Exploring Congestion Control

Advanced Networking - Università della Svizzera Italiana (USI)

Luca Di Bello

March 23, 2024

Contents

1 Introduction and environment setup

To analyze the differences between the congestion control algorithms proposed in the assignment (Reno, Cubic, and Vegas) I created a simple virtual dumbbell topology using an OS-level Hypervisor (Parallels) and two Debian-based virtual machines connected via a virtual network created by the hypervisor. In addition, to simulate the network congestion, I used the provided script `rate-limiter.sh` to limit the bandwidth of the virtual network interface of the two virtual machines to a fixed transfer rate of 1 Mbps and 5 Mbps. [1]

Figure 1 shows the network topology used for the experiments with the two virtual machines connected to the hypervisor virtual network gateway.

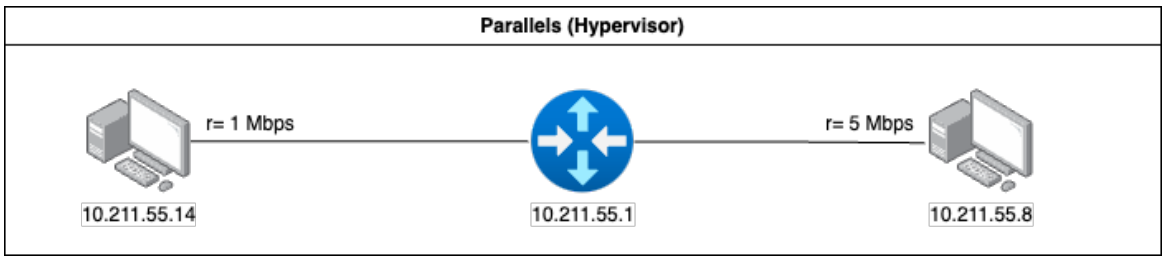

Figure 1: Network topology

Note: The hypervisor virtual network gateway is connected to the physical network interface of the host machine. In the diagram above, this feature is not shown as it is not relevant for the purpose of the experiments.

2 Traffic control setup

To simulate the network congestion, I used the provided script `rate-limiter.sh` to limit the bandwidth of the virtual network interface of the two virtual machines to 1 Mbps (left VM) and 5 Mbps (right VM). In addition, the script also adds a delay of 100 ms to the packets exchanged between the two virtual machines. The script leverages the `tc` command to configure the traffic control settings of the network interface. This is the command used to limit the bandwidth of the network interface, to add a delay to the packets, and to limit the number of packets in the queue:

```
sudo tc qdisc add dev "$INTERFACE" root netem rate "$RATE"
sudo tc qdisc change dev "$INTERFACE" root netem limit 10
sudo tc qdisc change dev "$INTERFACE" root netem delay 100ms \
    50ms 50 distribution normal
```

Unfortunately, after some testing, I was not able to create timeouts using the provided script. Therefore, I decided to introduce packet loss, duplications, and packet corruption via `tc` to simulate problems in the network and test how the congestion control algorithms behave in these scenarios. To do so, I added the following commands to `setup` routine of the script:

```
sudo tc qdisc change dev "$INTERFACE" root netem loss 5%
sudo tc qdisc change dev "$INTERFACE" root netem duplicate 2%
sudo tc qdisc change dev "$INTERFACE" root netem corrupt 1%
```

This approach resulted in a more realistic simulation of network congestion, as it allowed me to test how the window size changes in response to these events.

[1] I decided to use two different transfer rates to simulate a real network scenario where the bandwidth is not the same in both directions.

3 Test Plan

To analyze the behavior of the congestion control algorithms, I created a simple test plan that consists of the following steps:

1. Compile the provided source code for the TCP server and client on both virtual machines using the provided `Makefile`.

2. Start the `rate-limiter.sh` script on both virtual machines to limit the bandwidth of the network interface. This script will be executed only once on each virtual machine.

