter is preserved despite the ongoing flood.



Validation Using Wireshark:



In the screenshot above, it is clearly visible that out of **11,438 packets**, **9,928 (86.8%)** are SYN packets without proper closure. This indicates that a large number of connections remain incomplete, confirming the successful execution of the SYN flood attack. Even though the SYN flood occurred, the server was now able to distinguish between the flood and the real traffic.

In this experiment, we successfully mitigated a SYN flood attack by implementing key kernel-level strategies such as: enabling SYN cookies (net.ipv4.tcp_syncookies=1) and increasing the backlog queue size (net.ipv4.tcp_max_syn_backlog=2048). These measures were sufficient to distinguish legitimate traffic from malicious SYN requests, as evidenced by the plotted graph showing normal connection durations for legitimate traffic even during the attack period. While these methods effectively prevented resource exhaustion and ensured uninterrupted service, additional techniques such as rate limiting using iptables, deploying intrusion detection systems (IDS), or using load balancers can further enhance protection against SYN flood attacks in high-security environments.

Task-3

Nagle's algorithm is a technique used in TCP to improve network efficiency by reducing the number of small packets sent over the network. It works by buffering data until either a full-sized packet can be sent or an acknowledgment (ACK) is received from the other end. This reduces network overhead and improves throughput, especially for applications that send small amounts of data frequently.

For this task, we have written the following code to implement Nagle's algorithm:

1. nagle.py

This script checks for dependencies, ensures all necessary scripts have execute permissions, creates a results directory, and runs the experiment using experiment_runner.py.

Key Functions:

- check_dependencies(): Verifies if Mininet and required files are present.
- ensure_permissions(): Sets execute permissions on scripts.
- create_results_dir(): Creates a directory for storing experiment results.
- run_experiment(): Executes the experiment using experiment_runner.py.

2. client.py

This script simulates a TCP client that sends data at a specified transfer rate. It configures Nagle's algorithm and delayed ACK settings based on input parameters.

Key Features:

- Configures socket options for Nagle's algorithm (TCP_NODELAY) and delayed ACK (TCP_QUICKACK).
- Sends data in chunks, simulating a transfer at a specified rate.
- Calculates and prints metrics like throughput, goodput, and packet loss rate.

3. server.py

This script sets up a TCP server that listens for incoming connections. It also configures Nagle's algorithm and delayed ACK settings.

Key Features:

- Configures socket options similar to the client.
- Accepts connections and receives data from the client.
- Calculates and prints metrics like throughput and maximum packet size.

4. experiment_runner.py

This script runs experiments using Mininet to test the performance of Nagle's algorithm and delayed ACK under different configurations.

Key Features:

- Creates a simple network topology with one switch and two hosts using Mininet.
- Runs the client and server scripts with different Nagle and delayed ACK settings.
- Parses client output to extract metrics and saves results to a JSON file.
- Generates an analysis report comparing the performance of different configurations.

Methodology:

The objective of this experiment was to analyze the impact of Nagle's algorithm and delayed ACK on TCP performance using Mininet. The experiment aimed to measure throughput, goodput, and packet loss rates under different configurations.

1. Setup:
   - Mininet was used to create a simple network topology with one switch and two hosts.
   - The client and server scripts were configured to test four combinations of Nagle's algorithm and delayed ACK settings.
2. Configurations Tested:
   - Nagle on, Delayed ACK on
   - Nagle on, Delayed ACK off
   - Nagle off, Delayed ACK on
   - Nagle off, Delayed ACK off
3. Metrics Collected:
   - Throughput
   - Goodput
   - Packet Loss Rate
   - Average Packet Size

Results:

The following are the results, which can also be found in the ***Part-3/tcp_results*** folder.

Configuration	Throughput (B/s)	Goodput (B/s)	Packet Loss Rate	Avg Packet Size (B)
nagle_on_delack_on	38.93	39.13	0.00%	39.80
nagle_on_delack_off	38.93	39.13	0.00%	39.80
nagle_off_delack_on	38.92	39.12	0.00%	39.80
nagle_off_delack_off	38.92	39.12	0.00%	39.80

Analysis and Observations:

1. Effect of Nagle's Algorithm:

- Average throughput with Nagle on: 38.93 B/s
- Average throughput with Nagle off: 38.92 B/s
- Nagle's algorithm increases throughput by 0.01 B/s (0.03%)

2. Effect of Delayed ACK:

- Average throughput with Delayed ACK on: 38.92 B/s
- Average throughput with Delayed ACK off: 38.92 B/s
- Delayed ACK decreases throughput by 0.00 B/s (0.00%)

3. Best Configuration:

- nagle_on_delack_on provides the highest goodput at 39.13 B/s

4. Explanation of Observations:

- Nagle's Algorithm aims to reduce the number of small packets by buffering data until either a full-sized packet can be sent or an ACK is received.
- Delayed ACK reduces the number of ACKs by delaying them, which can cause Nagle's algorithm to wait unnecessarily.
- When both are enabled, they can create a 'lock-step' behavior where each is waiting for the other.
- Disabling both typically gives the best interactive performance but may increase network overhead.

5. Recommendations:

- For bulk transfers: Nagle on, Delayed ACK on - Reduces overhead, maximizes efficiency
- For interactive applications: Nagle off, Delayed ACK off - Minimizes latency
- For mixed workloads: Nagle off, Delayed ACK on - Good compromise