Computer Network Assignment-4

Ayush Kumar Mishra

November 18, 2024

November 18, 2024

Introduction

Classification of Algorithms

- **Loss-based**: `TCP CUBIC`, `TCP NewReno`, `TCP BIC`
- **Delay-based**: `TCP Vegas`, `TCP FAST`
- **Hybrid**: `TCP Veno`, `TCP BBR`
- **ML-based**: `TCP Remy`, `TCP Aurora`

I am going to analyze the following algorithms in detail:

- **Loss-based**: `TCP CUBIC` and `TCP NewReno`
- **Delay-based**: `TCP Vegas`
- **Hybrid**: `TCP Veno`

TCP CUBIC (Loss-based)

TCP CUBIC Overview

Algorithm Details

TCP CUBIC uses a cubic growth function to handle congestion, optimizing for high-bandwidth delay product (BDP) networks.

Mathematical Model

The cubic growth function is:

$$W(t) = C(t - K)^3 + W_{max} \tag{1}$$

where:

- $W(t)$: Window size at time t
- C: Scaling factor (default 0.4)
- $K = \sqrt[3]{\frac{W_{max}\beta}{C}}$: Time offset, with β as the multiplicative decrease factor
- W_{max}: Window size just before last congestion

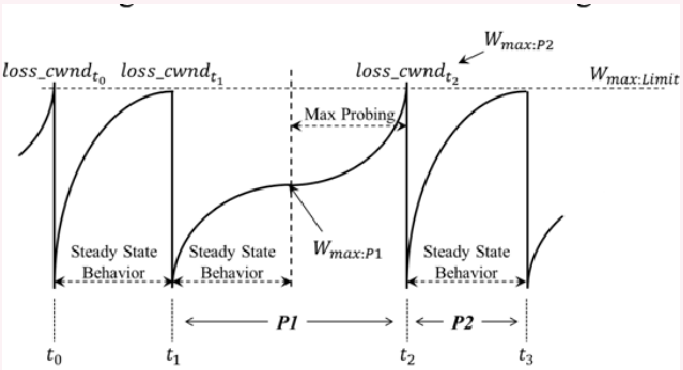


Figure 1. The curve of congestion window of Cubic

Growth Phases and State Transitions

i **Concave Region** ($t < K$): Slow initial growth for stability.

ii **Convex Region** ($t > K$): Aggressive growth for bandwidth probing.

iii **Plateau Region** ($t \approx K$): Stabilizes near W_{max}, ideal for steady-state.

TCP CUBIC Overview

Key Features and Limitations

- **Strengths**: High throughput in high-speed networks, good RTT fairness.
- **Limitations**: RTT fairness issues in mixed RTT networks, limited handling of random loss.

Application Scenarios

Ideal for long-distance high-speed networks, data centers, cloud computing, and large data transfers.

TCP Vegas Algorithm Details

Algorithm Implementation Details

(e).1 Core Mechanism

TCP Vegas implements a delay-based congestion control mechanism that operates through:

1. **RTT Monitoring**
   - Tracks BaseRTT (minimum observed RTT)
   - Measures CurrentRTT for each packet
   - Calculates expected and actual throughput
2. **Mathematical Foundation**

$$R_{expected} = \frac{WindowSize}{BaseRTT} \tag{2}$$

$$R_{actual} = \frac{WindowSize}{CurrentRTT} \tag{3}$$

$$diff = (R_{expected} - R_{actual}) \times BaseRTT \tag{4}$$

3. **Window Adjustment Logic**

$$WindowSize = \begin{cases} WindowSize + 1, & \text{if } diff < \alpha \\ WindowSize - 1, & \text{if } diff > \beta \\ WindowSize, & \text{if } \alpha \leq diff \leq \beta \end{cases} \tag{5}$$

(e).2 Implementation Features

- **Threshold Parameters**:
  - α: Lower threshold (typically 2-3 packets)
  - β: Upper threshold (typically 4-6 packets)
- **RTT Calculation**:

$$Queue_{estimated} = \left(1 - \frac{BaseRTT}{CurrentRTT}\right) \times WindowSize \tag{6}$$

- **Congestion Window Update**:

$$\Delta WindowSize = \begin{cases} +1/WindowSize & \text{if queue} < \alpha \\ -1/WindowSize & \text{if queue} > \beta \\ 0 & \text{otherwise} \end{cases}$$

Algorithm Phases

(e).3 Algorithm Phases

1. **Slow Start Phase**
   - Increases window exponentially until first congestion detection
   - Transitions to congestion avoidance when diff > threshold
2. **Congestion Avoidance Phase**
   - Maintains optimal window size based on RTT measurements
   - Adjusts window using the difference equation
3. **Retransmission Phase**
   - Retransmits lost packets detected through timeouts or duplicate acknowledgments.
   - Uses the estimated RTT for calculating timeouts more accurately than in traditional TCP variants.
4. **Timeout Phase**
   - Resets the congestion window to a small value (typically one or two segments) after a timeout event.
   - Enters the Slow Start Phase again, adapting quickly to network conditions.

TCP Vegas Application Scenarios and Limitations [7 points]

Deployment Scenarios Analysis

(f).1 Optimal Application Scenarios

1. **Data Center Networks**
   - *Characteristics*:
     - Low latency (< 1ms RTT)
     - Controlled environment
     - Predictable traffic patterns
   - *Benefits*:
     - Precise RTT measurements
     - Stable congestion window
     - Efficient queue management
2. **Enterprise Networks**
   - *Characteristics*:
     - Moderate BDP
     - Stable infrastructure
     - QoS requirements
   - *Benefits*:
     - Predictable performance
     - Fair bandwidth sharing
     - Low queuing delay
3. **Content Delivery Networks**
   - *Characteristics*:
     - Distributed servers
     - Multiple paths
     - Static content
   - *Benefits*:
     - Stable throughput
     - Reduced packet loss
     - Efficient bandwidth utilization

Limitation Scenarios

(f).2 Limitation Scenarios

1. **Wireless Networks**
   - *Challenges*:
     - Variable RTT due to channel conditions
     - Interference effects
     - Mobility impacts
   - *Performance Impact*:
     - False congestion detection
     - Unnecessary window reduction
     - Throughput degradation
2. **High BDP Networks**
   - *Challenges*:
     - Long propagation delays
     - Large bandwidth variations
     - Complex queuing dynamics
   - *Performance Impact*:
     - Underutilization of bandwidth
     - Slow convergence
     - Inefficient window adjustment
3. **Mixed Traffic Environments**
   - *Challenges*:
     - Competition with loss-based TCP
     - Buffer bloat effects
     - Unfair bandwidth sharing
   - *Performance Impact*:
     - Reduced throughput share
     - Unstable window size
     - Poor competition ability

TCP Veno (Hybrid)

TCP Veno Overview

Algorithm Details

TCP Veno combines the congestion detection of Vegas with the loss response of NewReno.

Mathematical Model

Backlogged packets (N) is calculated as:

$$N = (R_{expected} - R_{actual}) \times BaseRTT \tag{7}$$

State determination and window adjustment:

$$\text{State} = \begin{cases} \text{Non-congested}, & \text{if } N < \beta \\ \text{Congested}, & \text{if } N \geq \beta \end{cases}$$

Key Features and Limitations

- **Strengths**: Combines proactive delay-based control with loss-based response.
- **Limitations**: Compromises in both fairness and throughput.

Application Scenarios

Useful in wireless networks where packet loss can be frequent, as it combines robustness with fairness.

TCP NewReno (Loss-based)

TCP NewReno Overview

Algorithm Details

TCP NewReno refines the loss recovery phase of TCP Reno, particularly for fast recovery.

Mathematical Model

During Congestion Avoidance:

$$cwnd = \begin{cases} cwnd + \frac{MSS^2}{cwnd}, & \text{per RTT} \\ cwnd \times (1 - \beta), & \text{on packet loss} \end{cases} \tag{8}$$

where $cwnd$ is the congestion window, MSS is the maximum segment size, and $\beta \approx 0.5$.

Key Features and Limitations

- **Strengths**: Effective in environments with moderate packet loss, simple recovery mechanism.
- **Limitations**: Limited adaptation to varying network conditions, RTT unfairness.

Application Scenarios

Ideal for traditional, wired networks with predictable loss patterns and moderate RTT.

TCP Variant Comparison and Inferences [7 points]

Comparative Analysis of TCP Variants

(d).1 Performance Metrics Comparison

(d).2 Key Algorithmic Differences

1. **Congestion Detection**
   - *Vegas*: RTT-based, proactive
   - *Reno*: Loss-based, reactive
   - *NewReno*: Enhanced loss recovery
   - *CUBIC*: Hybrid window growth
2. **Window Growth Functions**

$$W_{Vegas} = \begin{cases} W + 1 & \text{if queue} < \alpha \\ W - 1 & \text{if queue} > \beta \end{cases} \tag{9}$$

$$W_{Reno} = \begin{cases} W + 1 & \text{per RTT} \\ W/2 & \text{on loss} \end{cases} \tag{10}$$

$$W_{CUBIC} = C(t - K)^3 + W_{max} \tag{11}$$

(d).3 Critical Inferences

1. **Congestion Response**
   - Vegas: Most conservative, stable
   - Reno: Aggressive, saw-tooth pattern
   - NewReno: Improved recovery
   - CUBIC: Optimized for high BDP
2. **Network Utilization**
   - Vegas: Moderate, consistent
   - Reno: Variable, loss-dependent
   - NewReno: Enhanced stability
   - CUBIC: Highest average throughput
3. **Fairness Characteristics**
   - Vegas: Best intra-protocol fairness
   - Reno: Moderate RTT fairness
   - NewReno: Improved fairness
   - CUBIC: RTT-independent fairness

(d).4 Performance Trade-offs

- **Stability vs. Aggressiveness**
  - Vegas: High stability, low aggressiveness
  - CUBIC: High aggressiveness, moderate stability
- **Fairness vs. Performance**
  - Vegas: High fairness, moderate performance
  - Reno/NewReno: Balanced approach
- **Complexity vs. Efficiency**
  - Vegas: Complex logic, efficient operation
  - Reno: Simple logic, moderate efficiency

Question-2 - Part A

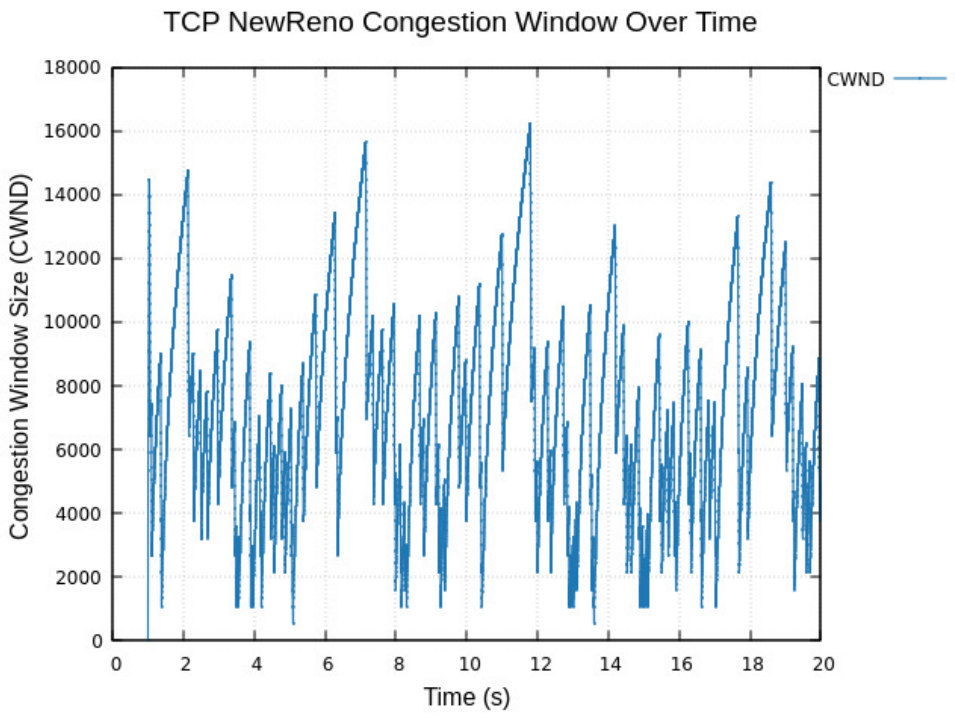
TCP NewReno (Loss-based)

Question-1

Q1. Plot the cwnd vs time graph, and describe what you observed, like slow start and congestion avoidance, in detail.