   On the left VM, this script will limit the bandwidth to 1 Mbps:

   ```
   $ sudo ./rate-limiter.sh -i enp0s5 -r 1mbps
   ```

   On the right VM, this script will limit the bandwidth to 5 Mbps:

   ```
   $ sudo ./rate-limiter.sh -i enp0s5 -r 1mbps
   ```

3. Start a TCP server on the virtual machine on the left side of the network topology (`10.211.55.14`) to listen on port 1234. The following command will be executed only once:

   ```
   $ ./server.py 1234
   ```

4. Start Wireshark on the right side of the virtual network (`10.211.55.8`) in order to capture the packets exchanged between the client and the server. In order to be able to sniff the packets, the user must execute the program with superuser privileges:

   ```
   $ sudo wireshark -i enp0s5
   ```

5. Start a TCP client on the virtual machine on the right side of the network (`10.211.55.8`), in order to connect to the server on port 1234, and send 10000000 random bytes (10 MB) of data. The following command will be executed 3 times, each with a different congestion control algorithm:

   ```
   $ ./client -p 1234 -n 10000000 -c <reno|cubic|vegas> 10.211.55.14
   ```

4 Congestion Control Algorithms

As mentioned in the introduction (refer to section 1), this report will analyze the behavior of three different congestion control algorithms: Reno, Cubic, and Vegas. The tests have been conducted using the source code provided in the submission, using the same configuration for all the algorithms to ensure a fair comparison.

4.1 Reno Congestion Control Algorithm

This algorithm presents a simple and efficient way to control congestion in TCP networks. It is based on the *Additive Increase Multiplicative Decrease* (AIMD) principle [2].

By analyzing the Wireshark capture of the packets exchanged between the client and the server, this behavior can be observed. Figure 2 shows the congestion window size in function of the time for the Reno congestion control algorithm (generated by Wireshark) during a file transfer of 1 MB.

