**Congestion Control Algorithm (CCA) Implementation Report**

## **Overview**

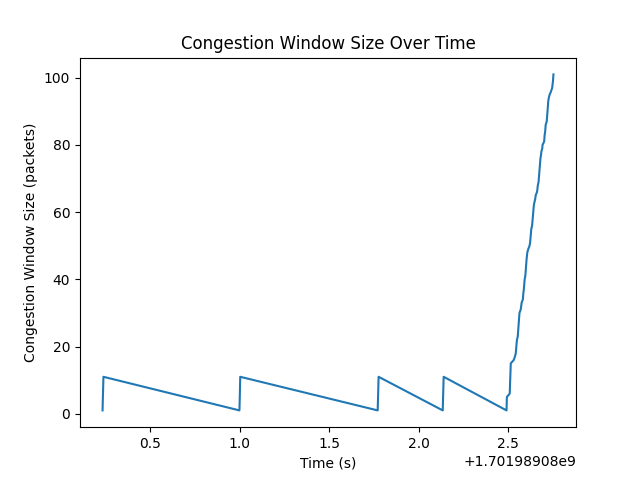
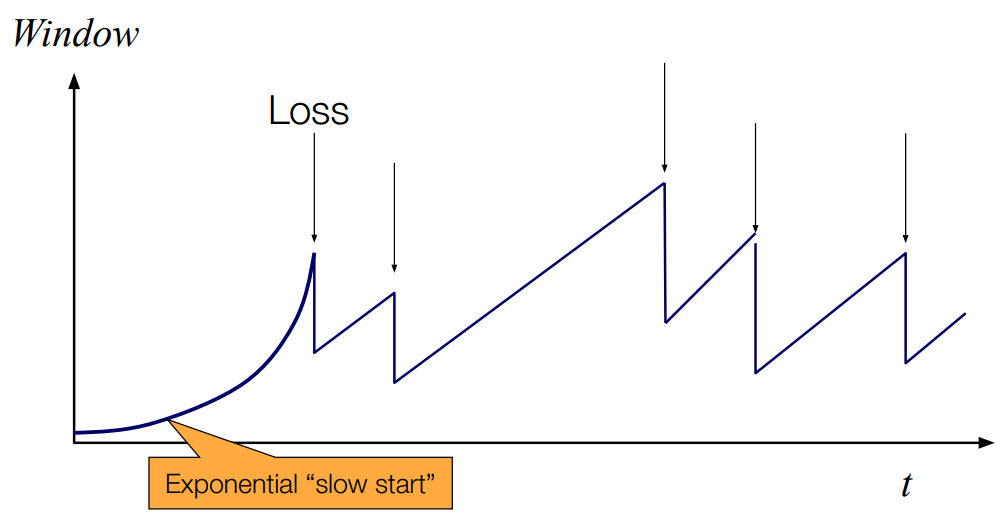
The implemented Congestion Control Algorithm (CCA) is based on the TCP Reno protocol. It is designed to manage network congestion by adjusting the rate of packet transmission based on network conditions. The algorithm operates in three phases: slow start, congestion avoidance, and fast recovery.

## **Key Design Decisions**

1. **Slow Start**: The algorithm begins in the slow start phase, where the congestion window size (`cwnd`) is initially set to 1 and then increased by 1 for each acknowledgment received. This allows the sender to quickly ramp up transmission rate until it reaches a threshold (`ssthresh`).
2. **Congestion Avoidance**: Once the `cwnd` exceeds `ssthresh`, the algorithm enters the congestion avoidance phase. In this phase, `cwnd` is increased by 1/`cwnd` for each acknowledgment, which results in a more conservative increase in the transmission rate.
3. **Fast Recovery**: If a packet is lost (detected by receiving three duplicate acknowledgments), the algorithm enters the fast recovery phase. `ssthresh` is set to half of the current `cwnd`, and `cwnd` is set to `ssthresh` + 3. This allows the sender to quickly recover from packet loss.
4. **Retransmission**: If a packet is not acknowledged before its timeout, it is retransmitted. The timeout is initially set to twice the round-trip time (`rtt`), and is doubled for each subsequent retransmission of the same packet.

## **Initial Observation**

After plotting a graph of my implementation and comparing it with the generally available on, I saw that the graph was a bit varying than the one which I saw in the class. Since I’m increasing `cwnd` linearly, I could observe that the slope of the graph is increasing constantly instead of exponential growth as observed in the class.

