# Networking Lab 8 – Impact of packet loss on Throughput

In the previous laboratory, we have observed how the goodput changes when considering different scenarios, with TCP or UDP, with half/full duplex links, and with wired or WiFi links. In all cases, the RTT was very small, and the link capacity was either in the order of 10Mb/s or more. In this laboratory, instead, we consider some simple scenarios where a single TCP stream is used to transmit data, but under different network conditions. We learn how to control RTT and the packet loss, and how these affect TCP goodput.

To emulate and control the link path, we use the *Linux Traffic Control project*. Traffic control is the name given to the sets of queuing discipline (QDISC) systems and mechanisms by which packets are received and transmitted on a Linux router. This includes deciding which (and whether) packets to accept at what rate on the input of an interface, and determining which packets to transmit in what order at what rate on the output of an interface. We use two modules:

- `netem`: Linux Network Emulator
- `TBF`: Linux Token Buffer Filter

`tc` is used to configure Traffic Control in the Linux kernel. The generic syntax to configure it is

```
tc qdisc add dev DEV root QDISC QDISC-PARAMETERS

tc qdisc change dev DEV root QDISC QDISC-PARAMETERS

tc qdisc del dev DEV root

tc qdisc show dev DEV root
```

**TC COMMANDS**

The following commands are available for qdisc, classes and filter:

`add` Add a qdisc, class or filter to a node. For all entities, a **parent** must be passed, either by passing its ID or by attaching directly to the `root` of a device. When creating a qdisc or a filter, it can be named with the **handle** parameter. A class is named with the **classid** parameter.

`delete` A qdisc can be deleted by specifying its handle, which may also be 'root'. All subclasses and their leaf qdiscs are automatically deleted, as well as any filters attached to them.

`change` Some entities can be modified 'in place'. Shares the syntax of 'add', with the exception that the handle cannot be changed and neither can the parent. In other words, change cannot move a node.

`replace` Performs a nearly atomic remove/add on an existing node id. If the node does not exist, it is created.

`show` report the status of the node id.

**NetEm qdisc**

NetEm provides Network Emulation functionality for testing protocols by emulating the properties of wide area networks. NetEm is an enhancement of the Linux traffic control facilities that allow to **add delay**, **packet loss**, duplication and more other characteristics to packets outgoing from a selected network interface.

## 1. Emulating wide area network delays

**WARNING: disable offloading capabilities of the Ethernet driver since those are known to interfere with linux TC.**

This is the simplest example; it just adds a fixed amount of delay to all packets going out of the local Ethernet.

```
# tc qdisc add dev eth0 root netem delay 100ms
```

Now a simple ping test to host on the local network should show an increase of 100 milliseconds. The delay is limited by the clock resolution of the kernel (HZ). On most 2.4 systems, the system clock runs at 100hz which allows delays in increments of 10ms. On 2.6, the value is a configuration parameter from 1000 to 100 hz.

You can change the delay with

```
# tc qdisc change dev eth0 root netem delay 10ms
```

Real wide area networks show variability so it is possible to add random delay variation. Supported random variables can be specified via the following syntax:

```
DELAY := delay TIME [ JITTER [ CORRELATION ]]]

      [ distribution { uniform | normal | pareto |  paretonormal } ]
```

```
# tc qdisc change dev eth0 root netem delay 100ms 10ms
```

This causes the added delay to be 100ms ± 10ms, according to a uniform distribution.

**Delay distribution**

Typically, the delay in a network is not uniform. It is more common to use a something like a normal distribution to describe the variation in delay. The netem discipline can take a table to specify a non-uniform distribution.

```
# tc qdisc change dev eth0 root netem delay 100ms 20ms distribution normal
```

The actual tables (normal, pareto, paretonormal) are generated as part of the iproute2 compilation and placed in /usr/lib/tc; so it is possible with some effort to make your own distribution based on experimental data.