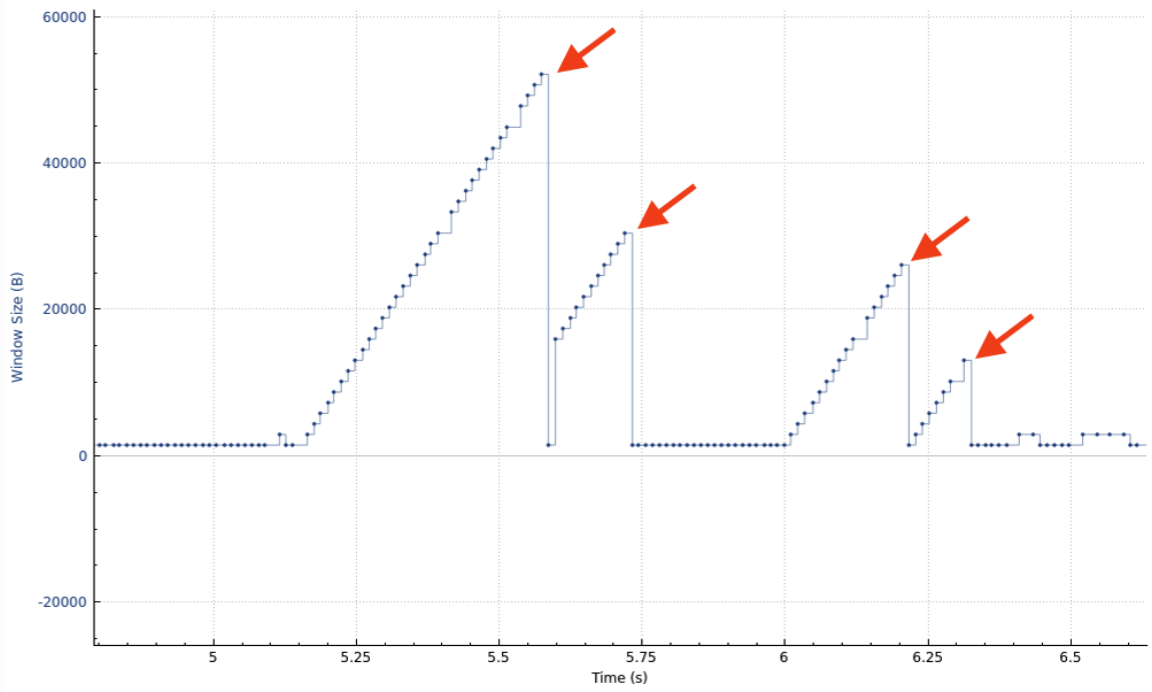

Figure 2: Reno Congestion Control Algorithm - Congestion Window Size over Time during a file transfer of 1 MB.

We can see that the Reno congestion control algorithm behaves as expected. The algorithm starts by increasing the congestion window size linearly until it detects a duplicate ACK (packet loss). When a packet loss is detected, the algorithm decreases the congestion window size by half and starts increasing it linearly again. On the other hand, if there is a timeout, the algorithm decreases the congestion window size to 1 and starts increasing it linearly again.

Is it possible to estimate the window size of the congestion control algorithm using the following information:

[2] AIMD is a feedback control algorithm that controls the rate of data transmission in a network. It increases the transmission rate linearly until it detects a congestion (NACKs or timeout) and then decreases the rate by an exponential factor.[3]

- Slow start: the congestion window increases by 1 MSS for each ACK received, doubling the window size each RTT until reaching the slow start threshold ($ssthresh$): $cwnd = cwnd{+}1\ MSS$.
- Congestion avoidance: After reaching the slow start threshold, the congestion window increases by a factor of $1/cwnd$ for each RTT: $cwnd = cwnd + 1/cwnd$.
- Fast recovery: When a packet loss is detected, the congestion window is halved, setting the slow start threshold to half the current congestion window size: $cwnd = cwnd/2$, $ssthresh = ssthresh/2$.
- Timeout: When a timeout is detected, the congestion window is set to 1 MSS and the slow start threshold is set to half the current congestion window size: $cwnd = 1$, $ssthresh = cwnd/2$.

For example, in Figure 2, we can see a segment of the overall communication between client and server, where multiple packet loss and timeouts are detected in a time frame of 1.5 seconds. The congestion window size starts at 1 MSS, then increases exponentially until it reaches its peak at around 50k bytes (about 32 MSS). After a packet loss is detected, the congestion window size is halved and continues to increase linearly until it reaches the slow start threshold. Unfortunately, during the increase of the congestion window size, a timeout is detected, and the congestion window size is set to 1 MSS to then start increasing linearly again.

4.2 Cubic Congestion Control Algorithm

The Cubic congestion control algorithm is present in the Linux kernel since version 2.6.13 (around 2006). It is optimized for *long fat networks* (LFNs) as it can achieve reliable high bandwidth connections even in high latency situations. [2]

This algorithm does not use the AIMD principle as the Reno algorithm does (refer to section 4.1). Instead, it uses a cubic function to calculate the congestion window size based on the time elapsed since the last congestion event with the inflection point of the cubic function being the last congestion event, allowing a slow start of the congestion window size to avoid possible additional congestion events. Then, due to the cubic function, the congestion window size increases exponentially until it reaches the last congestion window size before the congestion event.

In Figure 3, is possible to observe the behavior of the Cubic congestion control algorithm during a file transfer of 1 MB with some packet loss, duplication, and corruption events.

