CSE322: Computer Networks Sessional
NS-3 Assignment on Transport Layer

Ananya Shahrin Promi
ID: 2005079

16 December 2024

Introduction

Here,

$$x = 2005079$$

$$(x + 3)\%17 = 0$$

$$(x^2 + 12)\%17 = 4$$

Hence, I need to work with the two variants of congestion control `TcpNewReno` and `TcpHtcp`.

Here,

$$\texttt{num\_flows} = 1 + x^3\%7 = 7$$

$$\texttt{bandwith} = 1 + x\%3 = 3\text{Mbps}$$

Analysis of TcpHtcp

1. Congestion Window (`cwnd`)

The `cwnd` metric represents the congestion window size, which controls the maximum amount of data the sender can send before receiving an acknowledgment.

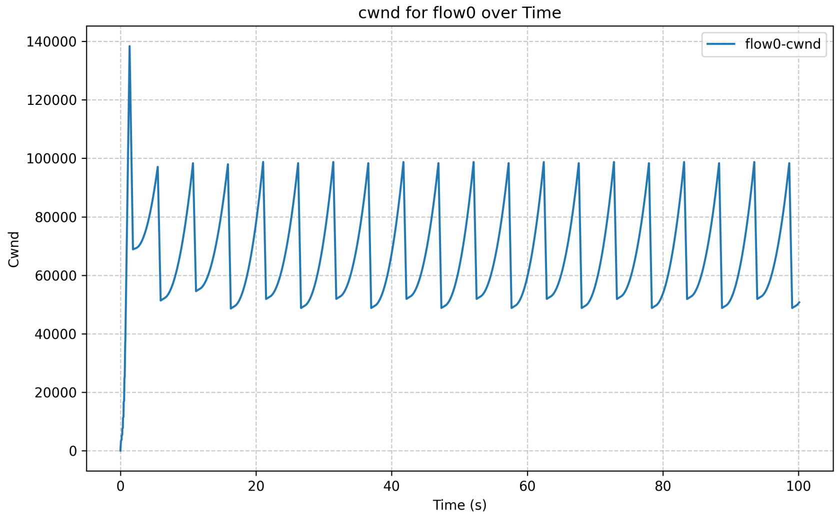

Figure 1: `cwnd` for Flow0 over Time

Graph Observation

The graph shows a steady increase in `cwnd` during the congestion avoidance phase, with possible sharp drops during congestion events (e.g., packet loss or timeout).

Justification

The `cwnd` growth is governed by the `CongestionAvoidance` method in `TcpHtcp`:

```
double adder = static_cast<double>(((tcb->m_segmentSize * tcb->m_segmentSize)
    + (tcb->m_cWnd * m_alpha)) / tcb->m_cWnd);
adder = std::max(1.0, adder);
tcb->m_cWnd += static_cast<uint32_t>(adder);
```

During congestion, `cwnd` is reduced, influenced by `m_beta` in the `GetSsThresh` method:

```
uint32_t ssThresh = std::max(segWin, bFlight);
```

2. Bytes in Flight (`inflight`)

Represents the number of bytes currently in transit (unacknowledged data).

Graph Observation

The `inflight` graph closely follows `cwnd`.

Justification

`inflight` depends directly on `cwnd` and the rate of acknowledgments:

```
auto bFlight = static_cast<uint32_t>(bytesInFlight * m_beta);
uint32_t ssThresh = std::max(segWin, bFlight);
```

During congestion, `ssthresh` reduces, limiting the `cwnd` and subsequently reducing `inflight`. The alignment of `inflight` with `cwnd` in the graph reflects the direct relationship between these metrics.