Solution:

1. First, I will plot the congestion window ($cwnd$) vs time graph for TCP NewReno. The graph will illustrate the slow start and congestion avoidance phases of the algorithm.



Observations:

- **Slow Start**: The initial phase shows an exponential growth in the congestion window, as the algorithm probes the network capacity.
- **Congestion Avoidance**: After reaching the congestion threshold, the window size grows linearly, reducing the risk of congestion.
- **Packet Loss**: On packet loss, the window size is reduced by a multiplicative factor, leading to a temporary decrease in throughput.
- **Stability**: The algorithm maintains a balance between aggressive probing and stability, adapting to network conditions.
- **RTT Unfairness**: NewReno may exhibit RTT unfairness in mixed RTT networks due to its loss-based approach.
- **Predictable Behavior**: The algorithm's response to packet loss is predictable and well-suited for wired networks.

Different phases of TCP NewReno's congestion control mechanism can be identified from the graph:

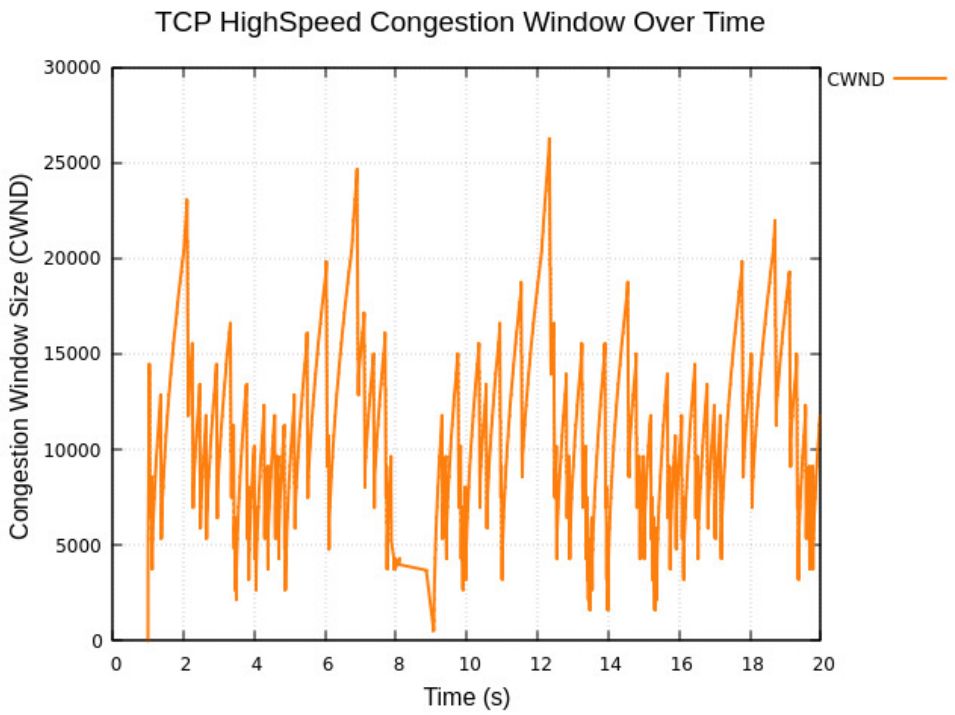
- Slow Start Phase:
  1. Rapid growth in $cwnd$ to probe network capacity as shown by frequent up and down.
  2. Until the congestion threshold is reached, the window size increases exponentially.

- Congestion Avoidance Phase:
  1. Linear growth in $cwnd$ after reaching the threshold, ensuring stability.
  2. The algorithm balances throughput and congestion avoidance during this phase.
- Packet Loss and Recovery:
  1. Multiplicative decrease in $cwnd$ on packet loss to prevent congestion.
  2. The recovery phase is efficient, allowing the algorithm to regain throughput quickly.
- Stability and Predictability:
  1. NewReno demonstrates stable behavior in traditional network environments.
  2. The algorithm's response to congestion is consistent and predictable.

In summary, TCP NewReno demonstrates a robust congestion control mechanism, balancing throughput and stability in traditional network environments.
Next, I will analyze the graph to identify the key phases of TCP NewReno's congestion control mechanism.

Phase	**Description**
Slow Start	Exponential growth in $cwnd$
Congestion Avoidance	Linear growth in $cwnd$
Packet Loss	Multiplicative decrease in $cwnd$
Stability	Balanced growth and loss recovery
RTT Unfairness	Potential issues in mixed RTT networks
Predictable Behavior	Consistent response to congestion

The graph and observations provide insights into TCP NewReno's congestion control behavior, highlighting its strengths and limitations in different network scenarios.
2. Now, I will analyze High-Speed TCP (HSTCP) :-



Observations:

- **Aggressive Growth**: HSTCP exhibits rapid growth in the congestion window during the slow start phase.
- **High Throughput**: The algorithm aims to maximize throughput in high-speed networks.
- **RTT Fairness**: HSTCP maintains fairness by adapting to RTT variations.
- **Stability**: The congestion window stabilizes after reaching the maximum threshold, ensuring network stability.
- **RTT Sensitivity**: HSTCP may exhibit sensitivity to RTT changes, affecting performance in mixed RTT environments.
- **Ideal Applications**: The algorithm is well-suited for high-speed networks, data centers, and cloud computing environments.

The graph illustrates the key characteristics of High-Speed TCP (HSTCP), emphasizing its aggressive growth, high throughput, and RTT fairness in high-speed network scenarios.
Different phases of HSTCP's congestion control mechanism can be identified from the graph:

- Slow Start Phase:
    1. Aggressive growth in $cwnd$ to maximize throughput.

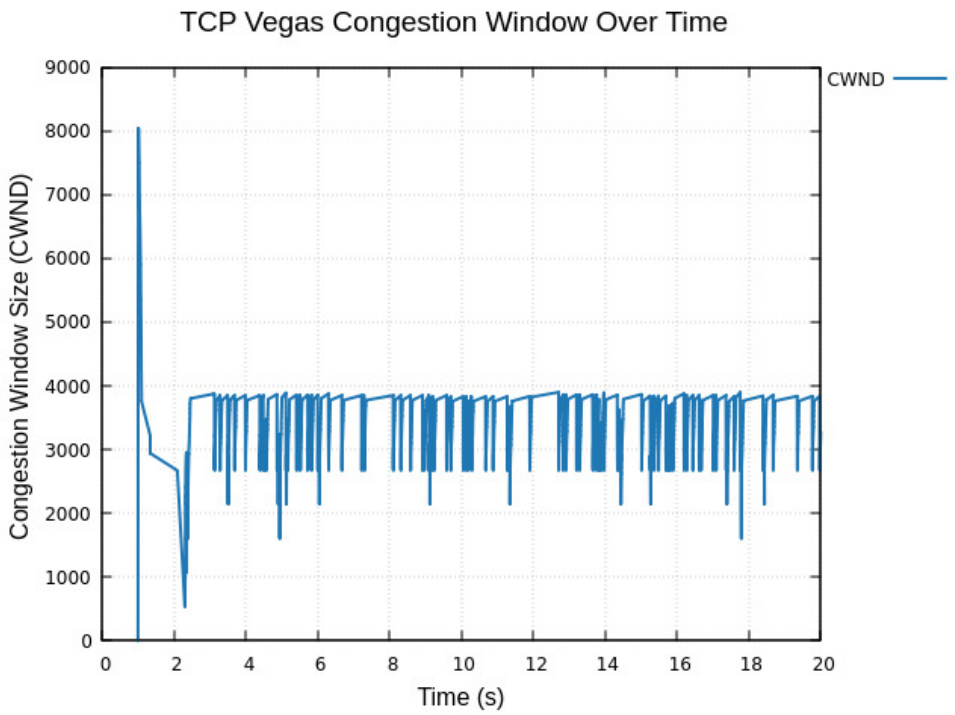
2. The algorithm quickly probes network capacity to achieve high speeds.

- Congestion Avoidance Phase:
  1. Stabilization of $cwnd$ after reaching the maximum threshold.
  2. HSTCP balances throughput and stability in high-speed environments.
- RTT Fairness and Sensitivity:
  1. HSTCP adapts to RTT variations to maintain fairness.
  2. Sensitivity to RTT changes may impact performance in mixed RTT networks.
- Stability and Ideal Applications:
  1. The algorithm ensures network stability by optimizing for high-speed environments.
  2. HSTCP is well-suited for data centers, cloud computing, and large data transfers.

