Academic Year 2023/2024

Internet Performance and Troubleshooting Lab

Politecnico di Torino

Report IV

Impact of RTT and Packet Loss

Group 3

Brendon Mendicino (s317639)

Alessandro Ciullo (s310023)

Davide Colaiacomo (s313372)

1 Configuration

Before conducting the experiment, we set up the client with the following configuration: first of all, we enabled the `TCP_Diag` module with the command:

```
    modprobe tcp_diag
```

Then, we enabled the `tcp log tracing` by means of the following command in the `/sys/kernel/debug/tracing` directory:

```
    echo 1 > events/tcp/tcp_probe/enable
```

By using the `systemctl` tool, we enabled the `TCP Selective Acknowledgment`:

```
    sysctl net.ipv4.tcp_sack=1
```

During the experiments, all the offloading capabilities of the **NIC** were disabled with the command:

```
    ethtool -K ${INTF} rx off tx off sg off tso off gso off gro off
```

Finally, the channel speed was set to 10Mbps. All experiments were done with two PC connect with point-to-point Ethernet cable, and all the captures where performed using `iperf3`.

2 tcp_diag

`tcp_diag` is a kernel module used to monitor the internal values of the **TCP** algorithm, such as the value of the `Congestion Window` (**CWND**), the `Slow start Threshold` (**SSTRESH**), and other parameters. Unfortunately, this way of tracing the **TCP** state is not the most precise from the point of view of the packets sent; the reason is that all of these values are fetched when an **ACK** is received by the system and not when a packet is sent, thus causing some of them to be not precise, like the `Seq-No.` or the `packet length`. This is done in order to avoid overloading the system when it is sending the segments in loop; in fact, we can see that the majority of the segments have `payload length` equal to 0, signaling that there are no data being sent in that specific moment by the kernel. This way of reading the data is consistent with the fact that, in the `Linux kernel`, parameters such as **CWND** and **SSTRESH** are updated when an **ACK** is received; as a matter of fact, by looking at the implementation of the `slow-start` phase in the kernel, it is possible to see that the **CWND** is updated when the next packet is acknowledged:

```
__bpf_kfunc u32 tcp_slow_start(struct tcp_sock *tp, u32 acked)
{
  u32 cwnd = min(tcp_snd_cwnd(tp) + acked, tp->snd_ssthresh);

  acked -= cwnd - tcp_snd_cwnd(tp);
  tcp_snd_cwnd_set(tp, min(cwnd, tp->snd_cwnd_clamp));

  return acked;
}
```

It is also interesting to note that this piece of code is the source of the exponential growth of the `slow-start` phase.

3 Delay vs. Loss

All the algorithm we tested were based on either one of two premises: *"Decrease the CWND based on the lost segments recorded"* vs *"Decrease the CWND based on the delay of the ACK segments recorded"*, one of them though (**BBR**) is based on delay to adjust the **CWND**, but it doesn't quite fit how the others **TCP** CCAs are fair between each other.

4 Reno

Reno was among the first algorithms implemented for the `TCP Congestion Control`; the implementation consists of a simple `Additive Increase - Multiplicative Decrease`, which means that it will be very sensible to packet losses, especially with modern connections that, in some instances, might also reach speeds of 10Gb/s. The **Reno** implementation that is currently adopted in the `Linux kernel` is the **New Reno**, which is an improved version of the original one; more than one change has been made in **New Reno**, but the one we can observe from the graphs is the following: in **New Reno**, losses do not make the algorithm get back to the `Slow start` phase, they just halve the **CWND** and continue from there in `Congestion Avoidance` mode.

By observing image 5.1 (b) and image 10.1, we can understand the differences that become more evident when different delays and different packet loss percentages are used. The first thing that catches our attention is the growth of the **CWND**, with 0% or 0.01% of packet loss, that reaches a size of `4500` (a limit set by the kernel) and then stops growing; furthermore, the **CWND** seems to grow linearly, instead of exponentially. This is caused by the channel speed, in the sense that a speed of 10Mbps with small **RTT** (10ms or 50ms) can be supported by the `Congestion Control` when the **CWND** size is about $\frac{10Mbps*RTT}{8*1538}$; with a delay of 10ms, the needed size is around 10, so only 4 `round-trip` times (**RTT**) must pass for the channel to be saturated, making it hard for it to appear on the graph; we can only see a little trace of the classical exponential `Slow start` shape in the first part of image 10.1 with 200ms of delay. After that, even if the congestion window increases (as observable from the graph and the `Linux kernel` source code), the growth is limited by the channel's speed, and we know that if the growth is constant, the function is linear. The `0.01% packet loss` graph acts in the same way as the `0% packet loss` graph because, in the sample that has been taken, the loss never happened.

