Computer Networks Assignment 4

Harsh Prajapati - 12241300

October 29, 2024

Part 1 : Study of TCP Congestion Control Algorithms

TCP Cubic

Cubic is a congestion control Algorithm that uses a cubic function to improve scalability and stability over long-distance networks and fast networks. Cubic has been adopted by Linux, Windows and Apple stacks as default congestion control algorithm.

CUBIC tries to provide fair bandwidth sharing among connections with different roundtrip times (RTTs) by making its congestion window grow independent of RTT. Thus, all connections will scale up their window size at the same rate irrespective of RTTs. This allows CUBIC to quickly ramp up the window size when the network is not fully utilized, while slowing down as it nears capacity of an already full pipe. That is the reason why CUBIC can be scalable for high-bandwidth, long distance network and has a reasonably good level of stability besides having proved itself to provide fairness against standard TCP connections.

How does it work?

1. **Slow Start Phase :** The algorithm starts with a small congestion window (cwnd) and quickly increases it until it detects network congestion.

2. **Recording Maximum Window Size :** When congestion occurs (like packet loss), the current Cwnd is saved as W_{max} , the highest window size reached without congestion.

3. **Setting W_{max} as a Key Point :** W_{max} value is used as the "turning point" for a cubic function that will control how the cwnd grows from now on.

4. **Restart with a Smaller Window :** After recording W_{max} , Cubic restarts transmission but with a slightly smaller cwnd to avoid immediate congestion.

5. **Increasing Window Size in a Curved Pattern** : f no congestion is detected, Cubic increases the cwnd according to a smooth curve (the cubic function), which grows quickly at first but slows down as it approaches W_{max} .

6. **Slower Growth Near W_{max} :** When cwnd is close to W_{max} , Cubic's growth slows down to avoid hitting network limits, increasing more cautiously.

7. **Further Growth if No Congestion :** If network still not observes congestion, the cwnd continues to grow beyond W_{max} , gradually accelerating as it moves along teh upward curve of the cubic function.

This is how TCP Cubic make full use of available bandwidth while staying stable and fair across different network paths. Cubic algorithm increases its window to be real-time dependent, not RTT dependent.

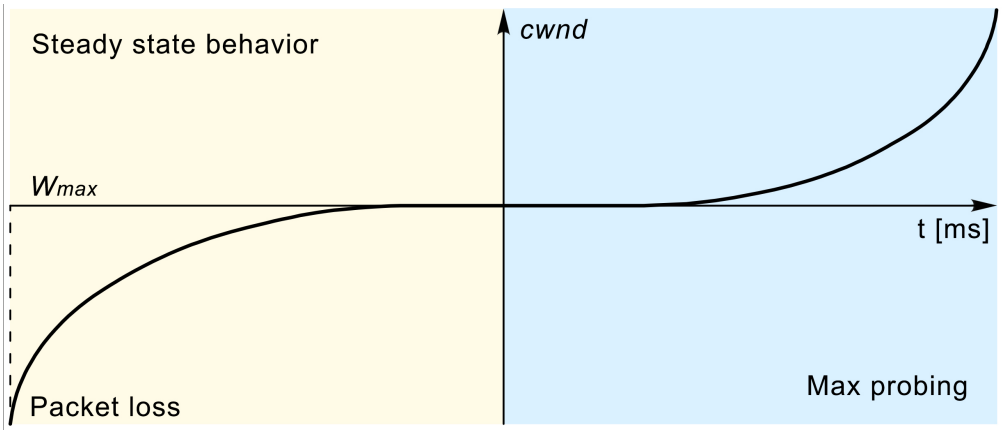

Figure 1: Cubic function

Suitable Scenarios for Cubic

- TCP Cubic is optimized for networks with high bandwidth and long distances, known as high bandwidth-delay products.

- Cubic can fully use available bandwidth and minimizes recovery time after congestion, making it ideal for high-throughput needs.

- Its fast growth enables efficient data transfers within and between data centers where low latency and high throughput are critical.

- Cubic is suitable when maximizing data throughput is a priority in network performance.

Limitations of Cubic

- Cubic's aggressive growth may cause repeated congestion in low-bandwidth environments, reducing efficiency.
- In networks with highly variable latency (like mobile or wireless networks), Cubic may misinterpret delay fluctuations, leading to unnecessary congestion reactions.
- It can show unfairness between flows since it prioritizes throughput.

TCP - New Reno (Loss Based)

New Reno is an enhancement of TCP Reno algorithm. It improves the performance during packet loss recovery, especially when multiple packets are lost. New Reno focuses on improving the **fast recovery phase** by avoiding multiple retransmissions in a single round-trip time (RTT). It introduces a **modified fast recovery** mechanism that allows the sender to recover from multiple lost packets more efficiently.

1. **Slow Start Phase:** New Reno starts with a minimum cwnd which doubles on each congestion-free RTT. However, in most cases, it detects congestion either by hitting ssthresh or with the arrival of three DUPACKs.

2. **Congestion Avoidance Phase:** Once New Reno reaches the slow start threshold (ssthresh), it switches to congestion avoidance, where cwnd increases one segment per RTT.

3. **Fast Retransmit:** At this stage, on receiving three DUPACKs, the retransmission enters a fast retransmit

4. **Update ssthresh:** When loss is detected, ssthresh is updated to half the current value of cwnd.

$$\text{ssthresh} = \frac{cwnd}{2}$$

5. **Fast Recovery:** During fast recovery, New Reno avoids drastic reductions in cwnd. If congestion occurs, cwnd may be decreased but not reset to its initial value.

$$\text{cwnd} = \text{cwnd} - 1$$

6. **Handling Partial Acknowledgments:** If an acknowledgment only confirms some of the outstanding packets (a partial ACK), New Reno retransmits the next unacknowledged packet without leaving the fast recovery phase.

7. **Post-Recovery:** Once lost packets are acknowledged:
   - If cwnd is below ssthresh, New Reno re-enters the slow start phase, growing cwnd exponentially.
   - If cwnd is above ssthresh, it continues in the congestion avoidance phase with linear growth.

Suitable Scenarios for New-Reno

- New Reno is effective in networks with average bandwidth and lower delay, where the probability of losses of multiple packets is present but not at extreme levels.

- New Reno is appropriate for networks of moderate bandwidth and low delay where the possibility of more packet losses is anticipated but not severe.

- New Reno is backward compatible with standard TCP and is useful when fairness with older TCP variants, such as Reno, is a concern.

Limitations of New-Reno

- New Reno uses DUPACKs and conservative retransmission techniques, which delay recovery in high-bandwidth, long-delay networks since it does not scale well with a higher bandwidth-delay product.

- However, being better than Reno is based on struggling with high-loss networks for New Reno as limited acknowledgment-based recovery fails to handle packet losses totally.

- New Reno probably won't perform well under variable latency, as it lacks a delay component based on adaptation to varying RTTs.

TCP Vegas (Delay Based)

Vegas is a network congestion control algorithm which uses network delay as an indicator of congestion. Basically, it tries to detect congestion before packet loss and changes the cwnd. It measures Expected and current throughput whose difference is compared with some minimum and maximum threshold values. On the basis of the comparison we increase, decrease, or don't change the congestion window.
We can find extra data by subtracting actual output of the network from expected output of the network. Then the extra data is compared two threshold values say alpha and beta, and accordingly window size is increased or decreased.

How does it work?

1. **Slow Start Phase :** It starts with a small window size and gradually increases the size of the window until it detect congestion through RTT measurements.

2. **Calculating Actual and Expected Flow Rates :**

$$\text{expectedOutput} = \frac{\text{cwnd}}{\text{BaseRTT}}$$

, where baseRTT = minimum RTT value measured so far.

$$\text{actualOutput} = \frac{\text{cwnd}}{\text{currentRTT}}$$

3. **Computing Extra Data in the Queue :** TCP Vegas computes the amount of extra data in the network queue using the formula:

$$\text{extra\_data} = (\text{expectedOutput} - \text{actualOutput}) \times \text{BaseRTT}$$

This extra data estimate helps TCP Vegas assess how congested the network is and decide whether to increase, decrease, or hold the current cwnd.

4. **Adjusting the Window Size Based on Extra Data :** Vegas uses two thresholds , α and β to make precise adjustments:

Case 1: If the extra data in the network is greater than β, congestion is likely approaching, so **cwnd is decreased :**

$$\text{cwnd} = \text{cwnd} - 1$$

Case 2: If the extra data in the network is less than α, the network has capacity for more data, so **cwnd is increased**:

$$\text{cwnd} = \text{cwnd} - 1$$

Case 3: If the extra data is within the range [α,β], the network is stable, so **cwnd remains unchanged**:

$$\text{cwnd} = \text{cwnd}$$

5. **Proactive Window Adjustment Based on Delay Trends :**By adjusting cwnd before packet loss occurs, TCP Vegas manages congestion more smoothly than traditional loss-based algorithms, reducing retransmissions and keeping data flow steady.