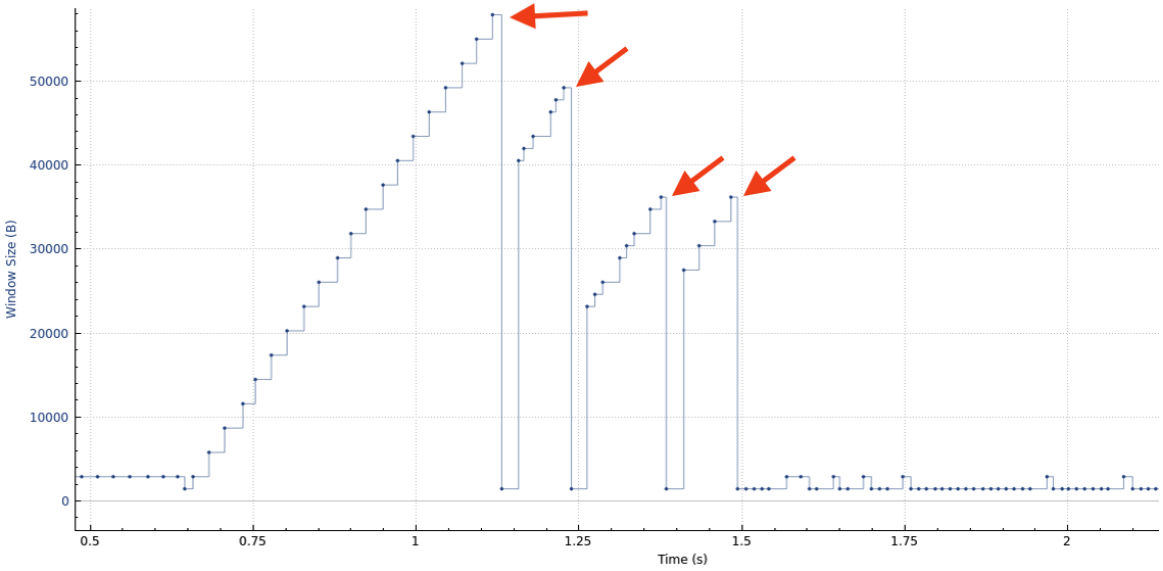

Figure 3: Cubic Congestion Control Algorithm - Congestion Window Size over Time during a file transfer of 10 MB.

As it is possible to see from the figure above, the Cubic congestion control algorithm has a different behavior compared to the Reno algorithm. In this case, the congestion window size could be estimated using the following formula: [2]

$$cwnd(t) = C(t - K)^3 + W_{max}$$

Where:

$cwnd(t)$ is the congestion window size at time t

C is a constant that determines the rate of increase of the congestion window size

$$K = \sqrt[3]{\frac{W_{max}(1 - \beta)}{C}}$$

β is the congestion window reduction factor

W_{max} is the maximum congestion window size before the last congestion event

4.3 Vegas Congestion Control Algorithm

To conclude, the Vegas congestion control algorithm is a TCP congestion control algorithm that uses the *Additive Increase Multiplicative Decrease* (AIMD) principle to control the congestion in the network. The main difference between Vegas and Reno is that Vegas uses the variation in RTT to detect congestion in the network, while Reno uses packet loss. [3]

By measuring the actual throughput and the expected throughput, Vegas can detect congestion in the network and adjust the window size to avoid packet loss.