**Packet loss**

Random packet loss is specified in the 'tc' command in percent.

```
# tc qdisc change dev eth0 root netem loss 0.1%
```

This causes 1/10th of a percent (i.e 1 out of 1000) packets to be randomly dropped, according to an i.i.d. process. An optional correlation may also be added. This causes the random number generator to be not i.i.d. anymore, and can be used to emulate packet burst losses.

```
# tc qdisc change dev eth0 root netem loss 0.3% 25%
```

This will cause 0.3% of packets to be lost, and each successive probability depends by a quarter on the last one, i.e., $Ploss_n = .25 * Ploss_{n-1} + .75 * Random(0.3\%)$

## 2. TBF – Token Buffer Filter

The Token Bucket Filter is a classless queueing discipline available for traffic control with the `tc` command. TBF is a pure shaper and never schedules traffic. It is non-work-conserving and may throttle itself, although packets are available, to ensure that the configured rate is not exceeded.

As the name implies, traffic is filtered based on the expenditure of **tokens.** Tokens roughly correspond to bytes, with the additional constraint that each packet consumes some tokens, no matter how small it is. This reflects the fact that even a zero-sized packet occupies the link for some time. On creation, the TBF is stocked with tokens which correspond to the amount of traffic that can be burst in one go. Tokens arrive at a steady rate, until the bucket is full. If no tokens are available, packets are queued, up to a configured limit. The TBF now calculates the token deficit, and throttles until the first packet in the queue can be sent. If it is not acceptable to burst out packets at maximum speed, a peakrate can be configured to limit the speed at which the bucket empties. This peak rate is implemented as a second TBF with a very small bucket, so that it doesn't burst.

This limit is caused by the fact that the kernel can only throttle for at minimum 1 'jiffy', which depends on HZ as 1/HZ. For perfect shaping, only a single packet can get sent per jiffy - for HZ=100, this means 100 packets of on average 1000 bytes each, which roughly corresponds to 1mbit/s.

To attach a TBF with a sustained maximum rate of 5mbit/s, a 5kilobyte burst, with a pre-bucket queue size limit calculated so the TBF causes at most 10ms of latency, issue:

```
# tc qdisc add dev eth0 root tbf rate 5mbit burst 5kb latency 10ms
```

**Rate control**

By combining NetEm and a TBF is then possible to emulate a link with given delay, packet loss and capacity:

```
# tc qdisc add dev eth0 root handle 1:0 netem delay 100ms

# tc qdisc add dev eth0 parent 1:1 handle 10: tbf rate 5mbit burst 5kb
latency 10ms
```

Check on the options for buffer and limit as you might find you need bigger defaults than these (they are in bytes)

## 3. TCP Probe

Linux offers lots of possibilities for experimenting with TCP. The Linux kernel allows one to easily change the TCP congestion control algorithm, and to print internal TCP state variables, like congestion and receiver windows (CWND or RWND). `tcpprobe` is a kernel module that records the state of a TCP connection in response to incoming packets. To enable it, you need to load the module via the `modprobe` command. On the host you monitor, do:

```
H1: modprobe tcp_probe port=5001 full=1 bufsize=80
```

This enables the logging of TCP statistics on flows using TCP port 5001 (e.g., used by `iperf`). To check the log, you need to monitor the output that is recorded to virtual file in the `/proc/net/tcpprobe` file handle. For instance, on H1

```
cat /proc/net/tcpprobe >/tmp/tcpprobe.out &    # start logging data
sleep 1                # just wait 1s to be sure the logger is started…
pid=$!                 # get the cat process ID to later kill it
iperf -c H2            # run the iperf test
kill $pid              # now kill the logger
```