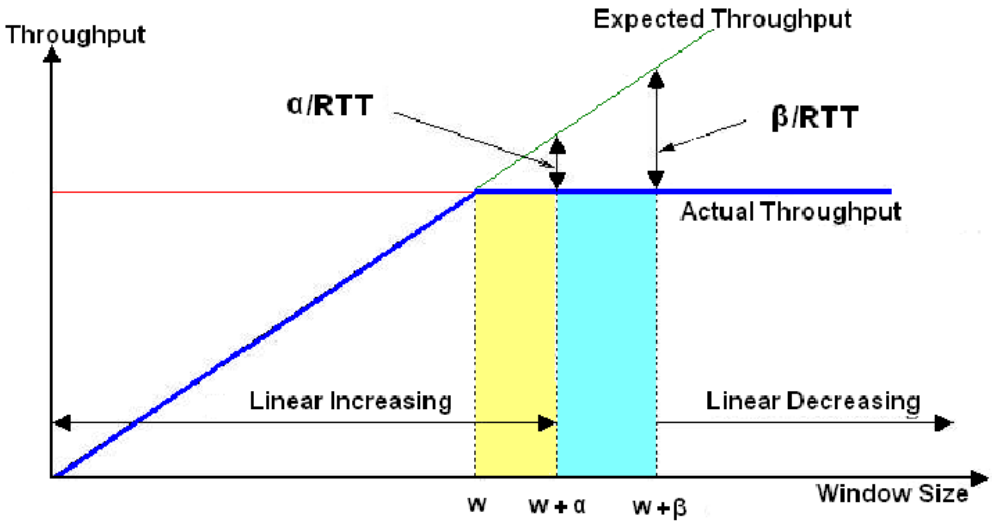

Figure 2: TCP Vegas

Suitable Scenarios for Vegas

- TCP Vegas works best in networks with stable RTTs (like wired LANs) where delay can be accurately measured.
- Since Vegas reduces congestion-induced delays, it's ideal for applications sensitive to delay, like interactive applications or real-time video calls.
- Vegas is effective on networks where RTT doesn't vary significantly and where the connection is low to moderate in bandwidth.

Limitations of Vegas

- In networks with high or fluctuating RTT (e.g., wireless or mobile networks), Vegas may struggle to detect true congestion, as RTT variation can be misinterpreted as a congestion signal.
- TCP Vegas may underutilize available bandwidth compared to algorithms like Cubic, especially in networks with large bandwidth-delay products, as it prioritizes reducing delay over maximizing throughput.

- In shared environments with other loss-based algorithms (like Reno or Cubic), Vegas can be at a disadvantage because it responds to delay rather than packet loss, potentially leading to lower bandwidth allocation.

TCP Veno (Hybrid)

Veno is a hybrid congestion control algorithms, combining loss-based and delay-based approaches. Though originally designed as an enhancement to Reno, it is optimal for networks with variable loss sources like wireless networks. The losses in such networks do not necessarily arise from the fact that there is some congestion in the network but they are mainly caused by errors. Hence, TCP Veno's ability to use delay measurements, in addition to loss detection, improves its handling of congestion. Its behavior adapts to the observed RTT.

How does it work?

1. **Slow Start Phase:** TCP Veno begins with a small congestion window (cwnd) and gradually increases it until it detects congestion through RTT measurements.

2. **Calculating Expected and Actual Throughput:** Veno calculates two key throughput values:

   - **Expected throughput** represents the ideal sending rate based on the minimum RTT (*BaseRTT*):
   $$\text{expectedOutput} = \frac{\text{cwnd}}{\text{BaseRTT}}$$
   - **Actual throughput** represents the observed sending rate based on the current RTT:
   $$\text{actualOutput} = \frac{\text{cwnd}}{\text{currentRTT}}$$

3. **Computing Extra Data in the Network:** Veno estimates the level of network congestion by calculating "extra data" (the difference between expected and actual throughput, scaled by *BaseRTT*):

   $$\text{extra\_data} = (\text{expectedOutput} - \text{actualOutput}) \times \text{BaseRTT}$$

   This value of *extra_data* helps Veno determine the network's congestion level and adjust *cwnd* accordingly.

4. **Adjusting *cwnd* Based on Extra Data:** TCP Veno uses thresholds, α and β, to adjust *cwnd*:

- **Case 1:** If extra_data $> \beta$, congestion is likely. Thus, *cwnd* is decreased:

$$\text{cwnd} = \text{cwnd} - 1$$

- **Case 2:** If extra_data $< \alpha$, the network can handle more data, so *cwnd* is increased:

$$\text{cwnd} = \text{cwnd} + 1$$

- **Case 3:** If $\alpha \leq$ extra_data $\leq \beta$, the network is stable, so *cwnd* remains unchanged:

$$\text{cwnd} = \text{cwnd}$$

5. **Adjusting *ssthresh* and *cwnd* After Packet Loss:** When packet loss is detected, Veno adjusts the slow start threshold (*ssthresh*) and *cwnd* to maintain stability. It updates *ssthresh* to half of the current *cwnd*:

$$\text{ssthresh} = \frac{\text{cwnd}}{2}$$

Then, it temporarily reduces *cwnd* but continues with a more gradual increase based on delay feedback.

Suitable Scenarios for Vegas

- TCP Veno is particularly effective in wireless or error-prone networks where packet loss may occur from link errors rather than congestion.

- It is well-suited for networks with moderate delay fluctuations and mixed traffic where both throughput and stability are essential.

- Ideal for networks that benefit from fine-grained adjustments to congestion, as Veno provides a more adaptive approach to network changes.

Limitations of Vegas

- TCP Veno may not perform optimally in high-bandwidth, low-latency networks, as its reliance on RTT variation for congestion detection may become ineffective.

- It can be slower in adjusting to sudden, high-congestion scenarios compared to purely loss-based TCP variants.

- The hybrid approach with multiple thresholds can introduce complexity, affecting predictability in performance across diverse network conditions.

Comparision of all 4 Algorithms

Feature	TCP CUBIC	New Reno	TCP Vegas	TCP Veno
Base Variant	BIC	TCP Reno	TCP Reno	TCP Reno
Congestion Detection	Loss-based	Loss-based	Delay-based	Hybrid (Loss & Delay)
Network Environment	High-Speed, Long-Delay	High-Speed, Moderate-Delay	Wired Networks	Wired & Wireless Networks
Congestion Control Mechanism	Cubic function of time since last congestion event	Fast retransmit and recovery, exponential growth in slow start	Expected vs. actual throughput estimation	Expected vs. actual throughput with additional feedback
Window Size Growth	Grows quickly after congestion; slows as it approaches W_{max}	Doubles in slow start; linear in congestion avoidance	Increases slowly, based on RTT	Increases cautiously after loss detection
Fairness	Fair across different RTTs	Fairness issues with multiple flows	Fair among connections with different RTTs	Fairness issues with TCP Reno in high-loss environments
Ideal for	High-bandwidth, high-delay connections	High-bandwidth connections with moderate delays	Low to moderate bandwidth, stable connections	Heterogeneous networks with wireless and wired components
Limitations	Not ideal for low bandwidth or high loss rates	Fairness issues with Reno; may underutilize bandwidth	Susceptible to fluctuations in RTT; may be conservative in low bandwidth	Performance degradation in high-loss environments; can be overly cautious

