(Pre) GENI Lab 5

Due: There is nothing to submit. However, make sure you watch all my videos, read Section 0, and complete Section 1 below by 3:30pm Oct 2, 2019.

0. Introduction

0.1. Overview

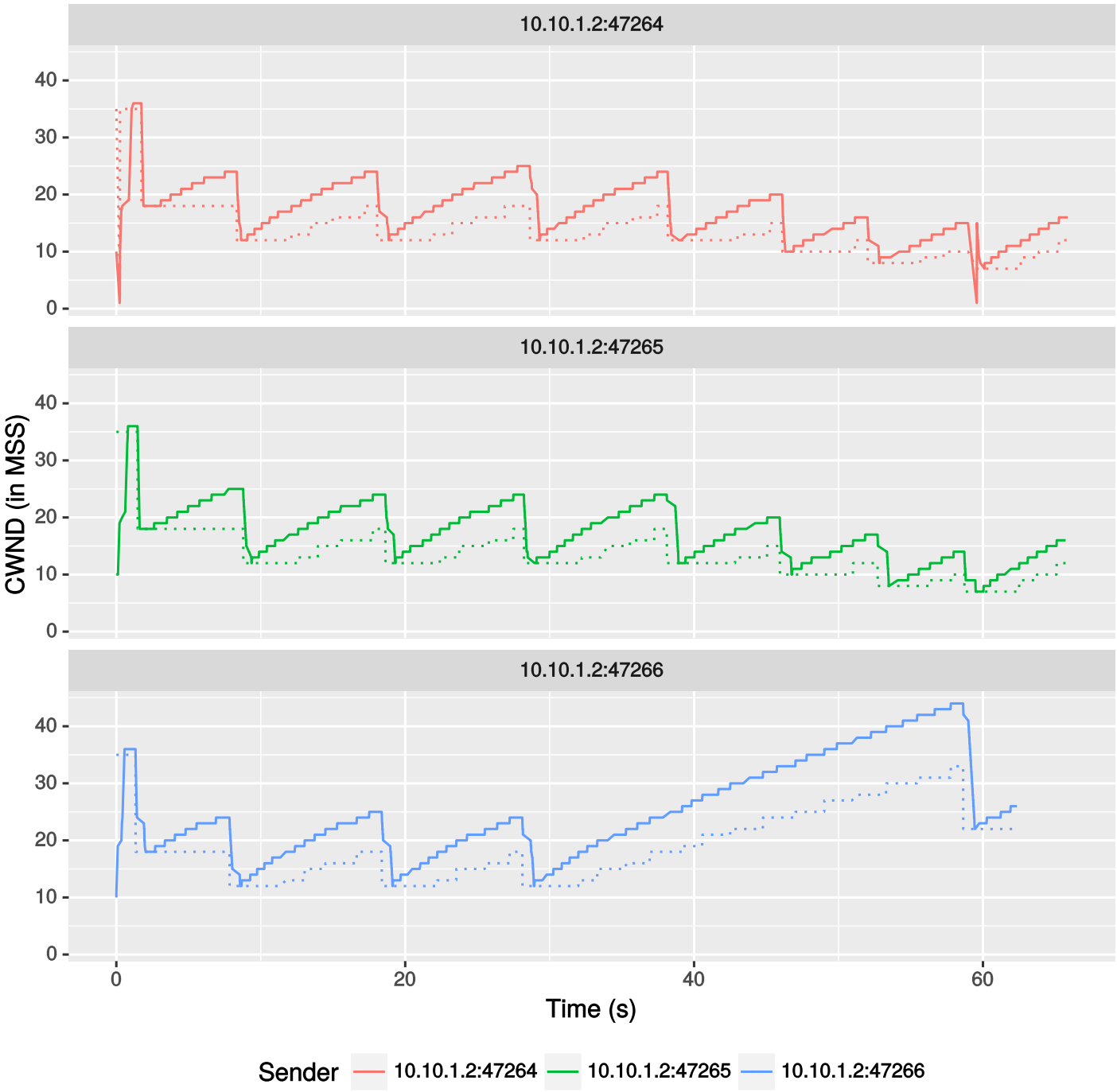
On the transport layer, congestion control is one of the most important tasks for the TCP protocol.

> The basic idea of TCP Congestion control is that the TCP sender starts by sending certain small amount of data governed by a variable `CWND` which stands for Congestion Window, typically in the unit of "MSS" or *Max Segment Size*, into the network. If it receives an acknowledgement (ACK) within a reasonable period of time, it increases `CWND` and consequently sends more in the next round. This way, the sender keeps probing the network by sending more and more until it realizes that it just sent too much too fast for the network to handle, i.e. congestion occurred. It punishes itself in two ways: if it detected the congestion by receiving triple duplicate ACKs, it cuts `CWND` in half and sends data at half speed in the next round; on the other hand, if it got "timeout", i.e. no ACK was received at all even after a pre-set timer has expired, it brings `CWND` down all the way the lowest possible (typically 1 MSS) and goes from there.

This experiment shows the basic behavior of TCP congestion control. You'll see the classic "sawtooth" pattern in a TCP flow's congestion window, and you'll see how a TCP flow responds to indications of congestion.