By continuing the analysis, we can observe how with packet loss percentage 0.01%, the graphs are pretty different; after a first `slow start` phase, when a loss occurs, the `Congestion Control` passes to the |Congestion Avoidance— phase by setting the threshold to the last —CWND/2— and starting an additive increase of 1 `Maximum Segment Size` (**MSS**) for each `RTT`. Since the amount of loss is significant, the growth of the —CWND— is slower than the reduction caused by the loss and the congestion window inevitably reaches values close to zero.

Another curious phenomenon is the variation of the `Round-Trip Time` (**RTT**) across different loss percentages graphs; by observing figure 10.2, what is surprising to notice are the around two seconds —RTT— we get over an Ethernet cable when no loss or nearly no loss occur. This is caused by the enormous congestion window we discussed before; the kernel generates and send to the **NIC** all those packets until the **NIC**'s memory is full, creating a roughly 2 seconds long queue for every packet after a certain point. For packet losses ¿ 0.01%, the **RTT** has a big growth during the `slow start` phase, but as soon as losses occur, the —CWND— decreases drastically and no queue is created; the `RTT` is the delay we set plus the Ethernet propagation and computation overhead.

5 Cubic

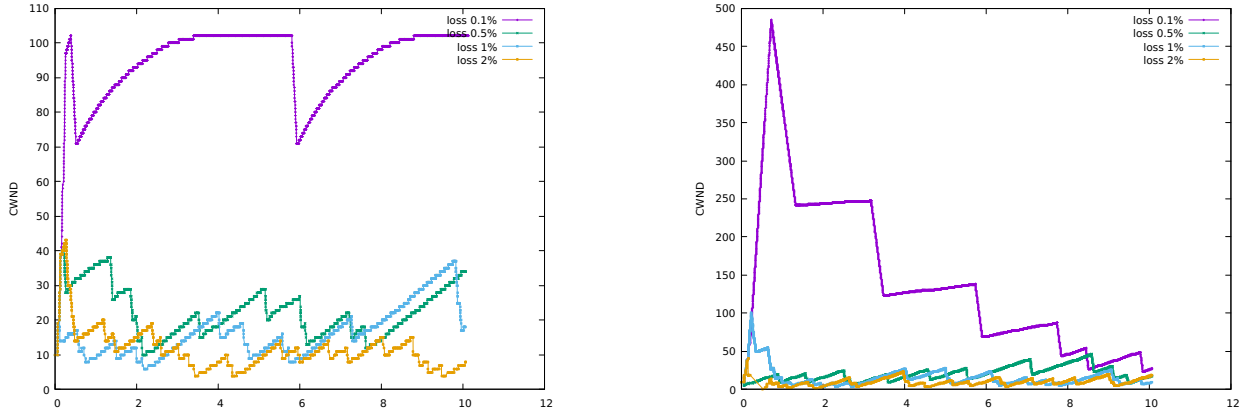
Cubic is the default **CCA** in the `Linux` distributions; **Cubic** tries to improve over **New Reno** with its less aggressive `congestion avoidance` phase. After a loss, the **CWND** will not halve like in **Reno**, but it will be multiplied by $\beta = 0.7$ (as stated in the **RFC8312**, Section 4.5); the **CWND** will then follow the curve described by the following equation (`congestion avoidance` phase):

$$CWND = C\left(T_{\text{elapsed}} - \sqrt[3]{\frac{w_{\max}(1-\beta)}{C}}\right)^3 + w_{\max}$$

where T_{elapsed} is the time elapsed since the last congestion event and $w_{\max}$ is the **CWND** value before the loss. The inflection point will be at same value of the previous **CWND**, and this causes the actual **CWND** value to start raising from below the inflection point of the cubic function, thus guaranteeing a more stable raise near the inflection point, which should be close to where packet loss occurred; this is the theoretical point of congestion of the network.

The cubic approach also achieves a faster increase of the **CWND** during the first**RTT**s of the `congestion avoidance` phase: making a comparison with **New Reno**, it's possible to see, in figure 5.1, how the two algorithms are affected by the losses: on one hand, with 0.1% `packet loss`, we can see how well **Cubic** performs, basically regaining its entire `CWND` after the losses, while **New Reno** drops under a **CWND** of 70 after many losses, not being able to recover fast enough.

This kind of algorithms performs very poorly with losses; their curve is driven downward very rapidly by the losses and they are called **Black Box** because they do not get any information from the state of the network, but only base their predictions on the amount of losses they get. We can see that, even with very low `packet loss` percentages like 1%, the graph shows that it is not able to recover fast enough, which is a problem as the cause of the losses is not always a symptom of a congested network, but could also be caused by a poor Wi-Fi connection that may lead to some losses. More modern algorithms also try to take into consideration the **RTT** of the packets, just to avoid this main issue; some of them are **Vegas** and **BBR**.



