1 TCP Congestion Control Algorithms

1.1 Introduction

A key component of TCP is end-to-end congestion control algorithm. The TCP congestion control algorithm limits the rate at which the sender sends traffic into the network based on the perceived network congestion. The TCP congestion control algorithm at the sender maintains a variable called congestion window, commonly referred as *cwnd*, that limits the amount of unacknowledged data in the network. The congestion window is adapted based on the network conditions, and this affects the sender's transmission rate. The TCP sender reacts to congestion and other network conditions based on new acknowledgements, duplicate acknowledgements and timeouts. The TCP congestion control algorithms describe the precise way in which TCP adapts *cwnd* with the different events.

The TCP congestion control algorithm has three major phases (a) slow start, (b) congestion avoidance, and (c) fast recovery. In slow-start, TCP is aggressive and increases *cwnd* by one MSS with every new acknowledgement. In congestion avoidance, TCP is cautious and increases the *cwnd* by one MSS per round-trip time. Slow-start and congestion avoidance are mandatory components of all TCP congestion control algorithms. In the event of a packet loss (inferred by timeout or triple duplicate acknowledgements), the TCP congestion control algorithm reduces the congestion window to 1 (e.g., Old Tahoe, Tahoe) or by half (e.g., New Reno). In fast recovery, TCP seeks to recover from intermittent packet losses while maintaining a high congestion window. The new versions of TCP, including TCP New Reno, incorporate fast recovery as well. Figure 1-1 presents a simplified view of the TCP New Reno congestion control algorithm highlighting slow-start, congestion avoidance and fast recovery phases.

TCP congestion control algorithm is often referred to as additive-increase multiplicative-decrease (AIMD) form of congestion control. The AIMD congestion control algorithm often leads to a "saw tooth" evolution of the congestion window (with linear increase of the congestion window during bandwidth probing and a multiplicative decrease in the event of packet losses), see Figure 1-6.

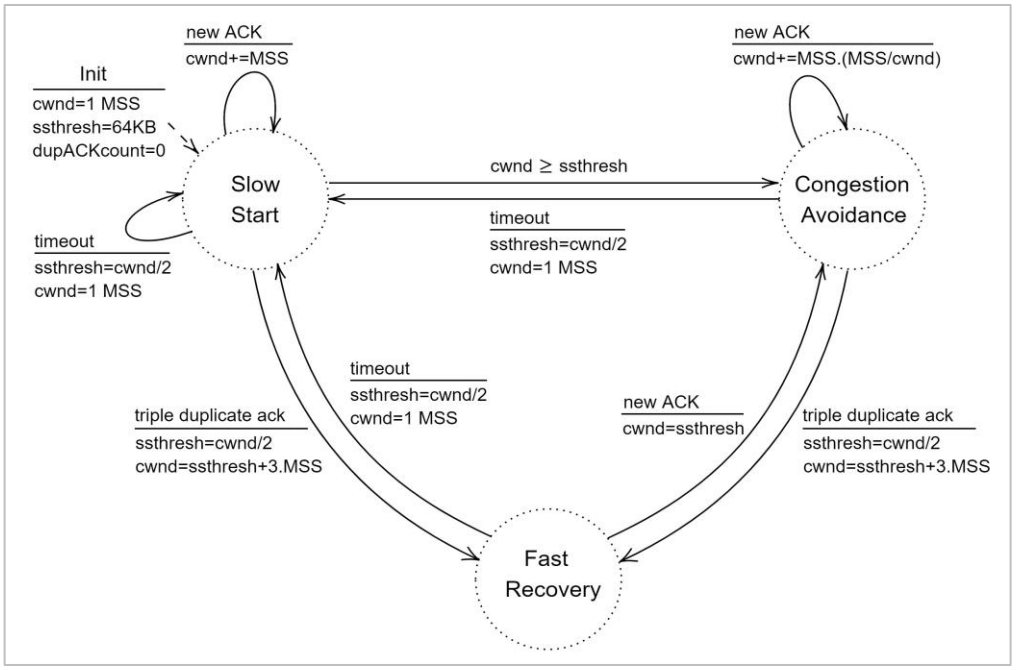

Figure 1-1: A simplified view of FSM of the TCP New Reno congestion control algorithm

1.2 Network Setup

We will seek a large file transfer with TCP over a lossy link to study the TCP congestion control algorithms. We will simulate the network setup illustrated in Figure 1-3 with the configuration parameters listed in detail in steps to study the working of TCP congestion control algorithms.