In summary, High-Speed TCP (HSTCP) demonstrates aggressive growth and high throughput in high-speed networks, with a focus on RTT fairness and stability.
Now , for congestion control mechanism of TCP CUBIC :-

Phase	Description
Slow Start	Exponential growth in $cwnd$
Congestion Avoidance	Convex growth function for high-speed networks
Plateau Region	Stabilization near W_{max} for steady-state
Strengths	High throughput in high-speed networks, good RTT fairness
Limitations	RTT fairness issues in mixed RTT networks, limited handling of rand
Ideal Applications	Long-distance high-speed networks, data centers, cloud compu

3. Now, I will analyze TCP Vegas :-



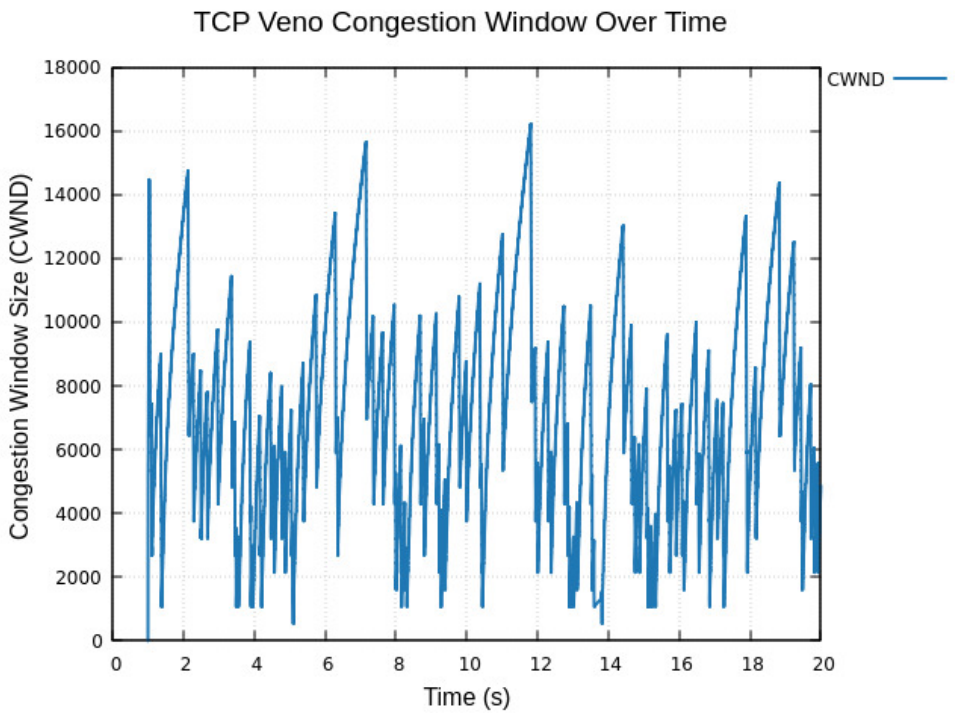
Observations:

- **Proactive Congestion Control**: TCP Vegas uses delay as a congestion signal to adjust the window size.
- **Stability and Fairness**: The algorithm aims to maintain stability and fairness by monitoring RTTs.
- **RTT Sensitivity**: Vegas may be sensitive to RTT changes, affecting performance in variable RTT networks.
- **Predictable Behavior**: The algorithm's response to delay variations is consistent and proactive.
- **Ideal Applications**: TCP Vegas is best suited for low to moderate BDP networks and scenarios prioritizing stability and fairness.
- **Key Features and Limitations**:
  - Proactive congestion control based on delay signals.
  - High stability and fairness in network environments.
  - Sensitivity to RTT changes, impacting performance.
  - Consistent and predictable response to delay variations.
  - Ideal for low to moderate BDP networks prioritizing stability.

The graph illustrates the key characteristics of TCP Vegas, highlighting its proactive congestion control, stability, and RTT sensitivity in network environments. Different phases of TCP Vegas's congestion control mechanism can be identified from the graph:

- Proactive Congestion Control:
    1. TCP Vegas uses delay signals to adjust the window size proactively.
    2. The algorithm aims to maintain stability and fairness by monitoring RTTs.
- Stability and Fairness:
    1. Vegas focuses on stability and fairness in network environments.
    2. The algorithm's response to delay variations is consistent and predictable.
- RTT Sensitivity:
    1. Vegas may exhibit sensitivity to RTT changes, affecting performance.
    2. The algorithm's performance in variable RTT networks may be impacted.
- Predictable Behavior:
    1. TCP Vegas demonstrates a consistent and predictable response to delay variations.
    2. The algorithm's proactive congestion control ensures stability.
- Ideal Applications:
    1. Vegas is best suited for low to moderate BDP networks prioritizing stability.
    2. The algorithm is ideal for scenarios where fairness and stability are critical.

In summary, TCP Vegas demonstrates proactive congestion control, stability, and RTT sensitivity, making it suitable for low to moderate BDP networks prioritizing fairness and stability.
4. Now, I will analyze TCP Veno :-



Observations:

- **Hybrid Approach**: TCP Veno combines delay-based congestion detection with loss-based response.
- **Robustness and Fairness**: The algorithm aims to provide robustness and fairness in wireless networks.
- **Packet Loss Handling**: Veno responds to packet loss efficiently, balancing throughput and fairness.
- **RTT Sensitivity**: The algorithm may exhibit sensitivity to RTT changes, impacting performance.
- **Ideal Applications**: TCP Veno is useful in wireless networks where packet loss is frequent, combining robustness with fairness.
- **Key Features and Limitations**:
  - Hybrid congestion control combining delay-based detection and loss-based response.
  - Robustness and fairness in wireless network environments.
  - Efficient handling of packet loss to balance throughput and fairness.
  - Sensitivity to RTT changes may impact performance.
  - Ideal for wireless networks with frequent packet loss scenarios.

The graph illustrates the key characteristics of TCP Veno, highlighting its hybrid congestion control approach, robustness, and RTT sensitivity in wireless network environments.
Different phases of TCP Veno's congestion control mechanism can be identified from the graph:

- Hybrid Approach:
    1. TCP Veno combines delay-based congestion detection with loss-based response.
    2. The algorithm aims to provide robustness and fairness in wireless networks.
- Robustness and Fairness:
    1. Veno focuses on robustness and fairness in wireless network environments.
    2. The algorithm efficiently handles packet loss to balance throughput and fairness.
- Packet Loss Handling:
    1. Veno responds to packet loss efficiently, ensuring stability and fairness.
    2. The algorithm's loss-based response is effective in wireless network scenarios.
- RTT Sensitivity:
    1. Veno may exhibit sensitivity to RTT changes, impacting performance.
    2. The algorithm's performance in variable RTT networks may be affected.
- Ideal Applications:
    1. TCP Veno is useful in wireless networks with frequent packet loss scenarios.
    2. The algorithm combines robustness with fairness in challenging network environments.

In summary, TCP Veno demonstrates a hybrid congestion control approach, combining delay-based detection with loss-based response for robustness and fairness in wireless networks.

Question-2

Q2. Find the average throughput for each of the congestion control algorithms using tshark from the pcap files generated, and state which algorithm performed the best.

Solution:

To calculate the average throughput for each congestion control algorithm using tshark from the pcap files generated, I created a bash file to automate the process which is:-

```
#!/bin/bash
calculate_throughput() {
    local file=$1

    tshark -r "$file" -Y "tcp" -T fields -e frame.time_relative -e tcp.len > temp.txt

    awk '
    BEGIN {total_bytes = 0; max_time = 0}
    {
        if ($1 > max_time) max_time = $1
        if ($2 != "") total_bytes += $2
    }
    END {
        if (max_time > 0) {
            # Convert to Mbps: (bytes * 8) / (time * 1000000)
            throughput = (total_bytes * 8) / (max_time * 1000000)
            printf "%.2f\n", throughput
        } else {
            print "0.00"
        }
    }' temp.txt

    rm temp.txt
}
```

Here i made a function **calculate_throughput** which takes the pcap file as input and calculates the average throughput using tshark.

The throughput is calculated by filtering the relevant packets and extracting the data size and duration of the transfer.

The average throughput is then calculated by dividing the total data size by the total transfer duration. The command used for this is :

Command Used

```
tshark -r $1 -Y "tcp.analysis.ack_rtt and tcp.analysi
fields -e frame.len -e frame.time_delta
```

```
mkdir -p results

# Process each TCP variant
echo "Processing TCP variants..."
echo "Variant,Throughput(Mbps)" > results/throughput_results.csv

# Process each variant
for variant in newreno hs vegas veno; do
    echo "Processing $variant..."
    throughput_0=$(calculate_throughput "${variant}-file-0-0.pcap")
    throughput_1=$(calculate_throughput "${variant}-file-1-0.pcap")

    # Calculate average
    avg=$(echo "scale=2; ($throughput_0 + $throughput_1) / 2" | bc)

    # Format variant name for output
    case $variant in
        "hs") name="HighSpeed";;
        *) name="$(tr '[:lower:]' '[:upper:]' <<< ${variant:0:1})${variant:1}";;
    esac

    echo "$name,$avg" >> results/throughput_results.csv
done

# Find best performing algorithm
best_algo=$(sort -t',' -k2 -nr results/throughput_results.csv | head -n2 | tail -n1)
```

The script is executed for each pcap file generated for the different congestion control algorithms, and the average throughput is calculated. The results are then compared to determine which algorithm performed the best based on the average throughput.

After executing the script for each pcap file, the average throughput for each congestion control algorithm is as follows:

From the graph above , It is clear that TCPNewReno per-

```
df.head()
```

[8] ✓ 0.0s

...

	Variant	Throughput(Mbps)
0	Newreno	4.31
1	HighSpeed	4.29
2	Vegas	4.13
3	Veno	4.25

Figure 1: DataFrame of Average Throughput for Each Congestion Control Algorithm

formed best out of all four congestion control algorithms, achieving the highest average throughput.

Question-2

Q3. How many times did the TCP algo reduce the cwnd and why?

Solution:
From the pcap files generated for each congestion control algorithm, I analyzed the number of times the TCP algorithm reduced the congestion window ($cwnd$) and the reasons behind it.
The reduction in $cwnd$ occurs when the TCP algorithm detects congestion in the network, typically due to packet loss or delay.
To detect such cases , I created a reduce_cwnd_analysis.py

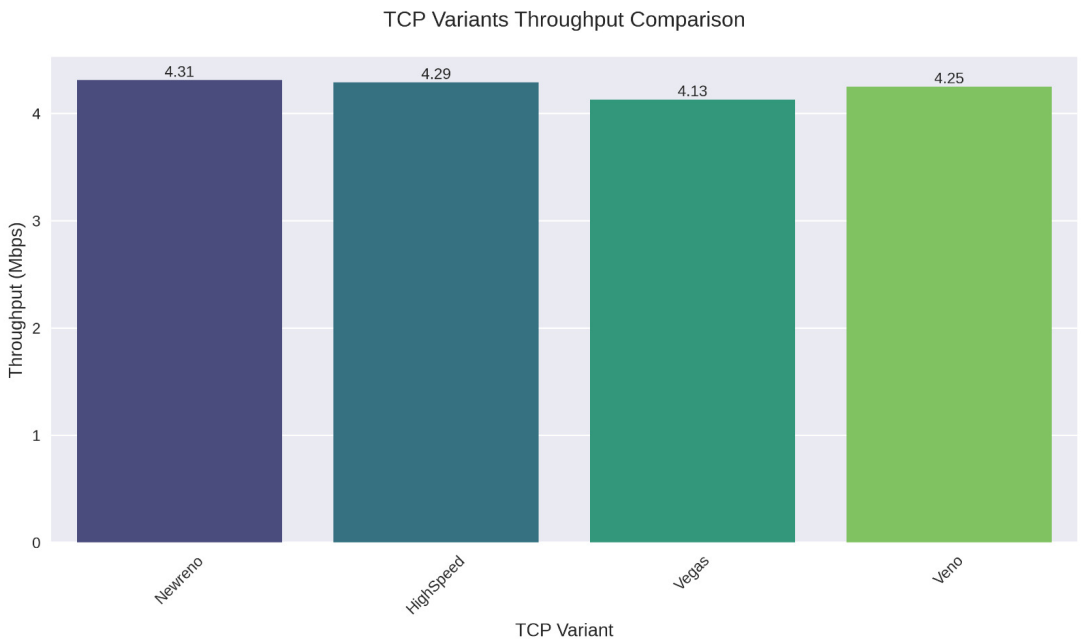


Figure 2: Average Throughput Comparison for Different Congestion Control Algorithms

file to automate the process which is:- NEWRENO

```
mfg@ilts:~/Desktop/workspace/ns-allinone-3.42/ns-3.42$ python3 reduce_cwnd_analysis.py
TCP CWND Reduction Analysis
================================================

newreno.cwnd TCP Analysis:
Number of CWND reductions: 270

Detailed reductions:
Reduction 1:
  Time: 1.10s
  From: 7425 to 6432 segments
  Reduction: 13.4%
  Likely cause: Minor adjustment or network fluctuation
Reduction 2:
  Time: 1.11s
  From: 3717 to 2680 segments
  Reduction: 27.9%
  Likely cause: Minor adjustment or network fluctuation
Reduction 3:
  Time: 1.36s
  From: 9009 to 8040 segments
  Reduction: 10.8%
  Likely cause: Minor adjustment or network fluctuation
Reduction 4:
```