(a) Cubic CWND with 50ms delay

(b) Reno CWND with 50ms delay

Figure 5.1: Cubic vs New Reno

6 Highspeed TCP

Highspeed TCP is another loss-based `CCA` for **TCP**; it tries to overcome the flaws of it and to improve the performances; one of the main issues with **Reno** is that, as we already said, losses during a connection have a great impact on the reduction of the value of the **CWND**, reducing the performances. The approach that **Highspeed TCP** takes is that, when we have an high-speed connection and we want to saturate the channel bandwidth, after a certain threshold, **Highspeed TCP** does not rely on the `AIMD` approach, but uses a more aggressive `Multiplicative Increase - Multiplicative Decrease`, where the decrease is meant to be less than half of the **CWND**. **Highspeed TCP** calculates the new **CWND** in function of the `packet-drop rate` p, and to achieve this, there are four constants that can be fine tuned to obtain ideal performances:

$$CWND = \left(\frac{p}{\text{Low\_P}}\right)^{S} \text{Low\_Window}$$

where S is

$$S = \frac{\log \text{High\_Window} - \log \text{Low\_Window}}{\log \text{High\_P} - \log \text{Low\_P}}$$

This formula is only used for **CWND** above `Low_Window`, while below that the classic `Additive Increase` is used. From our experiments, we did not experience a noticeable steepening in the **CWND** graphs of **Highspeed TCP**, that otherwise would have a exponential curve, probably due the maximum throughput being too low (just 10Mb/s), and consequentially being not enough to trigger the exponential growth after a loss. Still, in comparison to **Reno**, there are cases in which **Highspeed TCP** performs better, like in

figure 6.1: the graphs show the next `Seq-No.` to send over time, where the curves are grouped by **loss** and **delay**; the same tuples have the same colours and **Reno** is a dashed line while **Highspeed** is a solid line. We can observe that the **Highspeed TCP** is almost always on top of **Reno**, and even in the cases where **Reno** surpasses, **Highspeed TCP** still has a greater derivative, hinting that despite many losses, not a lot of throughput was lost.

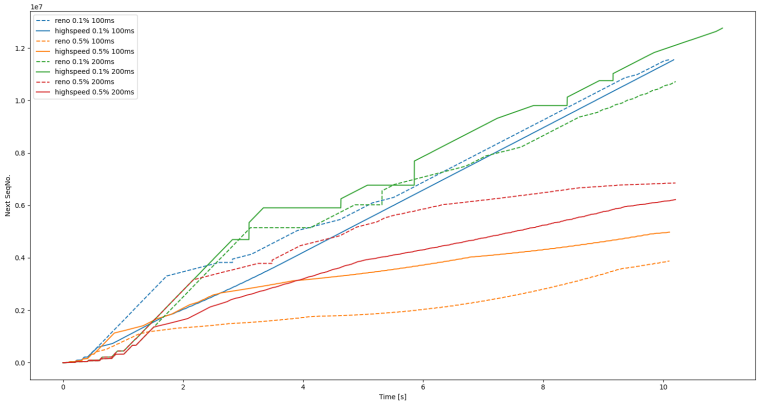


Figure 6.1: Reno vs Highspeed TCP

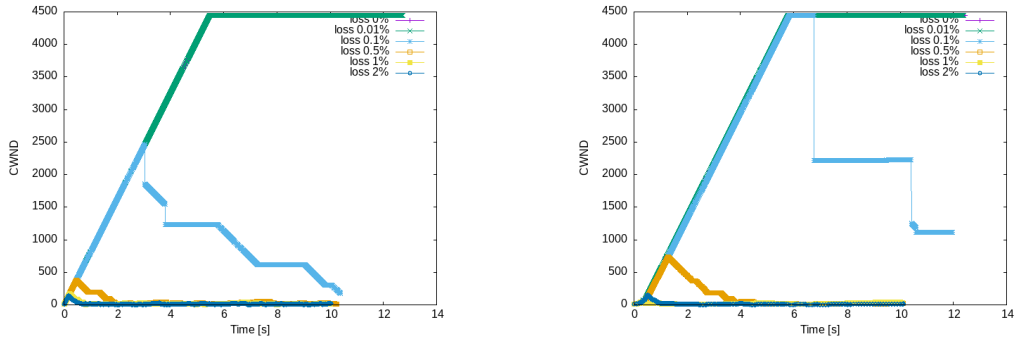
7 DCTCP

`Data Center TCP` (**DCTCP**) congestion control is an an algorithm developed to sustain the needs of large data centers. In general, the traffic in a data center is composed of both long and short flows, that require respectively high throughputs and low latencies; moreover, data centers can often experience sudden bursts, for which many servers address their traffic to the same server at the same time. Considering these aspects, the switches used in data centers need to search for the right trade off between short queues for short flows and long queues for bursts and long flows. **RFC3168** proposes the `Explicit Congestion Notification` (**ECN**) from switches to detect the presence of congestions, but not the extent; in case of not severe congestion, the TCP congestion window is reduced way too much and the throughput of long flows decreases unnecessarily.