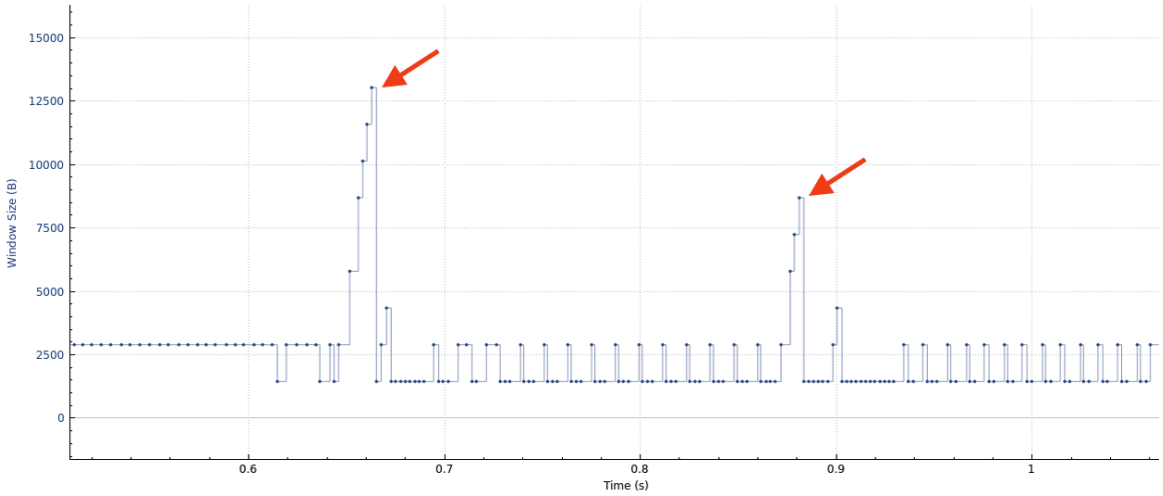

Figure 4: Vegas Congestion Control Algorithm - Congestion Window Size over Time during a file transfer of 10 MB.

In Figure 4, we can see the behavior of the Vegas congestion control algorithm during a file transfer of 10 MB. The congestion window size increases linearly until a congestion event is detected due to the difference in RTT between the expected and the actual throughput.

The goal of the Vegas congestion control algorithm is to keep the congestion window size as close as possible to the expected throughput in order to prevent future congestion events.

To estimate the congestion window size of the Vegas congestion control algorithm, we can use the following formula: [1]

$$W = \begin{cases} W + 1, & \text{if diff} < \alpha \\ W, & \text{if } \alpha \leq \text{diff} \leq \beta \\ W - 1, & \text{if diff} > \beta \end{cases}$$

Where:

$$diff = (\frac{W}{baseRTT} - \frac{W}{RTT})$$

$$baseRTT = W\frac{RTT - baseRTT}{RTT}$$

W is the congestion window size

RTT is the actual Round Trip Time

$baseRTT$ is the expected Round Trip Time

α Vegas throughput thresholds, measured in packets

This behavior can be observed in Figure 4, where the congestion window size is increased due

to RTT variations until a congestion event is detected. Then, as soon as a timeout is detected, the congestion window size is reset to 1 MSS and starts increasing linearly again.

Is it also possible to see that the congestion window size fluctuates between 1-3 MSS due to the RTT variations in the network. On the other hand, in certain timeframes the size of the window is static due to stable RTT values.

5 Conclusions

There is a clear difference between the Vegas congestion control algorithm and the Reno and Cubic algorithms. As cited in the previous sections, while Reno and Cubic use packet loss to detect congestion in the network, Vegas uses the variation in RTT to adjust the congestion window size. For this reason, is possible to see that the Vegas congestion window size fluctuates more than the other two algorithms. This is clear when comparing the diagrams of the congestion window size over time for the three algorithms.

On the other hand, the Reno and Cubic algorithms have a similar behavior, as they both use the AIMD principle to control the congestion in the network. The main difference between the two algorithms is that Cubic uses a cubic function to calculate the congestion window size, while Reno uses a linear function. This slight difference in the behavior of the two algorithms can be observed by comparing the two plots, but is less evident than the difference between Vegas and the other two algorithms.

References

[1] D. o. C. S. University of Winsconsin. Modeling the throughput of tcp vegas. `https://pages.cs.wisc.edu/~vernon/papers/sword.03sigm.pdf`. Accessed: 23.03.2024.

[2] Wikipedia. Cubic tcp. `https://en.wikipedia.org/wiki/CUBIC_TCP`. Accessed: 23.03.2024.

[3] Wikipedia. Tcp congestion control. `https://en.wikipedia.org/wiki/TCP_congestion_control`. Accessed: 22.03.2024.