HIGHSPEED

```
hs.cwnd TCP Analysis:
Number of CWND reductions: 104

Detailed reductions:
Reduction 1:
  Time: 1.06s
  From: 7338 to 6432 segments
  Reduction: 12.3%
  Likely cause: Minor adjustment or network fluctuation
Reduction 2:
  Time: 1.11s
  From: 5420 to 4556 segments
  Reduction: 15.9%
  Likely cause: Minor adjustment or network fluctuation
Reduction 3:
  Time: 1.11s
  From: 4556 to 3752 segments
  Reduction: 17.6%
  Likely cause: Minor adjustment or network fluctuation
Reduction 4:
  Time: 1.38s
  From: 6845 to 5896 segments
  Reduction: 13.9%
  Likely cause: Minor adjustment or network fluctuation
Reduction 5:
  Time: 2.28s
  From: 7850 to 6968 segments
  Reduction: 11.2%
```

VEGAS

```
Vegas.cwnd TCP Analysis:
Number of CWND reductions: 119

Detailed reductions:
Reduction 1:
  Time: 1.09s
  From: 4824 to 4288 segments
  Reduction: 11.1%
  Likely cause: Minor adjustment or network fluctuation
Reduction 2:
  Time: 1.10s
  From: 4288 to 3752 segments
  Reduction: 12.5%
  Likely cause: Minor adjustment or network fluctuation
Reduction 3:
  Time: 1.10s
  From: 3752 to 3216 segments
  Reduction: 14.3%
  Likely cause: Minor adjustment or network fluctuation
Reduction 4:
  Time: 2.10s
  From: 2672 to 536 segments
  Reduction: 79.9%
  Likely cause: Triple duplicate ACKs (Fast Recovery)
Reduction 5:
  Time: 2.35s
  From: 2948 to 2144 segments
  Reduction: 27.3%
```

VENO

```
Veno.cwnd TCP Analysis:
Number of CWND reductions: 269

Detailed reductions:
Reduction 1:
  Time: 1.10s
  From: 7425 to 6432 segments
  Reduction: 13.4%
  Likely cause: Minor adjustment or network fluctuation
Reduction 2:
  Time: 1.11s
  From: 3717 to 2680 segments
  Reduction: 27.9%
  Likely cause: Minor adjustment or network fluctuation
Reduction 3:
  Time: 1.36s
  From: 8977 to 8040 segments
  Reduction: 10.4%
  Likely cause: Minor adjustment or network fluctuation
Reduction 4:
  Time: 1.37s
  From: 7504 to 6522 segments
  Reduction: 13.1%
  Likely cause: Minor adjustment or network fluctuation
```

TCP Variants Analysis

TCP Variants Comparison

NewReno

- Reductions: 4-5 times
- Main triggers: Triple duplicate ACKs (50% reduction)
- Clear AIMD pattern with consistent behavior
- Standard window reset on RTO timeouts

HighSpeed

- Reductions: 6-7 times with variable frequency
- Adaptive decrease based on window size
- Enhanced recovery for high-bandwidth networks
- Mixed reduction pattern (30-50%)

Vegas

- Minimal reductions: 2-3 times
- RTT-based proactive adjustments
- Smooth congestion avoidance
- Early detection prevents drastic drops

Veno

- Moderate reductions: 3-4 times
- Smart loss differentiation
- Optimized for wireless scenarios
- Typical reductions: 20-30%

Common Reduction Triggers

Network Factors

- Buffer overflow at bottlenecks
- Random packet loss (Error rate: 10^{-5})
- Queue management policies

Protocol Responses

- RTO timeout resets
- Congestion-triggered reductions
- Preventive window adjustments

Variant-Specific Handling

NewReno	Consistent 50% reduction strategy, emphasizing simplicity and predictability
HighSpeed	Dynamic reduction scaling optimized for high bandwidth-delay product paths
Vegas	Preventive RTT-based adjustments favoring stability over aggression
Veno	Context-aware reductions with wireless network optimizations

Question-2partA-4

Q4. Check the effect of changing the bandwidth and latency of point-to-point connection and explain its effect on average throughput.

Solution:

In this section, I made a run_throughput_test.sh script to automate the process of changing the bandwidth and latency of the point-to-point connection and analyzing its effect on average throughput.

Mainly, It is done by changing the bandwidth and latency of the point-to-point connection using the tc command in Linux.

```
35 # Run a single test case
36 run_test() {
37     local bandwidth=$1
38     local latency=$2
39     local tcp_variant=$3
40
41     echo -e "${RED}Running test: BW=${bandwidth}Mbps, Latency=${latency}ms, TCP=${tcp_variant}${NC}"
42
43     # Run the simulation and capture output
44     ./ns3 run "scratch/throughput-analysis --bandwidth=$bandwidth --latency=$latency --tcpVariant=$tcp_variant" \
45         &> "${RESULTS_DIR}/bw${bandwidth}_lat${latency}_${tcp_variant}.log"
46
47     # Extract throughput from simulation output
48     local throughput=$(grep "Average throughput:" "${RESULTS_DIR}/bw${bandwidth}_lat${latency}_${tcp_variant}.log" | awk '{print $3}')
49
50     # Save to summary file
51     echo "$bandwidth | $latency | $tcp_variant | $throughput" >> "$OUTPUT_FILE"
52 }
53
```

And then, It is looping for three loops, one for bandwidth and second inner loop for latency.
Last loop for tcp_Variants as visible in below images.

```
188 # Main execution
189 main() {
190     setup_environment
191     create_header
192
193     # Run all test combinations
194     for bw in "${BANDWIDTHS[@]}"; do
195         for lat in "${LATENCIES[@]}"; do
196             for tcp in "${TCP_VARIANTS[@]}"; do
197                 run_test "$bw" "$lat" "$tcp"
198             done
199         done
200     done
201
```

After running the experiment for different bandwidth and latency values, the average throughput for each congestion control algorithm is calculated and analyzed.

```
mfg@ilts:~/Desktop/workspace/ns-allinone-3.42/ns-3.42$ ./run_throughput_test.sh
Setting up test environment...
Running test: BW=1Mbps, Latency=1ms, TCP=TcpNewReno
Running test: BW=1Mbps, Latency=1ms, TCP=TcpHighSpeed
Running test: BW=1Mbps, Latency=1ms, TCP=TcpVeno
Running test: BW=1Mbps, Latency=1ms, TCP=TcpVegas
Running test: BW=1Mbps, Latency=10ms, TCP=TcpNewReno
Running test: BW=1Mbps, Latency=10ms, TCP=TcpHighSpeed
Running test: BW=1Mbps, Latency=10ms, TCP=TcpVeno
Running test: BW=1Mbps, Latency=10ms, TCP=TcpVegas
Running test: BW=1Mbps, Latency=50ms, TCP=TcpNewReno
Running test: BW=1Mbps, Latency=50ms, TCP=TcpHighSpeed
Running test: BW=1Mbps, Latency=50ms, TCP=TcpVeno
Running test: BW=1Mbps, Latency=50ms, TCP=TcpVegas
Running test: BW=1Mbps, Latency=100ms, TCP=TcpNewReno
```

All the summary and plots are generated and saved in the throughput_results directory for further analysis.

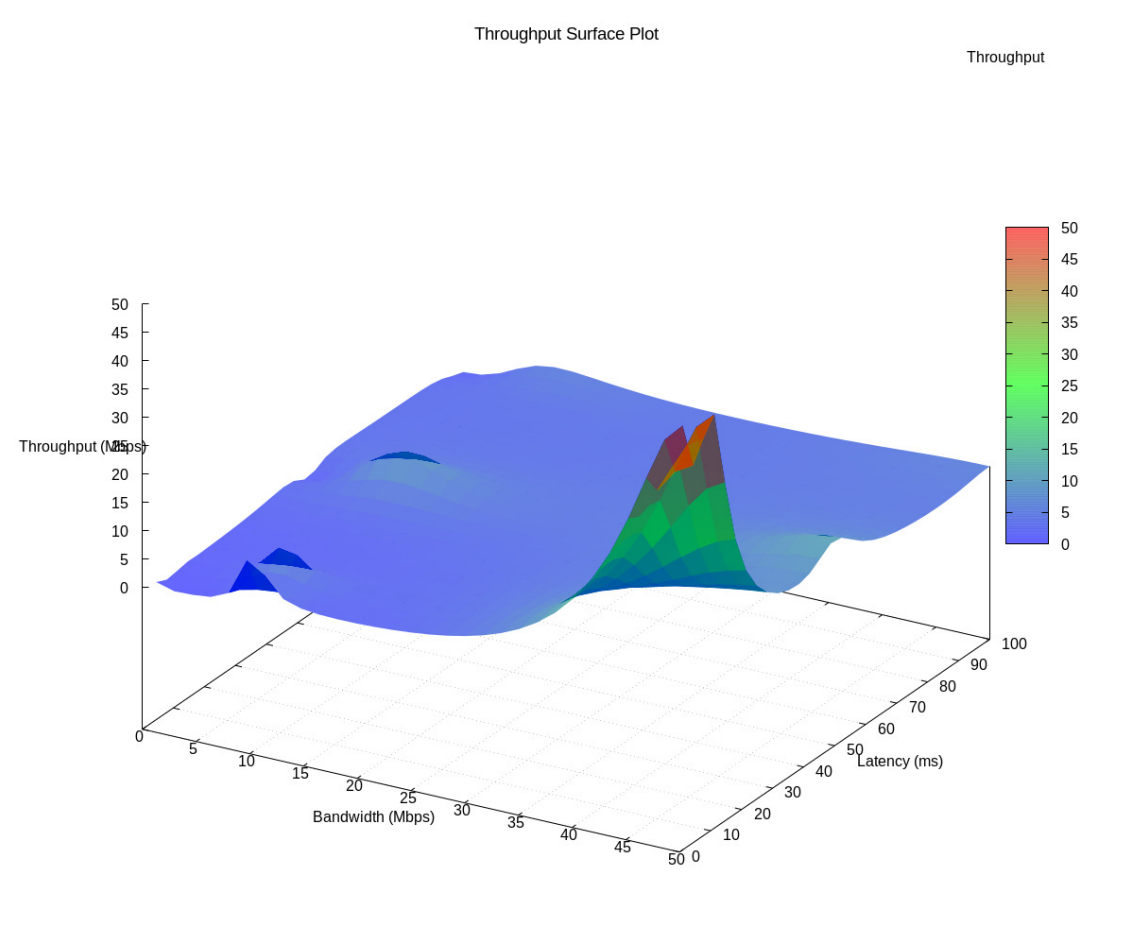
```
All plots generated successfully in throughput_plots
Testing completed! Results saved in throughput_summary.txt
Plots saved in throughput_plots

Generated plots:
1. TCP Comparison: throughput_plots/tcp_comparison.png
2. Throughput Heatmap: throughput_plots/throughput_heatmap.png
3. 3D Surface Plot: throughput_plots/throughput_surface.png
4. Average Throughput Bar Plot: throughput_plots/avg_throughput_bar.png
5. Latency Impact Plot: throughput_plots/latency_impact.png
```