DCTCP enhances the **ECN** approach to estimate the number of bytes that encounter congestion, modifying the **CWND** accordingly. Going more in depth, the **ECN** approach expects a specific flag (**CE**) in the `IP header` to be set when a packet reaches a switch and the queue in greater than a congestion threshold; then the receiver keeps setting the Explicit Congestion Notification Echo flag (**ECE**) until a packet with the `Congestion Window Reduced` (**CWR**) flag is set; the problem here is that, in **DCTCP**, it is necessary to have more detailed information from the sender about the congestion, as mentioned before. To achieve so, **DCTCP** adopts a new **TCP** state variable to keep the estimated fraction of bytes sent that encountered congestion, called **DCTCP.Alpha** and initialized to 1; this variable gets updated according to a specific algorithm and the sender will use it to update the **CWND** as follows:

$$CWND = CWND * \left(1 - \frac{\text{DCTCP.Alpha}}{2}\right)$$

It can be observed that, unless the estimated fraction of bytes sent that encountered congestion remains equal to 1, the **CWND** will not be halved, but it will be reduced by a smaller amount, as the algorithm proposed to achieve. Taking a look at the captures of the experiments conducted with different losses and different delays (figure 7.1), it can be noticed that, when the loss increases, the congestion window gets scaled back in a way that attempts to limit the reduction; of course, it is not easy to predict how the sender is going to update the **DCTCP.Alpha** parameter, as many details are up to the implementation.



(a) DCTCP 10ms delay

(b) DCTCP 100ms delay

Figure 7.1: DCTCP: 10ms vs 100ms delay

8 Vegas

Vegas TCP is a delay-based `Congestion Control` algorithm, which means that, during the connection, also the `RTT` is taken into consideration when updating the `CWND`. In **Reno**, we know that a re-transmission can only happen after three duplicate `ACK`s are received by the sender, while **Vegas** takes a different approach: it saves for each segment a `timestamp` and, after receiving a duplicated `ACK`, it will check if the difference between the `timestamp` and the current `CPU time` is greater than the `timeout`; if it is, it will re-send the segment. If **Vegas** receives an unacknowledged `ACK` that is the first or second after the re-transmission, it will check again if difference between the `timestamp` of the last packet it sent and the current `CPU time` is greater than the `timeout`; if it is, it will re-send the segment in in case a loss happend (it is to be recalled that **Reno** would have had to wait 3 duplicated `ACK`s). **Vegas** also has a different approach during the `congestion avoidance` phase, as, after two round trip delays, the`CWND` is calculated as follows:

$$(CWND_{\text{current}} - CWND_{\text{previous}}) \cdot (RTT_{\text{current}} - RTT_{\text{previous}})$$

If this value is positive, the `CWND` gets reduced by one-eight, while if it is negative or zero (either the `CWND` was just reduced or we are faster or equal than the previous sample), it is increased by one. This brings **Vegas** to oscillate around a stable point as we can clearly see in figure 8.1. Although **Vegas** tries to improve on the **Reno**'s limitations, it also introduces some new ones: unfortunately, **Vegas** under-performs when there is a high delay and the reason is that the `CWND` update is only done after two round trip delays; if there is a high delay, the `CWND` will not be updated very frequently and we can observe in figure 8.2 how this impacts on the overall speed of the connection; in the case of 500ms delay with just a few losses, the `CWND` is able to gain back some growth.