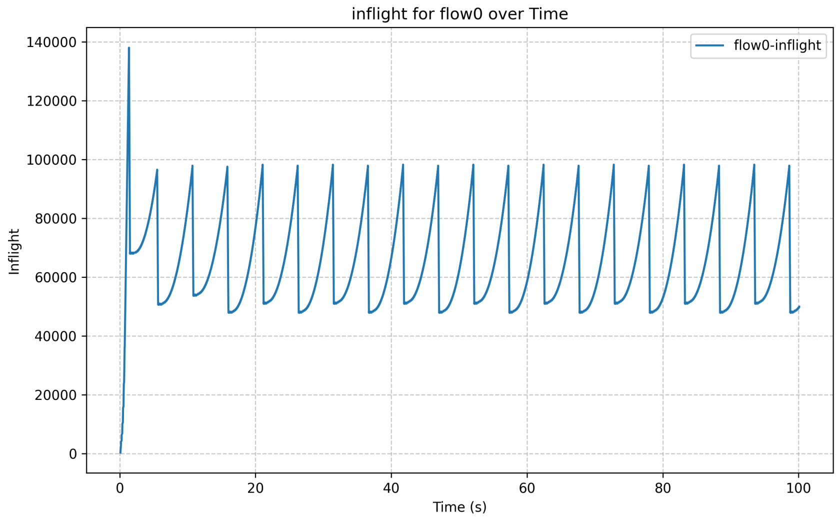

Figure 2: `inflight` for the Flow0 over Time

3. Next Expected Receive Sequence (`next-rx`)

Tracks the next expected sequence number to be received by the receiver.

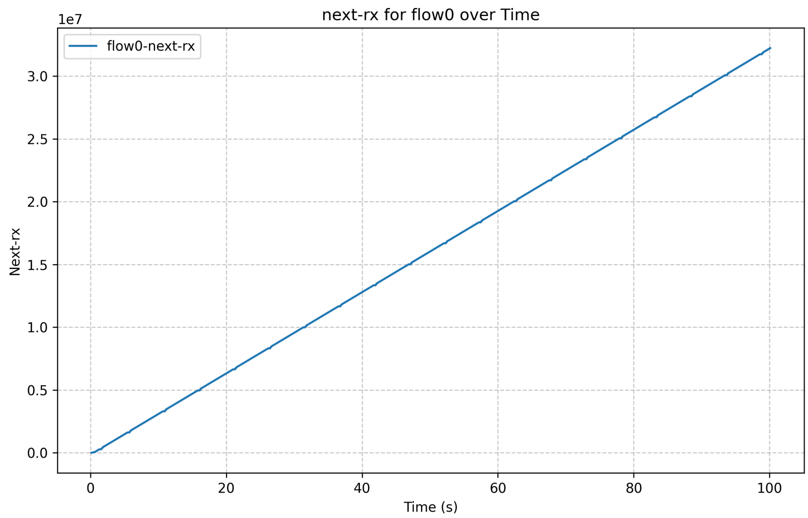

Figure 3: `next-rx` for Flow0 over Time

Graph Observation

The `next-rx` graph shows a steady linear increase, reflecting the orderly receipt of packets.

Justification

`next-rx` is indirectly influenced by `PktsAcked`, which processes acknowledgments.

```
if (tcb->m_congState == TcpSocketState::CA_OPEN)
    m_dataSent += segmentsAcked * tcb->m_segmentSize;
```

4. Next Expected Transmit Sequence (`next-tx`)

Represents the next sequence number the sender will transmit.

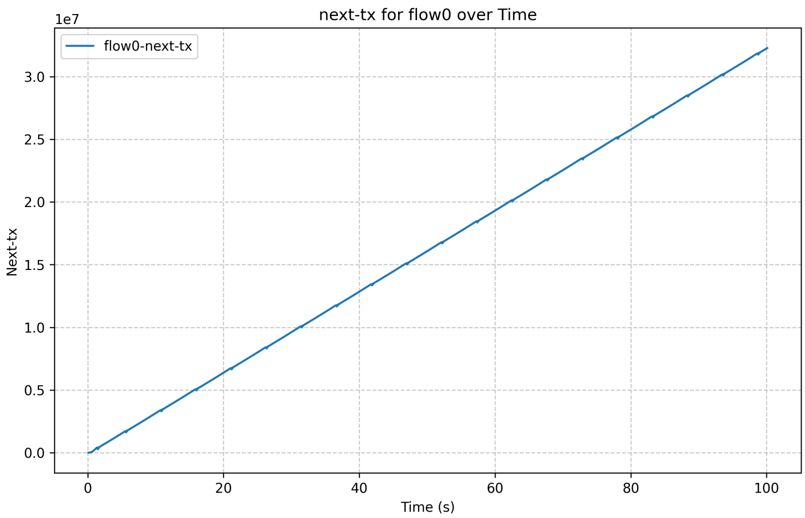

Figure 4: `next-tx` for Flow0 over Time

Graph Observation

The `next-tx` graph grows steadily during congestion avoidance and reset or pause during congestion events.