Table 1: Comparison of TCP CUBIC, New Reno, TCP Vegas, and TCP Veno

Part 2 : Understanding TCP Congestion Window using NS-3

PART A

Q1

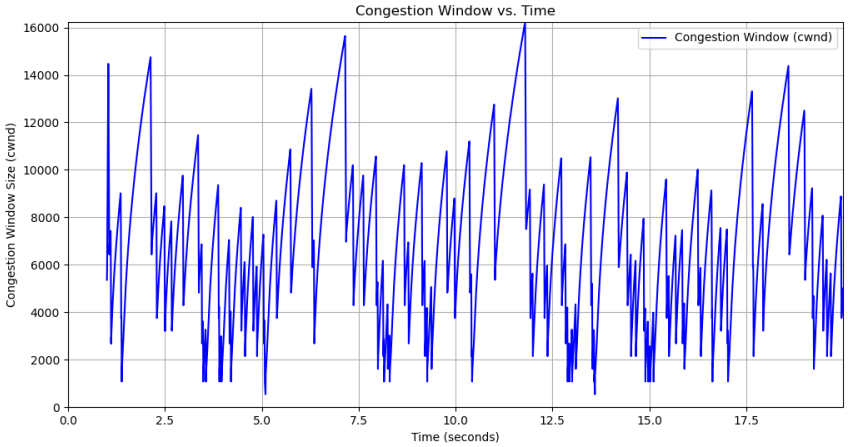

Figure 3: TCP NewReno

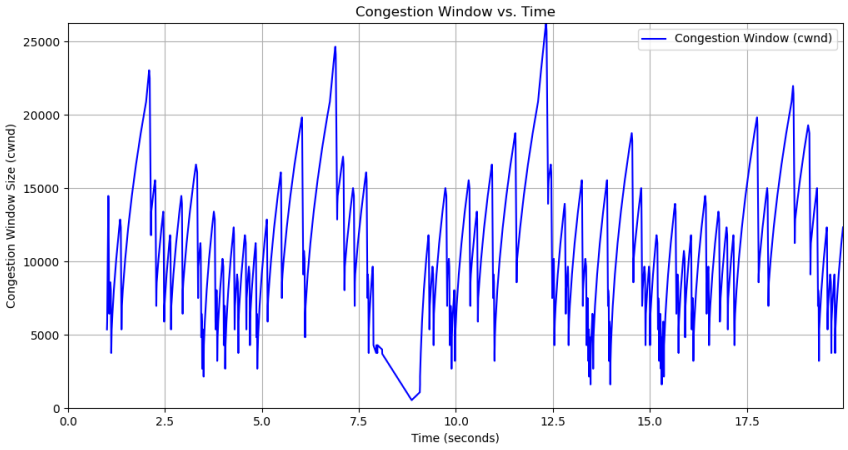

Figure 4: TCP HighSpeed

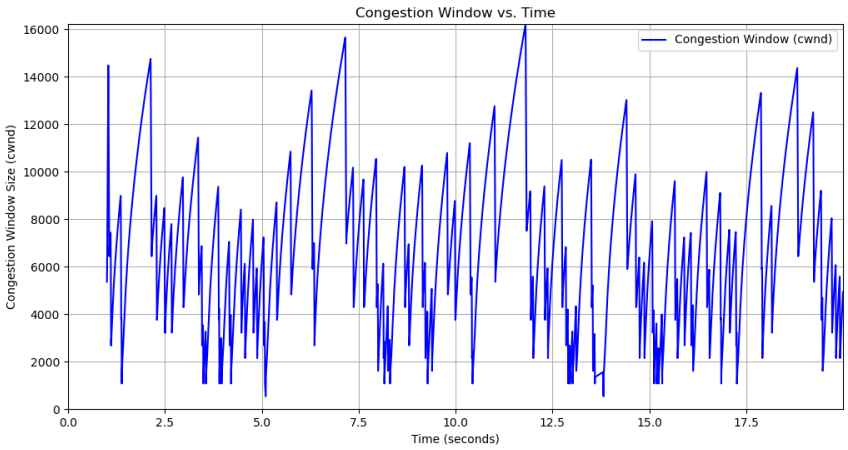

Figure 5: TCP Veno

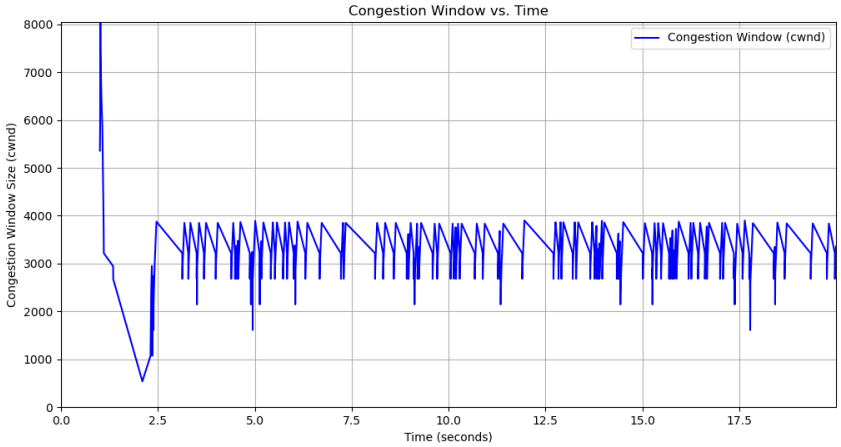

Figure 6: TCP vegas

NewReno: It shows a steep gain in cwnd during slow start, followed by the sawtooth pattern of Congestion avoidance. The congestion window often fluctuates with packet losses and subsequent fast retransmissions.
HighSpeed: It has a similar pattern as NewReno but steeper jumps in cwnd during congestion- tion avoidance, as it aims for high-throughput networks. This leads to more dramatic changes when congestion is detected.
Veno: TCP Veno, which is a variant of TCP Reno, specifically designed to enhance wireless network configurations, starts with a rapid growth of cwnd, marking the entry into the Slow Start phase. Loss- and delay-based. This way, the increase in cwnd is more moderate with adjustments. This results in a stable cwnd with fewer oscillations.
Vegas: Being a delay based protocol, Vegas increases its cwnd more slowly. It is designed to try to avoid congestion altogether. The cwnd is much more stable with less oscillations,

while the initial part of the plot, Vegas reaches a stable cwnd size fairly quickly: it is then that transition from Slow Start to Congestion Avoidance based on early RTT feedback.

Q2

For finding the throughput of the flow, we can go to the pcap file of the sink node and filter the packets which were received at the sink node.
after applying this filter we can go to statistics and then get the throughput.

there is an another way of finding throughput is that we can use **tshark (CLI for wireshark)** which shows us received bytes on the sink node.

Command to print Received bytes at sink node :
tshark -r "pcap-file" -Y "ip.src==10.1.1.1" -T fields -e frame.len | awk 'sum += 1 END print sum'

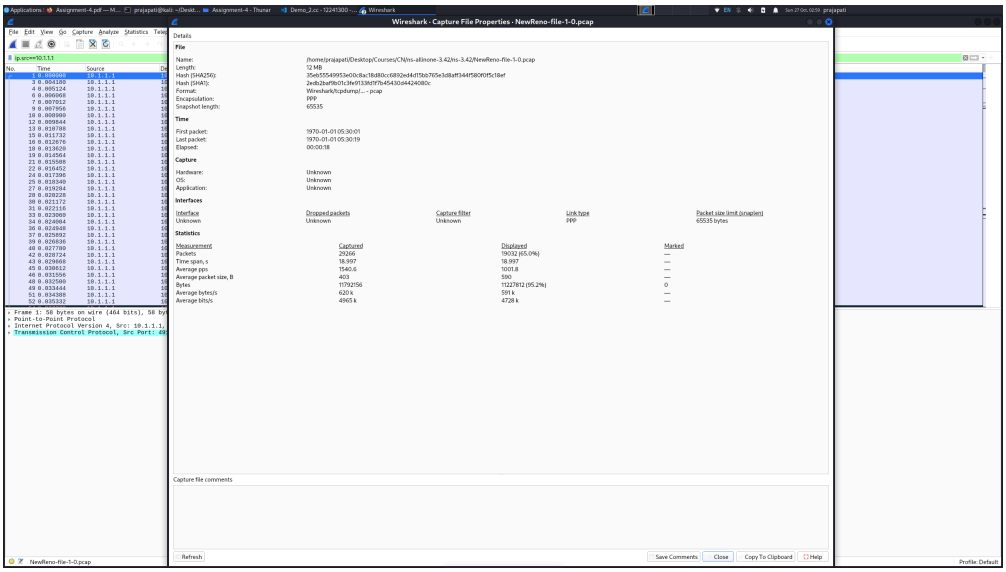

Figure 7: TCP NewReno statistics

```
┌──(prajapati㉿kali)-[~/Desktop/Courses/CN/ns-allinone-3.42/ns-3.42]
└─$ tshark -r NewReno-file-1-0.pcap -Y "ip.src==10.1.1.1" -T fields -e frame.len | awk '{sum += $1} END {print sum}'
11227812
```

Figure 8: Received bytes on the server side in TCP NewReno

Bytes received = 11227812
Throughput (TCP NewReno) : 4728 kBps

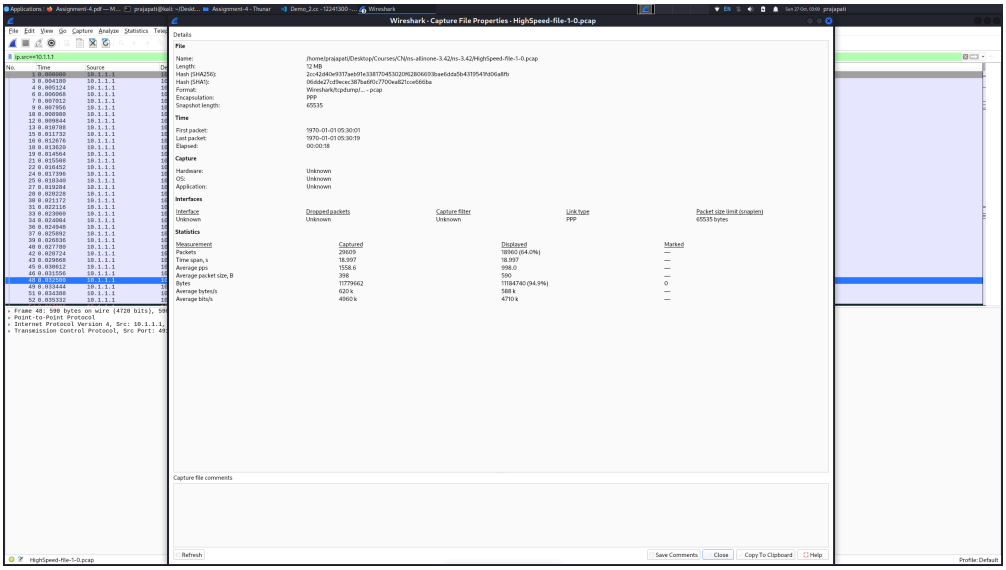

Figure 9: TCP HighSpeed statistics

```
┌──(prajapati㉿kali)-[~/Desktop/Courses/CN/ns-allinone-3.42/ns-3.42]
└─$ tshark -r HighSpeed-file-1-0.pcap -Y "ip.src==10.1.1.1" -T fields -e frame.len | awk '{sum += $1} END {print sum}'
11184740
```