The `tcpprobe` capture file contains one line for each packet sent, according to the following syntax (warning: some extra columns may be present depending on the version of the Linux Kernel you are using)

```
[1]     [2]             [3]              [4] [5]      [6]       [7][8]       [9]

0.073678 10.8.0.54:38644 192.168.1.42:5001 24 0xb6b19bb 0xb6b19bb 2  2147483647 5792
```

```
[1] Time seconds since tcpprobe was loaded
[2] Sender address:port
[3] Receiver address:port
[4] Packet length in Bytes
[5] snd_nxt: Sequence Number of the next packet to send
[6] snd_una: Sequence Number of the first unacknowledged packet
[7] snd_cwnd: Sender congestion window
[8] snd_ssthresh: Slow start threshold; -1 not known yet
[9] snd_wnd: Send Window size in Bytes
```

An event is logged **at the reception of a segment**. It works by inserting a hook into the tcp_recv processing path so that the congestion window and sequence number can be captured.

This text file can be easily filtered and modified with standard tools. A common usage is to make a plot of congestion window and slow start threshold over time using `gnuplot`.

```
set data style linespoints
show timestamp
set xlabel "time (seconds)"
set ylabel "Segments (cwnd, ssthresh)"
plot "/tmp/tcpprobe.out" using 1:7 title "snd_cwnd", \
     "/tmp/tcpprobe.out" using 1:($8>=2147483647 ? 0 : $8) title
"snd_ssthresh"
```

## 4. Impact of RTT and Packet Loss on Congestion Control

Before doing the tests, first double check the system configuration. Verify that Ethernet card is set to 100Mb/s Full Duplex, and offloading capabilities are disabled.

In addition, by default, Linux TCP implements a caching feature that remembers the last valid slow start threshold (ssthresh) achieved in previous connections to the same host. This modifies the result of the further tests, and can causes issues when the properties of the path to the same host have changed in the meanwhile. If the test involves repeated connections, you should also turn off the route caching metrics:

```
sysctl -w net.ipv4.tcp_no_metrics_save=1
```

Check which congestion control algorithms are supported by your Linux distribution. Those are kernel modules that can be loaded on demand. Check which are available by just listing the files in

```
ls /lib/modules/`uname -r`/kernel/net/ipv4/tcp_*
```

You can then load some module using the `modprobe` tool

```
modprobe tcp_vegas #load tcp_vegas for testing
```

and enable it by default by

```
sysctl net.ipv4.tcp_congestion_control=vegas
```

To check which is the currently default TCP congestion control, use

```
sysctl net.ipv4.tcp_congestion_control
```

To check which are the currently enabled, use

```
sysctl net.ipv4.allowed_tcp_congestion_control
```

1. Which is the default congestion control on your distribution?
2. Write a script that loads all TCP congestion control modules, enables them, then turn TCP RENO as default

The goal of this laboratory is to verify the impact of RTT and packet loss (p_loss) on TCP congestion control performance. To this end, we will

- Use linux TC to control RTT and p_loss on the sender
- Use tcp_probe to log detailed statistics of TCP tests
- Use iperf to run the test
- Use gnuplot to plot results

We will consider the impact of TCP parameters:

- Maximum allowed values for the TCP congestion window (`-w` option on `iperf`)
- Usage of multiple TCP connection in parallel (`-P` option in `iperf`)
- TCP congestion control algorithm (`-Z` option on `iperf`)

The goal is to obtain plots for different values of the above system parameters:

- Application goodput versus RTT
- Application goodput versus p_loss

Here are some possible experiments to be done. For each of them, define the goals of your experiment, describe how to setup the testbed, define the parameter range, and then run the experiment. Report and comment the results. To highlight differences, you must choose a proper scenario, e.g., consider small/large RTT, with/without losses. When choosing the parameters, some ingenuity must be used to properly set the ranges, eventually increasing the experiments in regions where the changes are sharper.