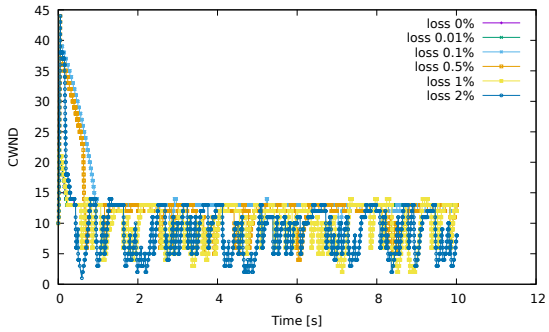
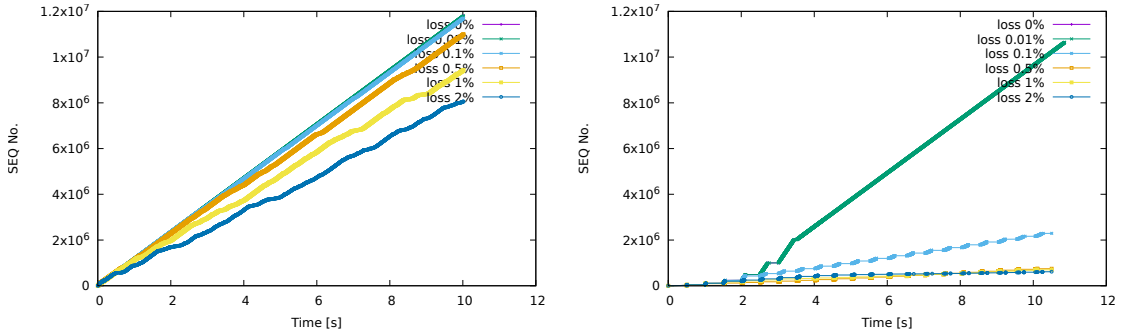


Figure 8.1: Vegas 10ms delay



(a) Vegas 10ms delay

(b) Vegas 500ms delay

Figure 8.2: Vegas 10ms vs 500ms delay

9 BBR

`Bottleneck Bandwidth and RTT` (**BBR**) congestion control is an algorithm developed by `Google`; the aim of this algorithm is to try and maximise the bandwidth of the connection. Unlike **Reno** or **Cubic**, which are based on packet losses to make adjustments to the `CWND`, **BBR** tries to guess how congested is the network in order to adjust its `CWND`; as a matter of fact, **BBR** resides in the kind of `CCA`s called **Grey Box** (not **White Box** because the information it gets is not the real state of the network, but it is just an internal model which may not reflect reality). **BBR** has four phases, which alternate between themselves, and they are **Startup**, **Drain**, **Probe Bandwidth**, **Probe RTT**; the most important ones are **Probe Bandwidth** and **Probe RTT**, as during these two phases, for each **ACK** received, **BBR** updates its internal values as follows:

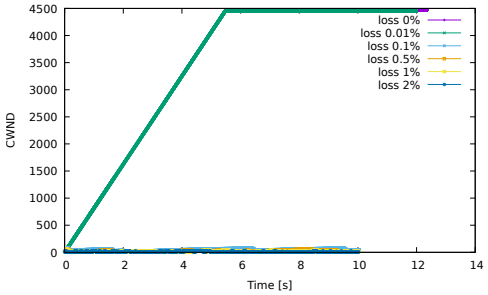
```
feareach ack
    bottleneck_bandwidth = windowed_max(delivered / elapsed, 10 round trips)
    min_rtt = windowed_min(rtt, 10 seconds)
pacing_rate = pacing_gain * bottleneck_bandwidth
cwnd = max(cwnd_gain * bottleneck_bandwidth * min_rtt, 4)
```

We can see how the `bottleneck_bandwidth` is calculated as the number of `packets delivered` divided by the `elapsed` from the first one, and the packets considered are inside a window which is big at max `10 round trips`, where each round is a burst that **BBR** sends. The `pacing_rate` is another important parameter which depends on the `bottleneck_bandwidth` and on the `pacing_gain`, which changes during each of the phases of **BBR**. Unlike the other `CCA`s, **BBR** bases how fast it sends packets on both the `CWND` and the `pacing_rate`; considering what is written on the documentation, the `CWND` is a countermeasure to stop **BBR** when it is in the `Probe Bandwidth` phase, because otherwise it would consume too much bandwidth, filling up the router's queues. About the other two phases, one is **Startup**, which is the beginning of the connection where the `CWND` increases like **Cubic**, until the maximum bandwidth is reached; the other phase is **Drain**, which has been implemented because **BBR** is not particularly disposed to share the bandwidth of a shared channel with other `CCA`s like **Reno** or **Cubic**. In order to avoid eating up the whole bandwidth, for a short period of time it decreases on purpose the `pacing_gain`, so that, if other loss-based `CCA`s like Reno need to take their share of the channel, the minimum `RTT` will increase for **BBR**, thus decreasing its `CWND`; after a certain amount in time, given that the event is happening in a stable system, they will come to a situation where they both share equally the channel.