Also, another thing which fascinated me was that the graph was depicted in a “reverse saw tooth” manner contrast to the one seen in the class which has forward leaning saw tooth. This could be possible because of the fact that the time taken for `cwnd` to get updated with 1 might’ve been a bit longer than a second causing the `reverse saw tooth` graph.

## **Evaluation**

1. **Correctness**: The implementation correctly follows the TCP Reno protocol. It properly transitions between the slow start, congestion avoidance, and fast recovery phases based on network conditions.

2. **Efficiency**: The algorithm efficiently manages network congestion by dynamically adjusting the transmission rate. By starting with a slow start phase, it quickly finds a suitable rate, and then fine-tunes it during the congestion avoidance phase.

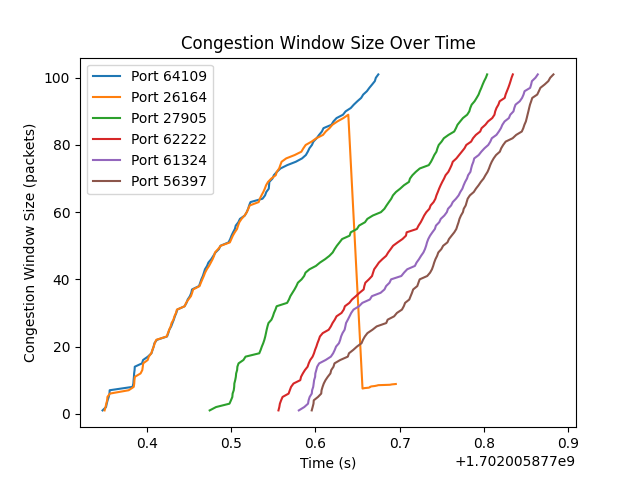
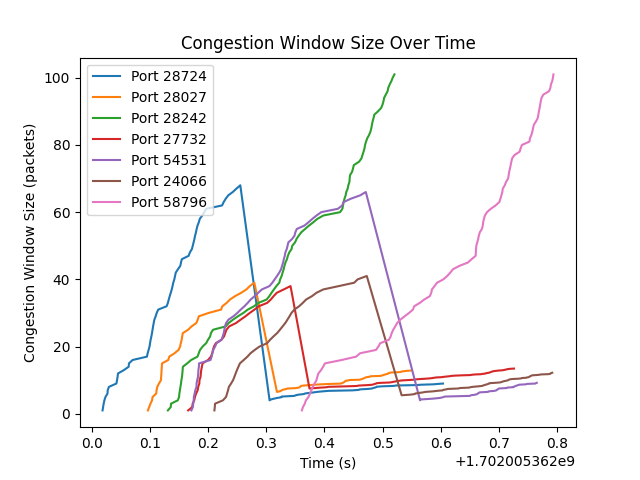
3. **Timeliness**: The use of timeouts and retransmissions ensures that data is delivered in a timely manner, even in the event of packet loss. The timeout value is dynamically adjusted based on the `rtt`, which helps to optimize the retransmission strategy.

4. **Fairness**: The algorithm is fair in the sense that it responds to congestion by reducing the transmission rate, which allows other senders to use the network. However, like TCP Reno, it may not be perfectly fair in all scenarios. For example, multiple TCP Reno flows do not necessarily share the network equally.

## **Experiments:**

The following experiments were conducted for the scenario where users are connected to a CDN.

