NDN-Testbed Loss

Sebastian Theuermann (stheuerm@edu.aau.at)

V 1.0 - August 2016

Contents

0 General

The goal is the introduction of a way to emulate packet loss in the NDN-Testbed.
Loss will affect the individual links between nodes, the probability of loss shall be specified per individual link, similar to the already existing emulation of delay.

Files, which may need to be changed.(For actual change proposals, please refer to section 6)

- rand_network.py
- node_parser.py
- deploy_network.py
- Scripts which change the parameters of rand_network.py (e.g. all_strategies_emulation.py)

0.1 Loss: egress vs. ingress

Given two nodes (A,B), connected by a link (A–>B):
Loss on egress (e.g. before leaving A) reduces the output-data rate (physical link utilization). Contrary to that, loss at ingress doesn't change the physical link utilization because packets are discarded after arriving at their destination-node (e.g. B).

0.2 Segmentation-Offloading

Segmentation-offloading reduces the cpu-load caused by TCP-IP. Packets whose size exceeds the MTU are still being split up in seperate packets (segmented), but the operation is deferred as long as possible (performed near Link-Layer, e.g. before adding the packet to the driver-queue/by NIC).

Because segmentation-offloading is deactivated/not available (e.g. UDP-Fragmentation-Offloading) on the BananaPi-routers (please refer to 7), it is not influencing the emulation of packet loss and therefore not further examined.

TCP/UDP - packets larger than the MTU are segemented/fragmented before leaving the IP-stack and being added to a output-queue. Packet loss on egress therefore causes the lost of single IP-fragments. IP-fragments are only assembled when arriving at their target destination, except for intermediate nodes specially instructed to do so, e.g. nodes performing connection-tracking (iptables), NAT, firewalls.

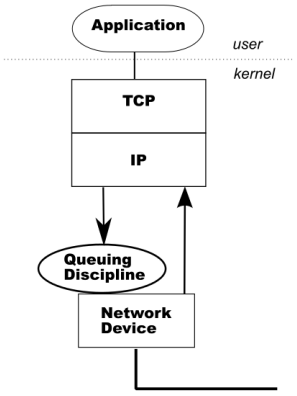
1 netem

The network-emulator netem represents a extension of the existing traffic-control facilities (traffic-shaping by queueing) of the linux kernel. Netem allows amongst other things for the emulation of late packets (delay), packet loss, duplication of packets, reordering of packets.
Netem itself as a classful queueing-discipline can be used in conjunction with other queueing-disciplines in a hassle-free manner.

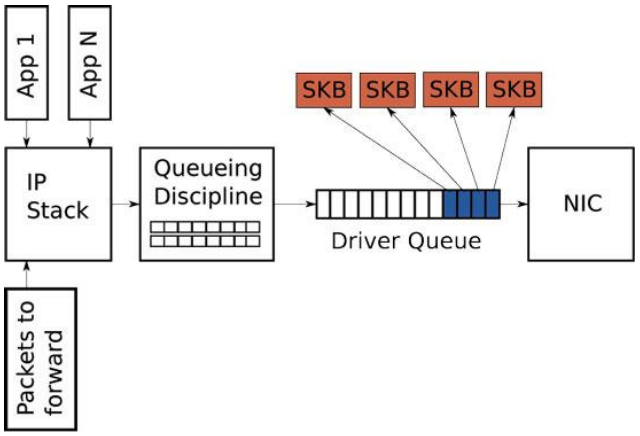
Packet loss in netem is achieved by dropping a percentage of packets,
before they are added to the queue. [1]

The smallest non-zero value possible for the specification of the packet loss probability is $2^{-32} = 0.0000000232\%$. [2]

Netem network-impairments can be readily combined.



(a) Linux-Queuing-Discipline [1]



(b) Queueing im Linux-Netwerkstack [4]

```
tc qdisc add dev eth2 root netem delay 10ms reorder 25% 50% loss 0.2
```

```
       +-------+                  +------+
       |ingress|    +----------+  |egress|
IN --->+qdisc  +--->|netfilter|--->+qdisc +---> OUT
       |       |    +----------+  |      |
       +-------+                  +------+
```

```
man (tc-)netem
```

Compared to iptables, netem offers significantly more ways to influence the probability of packet loss.