Test A – impact of RTT - Plot the goodput versus RTT in [0:300]ms range for:

1. different values of the congestion window
2. increasing number of parallel TCP connections
3. different TCP congestion control algorithms

Test B – impact of p_loss - Plot the goodput versus p_loss in [0:10]% range for:

1. RTT equal to 10, 50, 100, 200ms
2. RTT equal to 50ms, and different TCP congestion control algorithms

Test C – impact of congestion control algorithm - Plot the evolution of the cwnd for different congestion control algorithms, for:

1. For small RTT (e.g., 10ms), p_loss=0 and p_loss=0.5%
2. For large RTT (e.g., 300ms), p_loss=0 and p_loss=0.5%

Test D: TCP fairness: consider the scenario RTT=50ms, p_loss=0.1%. Consider two TCP flows that compete for the link capacity. One uses RENO, the second uses any other algorithm X. Compare

1. The evolution versus time of the congestion window of each flow
2. The average goodput obtained of TCP reno and TCP X

Instead of enforcing a p_loss=0.1% using netem, you could also consider the actual losses induced by the two TCP flows that compete for capacity in a shared link: H1 and H2 send data to H3, so that congestion is created on the output port from the switch to H3 (hint: disable PAUSE frame otherwise no losses will happen).

Suggestions: Choose the appropriate duration to get a reliable test. Repeat the tests several times. Write a script to run test automatically, save results, and then use gnuplot to check what you have obtained.

Here is an example.

```
#!/bin/bash

# choose parameter
TARGET=192.168.1.112  # iperf server to contact
DELAY=3ms             # extra delay for netem
LOSS="1%"             # losses for netem
LIMIT=1000            # limit for netem
DEV=eth0              # which Ethernet card to use
TCP_WIN="512k"        # default tcp buffer size (note to self: remember to
                      check the real one iperf will use)

# choose which TCP congestion control algorithms to test
# CongContr="bic cubic reno highspeed htcp hybla illinois lp scalable vegas
veno westwood yeah"
CongContr="bic cubic reno vegas"

# insert the tcp_probe module and bind it to iperf port.
# Enable full logging, and limit the buffering to 80 lines to be sure
# everything will be logged
rmmod tcp_probe
modprobe tcp_probe port=5001 full=1 bufsize=80

#configure linux tc
tc qdisc add dev $DEV root netem
tc qdisc change dev $DEV root netem delay $DELAY loss $LOSS limit $LIMIT
tc qdisc show dev $DEV

# run ping to be sure that everything is in place
ping $TARGET -c 3

# By default, TCP saves various connection metrics in the route cache
# when the connection closes, so that connections established in the
# near future can use these to set initial conditions.  Usually, this
# increases overall performance, but may sometimes cause performance
# degradation.  If set, TCP will not cache metrics on closing
# connections.
sysctl -w net.ipv4.tcp_no_metrics_save=1

# Disable/enable selected acknowledgments (SACKS)
sysctl -w net.ipv4.tcp_sack=0

# now run the tests versus different congestion control algorithms
for CC in $CongContr
do
  echo "Testing $CC as Congestion Control"
  # start the TCP logger
  cat /proc/net/tcpprobe >tcp_data_${CC}.out &
  # wait for 1 s to let cat start properly
  sleep 1
  pid=$!
  iperf -c $TARGET -t 10 -Z $CC -w $TCP_WIN
  sleep 1
  kill $pid
done
# now plot them with gnuplot
gnuplot plot.gnu
```

```
# This is plot.gnu file

 set terminal png
 set out “TCP-cwnd-vs-cong-control.png”
 set style data lines
 set title "TCP Congestion Control"
 set xlabel "time (seconds)"
 set ylabel "Segments (cwnd, ssthresh)"
 plot \
    "tcp_data_reno.out" using 1:7 title "snd_cwnd - reno", \
    "tcp_data_bic.out" using 1:7 title "snd_cwnd - bic", \
    "tcp_data_cubic.out" using 1:7 title "snd_cwnd - cubic", \
    "tcp_data_vegas.out" using 1:7 title "snd_cwnd - vegas"
```