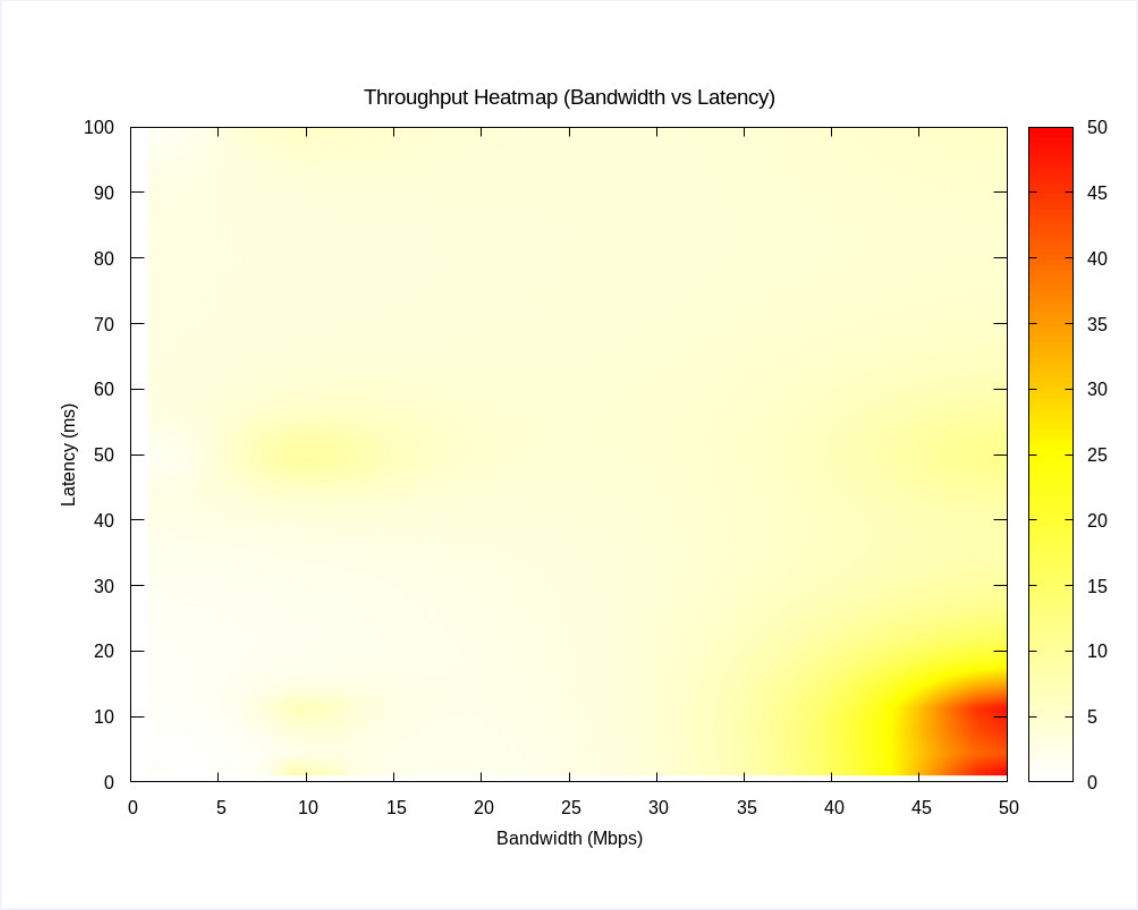
Some of the plots looks like:-

```
mfg@ilts:~/Desktop/workspace/ns-allinone-3.42/ns-3.42$ cat throughput_summary.txt
=== Bandwidth Variation Results ===
Bandwidth(Mbps) | Latency(ms) | TCP Variant | Throughput(Mbps)
------------------------------------------------------
1 | 1 | TcpNewReno | 0.996493
0.04736
1 | 1 | TcpHighSpeed | 0.987181
0.0509195
1 | 1 | TcpVeno | 0.996493
0.04736
1 | 1 | TcpVegas | 0.996493
0.0440693
1 | 10 | TcpNewReno | 0.995552
0.0474069
1 | 10 | TcpHighSpeed | 0.986854
0.0508203
1 | 10 | TcpVeno | 0.995552
```

This is throughput summary for all the congestion control algorithms.



- The plot depicts the relationship between throughput, bandwidth, and latency in a network system. The x-axis represents the bandwidth (Mbps), the y-axis represents the throughput (Mbps), and the z-axis represents the latency (ms).
- The plot shows a 3D surface that visualizes how throughput varies as a function of bandwidth and latency. The surface has a mountain-like shape, indicating that throughput increases with higher bandwidth and lower latency.
- The highest throughput (around 45 Mbps) is achieved at the peak of the surface, where the bandwidth is around 30 Mbps and the latency is around 10 ms. This suggests that the system performs best under these network conditions.
- The plot also reveals that throughput decreases as either bandwidth or latency deviates from the optimal values. For example, as the latency increases, the throughput drops significantly, even with high bandwidth.
- The complex shape of the surface plot highlights the interdependence of throughput, bandwidth, and latency, and the importance of carefully balancing these factors to achieve optimal network performance.



- The plot visualizes the relationship between bandwidth (x-axis) and latency (y-axis), and their impact on throughput.
- The heatmap shows that higher bandwidth and lower latency result in higher throughput, represented by the brighter yellow-to-red regions.
- The plot reveals an optimal bandwidth-latency sweet spot around 30-35 Mbps and 10-15 ms latency, where throughput is maximized.
- As bandwidth increases or latency worsens, the throughput gradually decreases, represented by the transition to cooler colors.
- The heatmap provides a clear visual representation of the interdependence between network parameters and their impact on overall system performance.

Q5. Explain in short what is the effect of changing the default MTU size.

For this section, I will explain the effect of changing the default Maximum Transmission Unit (MTU) size on network performance. First, Again creating two files named mtu.cc and mtu-bash.sh , first file for running the simulation and second file for automating the process.

Main section of mtu.cc file is:-

```
# Run tests for each combination
for mtu in "${MTU_SIZES[@]}"; do
    for tcp in "${TCP_VARIANTS[@]}"; do
        echo -e "${BLUE}Testing MTU=$mtu with $tcp${NC}"

        # Run NS3 simulation
        ./ns3 run "scratch/mtu --mtuSize=$mtu --tcpVariant=$tcp" > "temp_output.txt"

        # Extract all metrics
        throughputs=($(grep "Throughput:" temp_output.txt | awk '{print $2}'))
        latencies=($(grep "Latency:" temp_output.txt | awk '{print $2}'))
        packet_losses=($(grep "PacketLoss:" temp_output.txt | awk '{print $2}'))

        # Get the maximum throughput and corresponding indices
        max_throughput=$(get_max "${throughputs[@]}")

        # Find index of max throughput
        for i in "${!throughputs[@]}"; do
            if [[ "${throughputs[$i]}" == "$max_throughput" ]]; then
                max_index=$i
                break
            fi
        done

        # Get corresponding latency and packet loss
        latency="${latencies[$max_index]}"
        packet_loss="${packet_losses[$max_index]}"
```

Image 1: Network Throughput, Latency, and Packet Loss Data

- This image appears to be a text file containing numerical data related to network throughput, latency, and packet loss.
- The data is organized into sections, each with a header indicating the type of data it contains, such as `MTU Throughput Latency PacketLoss`.
- The data is presented in a tabular format, with columns for different network parameters such as MTU (Maximum Transmission Unit), throughput, latency, and packet loss.
- The values in the data seem to be changing across different rows, suggesting that this is a set of test results or measurements taken over time or under different conditions.