Open NetSim and click on **Experiments> Internetworks>TCP> TCP Congestion Control Algorithms > Old-Tahoe** then click on the tile in the middle panel to load the example as shown in below Figure 1-2.

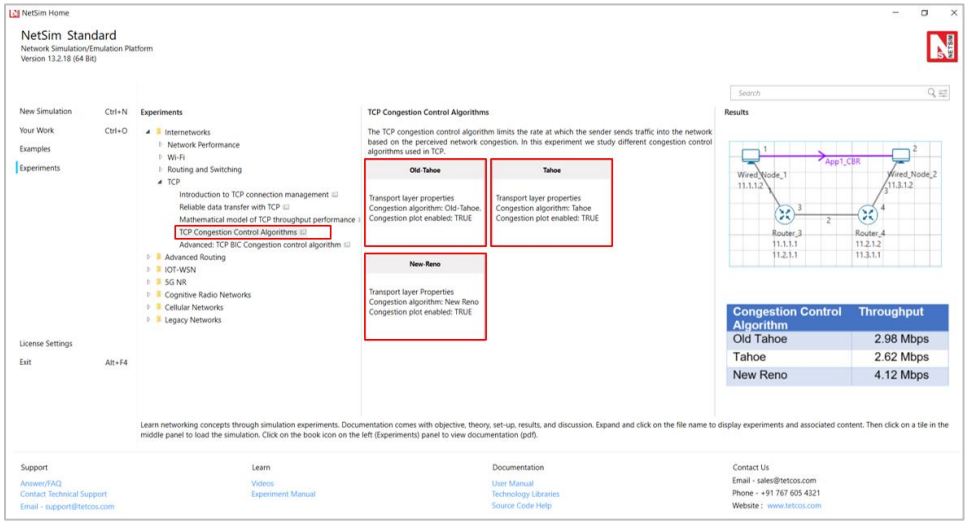

Figure 1-2: List of scenarios for the example of TCP Congestion Control Algorithms

NetSim UI displays the configuration file corresponding to this experiment as shown below:

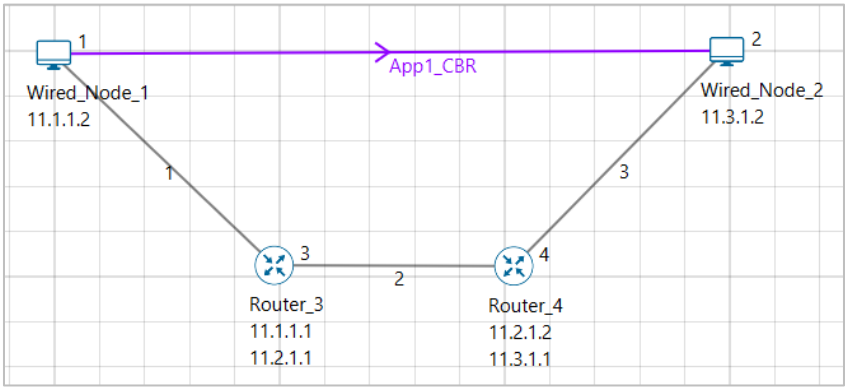

Figure 1-3: List of scenarios for the example of TCP Congestion Control Algorithms

1.3 Procedure

Old Tahoe

The following set of procedures were done to generate this sample.