Justification

`next-tx` advances based on `cwnd`:

```
double adder = ((tcb->m_segmentSize * tcb->m_segmentSize)
                + (tcb->m_cWnd * m_alpha)) / tcb->m_cWnd;
adder = std::max(1.0, adder);
tcb->m_cWnd += static_cast<uint32_t>(adder);
```

5. Retransmission Timeout (`rto`)

The timeout duration after which unacknowledged packets are retransmitted.

Graph Observation

The `rto` graph decreases over time, reflecting improved network conditions as `rtt` variability reduces. There are no spikes, suggesting minimal congestion or packet loss during the simulation.

Justification

The `rto` calculation depends on `rtt` measurements:

```
if (rtt < m_minRtt) m_minRtt = rtt;
if (rtt > m_maxRtt) m_maxRtt = rtt;
```

In `TcpHtcp`, `rtt` stabilization directly impacts `rto`. As `rtt` fluctuations diminish, the `rto` value is adjusted downward, signifying better network stability. A consistent decrease in `rto` indicates that congestion events are infrequent or completely absent.

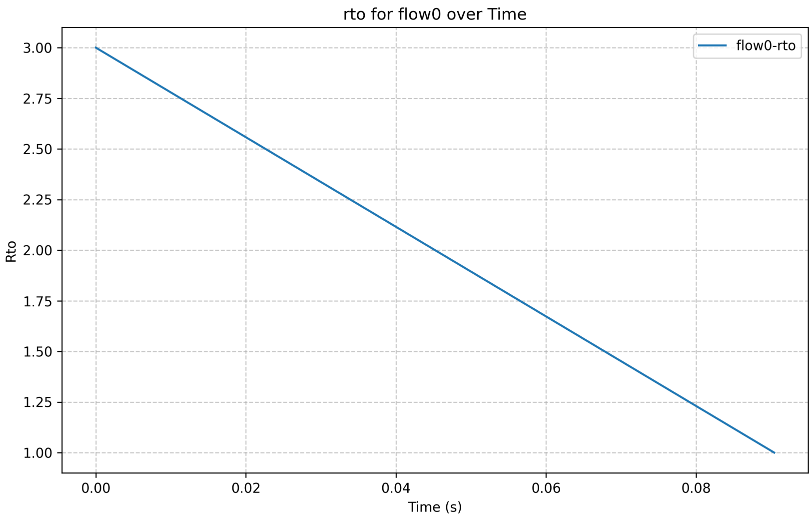

Figure 5: `rto` for Flow0 over Time

6. Round-Trip Time (`rtt`)

Measures the time taken for a packet to travel to the receiver and back.

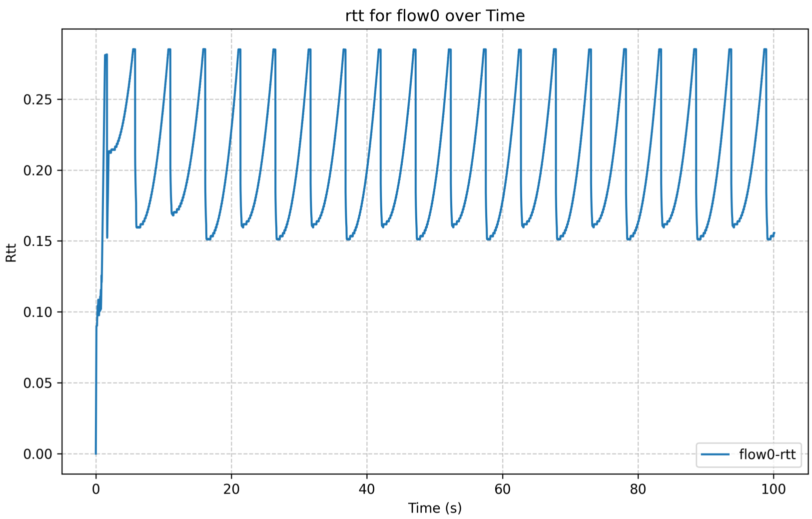

Figure 6: `rtt` for Flow0 over Time

Graph Observation

The `rtt` graph typically oscillates based on network conditions, with spikes indicating congestion or queuing delays.