Figure 10: Received bytes on the server side in TCP HighSpeed

Bytes received = 11184740
Throughput (TCP HighSpeed) : 4710 kBps

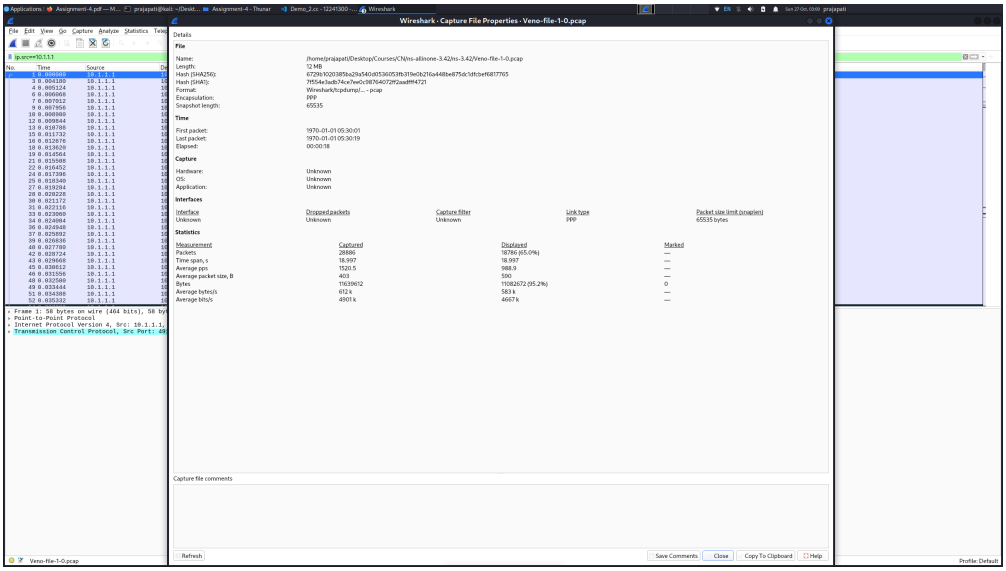

Figure 11: TCP Veno statistics

```
┌──(prajapati㉿kali)-[~/Desktop/Courses/CN/ns-allinone-3.42/ns-3.42]
└─$ tshark -r Veno-file-1-0.pcap -Y "ip.src==10.1.1.1" -T fields -e frame.len | awk '{sum += $1} END {print sum}'
11082672
```

Figure 12: Received bytes on the server side in TCP Veno

Bytes received = 11082672
Throughput (TCP Veno) : 4667 kBps

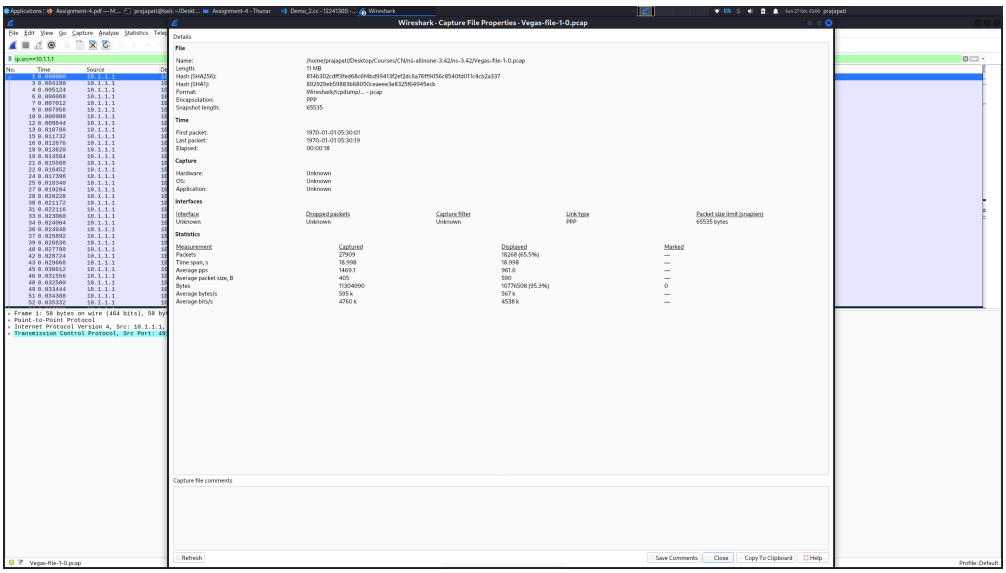

Figure 13: TCP Vegas statistics

```
┌──(prajapati㉿kali)-[~/Desktop/Courses/CN/ns-allinone-3.42/ns-3.42]
└─$ tshark -r Vegas-file-1-0.pcap -Y "ip.src==10.1.1.1" -T fields -e frame.len | awk '{sum += $1} END {print sum}'
10776508
```

Figure 14: Received bytes on the server side in TCP Vegas

Bytes received = 10776508
Throughput (TCP Vegas) : 4538 kBps

Q3

```
┌──(prajapati㉿kali)-[~/Desktop/Courses/CN/ns-allinone-3.42/ns-3.42]
└─$ awk '{if ($3 < $2) count++} END {print "Total cwnd reductions:", count}' NewReno.cwnd
Total cwnd reductions: 991

┌──(prajapati㉿kali)-[~/Desktop/Courses/CN/ns-allinone-3.42/ns-3.42]
└─$ awk '{if ($3 < $2) count++} END {print "Total cwnd reductions:", count}' HighSpeed.cwnd
Total cwnd reductions: 1513

┌──(prajapati㉿kali)-[~/Desktop/Courses/CN/ns-allinone-3.42/ns-3.42]
└─$ awk '{if ($3 < $2) count++} END {print "Total cwnd reductions:", count}' Veno.cwnd
Total cwnd reductions: 971

┌──(prajapati㉿kali)-[~/Desktop/Courses/CN/ns-allinone-3.42/ns-3.42]
└─$ awk '{if ($3 < $2) count++} END {print "Total cwnd reductions:", count}' Vegas.cwnd
Total cwnd reductions: 244
```

Figure 15: count of decrement in Cwnd for all the algorithms

TCP Algorithm	**Total cwnd Reductions**
New Reno	991
HighSpeed	1513
Veno	971
Vegas	244

Table 2: Total Congestion Window Reductions for Different TCP Algorithms

In the table shown above, we can see that Highspeed has so many reductions in cwnd because it uses aggressive behaviour for cwnd increase.

NewReno and Veno reduces cwnd in almost same range.

Vegas has the least reductions in cwnd because it anticipates congestion through RTT measurements.

Q4

```
| DataRate | Delay | Throughput (Bytes/sec) | TcpNewReno |
Parsed Bytes: 8653052
| 5Mbps | 5ms | Throughput : 3.46 Mbps |
Parsed Bytes: 10315082
| 10Mbps | 5ms | Throughput : 4.12 Mbps |
Parsed Bytes: 11227812
| 5Mbps | 2ms | Throughput : 4.49 Mbps |
Parsed Bytes: 17862152
| 10Mbps | 2ms | Throughput : 7.14 Mbps |
```

Figure 16: Effect of datarate and throughput change in TCPNewReno

```
| DataRate | Delay | Throughput (Bytes/sec) | TcpHighSpeed |
Parsed Bytes: 10797110
| 5Mbps | 5ms | Throughput : 4.31 Mbps |
Parsed Bytes: 15662758
| 10Mbps | 5ms | Throughput : 6.26 Mbps |
Parsed Bytes: 11184740
| 5Mbps | 2ms | Throughput : 4.47 Mbps |
Parsed Bytes: 20507838
| 10Mbps | 2ms | Throughput : 8.20 Mbps |
```

Figure 17: Effect of datarate and throughput change in TCPHighSpeed

```
| DataRate | Delay | Throughput (Bytes/sec) | TcpVeno |
Parsed Bytes: 8779312
| 5Mbps | 5ms | Throughput : 3.51 Mbps |
Parsed Bytes: 11037208
| 10Mbps | 5ms | Throughput : 4.41 Mbps |
Parsed Bytes: 11082672
| 5Mbps | 2ms | Throughput : 4.43 Mbps |
Parsed Bytes: 18755032
| 10Mbps | 2ms | Throughput : 7.50 Mbps |
```

Figure 18: Effect of datarate and throughput change in TCPVeno

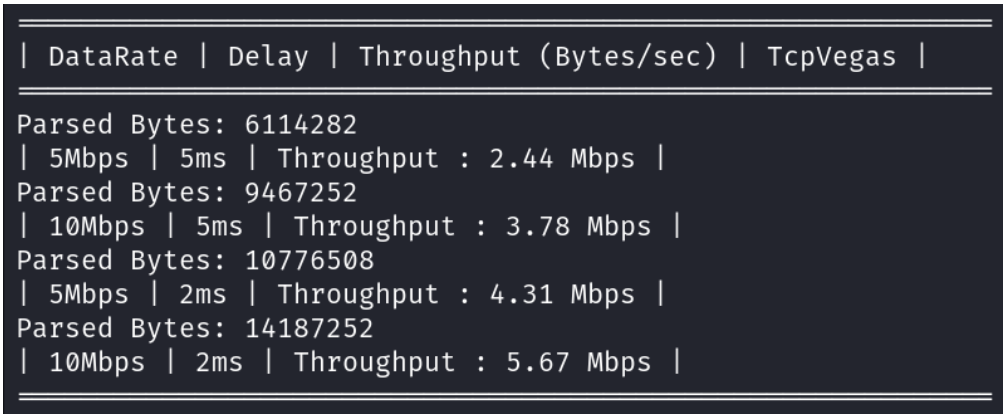

Figure 19: Effect of datarate and throughput change in TCPVegas

From the above results, we can say that Increasing datarate and decreasing delay help us getting better throughput. For all the algorithms, we found this to be true.