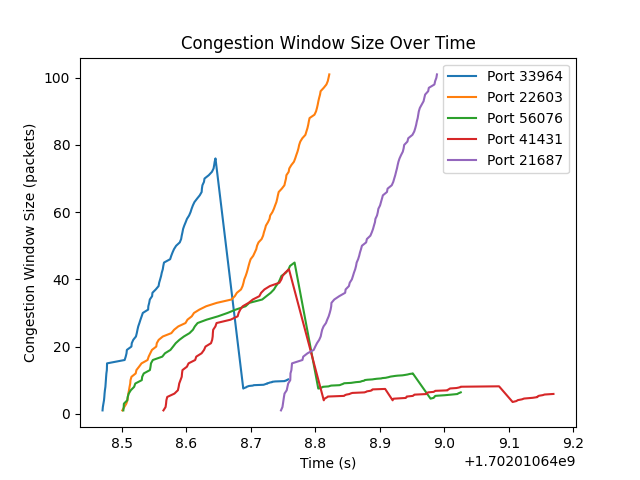
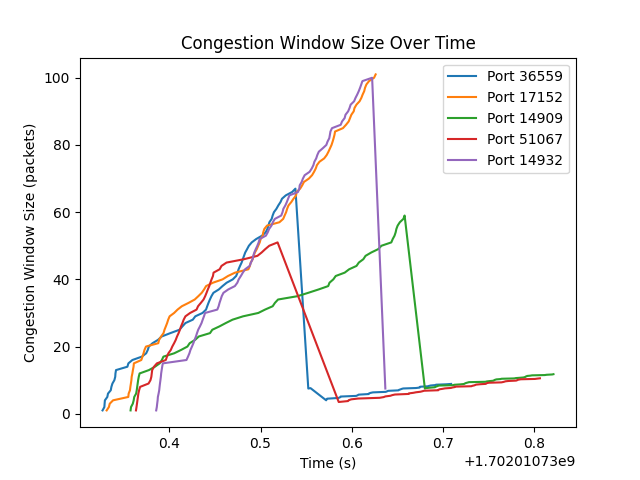
## Queue size:



(**a) Queue size 500 Mb (b) Queue size 1000 Mb**

In this scenario, when the router buffer size is 1000 Mb, it can hold twice as many packets as when the buffer size is 500 Mb. This means that the sender can have more unacknowledged packets in the network, which allows it to send packets more quickly without waiting for acknowledgments. This can lead to higher throughput, which is likely why the CCA performs better with a larger buffer size.

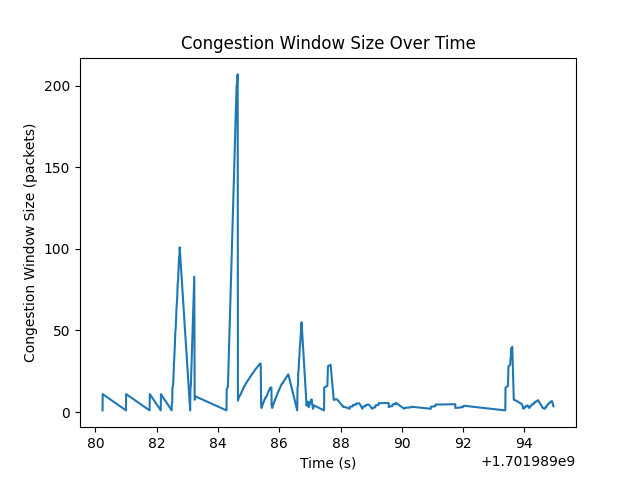
## Packet drop rate:



(**a) Drop rate of 10% loss (b) Drop rate of 1% loss**

## **Observation:**

When my implementation was run under a normal scenario a single a user is accessing a 1TB huge file on CDN with a decent bandwidth, low latency and packets drops, the below can be observed:



On the basis of this, the following are a few points worth noting for this specific implementation:

1. **Timeouts and Retransmissions**: The script uses a fixed timeout value (TIMEOUT = 0.1) for packet retransmissions. This could potentially lead to unnecessary retransmissions if the network latency is consistently higher than this value. A more sophisticated approach might involve dynamically adjusting the timeout based on the observed round-trip times.
2. **Slow Start and Congestion Avoidance**: The script correctly implements the slow start and congestion avoidance phases of TCP Reno. However, the congestion window (cwnd) is increased by 1/cwnd during the congestion avoidance phase, which results in a linear increase. This is less aggressive than the exponential increase during the slow start phase, which could potentially lead to slower throughput if the network conditions are favourable.
3. **Fast Recovery**: The script also implements the fast recovery phase of TCP Reno, where cwnd is set to ssthresh + 3 after a packet loss is detected. This allows the sender to quickly recover from packet loss, but it could potentially lead to more congestion if the network conditions are poor.

In terms of performance, this implementation would perform well in a network with low latency and low packet loss, as it aggressively increases the transmission rate during the slow start phase. However, it might perform worse in a network with high latency or high packet loss, as the fixed timeout value and aggressive fast recovery strategy could lead to unnecessary retransmissions and more congestion.