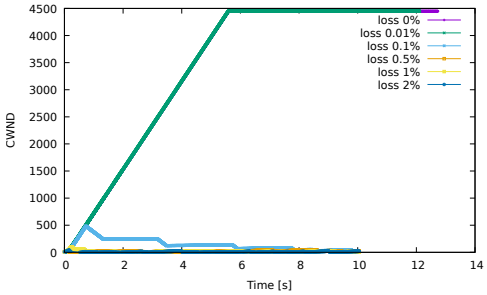
This characteristic of maximising the bandwidth provides **BBR** with very high resistance to packet losses; we can clearly see in figure 10.3 and in figure 10.4 how **BBR** maintains a high throughput on the channel also with a `2% packet loss`, while **Reno** and **Cubic** reach very low performances. This is even more noticeable when there is high delay, and we can observe that **Cubic**, with a `500ms delay` and `2% packet loss probability`, exchanges just a total of 10^6 bytes over 10 seconds on a 10Mb/s channel, while `BBR` is one degree of magnitude above him in the same conditions.

10 Appendix

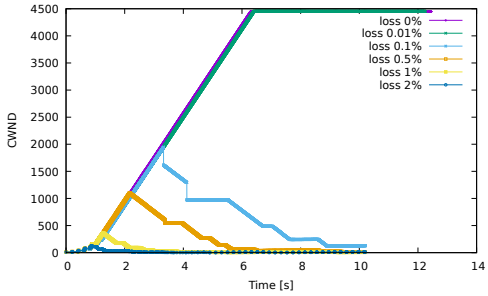
10.1 Reno



(a) Reno CWND with 10ms delay

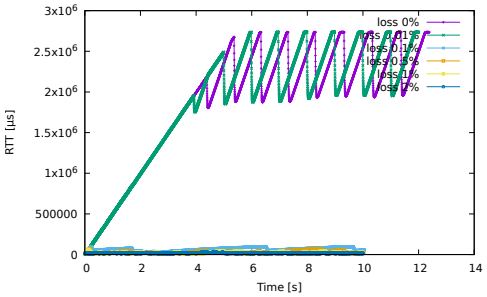


(b) Reno CWND with 50ms delay

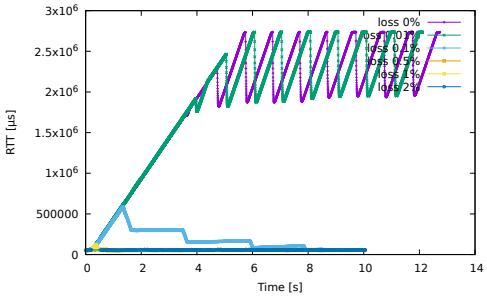


(c) Reno CWND with 200ms delay

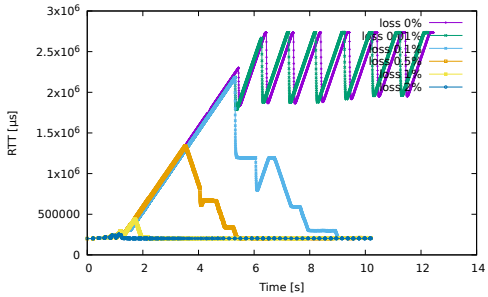
Figure 10.1: Reno CWND with different delay comparison



(a) Reno RTT with 10ms delay

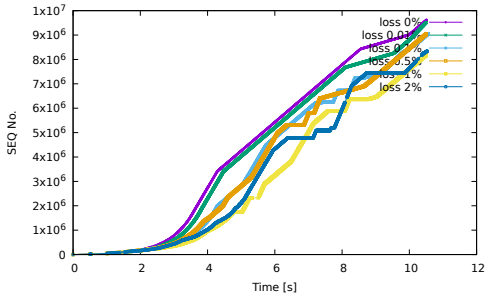


(b) Reno RTT with 50ms delay

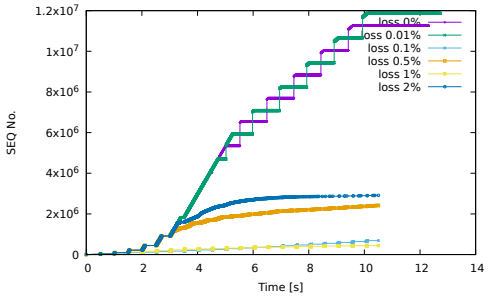


(c) Reno RTT with 200ms delay

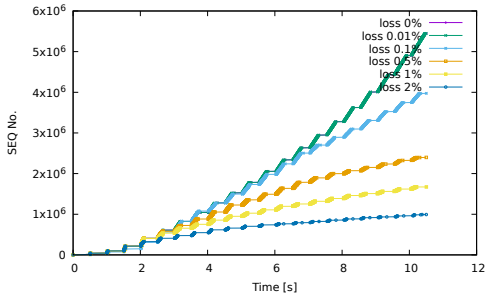
Figure 10.2: Reno RTT with different delay comparison