Q5

Larger MTU Size:

- **Higher Throughput:** A larger MTU allows us to transfer more data per packet by reducing the overhead from headers and enabling higher throughput, especially beneficial on high-speed networks.
- **Increased Latency and Fragmentation Risk:** Large packets take much longer to send than small ones, which may introduce delay, especially over slower networks. Even if all the links in the path have a lower MTU, packets would need to be fragmented or dropped, and thereby retransmissions may also be possible.

Smaller MTU Size:

- **Lower Latency:** Smaller packets bring down the time of transmission and prove handy for applications requiring low latency, like image download, as smaller packets take less time to process.
- **Increased Overhead:** While sending data in smaller packets may lead to better performance due to increased reliability and reduced round-trip delay, there are overheads from the header for each packet, thereby reducing efficiency and throughput on high-bandwidth networks.

Q6

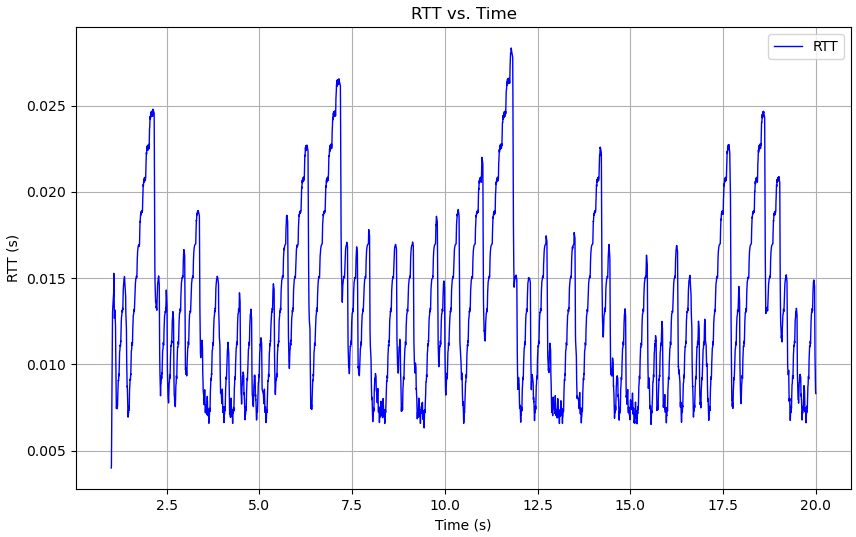

Figure 20: TCP NewReno

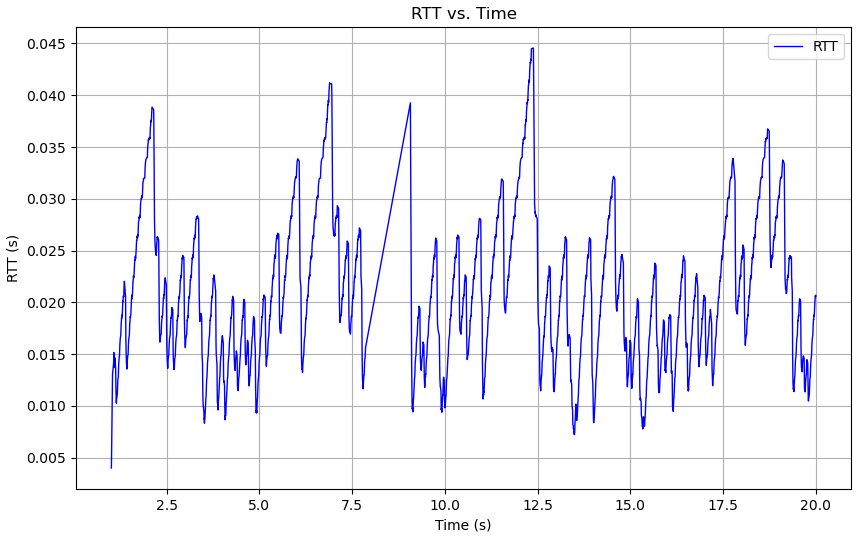

Figure 21: TCP HighSpeed

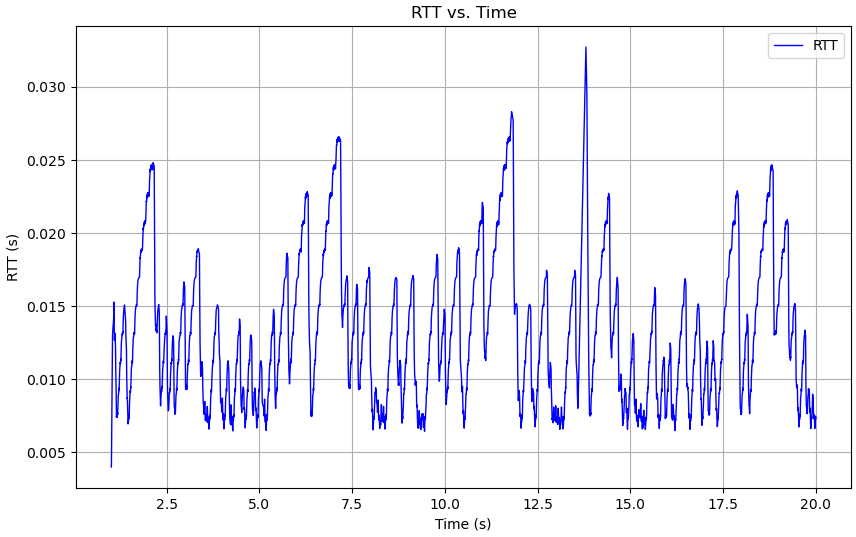

Figure 22: TCP veno

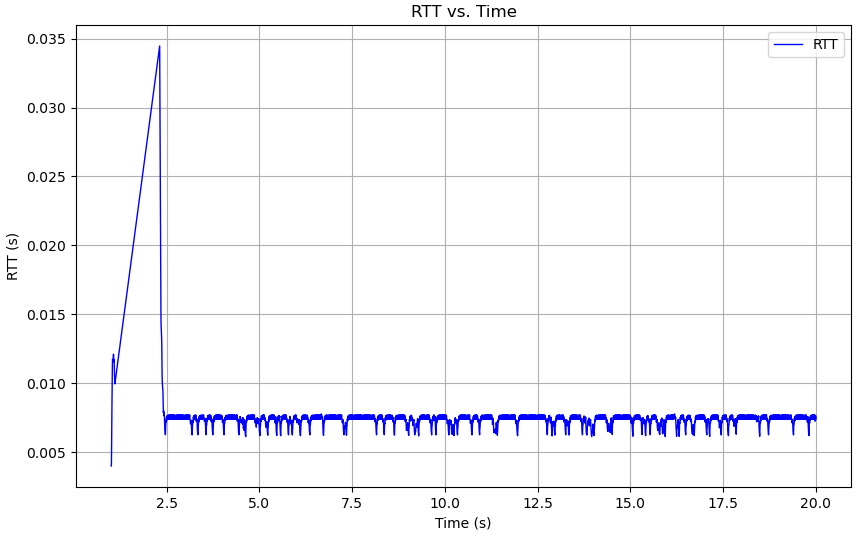

Figure 23: TCP Vegas

In above four graphs we can see that RTT for TCP Vegas is being found in a specific range of teh time since it is delay-based algorithm.
other algorithms are either hybrid or loss based , hence, we find more fluctuations in those algorithms.

PART B

Note : For any question of the part-B refer tcp-final.cc file only. Each part was done by making very small changes in the code of tcp-final.cc

(part-a)

To create our customCCA (in my case, it is TcpHarsh) , first we have to add tcp-harsh.cc and tcp-harsh.h file into CMakeLists.txt file in which we are supposed to implement a new CCA by replacing slowStart phase of TCP NewReno with TcpHystart method of TCP Cubic algorithm.

- Create **tcp-harsh.cc** and **tcp-harsh.h** file in **src/internet/model/** directory.
- Add both the files in **CMakeLists.txt** file which is present in **src/internet/** directory
- Since we have to improve performance of TcpNewReno CCA, I added the same codes of **tcp-congestion-ops.h** (Header file of NewReno) and **tcp-congestion-ops.cc** (NewReno CCA).
- In the next sections of this question, we will focus on improving TcpNewReno's performance.

(part-b)

To improve performance of TCPNewReno, I replaced the SlowStart phase of TcpNewReno with TcpHystart method of TCP Cubic algorithm.

Reference : Look at the file **tcp-harsh.cc**.