Justification

The `rtt` bounds are updated in `PktsAcked`:

```
if (rtt < m_minRtt) m_minRtt = rtt;
if (rtt > m_maxRtt) m_maxRtt = rtt;
```

7. Slow Start Threshold (`ssth`)

The threshold between slow start and congestion avoidance phases.

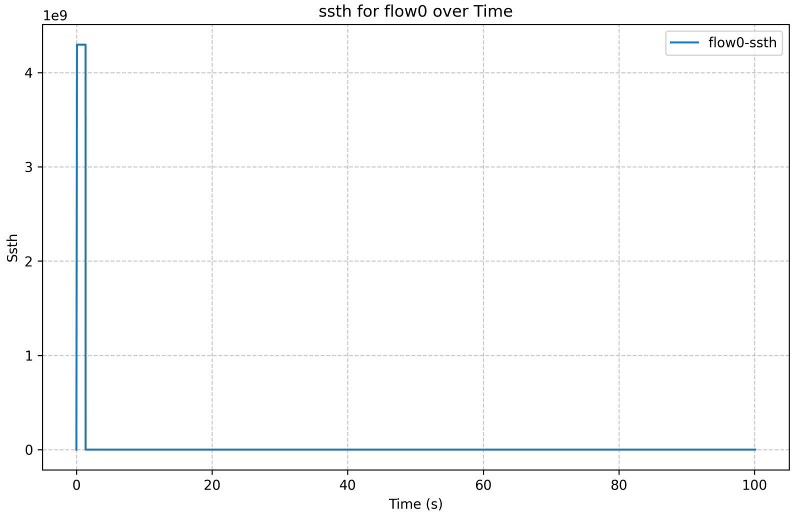

Figure 7: `ssth` for Flow0 over Time

Graph Observation

At the start of the simulation, `ssth` is initialized and sharply decreases during the first congestion event, forming the square shape. Afterward, `ssth` remains at zero, indicating that no further updates to the threshold occur, likely due to a transition into congestion avoidance or specific simulation parameters.

Justification

In TcpHtcp, `ssth` is updated when a congestion event occurs (e.g., packet loss or timeout):

```
uint32_t ssThresh = std::max(segWin, bFlight);
```

Initially, `ssth` is set to a large value. During congestion, it is sharply reduced to throttle the sending rate and control the network load.

Analysis of TcpNewReno

1. Congestion Window (`cwnd`)

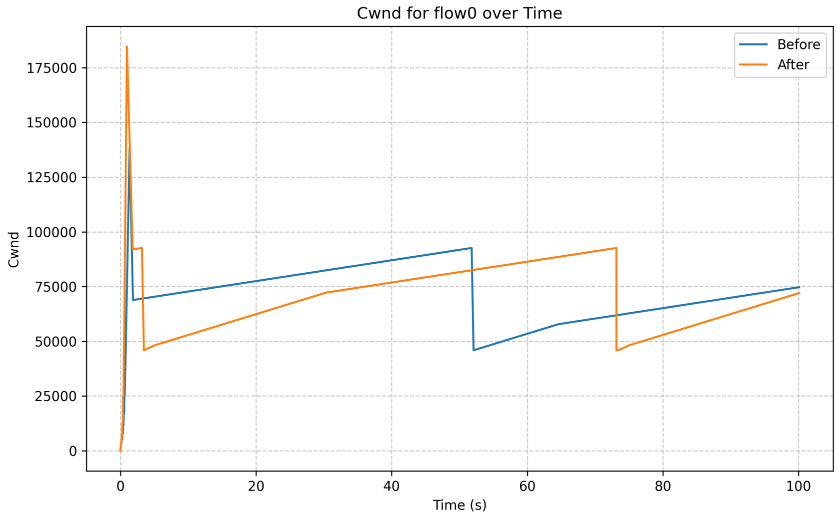

Figure 8: `cwnd` for Flow0 over Time

Graph Observation

Before: A sharp exponential growth during slow start, linear growth during congestion avoidance after reaching `ssth`, sudden drops in `cwnd` indicate congestion or loss events.
After: The exponential growth in slow start is smoother and tapers off around 75% of `ssth`. Faster linear growth in congestion avoidance leads to a higher `cwnd` overall.