(a) BBR with 500ms delay



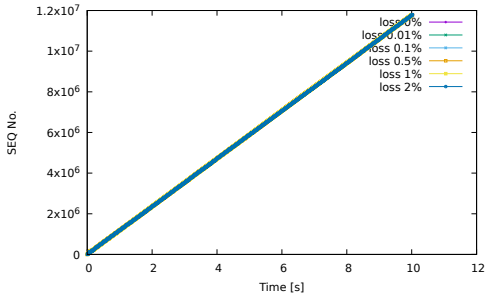
(b) Reno with 500ms delay



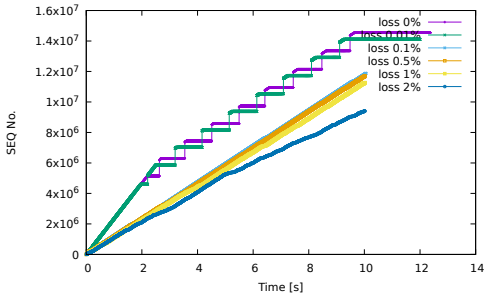
(c) Cubic with 500ms delay

Figure 10.4: BBR vs Reno vs Cubic (500ms delay)

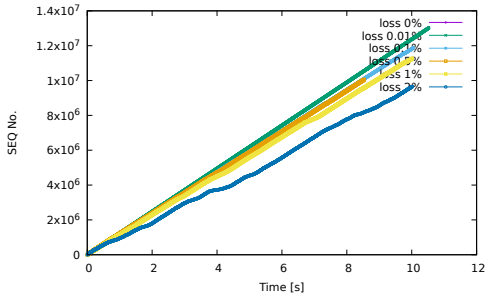
10.2 BBR vs Reno vs Cubic



(a) BBR with 10ms delay



(b) Reno with 10ms delay



(c) Cubic with 10ms delay

Figure 10.3: BBR vs Reno vs Cubic (10ms delay)