Below images will show how i implemented Hystart in TcpHarsh algorithm.
there are some explicit Hystart functions I implemented in TcpHarsh which are **HystartReset**, **HystartDelayThresh** and **HystartUpdate** and some other functions which I implemented from the reference of tcp-cubic.cc file.

```
Time
TcpHarsh::HystartDelayThresh (const Time& t) const
{
  Time ret = t;
  if (t > m_hystartDelayMax)
  {
    ret = m_hystartDelayMax;
  }
  else if (t < m_hystartDelayMin)
  {
    ret = m_hystartDelayMin;
  }
  return ret;
}

void
TcpHarsh::HystartUpdate (Ptr<TcpSocketState> tcb, const Time& delay)
{
  if (!m_found)
  {
    Time now = Simulator::Now ();
    if ((now - m_lastAck) <= m_hystartAckDelta)
    {
      m_lastAck = now;
      if ((now - m_roundStart) > m_delayMin)
      {
        m_found = true;
      }
    }

    if (m_sampleCnt < m_hystartMinSamples)
    {
      m_currRtt = std::min (m_currRtt, delay);
      ++m_sampleCnt;
    }
    else if (m_currRtt > m_delayMin + HystartDelayThresh (m_delayMin))
    {
      m_found = true;
    }

    if (m_found)
    {
      tcb->m_ssThresh = tcb->m_cWnd;
      NS_LOG_DEBUG ("Exiting Slow Start due to Hystart");
    }
  }
}
```

Figure 24: HyStart Functions implemented in TcpHarsh

TCPHarsh :: HystartDelayThresh

- HystartDelayThresh is the TCP congestion algorithm that regulates the delay in value, t, given within specific thresholds.
- It defines the minimum threshold given as m_hystartDelayMin and the maximum threshold as m_hystartDelayMax for the same delay.

- If the delay, t, attains the limit of m_hystartDelayMax, then it keeps at this value and exceeds no further.
- It ensures that if the delay is more than m_hystartDelayMax, then it will be limited to m_hystartDelayMax. In other words, it will treat this minimum delay.
- The function makes sure that the delay within the defined limits is kept when TCP Slow Start is under HyStart phase.
- These constraints of the function ensure a less congested and smoother starting phase of TCP connections.

TCPHarsh :: HystartUpdate

- The TCP Harsh Slow Start Extension HystartUpdate method speeds up the TCP Slow Start by examining round-trip time (RTT) and delays over a period of time for smoother transition to Congestion Avoidance.
- The first step is to determine the Slow Start Retrieval if it is already considered (m_found).
- If at a given interval (m_hystartAckDelta), there is an updating of the last acknowledged time (m_lastAck), extending the expiration probability of the old one to 0) in, for example, AIMS (Artificial Intelligence Markup Map lack of the quality)
- If the time from the beginning of the new round until now is larger than the minimum delay threshold (m_delayMin), then it computes the exit conditions.
- The version history contains the amount of samples (m_sampleCnt) and the measured m_currRtt until the desired sample size (m_hystartMinSamples) is received.
- If the RTT is bigger than the m_delayMin + HystartDelayThresh(m_delayMin), the system sets m_found as true.
- Every when the conditions are satisfied, the congestion window (m_cWnd) label is saved as the Slow Start threshold (m_ssThresh) which is the last phase of the Slow Start period before the slow & steady wind-down.

```
void
TcpHarsh::PktsAcked(Ptr<TcpSocketState> tcb, uint32_t segmentsAcked, const Time& rtt)
{
    NS_LOG_FUNCTION(this << tcb << segmentsAcked << rtt);

    /* Discard delay samples right after fast recovery */
    // if (m_epochStart != Time::Min() && (Simulator::Now() - m_epochStart) < m_cubicDelta)
    // {
    //     return;
    // }

    /* first time call or link delay decreases */
    if (m_delayMin == Time::Min() || m_delayMin > rtt)
    {
        m_delayMin = rtt;
    }

    /* hystart triggers when cwnd is larger than some threshold */
    if (m_hystart && tcb->m_cWnd <= tcb->m_ssThresh &&
        tcb->m_cWnd >= m_hystartLowWindow * tcb->m_segmentSize)
    {
        HystartUpdate(tcb, rtt);
    }
}

void
TcpHarsh::CongestionStateSet(Ptr<TcpSocketState> tcb, const TcpSocketState::TcpCongState_t newState)
{
    NS_LOG_FUNCTION(this << tcb << newState);

    if (newState == TcpSocketState::CA_LOSS)
    {
        m_found = false;
        HystartReset(tcb);
        m_delayMin = Time::Min();
    }
}

uint32_t
TcpHarsh::GetSsThresh (Ptr<const TcpSocketState> tcb, uint32_t bytesInFlight)
{
    // Sets the slow start threshold to half the current bytes in flight
    return std::max(2 * tcb->m_segmentSize, bytesInFlight / 2 );
}
```

Figure 25: HyStart Functions implemented in TcpHarsh

PktsAcked: This function deals with the acknowledgments for packets received. Based on the minimum delay values m_delayMin, an update may be triggered for the HyStart if the congestion window (m_cWnd) lies in some certain thresholds.

CongestionStateSet: This updates the TCP congestion state, sets HyStart parameters and min delay on a loss state or CA_LOSS state.

GetSsThresh: This function calculates the slow start threshold. It returns the maximum value between twice the segment size and half of the current bytes in flight.

```cpp
void
TcpHarsh::IncreaseWindow (Ptr<TcpSocketState> tcb, uint32_t segmentsAcked)
{
  if (tcb->m_cWnd < tcb->m_ssThresh)
  {
    if (m_hystart && tcb->m_lastAckedSeq > m_endSeq)
    {
      HystartReset (tcb);
    }
    tcb->m_cWnd += segmentsAcked * tcb->m_segmentSize;
    segmentsAcked = 0;
  }
  else
  {
    CongestionAvoidance (tcb, segmentsAcked);
  }
}


void TcpHarsh::CongestionAvoidance(Ptr<TcpSocketState> tcb, uint32_t segmentsAcked) {
  NS_LOG_FUNCTION(this << tcb << segmentsAcked);

    if (segmentsAcked > 0)
    {
        double adder =
            static_cast<double>(tcb->m_segmentSize * tcb->m_segmentSize) / tcb->m_cWnd.Get();
        adder = std::max(1.0, adder);
        tcb->m_cWnd += static_cast<uint32_t>(adder);
        NS_LOG_INFO("In CongAvoid, updated to cwnd " << tcb->m_cWnd << " ssthresh "
                                                     << tcb->m_ssThresh);
    }
}
```

Figure 26: Some General Functions of TcpNewReno implemented in TcpHarsh

IncreaseWindow: Increase the window size as a function of acknowledged segments. Transition to congestion avoidance when the window size exceeds the slow start threshold.

Congestion Avoidance : It is implemented using congestion control protocols to alter the available windows for transmitting given certain data in a transmitted congestion.

(part-c & part-e)

To evaluate the performance of an algorithm, TcpHarsh, We will take teh baseline of TcpNewReno algortihm's output and then compare results of both the algorithms.

```
uint32_t port = 8080;

for(uint i = 0; i < nflows; i++) {
    if(i % 2 == 0)
        CreateTcpFlowWithCcaAndTracing(leftNodes.Get(i),
                                       rightNodes.Get(i),
                                       rightInterfaces.GetAddress(i),
                                       port + i,
                                       "ns3::TcpNewReno",
                                       "NewReno"+std::to_string(i));
    else if(i % 2 == 1)
        CreateTcpFlowWithCcaAndTracing(leftNodes.Get(i),
                                       rightNodes.Get(i),
                                       rightInterfaces.GetAddress(i),
                                       port + i,
                                       "ns3::TcpNewReno",
                                       "NewReno"+std::to_string(i));
}
```

Figure 27: Logic behind assigning different algorithms to different flows

Above code snippet of the file tcp-final.cc used to assign different Congestion Control Algorithm to different flows.

```
void
CreateTcpFlowWithCcaAndTracing(Ptr<Node> srcNode,
                               Ptr<Node> destNode,
                               Ipv4Address destAddress,
                               uint16_t port,
                               std::string ccaTypeId,
                               std::string traceFilePrefix)
{
    // Configure CCA for the socket
    Config::Set("/NodeList/" + std::to_string(srcNode->GetId()) + "/$ns3::TcpL4Protocol/SocketType",
                StringValue(ccaTypeId));

    Address sinkAddress = InetSocketAddress(destAddress, port);

    PacketSinkHelper packetSinkHelper("ns3::TcpSocketFactory",
                                      InetSocketAddress(Ipv4Address::GetAny(), port));

    ApplicationContainer sinkApps = packetSinkHelper.Install(destNode); // sinking on node 1

    sinkApps.Start(Seconds(0.));
    sinkApps.Stop(Seconds(20.));

    Ptr<Socket> ns3TcpSocket = Socket::CreateSocket(srcNode, TcpSocketFactory::GetTypeId());

    Ptr<MyApp> app = CreateObject<MyApp>();
    app->Setup(ns3TcpSocket, sinkAddress, 1040, 1000000, DataRate("100Mbps"));
    srcNode->AddApplication(app);
    app->SetStartTime(Seconds(1.));
    app->SetStopTime(Seconds(20.));

    AsciiTraceHelper asciiTraceHelper; // this is for the cwnd
    Ptr<OutputStreamWrapper> stream = asciiTraceHelper.CreateFileStream(traceFilePrefix + ".cwnd");
    ns3TcpSocket->TraceConnectWithoutContext("CongestionWindow",
                                             MakeBoundCallback(&CwndChange, stream));

    Ptr<OutputStreamWrapper> stream4 =
        asciiTraceHelper.CreateFileStream(traceFilePrefix + "_rtt.txt");
    ns3TcpSocket->TraceConnectWithoutContext("RTT", MakeBoundCallback(&RttChange, stream4));

    Ptr<OutputStreamWrapper> stream35 =
        asciiTraceHelper.CreateFileStream(traceFilePrefix + "-ssthresh.txt");
    ns3TcpSocket->TraceConnectWithoutContext("SlowStartThreshold",
                                             MakeBoundCallback(&Ssthresh, stream35));
}
```