Justification:

In the tweaked code, `cwnd` growth slows down near 75% of `ssth`:

```
while (segmentsAcked > 0 && tcb->m_cWnd < tcb->m_ssThresh)
{
    tcb->m_cWnd += tcb->m_segmentSize;
    --segmentsAcked;
    if (tcb->m_cWnd > 0.75 * tcb->m_ssThresh) break;
}
```

The tweak multiplies the increment factor:

```
double adder = static_cast<double>(tcb->m_segmentSize * tcb->m_segmentSize)
                / tcb->m_cWnd.Get();
adder = std::max(1.0, adder * 1.25);
tcb->m_cWnd += static_cast<uint32_t>(adder);
```

2. Bytes in Flight (`inflight`)

Graph Observation

Before: Matches `cwnd` during growth phases, with sudden drops correlating with `cwnd` reductions.
After: It also tracks `cwnd` with occasional drops during congestion events.

Justification:

inflight mirrors `cwnd` behavior. The same snippets for `SlowStart` and `CongestionAvoidance` apply.

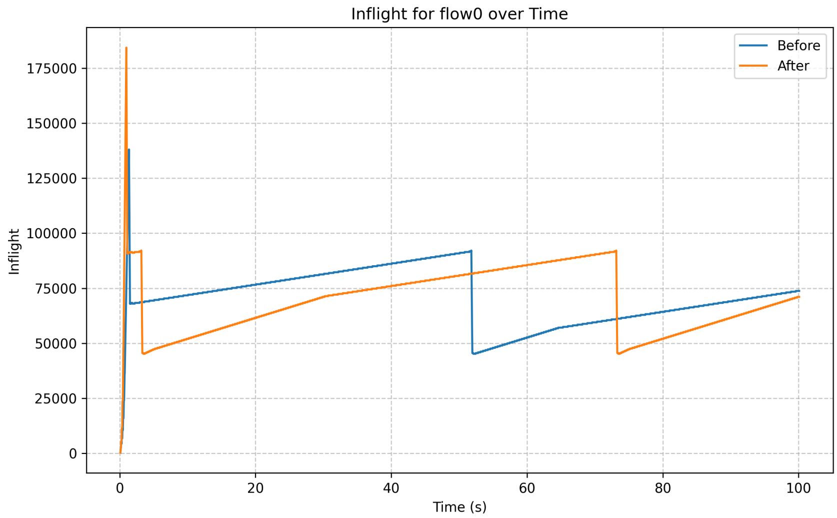

Figure 9: `inflight` for Flow0 over Time

3. Next Receive Sequence Number (`next-rx`)

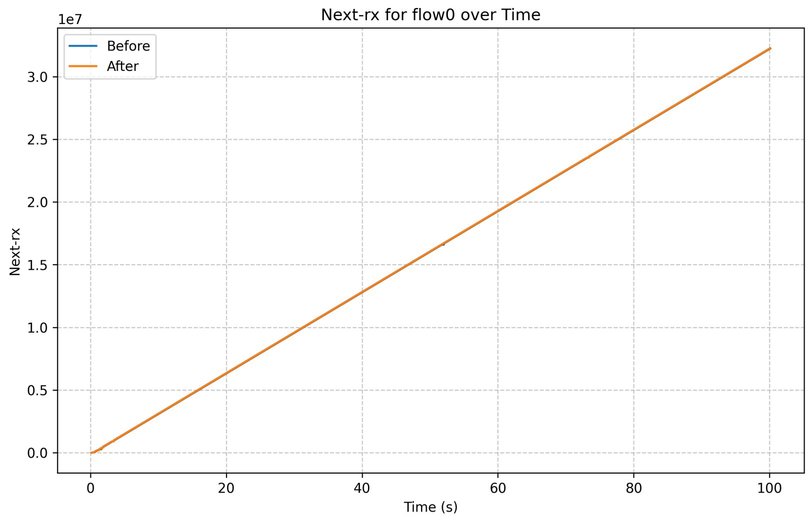

Figure 10: `next-rx` for Flow0 over Time

Graph Observation

Before: Progresses steadily.
After: Same as before.