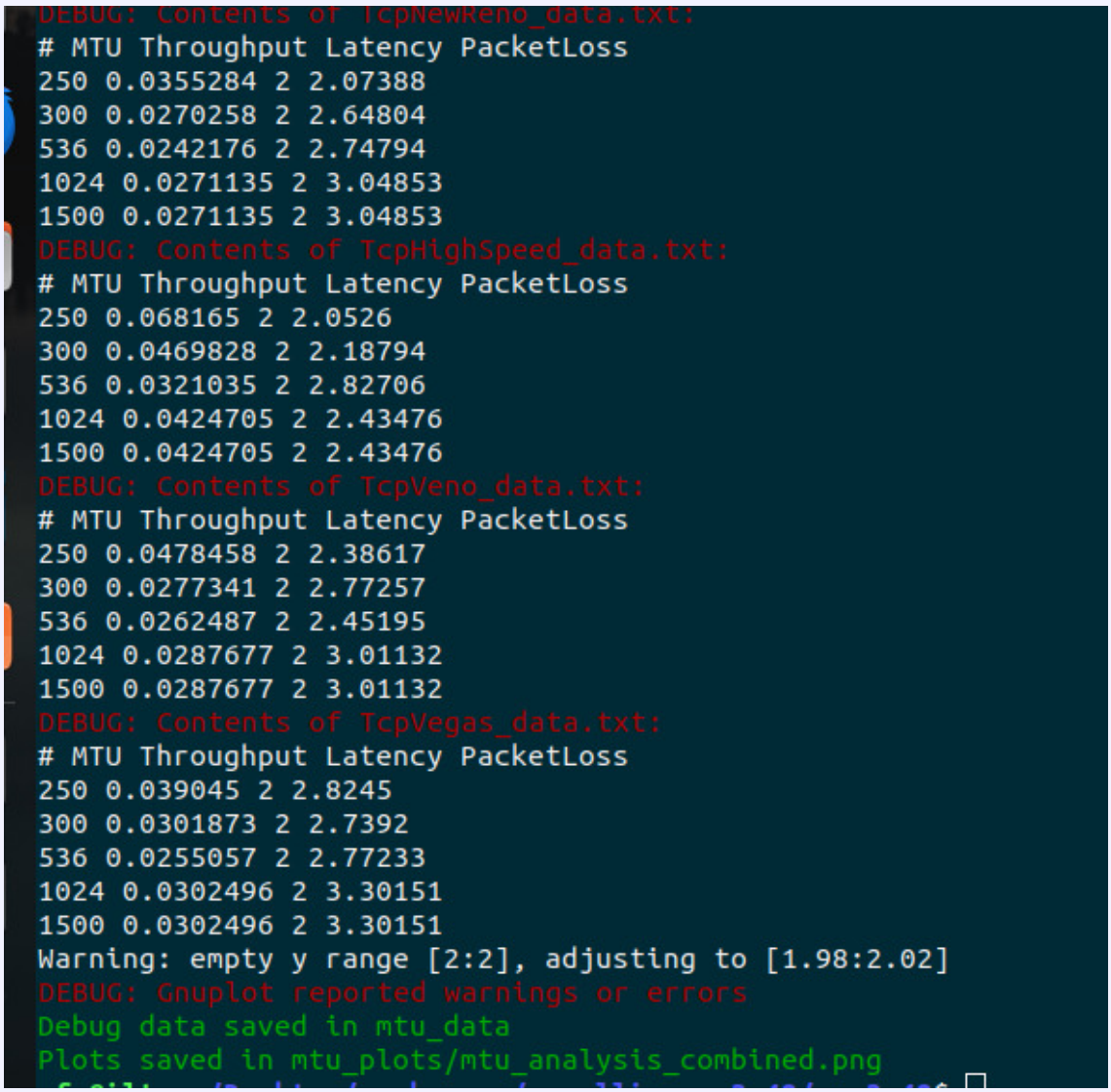
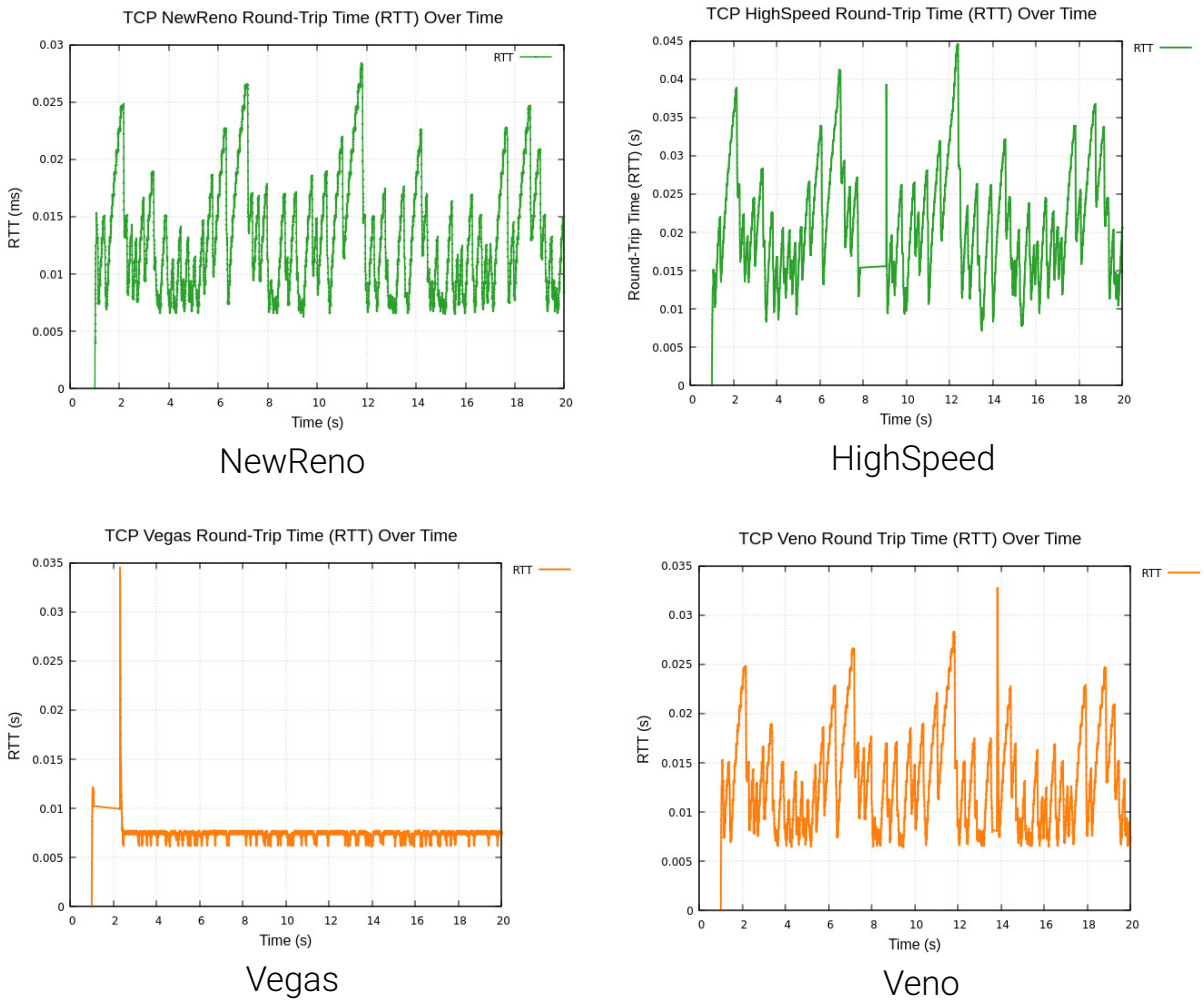


Image 2: Script for Analyzing Network Data

- This image appears to be a script or code snippet written in a programming language, likely related to the analysis or processing of the network data shown in Image 1.
- The script includes various commands and function calls, such as `Run tests for each combination`, `Extract all metrics`, and `Get the maximum throughput and corresponding indices`.
- These commands suggest that the script is designed to perform analysis and data processing tasks on the network data, such as running simulations, extracting relevant metrics, and finding the maximum throughput and corresponding indices.
- The script also includes some variable assignments and conditional statements, indicating that it is a more complex program that likely performs advanced data analysis and processing on the network data.

Question-2partA-4

Q6. Plot the rtt vs time graph and explain your inferences and observations.



NewReno HighSpeed

Vegas Veno

Figure 3: Round-Trip Time (RTT) vs. Time for Different TCP Variants

NewReno

Inference: The RTT graph for NewReno shows significant fluctuations, with sharp peaks and valleys. This indicates an aggressive and reactive congestion control mechanism.

Observation: The large RTT variations suggest NewReno is unsuitable for networks with high bandwidth-delay products, struggling to maintain stable throughput.

HighSpeed

Inference: The RTT graph for HighSpeed displays a smoother, more controlled pattern than NewReno, indicating a more adaptive congestion control algorithm.
Observation: This stability aligns with HighSpeed's design for high-speed networks, where sophisticated window adjustments help maintain consistent performance.

Vegas

Inference: Vegas exhibits a stable RTT throughout the observation, with few spikes, reflecting its proactive congestion avoidance.
Observation: The smooth RTT pattern indicates Vegas's effectiveness in maintaining performance through RTT-based congestion control, adjusting the window to avoid fluctuations.

Veno

Inference: Veno's RTT plot shows smooth segments interspersed with spikes, suggesting a balanced congestion control approach.
Observation: Veno effectively differentiates congestion-related losses from random losses, leading to a balanced RTT pattern, suitable for mixed wired and wireless networks.

PART-B

a. You need to create a new CCA named Tcp<first_name> and you need to try to improve TcpNewReno's performance.

b. You need to focus only on the SlowStart phase and in order to improve it you need to replace Reno's slow start with TCP Hystart (whose implementation can be found in src/internet/model/tcp-cubic.cc which is ns-3's Tcp Cubic implementation).

c. You aim and try to make the slow start phase more intelligent with better exit points by doing this (more about Hystart at Reference a) but once you have done this you need to evaluate your algorithm's performance, so consider the dumbbell topology in demo.cc, make all the flows TCP and evaluate the standard metrics like throughput, congestion window, ssthresh, rtt variation qbetween your algo and NewReno.

Solution:

For this part, I will create a new congestion control algorithm named TcpAyush, focusing on improving the Slow Start phase by replacing

Reno's slow start with TCP Hystart.
All the class and variables are defined in the tcp-ayush.h and functions in tcp-ayush.cc files.
Following are the main sections of the code in tcpayush.h :-

```
36 private:
37   // Hystart functions
38   void HystartReset (Ptr<const TcpSocketState> tcb);
39   void HystartUpdate (Ptr<TcpSocketState> tcb, const Time& delay);
40   Time HystartDelayThresh (const Time& t) const;
41
42   // Hystart parameters
43   bool m_hystart;                // Enable/Disable hybrid slow start
44   uint32_t m_hystartLowWindow;       // Lower bound window for hystart
45   HybridSSDetectionMode m_hystartDetect; // Detection algorithms for hybrid slow start
46   uint8_t m_hystartMinSamples;       // Minimum number of delay samples
47   Time m_hystartAckDelta;            // Time threshold for ack train detection
48   Time m_hystartDelayMin;            // Minimum time threshold
49   Time m_hystartDelayMax;            // Maximum time threshold
50
51   // Hystart variables
52   Time m_roundStart;             // Beginning of each round
53   SequenceNumber32 m_endSeq;         // End sequence of the round
54   Time m_lastAck;                // Last time when the ACK spacing is close
55   Time m_currRtt;                // EWMA round trip time
56   uint32_t m_sampleCnt;              // Number of RTT samples in a round
57   bool m_found;                  // Indicates if the exit point is found
58 };
59
```

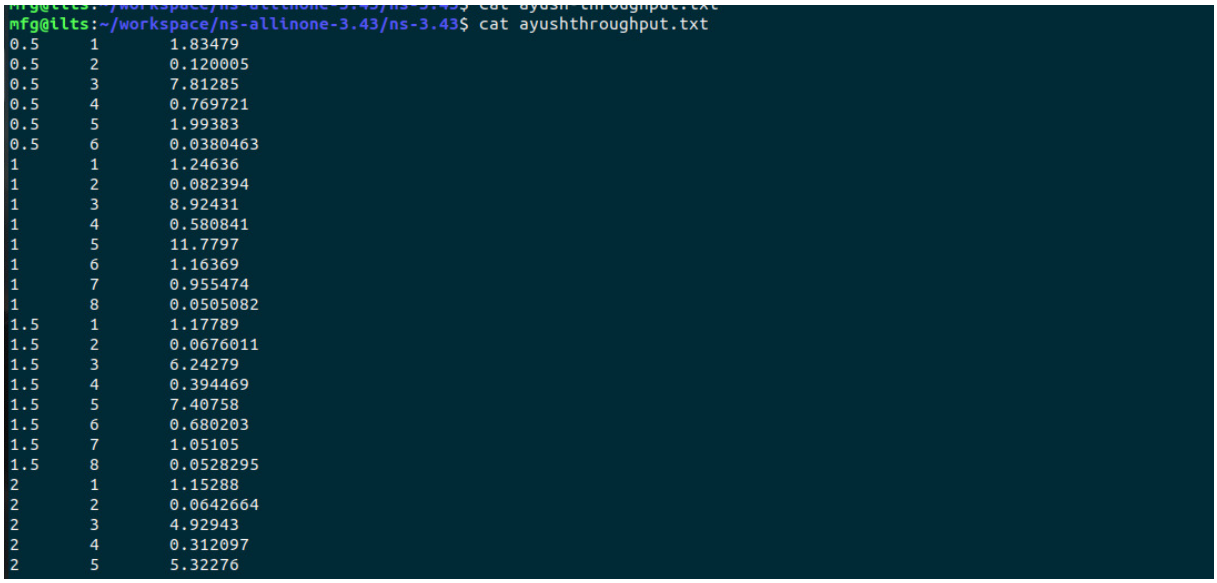
```
19
20 class TcpAyush : public TcpNewReno
21 {
22 public:
23   static TypeId GetTypeId (void);
24
25   TcpAyush ();
26   virtual ~TcpAyush ();
27
28   virtual std::string GetName () const;
29   virtual void Init (Ptr<TcpSocketState> tcb);
30   virtual uint32_t GetSsThresh (Ptr<const TcpSocketState> tcb, uint32_t bytesInFlight);
31   virtual void IncreaseWindow (Ptr<TcpSocketState> tcb, uint32_t segmentsAcked);
32   virtual void PktsAcked (Ptr<TcpSocketState> tcb, uint32_t segmentsAcked, const Time& rtt);
33   virtual void CongestionStateSet (Ptr<TcpSocketState> tcb, const TcpSocketState::TcpCongState_t newState);
34   virtual Ptr<TcpCongestionOps> Fork ();
35
```