Figure 28: function which defines the flow in our topology

The funciton of the above image implement the particular CCA to the flow and sets Tracing of some standard metrics of the flow.

case 1 : apply TcpNewReno on all the flows

```
┌──(prajapati㉿kali)-[~/Desktop/Courses/CN/ns-allinone-3.42/ns-3.42]
└─$ ./ns3 run "scratch/12241300/tcp-final.cc --nflows=8"
[0/2] Re-checking globbed directories...
[2/3] Linking CXX executable /home/prajapati/Desktop/Courses/CN/ns-allin
FlowID: 1 (TCP 10.1.1.1 / 49153 ⟶ 10.2.1.1 / 8080)
  Throughput: 0.563065 Mbps
FlowID: 2 (TCP 10.1.2.1 / 49153 ⟶ 10.2.2.1 / 8081)
  Throughput: 0.5596 Mbps
FlowID: 3 (TCP 10.1.3.1 / 49153 ⟶ 10.2.3.1 / 8082)
  Throughput: 0.666884 Mbps
FlowID: 4 (TCP 10.1.4.1 / 49153 ⟶ 10.2.4.1 / 8083)
  Throughput: 0.644104 Mbps
FlowID: 5 (TCP 10.1.5.1 / 49153 ⟶ 10.2.5.1 / 8084)
  Throughput: 0.561273 Mbps
FlowID: 6 (TCP 10.1.6.1 / 49153 ⟶ 10.2.6.1 / 8085)
  Throughput: 0.537047 Mbps
FlowID: 7 (TCP 10.1.7.1 / 49153 ⟶ 10.2.7.1 / 8086)
  Throughput: 0.558925 Mbps
FlowID: 8 (TCP 10.1.8.1 / 49153 ⟶ 10.2.8.1 / 8087)
  Throughput: 0.654142 Mbps
```

Figure 29: Throughput when all the flows using TcpNewReno

case 2 : apply TcpHarsh on all the flows

```
┌──(prajapati㉿kali)-[~/Desktop/Courses/CN/ns-allinone-3.42/ns-3.42]
└─$ ./ns3 run "scratch/12241300/tcp-final.cc --nflows=8"
[0/2] Re-checking globbed directories...
ninja: no work to do.
FlowID: 1 (TCP 10.1.1.1 / 49153 ⟶ 10.2.1.1 / 8080)
  Throughput: 0.58372 Mbps
FlowID: 2 (TCP 10.1.2.1 / 49153 ⟶ 10.2.2.1 / 8081)
  Throughput: 0.590394 Mbps
FlowID: 3 (TCP 10.1.3.1 / 49153 ⟶ 10.2.3.1 / 8082)
  Throughput: 0.584595 Mbps
FlowID: 4 (TCP 10.1.4.1 / 49153 ⟶ 10.2.4.1 / 8083)
  Throughput: 0.661455 Mbps
FlowID: 5 (TCP 10.1.5.1 / 49153 ⟶ 10.2.5.1 / 8084)
  Throughput: 0.582793 Mbps
FlowID: 6 (TCP 10.1.6.1 / 49153 ⟶ 10.2.6.1 / 8085)
  Throughput: 0.57822 Mbps
FlowID: 7 (TCP 10.1.7.1 / 49153 ⟶ 10.2.7.1 / 8086)
  Throughput: 0.581399 Mbps
FlowID: 8 (TCP 10.1.8.1 / 49153 ⟶ 10.2.8.1 / 8087)
  Throughput: 0.583904 Mbps
```

Figure 30: Throughput when all the flows using TcpHarsh

In the above two cases, we can clearly see that in most of the flows throughput in the case 2 (using TcpHarsh) is higher than TcpNewReno flows. Refer the table below for better comparision.

Table 3: Comparison of Throughput for TcpNewReno and TcpHarsh

FlowID	**TcpNewReno Throughput (Mbps)**	**TcpHarsh Throughput (Mbps)**	**Difference (Mbps)**
1	0.563065	0.58372	0.020655
2	0.559600	0.590394	0.030794
3	0.666884	0.584595	-0.082289
4	0.644104	0.661455	0.017351
5	0.561273	0.582793	0.021520
6	0.537047	0.578220	0.041173
7	0.558925	0.581399	0.022474
8	0.654142	0.583904	-0.070238

Hence we can say that our custom algorithm (TcpHarsh) is working better than TcpNewReno in terms of Throughput.

Now, let us take a look at cwnd changes , ssthresh changes and rtt changes of both the algorithms.

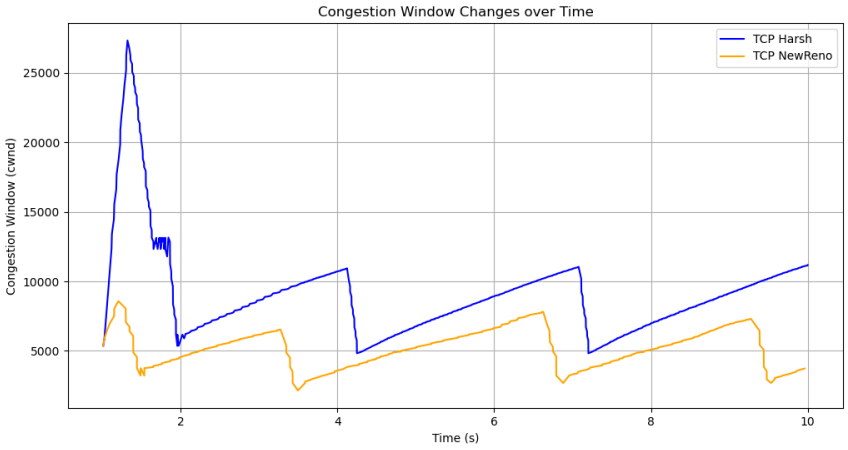

Figure 31: Trace of Cwnd

In the above graph we can clearly see that after implementing TcpHyStart in custom algorithm (TcpHarsh), Cwnd of TcpHarsh is always higher than TcpNewReno. This shows that we are able to send more data using TcpHarsh with comparision of TcpNewReno.

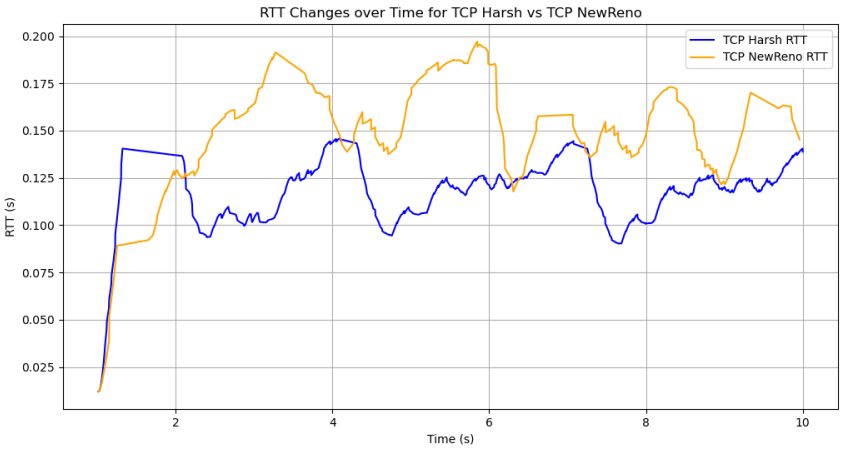

Figure 32: Trace of RTT

RTT values for most of the packets for the flow using TcpHarsh is found to be lesser than the flow which are using TcpNewReno. Hence, we can consider that TcpHarsh is performing better than TcpNewReno.

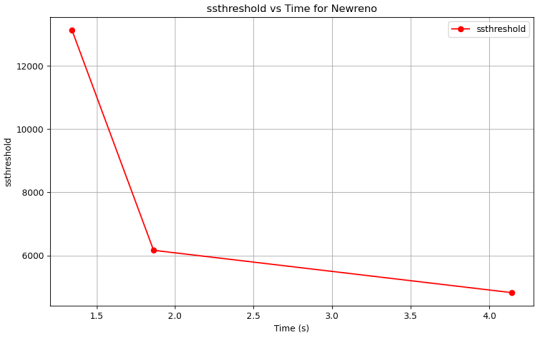

Figure 33: Harsh SsThresh

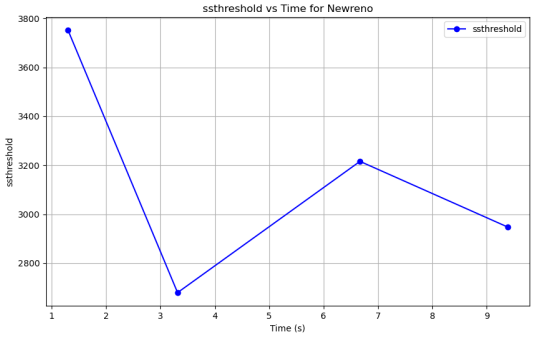

Figure 34: NewReno SsThresh

Figure 35: SsThresh trace of both the algorithms

Above graph shows that how the SsThresh changes in teh flows using TcpHarsh and TcpNewReno.