Justification:

This growth accelerates packet sending, reflected in the sequence numbers:

```
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);
```

4. Next Transmit Sequence Number (`next-tx`)

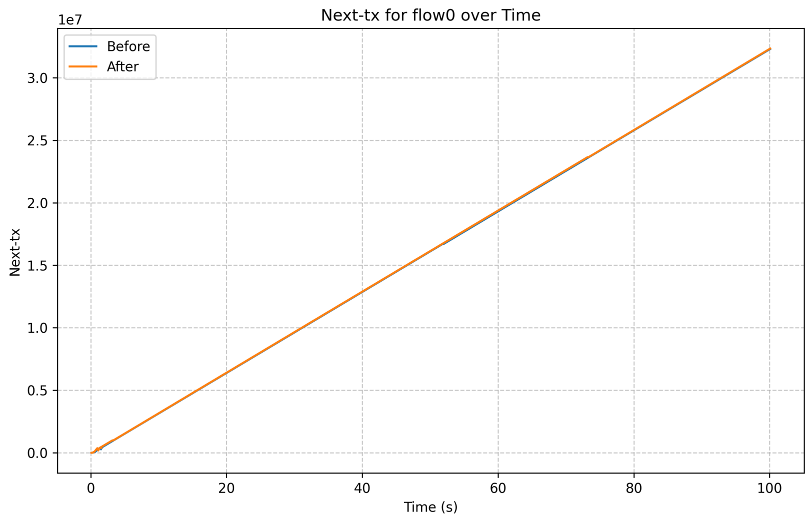

Figure 11: `next-tx` for Flow0 over Time

Graph Observation

Before: Progresses steadily.
After: Same as before.

Justification:

As with `next-rx`, `next-tx` behavior is tied to `cwnd` growth. Faster congestion avoidance means more data segments transmitted.

5. Retransmission Timeout (`rto`)

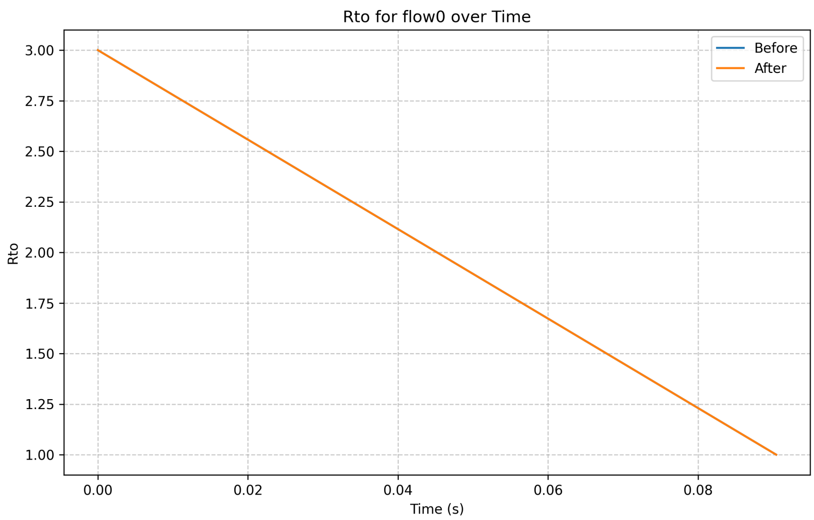

Figure 12: `rto` for Flow0 over Time

Graph Observation

Before: Decreases over time.
After: Same as before.

Justification:

The gradual ramp-up of the congestion window during slow start prevents queue buildup and packet drops early in the transmission phase. This leads to a more stable `rtt`, which is directly used to calculate `rto`. A stable `rtt` results in smaller variations and thus a lower `rto` value over time. With fewer retransmissions, the RTO graph decreases steadily.