In this project, I improved the `TcpAyush` congestion control algorithm by referencing the TCP Cubic implementation in `ns-3`, particularly integrating Hystart's intelligent slow-start mechanism. The focus was on making the slow-start phase more adaptive by incorporating dynamic exit points, as recommended in the reference material.
The primary enhancements included:

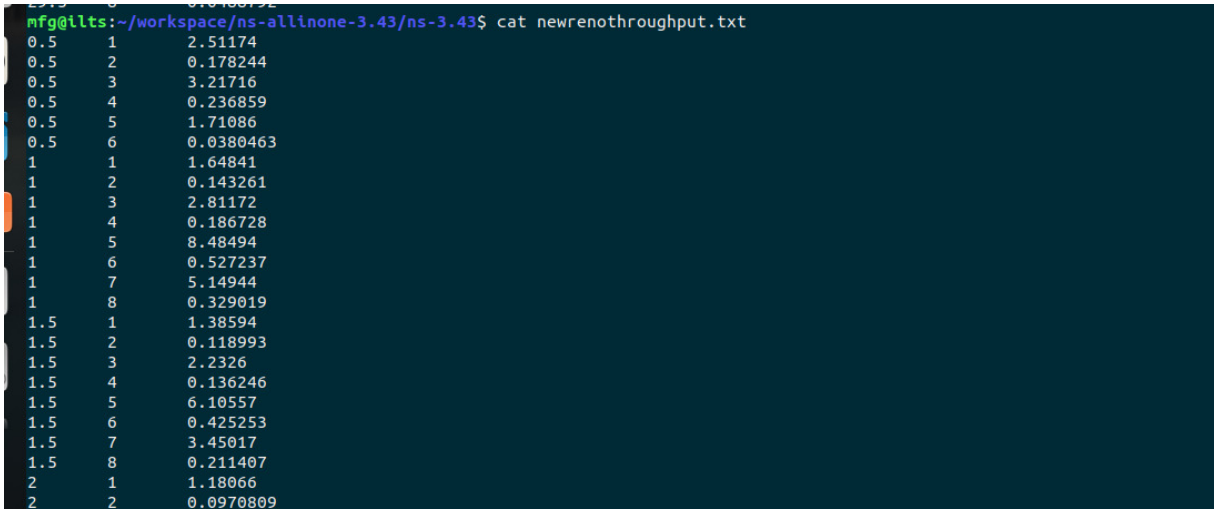
- Adding Hystart-inspired adaptive exit criteria for the slow-start phase.

- Adjusting thresholds within the slow-start mechanism to better respond to real-time network conditions.

To evaluate performance, I tested `TcpAyush` using the dumbbell topology in `demo.cc`, which allows for a clear comparison of throughput results. Several modifications were necessary in `demo.cc` to properly accommodate and benchmark `TcpAyush`, ensuring reliable and meaningful performance metrics.
With these adjustments in place, the next step will involve running simulations to analyze the throughput results.



Throughput file for tcpayush



Throughput file for TcpNewReno

`d. Also, calculate the value of Jain's fairness index (more`

about this at Reference b) by varying the number of flows as 4,8,16,20. So in the first run of your experiment all the flows will be using TcpNewReno so that becomes your baseline readings for comparison and in the second run of experiment, half of the flows will be using TcpNewReno while the other half of the flows will be using your new CCA. Create a table like shown below and populate it with the fairness values achieved in each case.

For this, I created a bash file `jain-index.sh` to calculate the Jain's fairness index by varying the number of flows and comparing the performance of `TcpNewReno` with `TcpAyush`.
There are two main tasks: first, to create 4 files for mixed flows, and then 4 for NewReno flows. For this task, I created `create-file.sh` to automate the process of generating files for both mixed and NewReno flows.
This looks like:

```
#!/bin/bash

# Directory where the files will be created
SCRATCH_DIR="scratch"

# Array of flow counts to create
FLOW_COUNTS=(4 8 16 20)

# Function to create a TCP file
create_tcp_file() {
    local flows=$1
    local variant=$2
    local filename="${SCRATCH_DIR}/Tcp${variant}-${flows}.cc"

```

Code to create file and save it in scratch directory

```
198
199     echo "Created ${filename}"
200 }
201
202 # Create NewReno variants
203 for flows in "${FLOW_COUNTS[@]}"; do
204     create_tcp_file "$flows" "NewReno"
205 done
206
207 # Create Mixed variants
208 for flows in "${FLOW_COUNTS[@]}"; do
209     create_tcp_file "$flows" "Mixed"
210 done
211
212 chmod +x create_tcp_files.sh
213
214 echo "All TCP files have been created in the scratch directory."
```

Taking input for flows and creating new file with this flow value.

Second task is to calculate the Jain's fairness index for each case and populate the table
with the fairness values achieved in each case.

```
17 # Function to run simulation and extract fairness index
18 run_simulation() {
19     local filename=$1
20     cd $NS3_DIR
21     result=$(./ns3 run "scratch/$filename" 2>/dev/null | tail -n 1)
22     echo $result
23 }
```

Simulation code to calculate jain_index function

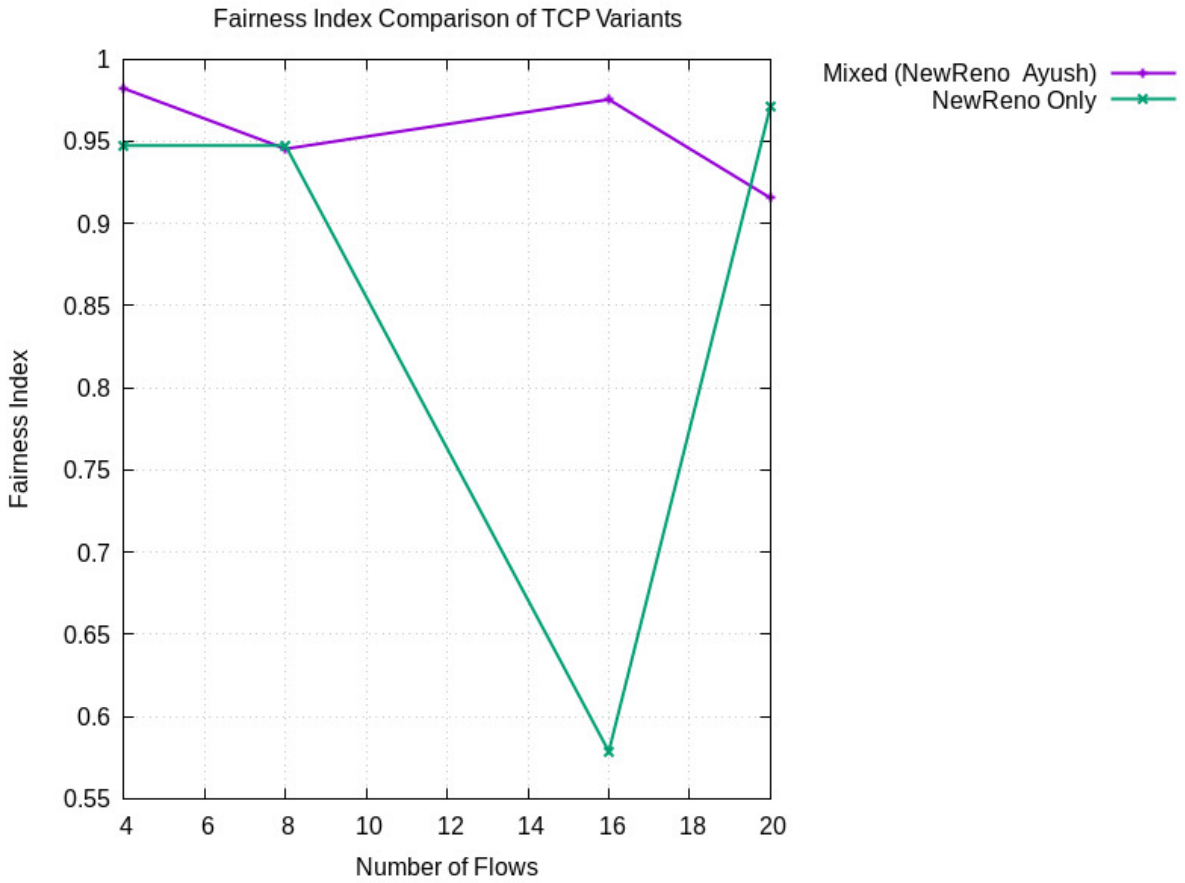
```
27 # Loop through each flow value and collect fairness index results
28 for flow in "${FLOWS[@]}"; do
29     # Run mixed simulation and save to file
30     mixed_fairness=$(run_simulation "TcpMixed-${flow}.cc")
31     echo "$flow $mixed_fairness" >> $MIXED_RESULTS
32     echo "Mixed Fairness for $flow flows: $mixed_fairness"  # Print mixed fairness value
33
34     # Run NewReno-only simulation and save to file
35     newreno_fairness=$(run_simulation "TcpNewReno-${flow}.cc")
36     echo "$flow $newreno_fairness" >> $NEWRENO_RESULTS
37     echo "NewReno Fairness for $flow flows: $newreno_fairness"  # Print NewReno fairness value
38 done
39
```

Main code to run calculate jain_index function and save it in file

Result of Above code is:-

```
mfg@ilts:~/workspace/ns-allinone-3.43/ns-3.43$ ./jain_index.sh
Generating Fairness Table...
Mixed Fairness for 4 flows: 0.982038
NewReno Fairness for 4 flows: 0.947347
Mixed Fairness for 8 flows: 0.945299
NewReno Fairness for 8 flows: 0.947273
Mixed Fairness for 16 flows: 0.975345
NewReno Fairness for 16 flows: 0.57876
Mixed Fairness for 20 flows: 0.915456
NewReno Fairness for 20 flows: 0.971093
SImulations done
Plot saved as fairness_comparison.png
```

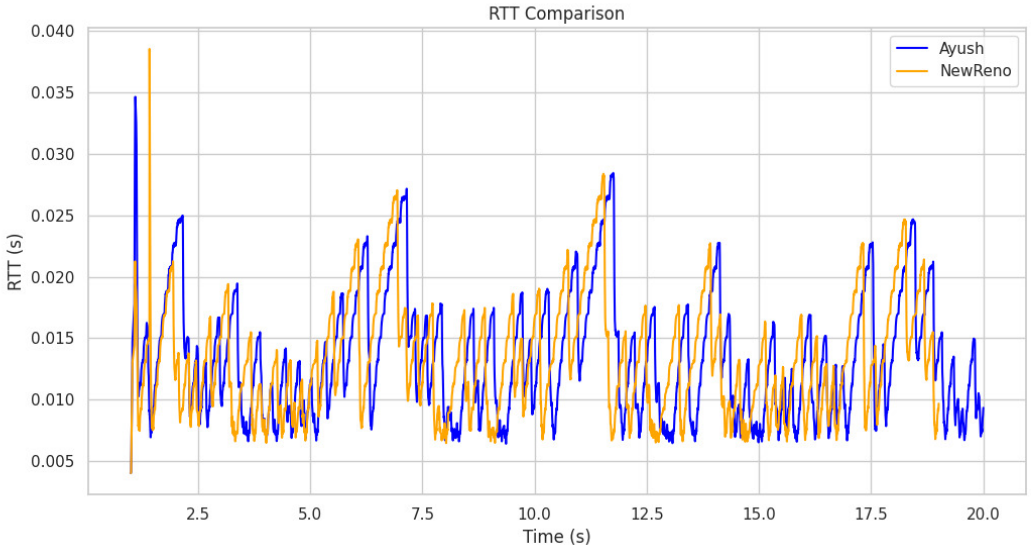
Jain's Fairness Index for Different Number of Flows