Further justification and evaluation points are mentioned in the f part.

part-d

```
double
CalculateJainsFairnessIndex(const std::vector<double>& throughputs)
{
    double sum = 0.0;
    double sumSquare = 0.0;

    for (double throughput : throughputs)
    {
        sum += throughput;
        sumSquare += throughput * throughput;
    }

    double n = throughputs.size();

    return (sum * sum) / (n * sumSquare);
}
```

Figure 36: Function for calculating Jain's Fairness index

```
std::map<FlowId, FlowMonitor::FlowStats> stats = monitor->GetFlowStats();
std::vector<double> throughputs;

for (auto& flow : stats)
{
    double throughput = flow.second.rxBytes * 8.0 /
                        (flow.second.timeLastRxPacket.GetSeconds() -
                         flow.second.timeFirstTxPacket.GetSeconds()) /
                        1000000; // Mbps
    throughputs.push_back(throughput);
}

// Calculate Jain's Fairness Index
double jainsFairnessIndex = CalculateJainsFairnessIndex(throughputs);
std::cout << "Jain's Fairness Index: " << jainsFairnessIndex << std::endl;
```

Figure 37: Calculating Jain's Fairness Index

Above functions are showing us that how we were able to calculate the Jain's Fairness index from the Stats of teh flows we monitored using Flowmon.
Note : Here, I considered all the flows (both uplink flows and downlink flows) for the calculation of Fairness index. That's why it will give index value around 0.5xxxxx. If I would have considered only downlink flows towards sink then it would have given the index value near 1.

```
uint32_t port = 8080;

for(uint i = 0; i < nflows; i++) {
    if(i % 2 == 0)
        CreateTcpFlowWithCcaAndTracing(leftNodes.Get(i),
                                       rightNodes.Get(i),
                                       rightInterfaces.GetAddress(i),
                                       port + i,
                                       "ns3::TcpNewReno",
                                       "NewReno"+std::to_string(i));
    else if(i % 2 == 1)
        CreateTcpFlowWithCcaAndTracing(leftNodes.Get(i),
                                       rightNodes.Get(i),
                                       rightInterfaces.GetAddress(i),
                                       port + i,
                                       "ns3::TcpHarsh",
                                       "Harsh"+std::to_string(i));
}
```

Figure 38: Code for assigning different CCAs to half of the flows

```
FlowID: 8 (TCP 10.2.4.1 / 8083 ⟶ 10.1.4.1 / 49153)
  Throughput: 0.0630139 Mbps
Total throughput of System: 0.62184 Mbps
Total packets transmitted: 14714
Total packets received: 14565
Total packets dropped: 149
Packet Lost Ratio: 0.0101264
Jain's Fairness Index: 0.546441
```

Figure 39: all 4 - newreno

```
FlowID: 8 (TCP 10.2.4.1 / 8083 ⟶ 10.1.4.1 / 49153)
  Throughput: 0.0614675 Mbps
Total throughput of System: 0.624252 Mbps
Total packets transmitted: 14965
Total packets received: 14798
Total packets dropped: 167
 Packet Lost Ratio: 0.0111594
Jain's Fairness Index: 0.550872
```

Figure 40: 2 - newreno , 2 - Harsh

Figure 41: Fairness index change in 4 flow topology

```
FlowID: 16 (TCP 10.2.8.1 / 8087 ⟶ 10.1.8.1 / 49153)
  Throughput: 0.0329391 Mbps
Total throughput of System: 0.311509 Mbps
Total packets transmitted: 14888
Total packets received: 14705
Total packets dropped: 183
Packet Lost Ratio: 0.0122918
Jain's Fairness Index: 0.546075
```

Figure 42: all 8 - newreno

```
FlowID: 16 (TCP 10.2.8.1 / 8087 ⟶ 10.1.8.1 / 49153)
  Throughput: 0.0363582 Mbps
Total throughput of System: 0.313417 Mbps
Total packets transmitted: 15306
Total packets received: 15093
Total packets dropped: 213
Packet Lost Ratio: 0.0139161
Jain's Fairness Index: 0.553059
```

Figure 43: 4 - newreno , 4 - Harsh

Figure 44: Fairness index change in 8 flow topology

```
FlowID: 32 (TCP 10.2.16.1 / 8095 ⟶ 10.1.16.1 / 49153)
  Throughput: 0.0152191 Mbps
Total throughput of System: 0.156231 Mbps
Total packets transmitted: 15149
Total packets received: 14873
Total packets dropped: 276
Packet Lost Ratio: 0.018219
Jain's Fairness Index: 0.549866
```

Figure 45: all 16 - newreno

```
FlowID: 32 (TCP 10.2.16.1 / 8095 ⟶ 10.1.16.1 / 49153)
  Throughput: 0.0161694 Mbps
Total throughput of System: 0.15637 Mbps
Total packets transmitted: 15230
Total packets received: 14937
Total packets dropped: 293
Packet Lost Ratio: 0.0192383
Jain's Fairness Index: 0.552189
```

Figure 46: 8 - newreno , 8 - Harsh

Figure 47: Fairness index change in 16 flow topology

```
FlowID: 40 (TCP 10.2.20.1 / 8099 ⟶ 10.1.20.1 / 49153)
  Throughput: 0.0125676 Mbps
Total throughput of System: 0.125241 Mbps
Total packets transmitted: 15352
Total packets received: 15013
Total packets dropped: 339
Packet Lost Ratio: 0.0220818
Jain's Fairness Index: 0.549174
```

Figure 48: all 20 - newreno

```
FlowID: 40 (TCP 10.2.20.1 / 8099 ⟶ 10.1.20.1 / 49153)
  Throughput: 0.0139716 Mbps
Total throughput of System: 0.125379 Mbps
Total packets transmitted: 15433
Total packets received: 15083
Total packets dropped: 350
Packet Lost Ratio: 0.0226787
Jain's Fairness Index: 0.552823
```

Figure 49: 10 - newreno , 10 - Harsh

Figure 50: Fairness index change in 20 flow topology

Table 4: Fairness Index Comparison for TcpNewReno and Your CCA

Protocol	4 Flows	8 Flows	16 Flows	20 Flows
Both CCAs	0.550872	0.553059	0.552189	0.552823
TCP NewReno	0.546441	0.546075	0.549866	0.549174

(part-f)

Our primary objective was to enhace teh TCP-SlowStart by implementing TCP-Hystart in our CCA for better exit points from slowstart phase.This approach aims to reduce congestion by avoiding abrupt increases in the congestion window (`cwnd`), allowing for a more adaptive and stable transition to the congestion avoidance phase.

Below are the observation and justification for the whole experiment.

1. Throughput Analysis

Observation: Throughput analysis showed that flows using TCP HyStart had consistently higher throughput than those using TCP NewReno.
Justification: The intelligent delay-based exit points in HyStart reduce packet loss in the Slow Start phase, leading to a smoother growth of the congestion window. It, therefore, leads to higher throughput because TCP HyStart does not prematurely leave the Slow Start phase just because of packet loss; it is a limitation of NewReno's approach.

2. Congestion Window Evolution (Cwnd)

Observation: The congestion window changes in TCP HyStart were smoother and more stable, while in TCP NewReno, abrupt increases were followed by drops due to packet loss.
Justification: TCP HyStart prevents the congestion window from overshooting in the Slow Start phase by using RTT and delay thresholds, thus ensuring a controlled and smoother growth. This stability in cwnd growth reduces the chances of inducing network congestion, thus maintaining higher and more consistent throughput. NewReno's purely loss-based mechanism lacks this adaptive response, leading to sharp drops in cwnd upon detecting packet loss, which disrupts throughput.

3. Slow Start Threshold (ssthresh)

Observation: TCP HyStart adjusts the ssthresh much more adaptively than NewReno in response to delay-based conditions, while NewReno sets the ssthresh after suffering packet loss.
Justification: The delay-based threshold in TCP HyStart enables more intelligent boundary setting for Slow Start, transitioning to congestion avoidance before excessive congestion builds up. This proactive adjustment results in fewer retransmissions and a more efficient utilization of network resources, whereas NewReno's loss-based ssthresh setting is more reactive, often reducing ssthresh after network congestion has already impacted throughput.

4. RTT Variation

Observation: HyStart TCP demonstrated a more stable RTT behavior over the time scale compared to TCP NewReno, which provided higher RTT variability.
Reasoning: Smaller RTT variations of TCP HyStart are a result of the controlled cwnd growth along with congestion detection by means of RTT monitoring. HyStart thus prevents unrequired retransmissions as well as packet loss which leads to fewer spikes of RTT; this means less variation of RTT as against TCP NewReno whose dependence on packet loss leads to higher variations in the form of bursts in RTT due to retransmission.

Table 5: Comparative Summary of TCP NewReno and TCP HyStart Performance

Metric	**TCP NewReno**	**TCP HyStart**	**TCP Harsh**
Throughput	Lower, due to reactive loss-based exit	Higher, due to proactive delay-based exit	Increased by adaptive exit points
Congestion Window Stability	Abrupt increases and drops	Smoother, stable growth	Fewer retransmissions, stable performance
Slow Start Threshold (ssthresh)	Set after packet loss	Set based on delay observations	Reduced packet loss events
RTT Variation	High, with noticeable spikes	Lower, more stable	Minimizes congestion-induced delay