6. Round-Trip Time (rtt)

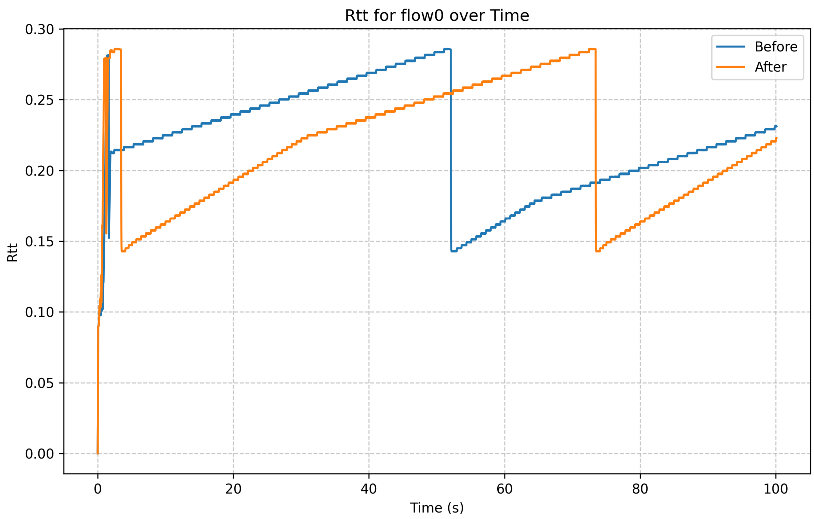

Figure 13: `rtt` for Flow0 over Time

Graph Observation

Before: Shows spikes during aggressive slow start and congestion events.
After: Lower initial `rtt` values due to smoother slow start. Slight increases later align with higher bandwidth usage.

Justification:

Lower `rtt` during slow start is due to the controlled ramp-up:

```
if (tcb->m_cWnd > 0.75 * tcb->m_ssThresh) break;
```

Slight increases in `rtt` later stem from increased queueing due to faster linear growth:

```
adder = std::max(1.0, adder * 1.25);
```

7. Slow-Start Threshold (ssth)

Graph Observation

Before: After initialization, the graph sharply decreases during the first congestion event, forming the square.
After: More stability, indicating fewer congestion-triggered reductions.

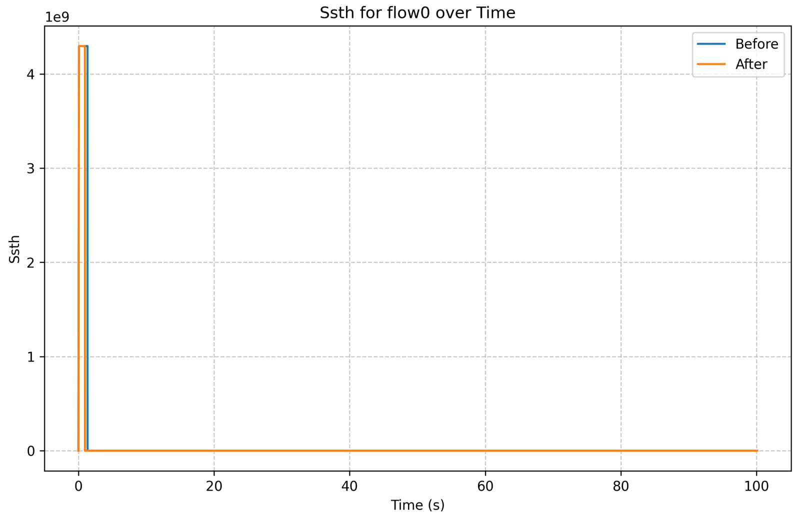

Figure 14: `ssth` for Flow0 over Time

Justification:

By preventing overshooting `ssth`, the tweaks reduce the likelihood of congestion:

```
if (tcb->m_cWnd > 0.75 * tcb->m_ssThresh) break;
```

The default behavior for setting `ssth` remains unchanged:

```
return std::max(2 * state->m_segmentSize, bytesInFlight / 2);
```

The stability of the tweaked curve reflects better handling of slow start transitions.

Imporved Congestion Control

The reduced frequency of changes in `cwnd` (congestion window), `inflight` bytes, and `rtt` in the tweaked version is a strong indication of improved congestion control.

1. Smoother Behavior Indicates Stability

The smoother growth of `cwnd` and `inflight` bytes reflects a more gradual and predictable adjustment to the network capacity. This indicates better stability, where the sender adapts without overloading the network.

2. Lower `rtt` Variability is Better for Applications

`rtt` variability occurs due to unstable queues in the network, caused by rapid changes in `cwnd` or `inflight` data. Lower `rtt` variability ensures consistent latency, which is critical for latency-sensitive applications such as voice over IP (VoIP), video streaming, online gaming.

3. Improved Handling of Congestion Signals

Reduced frequency of drops indicates proactive reduction in the congestion window before severe congestion occurs and fewer packet losses and retransmissions, improving throughput and delay.

4. Efficient Bandwidth Utilization

A stable congestion window allows efficient utilization of network capacity without overloading it. It reduces retransmissions caused by congestion-induced packet loss and maintains a stable state in the network.

5. Less Aggressive, More Responsive Control

Good congestion control strikes a balance between being aggressive enough to fully utilize available bandwidth and being responsive enough to congestion signals, reducing the sending rate when needed.