Step 1: A network scenario is designed in NetSim GUI comprising of 2 Wired Nodes and 2 Routers in the **"Internetworks"** Network Library.

Step 2: In the Source Node, i.e., Wired Node 1, in the TRANSPORT LAYER Properties, Congestion Control Algorithm is set to **OLD TAHOE**. **Congestion plot enabled** is set to **TRUE**.

Step 3: In the General Properties of Wired Node 1 i.e., Source, Wireshark Capture is set to Online.

***NOTE:* Accept default properties for Routers as well as the Links Properties should be changed.**

Step 4: Right-click the link ID (of a wired link) and select Properties to access the link's properties. Set Max Uplink Speed and Max Downlink Speed to **10** Mbps. Set Uplink BER and Downlink BER to **0**. Set Uplink Propagation Delay and Downlink Propagation Delay as **100** microseconds for the links 1 and 3 (between the Wired Node's and the routers). Set Uplink Propagation Delay and Downlink Propagation Delay as **50000** microseconds and Uplink BER and Downlink BER to **0.0000001** for the backbone link connecting the routers, i.e., 2.

Step 5: Right click on the Application Flow **App1 CBR** and select Properties or click on the Application icon present in the top ribbon/toolbar.

An CBR Application is generated from Wired Node 1 i.e., Source to Wired Node 2 i.e., Destination with Packet Size set to 1460 Bytes and Inter Arrival Time set to 1168 microseconds.

Step 6: Click on Display Settings > Device IP check box in the NetSim GUI to view the network topology along with the IP address.

Step 7: Click on **Plots** icon and select the **Enable Plots** checkbox. This enables us to view the throughput plot of the application **App1 CBR**.

Step 8: Click on Run simulation. The **simulation time** is set to 20 seconds. In the "**Static ARP Configuration**" tab, Static ARP is set to **disable**.

Tahoe

Step 1: In the Source Node, i.e., Wired Node 1, in the TRANSPORT LAYER Properties, Congestion Control Algorithm is set to **TAHOE**. **Congestion plot enabled** is set to **TRUE**.

Step 2: Click on Run simulation. The **simulation time** is set to 20 seconds. In the "**Static ARP Configuration**" tab, Static ARP is set to **disable**.

New Reno

Step 1: In the Source Node, i.e. Wired Node 1, in the TRANSPORT LAYER Properties, Congestion Control Algorithm is set to **NEW RENO**. **Congestion plot enabled** is set to **TRUE**.

Step 2: Click on Run simulation. The **simulation time** is set to 20 seconds. In the "**Static ARP Configuration**" tab, Static ARP is set to **disable**.

1.4 Output

We have enabled WireShark Capture in the Wired Node 1. The PCAP file is generated at the end of the simulation. From the PCAP file, the congestion window evolution graph can be obtained as follows. In Wireshark, select any data packet with a left click, then, go to **Statistics** > **TCP Stream Graphs** > **Window Scaling >** Select **Switch Direction**.

The congestion window evolution for Old Tahoe, Tahoe and New Reno congestion control algorithms are presented in Figure 1-4, Figure 1-5, and Figure 1-6, respectively.

Table 1-1 shows the throughput values of different congestion control algorithms (obtained from the Application Metrics).