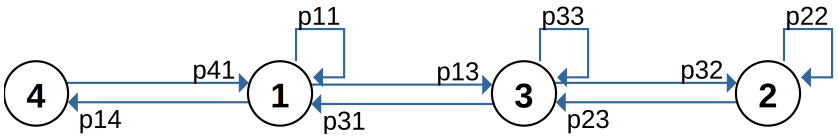
1.0.1 loss random

Adds an independent loss propability to the packets. (Specifying a correlation is possible but deprecated because of bad behaviour)

```
tc qdisc add dev eth0 root netem loss random 0.1%
```

1.0.2 loss state

Adds packet losses according to the 4-state Markov chain using transition-probabilities as input. The parameter



State	
1	good reception
2	good reception within a burst
3	burst losses
4	independent losses

Figure 1: 4 state markov model [15]

p13 is mandatory and if used alone corresponds to the Bernoulli model. The optional parameters allow to extend the model to 2-states (p31), 3-states (p23 and p32) and 4-states (p14). (man netem)

```
sudo tc qdisc add dev eth0 root netem loss state p13 [ p31 [ p32 [ p23 [ p14]]]]
```

1.0.3 loss gemodel

Adds packet losses according to the Gilbert-Elliot loss model (burst error patterns) or its special cases (Gilbert, Simple Gilbert and Bernoulli).

```
sudo tc qdisc add dev eth0 root netem loss gemodel p [ r [ 1-h [ 1-k ]]] }  [ ecn ]
```

Bernoulli

```
tc qdisc add dev eth0 root netem loss gemodel p
p     ... loss probability
```

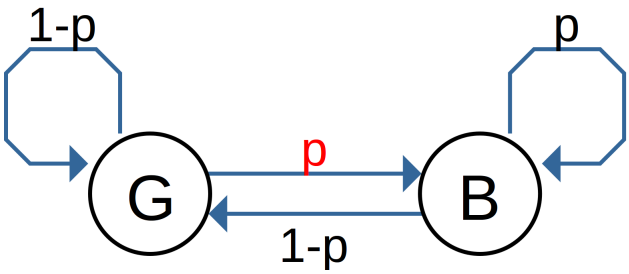


Figure 2: Bernoulli Loss-model (independent loss events)[16]

Simple Gilbert

```
tc qdisc add dev eth0 root netem loss gemodel p r
p     ... loss probability
(1-r) ... burst duration
```

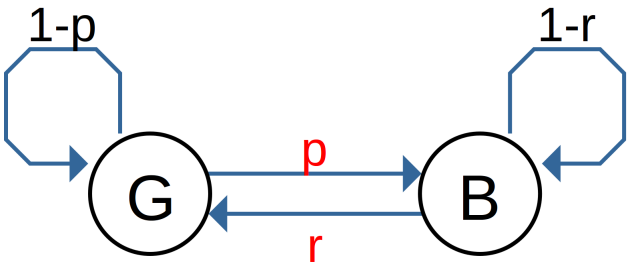


Figure 3: Simple Gilbert (consecutive loss events)[16]

Gilbert

```
tc qdisc add dev eth0 root netem loss gemodel p r 1-h
p     ... loss probability
(1-r) ... burst duration
(1-h) ... loss probability in the bad state
```

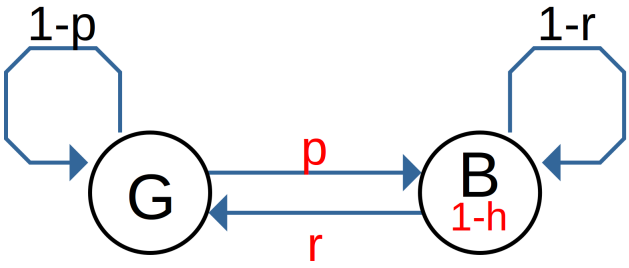


Figure 4: Gilbert Loss-model (correlated loss events)[16]

Gilbert-Elliot

```
tc qdisc add dev eth0 root netem loss gemodel p r 1-h 1-k
p     ... loss probability
(1-r) ... burst duration
(1-h) ... loss probability in the bad state
(1-k) ... loss probability in the good state
```

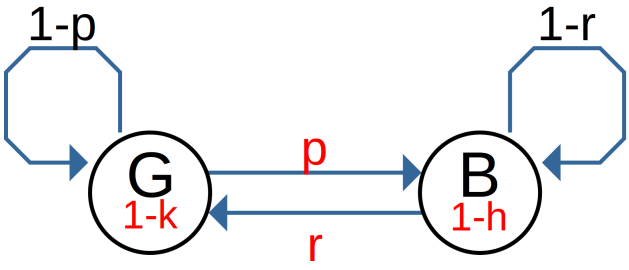


Figure 5: Gilbert-Elliot Loss-model (correlated and isolated loss events)[16]

1.1 netem - loss at egress

Packet loss is one of the standard traffic shaping methods for outgoing traffic offered by netem.