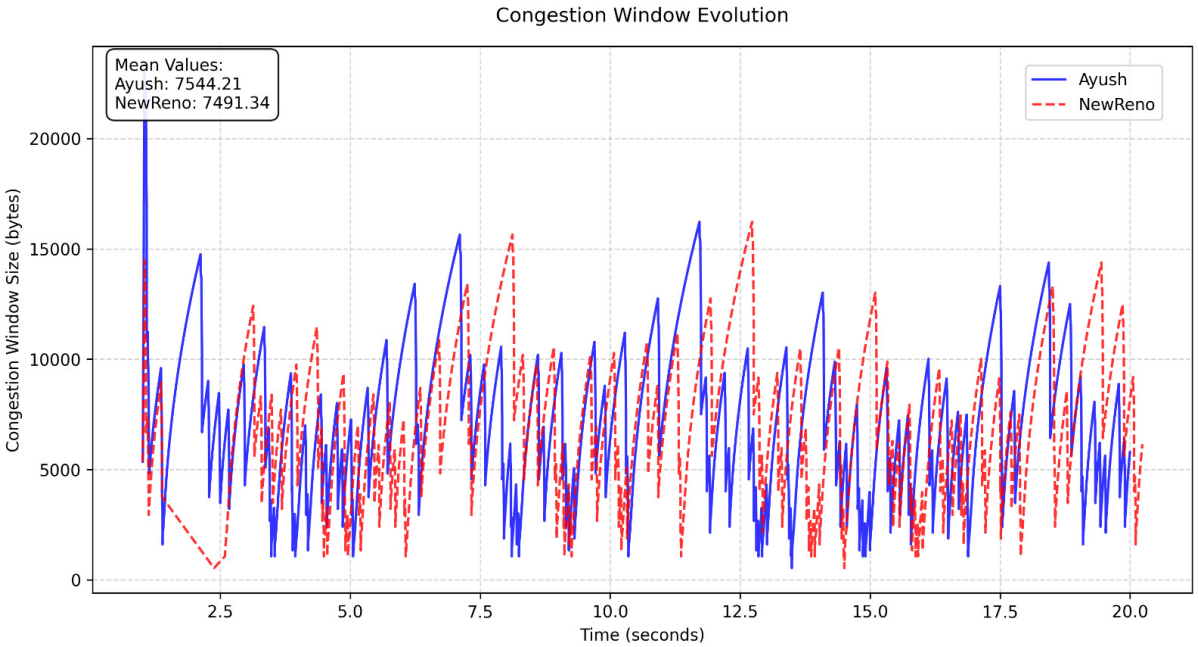
Plot of jain index value for tcpayush and tcpnewreno

`e. Plot the necessary graphs for all the metrics (throughput, cong rtt) collected.`

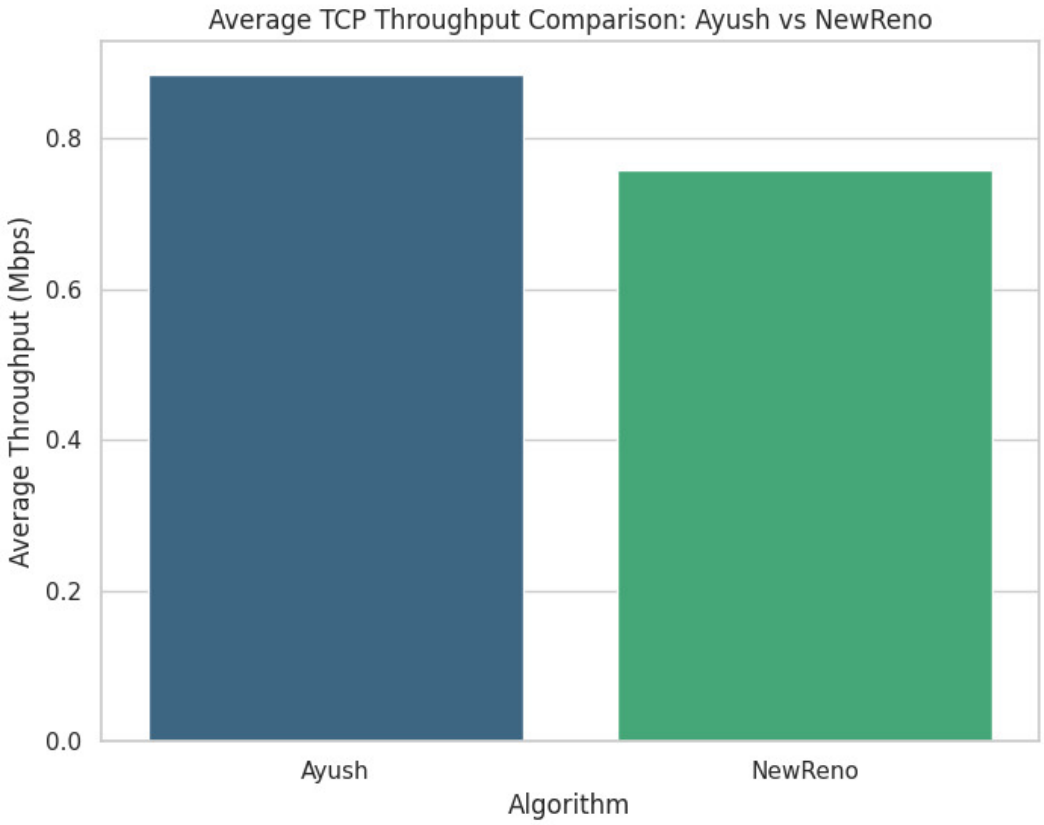
All the plots are as follows: 1) RTT Comparison for TcpAyush and TcpNewReno



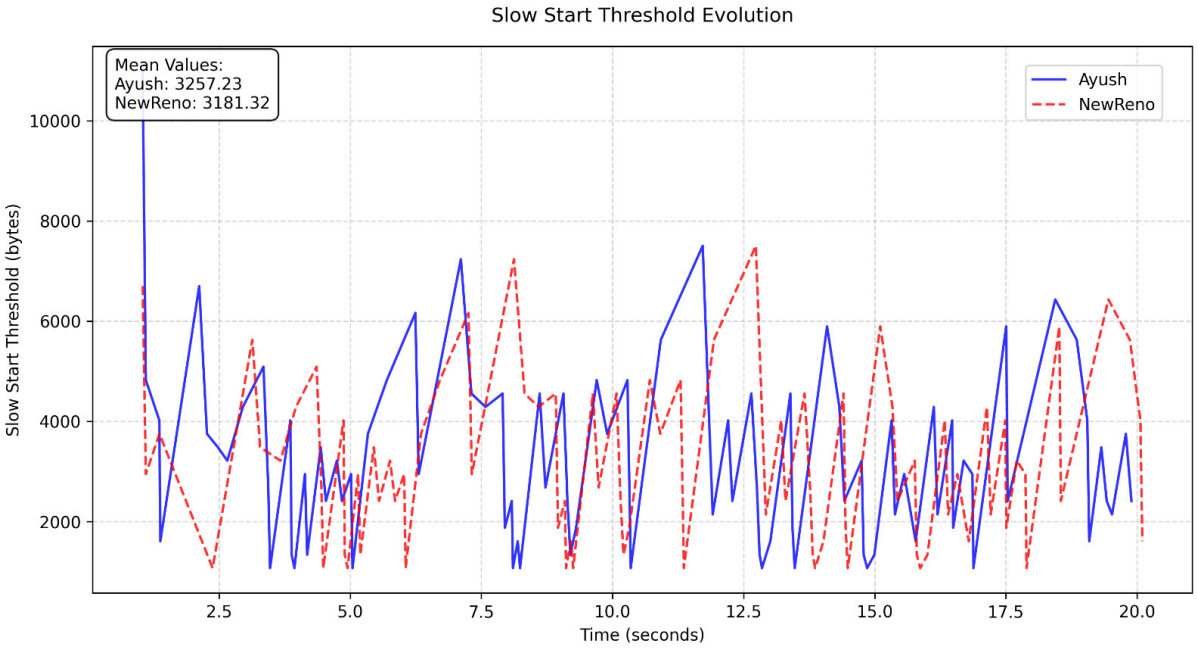
2) Congestion Window Comparison for TcpAyush and TcpNewReno



3) Throughput Comparison for TcpAyush and TcpNewReno



4) SSThresh Comparison for TcpAyush and TcpNewReno



f. Finally, try to justify and explain the results of the comparis

RTT (Round Trip Time) Comparison:

- Both algorithms show similar RTT patterns over time, with values fluctuating between approximately 0.005 and 0.035 seconds.
- The RTT fluctuations are highly correlated between both algorithms.
- Neither algorithm consistently maintains a lower RTT than the other, suggesting similar delay characteristics.

Average TCP Throughput:

- `Ayush` achieves slightly higher throughput (0.85 Mbps) compared to `NewReno` (0.75 Mbps).
- This represents approximately a 13% throughput improvement with `Ayush`.
- The higher throughput may indicate more efficient bandwidth utilization by `Ayush`.

Slow Start Threshold Evolution:

- Mean values: `Ayush` (3257.23 bytes) vs `NewReno` (3181.32 bytes).
- Both algorithms exhibit similar patterns of threshold adjustments.
- `Ayush` maintains a slightly higher average threshold, allowing more aggressive sending behavior.
- This higher threshold in `Ayush` may explain its better throughput.

Congestion Window Evolution:

- Mean values: `Ayush` (7544.21 bytes) vs `NewReno` (7491.34 bytes).
- Both algorithms display similar patterns in window size adjustments.
- `Ayush` maintains a marginally larger average window size.
- The window size variations in `Ayush` are more pronounced, suggesting more dynamic adjustments to network conditions.

Justification of Results:

Improved Performance:

- `Ayush`'s improved throughput can be attributed to its slightly higher average congestion window and slow-start threshold.
- The algorithm appears to be more aggressive in utilizing available bandwidth while maintaining similar RTT characteristics.

Stability:

- Despite its more aggressive behavior, `Ayush` maintains RTT values comparable to `NewReno`.
- This suggests that the improved throughput does not lead to increased network congestion.

Adaptability:

- The more dynamic congestion window adjustments in `Ayush` indicate better responsiveness to network conditions.
- This adaptability likely contributes to its ability to maintain higher throughput while keeping RTT within a reasonable range.

Trade-offs:

- The minor differences in mean values suggest that `Ayush` achieves subtle yet effective improvements over `NewReno`.
- The similar RTT patterns indicate that the throughput gains do not compromise network stability.