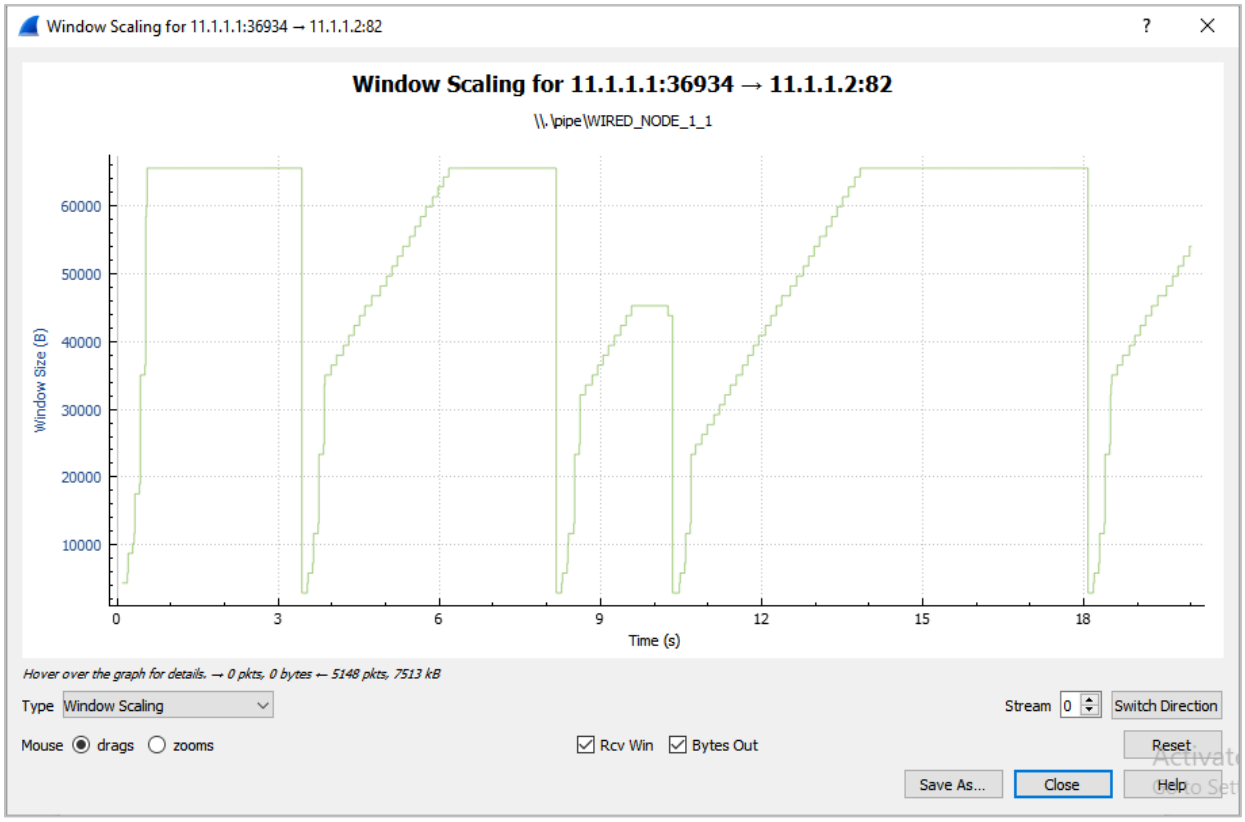

Figure 1-4: Congestion window evolution with TCP Old Tahoe. We note that Old Tahoe infers packet loss only with timeouts, and updates the slow-start threshold ssthresh and congestion window cwnd as ssthresh = cwnd/2 and cwnd = 1