0.2. Background

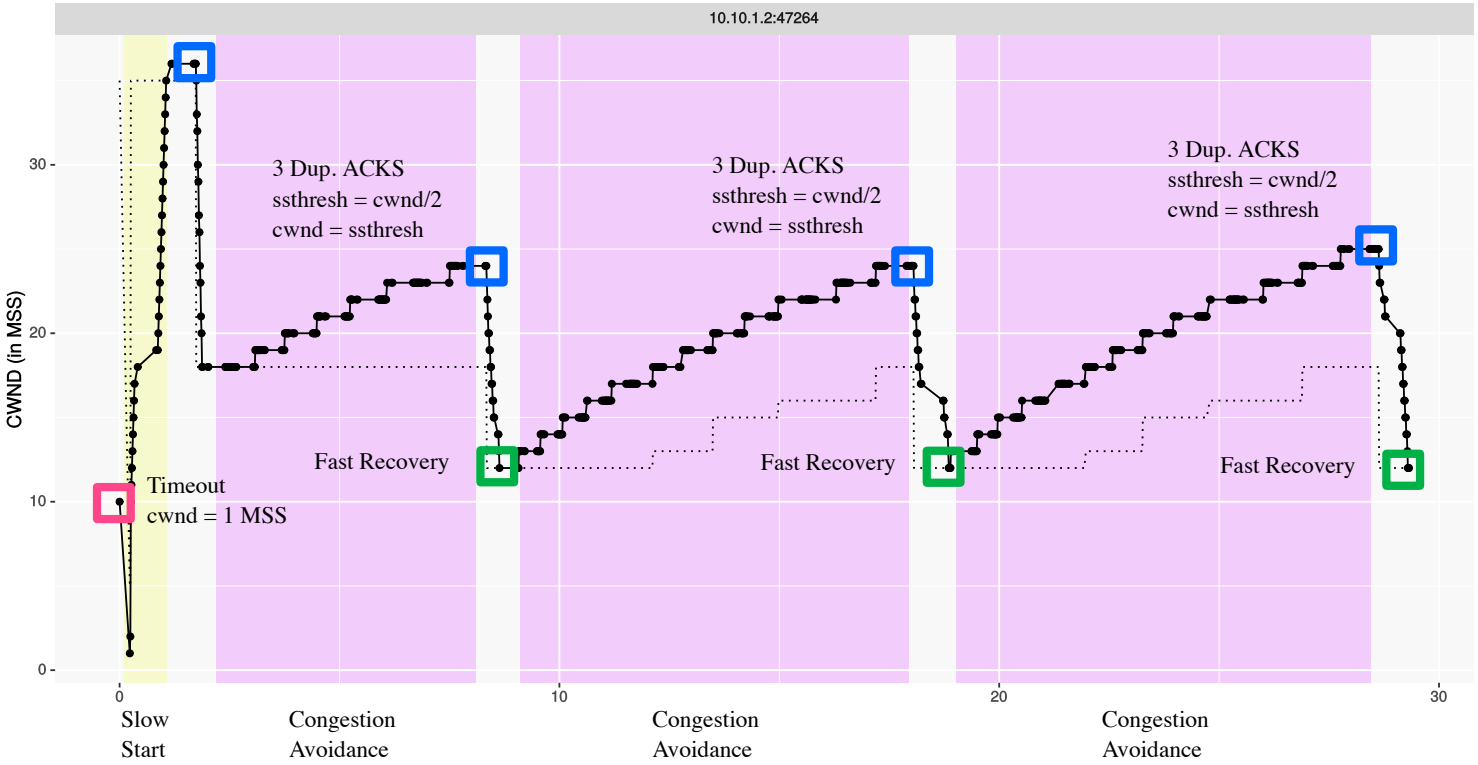
In this experiment, we'll see the classic "sawtooth" pattern of the TCP congestion window (CWND), shown as the solid line, and slow start threshold (dotted line) of three TCP flows sharing the same bottleneck in the plot below:



We can also identify

- *Slow start* periods (when the congestion window is less than the slow start threshold)
- *Congestion avoidance* periods (when the congestion window is greater than the slow start threshold)
- Instances where 3 duplicate ACKs were received. (We are using TCP Reno, which will enter "fast recovery" in response to 3 duplicate ACKs.)
- Instances of timeout. This will cause the congestion window to go back to 1 MSS, and trigger a "slow start" period. For example, the following annotated image shows a short interval in the top TCP flow of the figure

above:



Slow-start periods are marked in yellow; congestion avoidance periods are in purple. We can also see when indicators of congestion occur: instances of timeout (red), and instances where three duplicate ACKs are received (blue) triggering fast recovery (green).

1. Lab Configurations

Create a new slice (please name the new slice "**lab5-your-initial**") and request resources by loading the RSpec from the following URL: https://git.io/vSioM.

Be sure you have one ssh terminal to each of the three nodes.

On the end hosts (**Sender** and **Receiver**), install the `iperf` network testing tool, with the command:

```
sudo apt-get update
sudo apt-get install iperf
```

Also, on **Sender** and **Receiver** nodes, set TCP Reno as the default TCP congestion control algorithm on both, with

```
sudo sysctl -w net.ipv4.tcp_congestion_control=reno
```

On **Router** node, turn on packet forwarding with the command

```
sudo sysctl -w net.ipv4.ip_forward=1
```

Also, on **Router** node, set each experiment interface on the router so that the router is a 1 Mbps bottleneck, and buffers up to 0.1 MB, using a token bucket queue:

```
sudo tc qdisc replace dev eth1 root tbf rate 1mbit limit 0.1mb burst 32kB peakrate 1.
01mbit mtu 1600
sudo tc qdisc replace dev eth2 root tbf rate 1mbit limit 0.1mb burst 32kB peakrate 1.
01mbit mtu 1600
```

Finally, on **Sender** node, load the `tcp_probe` kernel module, and tell it to monitor traffic to/from TCP port 5001:

```
sudo modprobe tcp_probe port=5001 full=1
sudo chmod 444 /proc/net/tcpprobe
```

We will use this TCP probe module to monitor the behavior of the TCP congestion control algorithm on the sender.

TO BE CONTINUED....