10.3 Script to generate graphs

```
#!/bin/bash

if [ -z $1 ]; then
  echo "no parameter"
  exit 1
fi

# insert the tcp_diag module
modprobe tcp_diag
# enable the tcp log tracing
cd /sys/kernel/debug/tracing
echo 1 > events/tcp/tcp_probe/enable
cd -   # get back to the previous folder

#disable tcp metric saving
sysctl net.ipv4.tcp_no_metrics_save=0
#enable/disable SACK
sysctl net.ipv4.tcp_sack=1

INTF=enp2s0

ethtool -K ${INTF} rx off tx off sg off tso off gso off gro off

tc qdisc add dev ${INTF} root netem

algos=("cubic" "reno" "vegas" "dctcp" "highspeed" "bbr")
losses=("0%" "0.01%" "0.1%" "0.5%" "1%" "2%")

function plot_algos {
  mkdir -p afig.cwnd
  mkdir -p afig.sstresh
  mkdir -p afig.seqno
  mkdir -p afig.rtt

  # Plot scripts toghether
  for algo in ${algos[@]}; do
    PLOT_FILES=""
    TITLES=""
    for loss in ${losses[@]}; do
      PLOT_FILES="${PLOT_FILES} ${algo}_${loss}_${1}/data"
      TITLES="${TITLES}'loss $(sed 's/%/\\%/' <<< ${loss})' "
    done
    echo ${PLOT_FILES[*]}

    gnuplot <<EOF
set term pdf
set pointsize 0.3

files='${PLOT_FILES}'
titles="${TITLES}"

set output "afig.cwnd/${algo}_${1}.pdf"
set xlabel "Time [s]"
set ylabel "CWND"
ofs = 0
plot for [i=1:words(files)] word(files, i) using (ofs = (\$0 == 0 ? strcol(1) : ofs), strcol
    (1) - ofs):5 title word(titles, i) with linespoint

set output "afig.sstresh/${algo}_${1}.pdf"
set xlabel "Time [s]"
set ylabel "SSTRESH"
ofs = 0
plot for [i=1:words(files)] word(files, i) using (ofs = (\$0 == 0 ? strcol(1) : ofs), strcol
    (1) - ofs):(\$6 < 2000000 ? \$6 : 0) title word(titles, i) with linespoint
```

```
63
64 set output "afig.seqno/${algo}_${1}.pdf"
65 set xlabel "Time [s]"
66 set ylabel "SEQ No."
67 ofs = 0
68 seq = 0
69 plot for [i=1:words(files)] word(files, i) using (ofs = (\$0 == 0 ? strcol(1) : ofs), strcol
    (1) - ofs):(seq = (\$0 == 0 ? strcol(3) : seq), strcol(3) - seq >= 0 ? strcol(3) - seq :
     1/0) title word(titles, i) with linespoint
70
71 set output "afig.rtt/${algo}_${1}.pdf"
72 set xlabel "Time [s]"
73 set ylabel "RTT [ s ]"
74 ofs = 0
75 plot for [i=1:words(files)] word(files, i) using (ofs = (\$0 == 0 ? strcol(1) : ofs), strcol
    (1) - ofs):8 title word(titles, i) with linespoint
76 EOF
77   done
78 }
79
80 mkdir $1
81 cd $1
82
83 #plot_algos $1
84 #exit 0
85
86
87 ALLOW=""
88 for algo in ${algos[@]}; do
89   modprobe tcp_${algo}
90   ALLOW="${ALLOW}${algo} "
91   sysctl net.ipv4.tcp_allowed_congestion_control=$(echo ${ALLOW})
92
93   for loss in ${losses[@]}; do
94     mkdir ${algo}_${loss}_$1
95     cd ${algo}_${loss}_$1
96
97     #set link properties - note that you first have to ADD the netem qdisc before CHANGE it
98     #change your params
99     tc qdisc change dev ${INTF} root netem loss ${loss} delay $1
100    echo tc qdisc change dev ${INTF} root netem loss ${loss} delay $1
101
102    # reset the log file so that we don't get past values
103    echo > /sys/kernel/debug/tracing/trace
104
105    # follow the output of trace, the file has capped lenght
106    TRACE_FILE=trace
107    #tail --follow /sys/kernel/debug/tracing/trace > $TRACE_FILE &
108    #TAIL_PID=$!
109
110    SRC_PORT=12345
111    SERVER="10.0.3.1"
112    iperf3 -c ${SERVER} -C ${algo} --cport ${SRC_PORT}
113
114    #now extract data
115    # NOTE: you need to filter the proper TCP connection
116    # TBD: may be it's possible to automatize this by parsing iperf3 output?
117
118    cat /sys/kernel/debug/tracing/trace > $TRACE_FILE
119
120    # get time
121    cat $TRACE_FILE |grep cwnd| grep $SRC_PORT| tr -s ' '| cut -d ' ' -f 5|cut -d':' -f 1 >
    time
122    #get data len
123    cat $TRACE_FILE |grep cwnd| grep $SRC_PORT|cut -d '=' -f6|cut -d ' ' -f 1 >len
124    #get SeqNo
```

```
    cat $TRACE_FILE |grep cwnd| grep $SRC_PORT|cut -d '=' -f7|cut -d ' ' -f 1 >seqno
    #get ack
    cat $TRACE_FILE |grep cwnd| grep $SRC_PORT|cut -d '=' -f8|cut -d ' ' -f 1 >una
    #get CWND
    cat $TRACE_FILE |grep cwnd| grep $SRC_PORT|cut -d '=' -f9|cut -d ' ' -f 1 >cwnd
    #get ssthresh
    cat $TRACE_FILE |grep cwnd| grep $SRC_PORT|cut -d '=' -f10|cut -d ' ' -f 1 >ssthresh
    #get snd_wnd
    cat $TRACE_FILE |grep cwnd| grep $SRC_PORT|cut -d '=' -f11|cut -d ' ' -f 1 >snd_wnd
    #get rtt
    cat $TRACE_FILE |grep cwnd| grep $SRC_PORT|cut -d '=' -f12|cut -d ' ' -f 1 >rtt

    #put everything together
    paste time len seqno una cwnd ssthresh snd_wnd rtt > data


    #now just quick&dirt plot it - TBD: better prepare a plot.gnu
    gnuplot << EOF
set term png
set xlabel "Time [s]"

#TBD: set proper labels
set ylabel " Len "
set out "len_$algo.png"
plot 'data' using 1:2 title "" with linespoint

set ylabel " SeqNo "
set out "SeqNo_$algo.png"
plot 'data' using 1:3 title ""  with linespoint

set ylabel " UNA "
set out "una_$algo.png"
plot 'data' using 1:4 title "" with linespoint

set ylabel " CWND "
set out "cwnd_$algo.png"
plot 'data' using 1:5 title "" with linespoint

set ylabel " ssthresh "
set out "ssthresh_$algo.png"
plot 'data' using 1:6 title "" with linespoint

set ylabel " SND WND "
set out "snd_wnd_$algo.png"
plot 'data' using 1:7 title "" with linespoint

set ylabel " rtt "
set out "rtt_$algo.png"
plot 'data' using 1:8 title "" with linespoint

EOF

    cd ..
  done
done

plot_algos $1
```