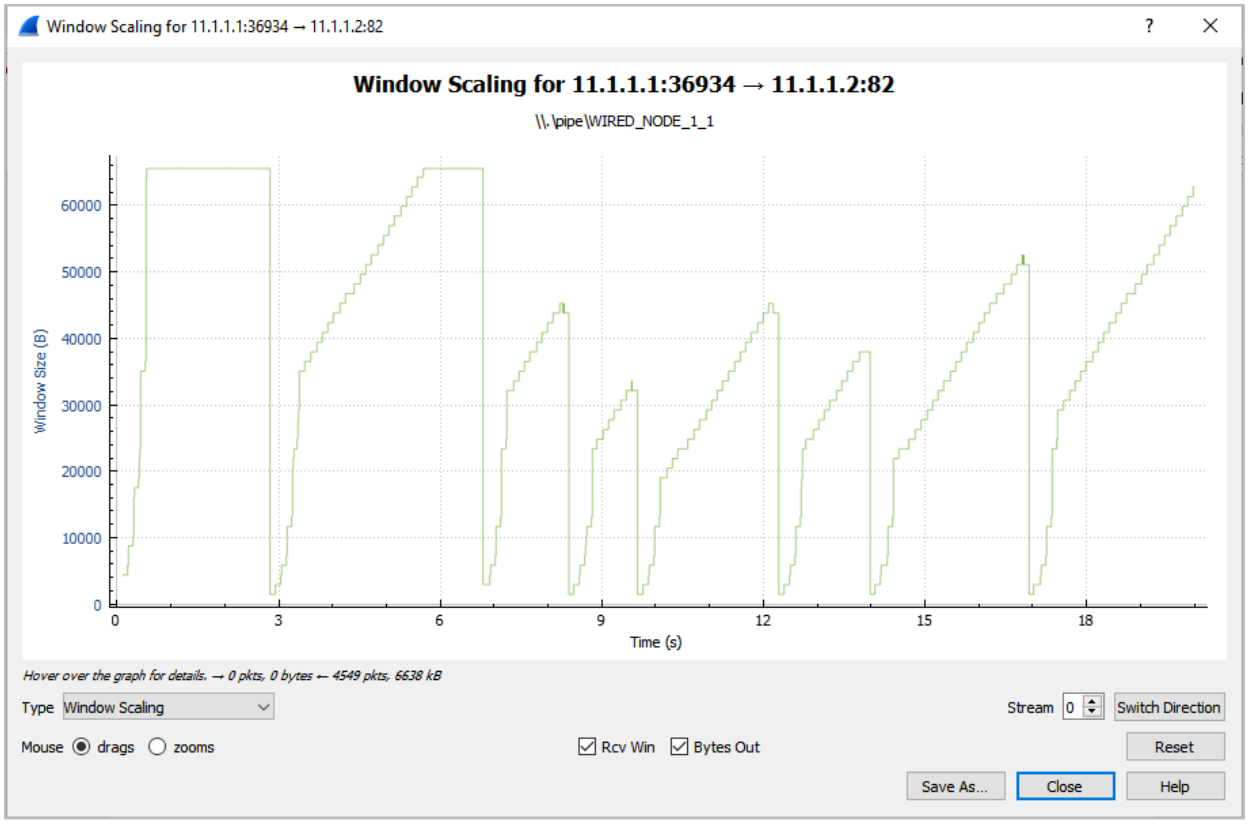

Figure 1-5: Congestion window evolution with TCP Tahoe. We note that Tahoe infers packet loss with timeout and triple duplicate acknowledgements, and updates the slow-start threshold ssthresh and congestion window cwnd as ssthresh = cwnd/2 and cwnd = 1

Figure 1-6: Congestion window evolution with TCP New Reno. We note that New Reno infers packet loss with timeout and triple duplicate acknowledgements, and updates the slow-start threshold ssthresh and congestion window cwnd as ssthresh = cwnd/2 and cwnd = ssthresh + 3MSS (in the event of triple duplicate acknowledgements).

Congestion Control Algorithm	Throughput
Old Tahoe	2.98 Mbps
Tahoe	2.62 Mbps
New Reno	4.12 Mbps

Table 1-1: Long-term average throughput of the different TCP congestion control algorithms

1.5 Observations and Inference

1. We can observe slow start, congestion avoidance, timeout, fast retransmit and recovery phases in the Figure 1-4, Figure 1-5, and Figure 1-6. In Figure 1-4, we note that Old Tahoe employs timeout, slow-start and congestion avoidance for congestion control. In Figure 1-5, we note that Tahoe employs fast retransmit, slow-start and congestion avoidance for congestion control. In Figure 1-6, we note that New Reno employs fast retransmit and recovery, congestion avoidance and slow-start for congestion control.

2. We note that TCP New Reno reports a higher long term average throughput (in comparison with Old Tahoe and Tahoe, see Table 1-1) as it employs fast retransmit and recovery to recover from packet losses.