According to [2], this method has following drawback: *When loss is used locally (not on a bridge or router), the loss is reported to the upper level protocols. This may cause TCP to resend and behave as if there was no loss. When testing protocol reponse to loss it is best to use a netem on a bridge or router*
(Currently, UDP is used as the overlay-protocol for the NDN-Testbed)

1.1.1 Bsp [7]

```
sudo tc qdisc add dev eth0 root netem loss 3%
sudo tc qdisc show dev eth0
```

1.1.2 network.sh - Beispiel

Packet loss realized using netem: Following is a example for the easiest kind of packet loss (loss random). Other loss models require / allow more parameters to be specified (state [Markov], gemodel [Gilbert-Elliot burst error patterns])

```
#!/bin/sh
sudo iptables --flush
sudo iptables -P INPUT DROP
sudo iptables -P FORWARD DROP
sudo iptables -P OUTPUT DROP
sudo tc qdisc del dev eth0 root
sudo tc qdisc add dev eth0 root handle 1: htb default 10
sudo tc class add dev eth0 parent 1: classid 1:10 htb rate 100mbit
sudo iptables -A INPUT -d 192.168.0.33 -s 192.168.0.100 -j ACCEPT
```

```
sudo iptables -A FORWARD -d 192.168.0.33 -s 192.168.0.100 -j ACCEPT
sudo iptables -A FORWARD -d 192.168.0.100 -s 192.168.0.33 -j ACCEPT
sudo iptables -A OUTPUT -s 192.168.0.33 -d 192.168.0.100 -j ACCEPT
sudo tc class add dev eth0 parent 1: classid 1:12 htb rate 100mbit
sudo tc class add dev eth0 parent 1: classid 1:15 htb rate 100mbit
sudo tc filter add dev eth0 protocol ip parent 1:0 prio 1
        u32 match ip dst 192.168.0.33 match ip src 192.168.0.100 flowid 1:12
sudo tc filter add dev eth0 protocol ip parent 1:0 prio 1
        u32 match ip dst 192.168.0.100 match ip src 192.168.0.33 flowid 1:15
sudo tc qdisc add dev eth0 parent 1:12 handle 12: tbf rate 2000kbit burst 2500 latency 100ms
sudo tc qdisc add dev eth0 parent 1:15 handle 15: tbf rate 2500kbit burst 3125 latency 100ms
sudo tc qdisc add dev eth0 parent 15:2 handle 18 netem delay 4ms loss random 10%
sudo nfd-stop
sleep 5
sudo nfd-start
sleep 5
sudo nfdc register /best-route/100 udp://192.168.0.100 -c 1
sudo nfdc set-strategy /best-route/ /localhost/nfd/strategy/best-route
```

1.1.3 netem - example for filtering based on the ethernet-destination-address [3]

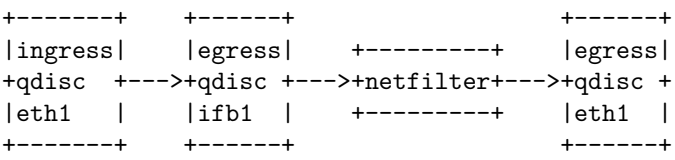
```
...
sudo tc filter add dev eth0 parent 1:0 protocol ip
        u32 match ether dst 02:01:09:02:04:80 flowid 1:12
sudo tc filter add dev eth0 parent 1:0 protocol ip
        u32 match ether dst 3c:97:0e:2f:0e:e9 flowid 1:15
...
```

1.2 netem - loss at ingress

Out of the box, linux does not support traffic-shaping for incoming traffic, just policing.

Usage of an indirection allows the execution of traffic-shaping for incoming traffic:
The plan is to redirect the incoming traffic over a virtual device (IFB [5], = kernel-module ifb), where the full traffic-shaping capabilities can be applied to the egress traffic of this virtual device. [2]

Position of the virtual device (IFB) in the linux-packetflow [6]



1.2.1 Bsp [2]

```
# modprobe ifb
# ip link set dev ifb0 up
# tc qdisc add dev eth0 ingress
# tc filter add dev eth0 parent ffff: \
   protocol ip u32 match u32 0 0 flowid 1:1 action mirred egress redirect dev ifb0
# tc qdisc add dev ifb0 root netem delay 750ms
```

1.2.2 network.sh - example

```
#!/bin/sh
sudo iptables --flush
sudo iptables -P INPUT DROP
sudo iptables -P FORWARD DROP
sudo iptables -P OUTPUT DROP
sudo tc qdisc del dev eth0 root
```

```
sudo tc qdisc add dev eth0 root handle 1: htb default 10
sudo tc class add dev eth0 parent 1: classid 1:10 htb rate 100mbit
sudo iptables -A INPUT -d 192.168.0.33 -s 192.168.0.100 -j ACCEPT
sudo iptables -A FORWARD -d 192.168.0.33 -s 192.168.0.100 -j ACCEPT
sudo iptables -A FORWARD -d 192.168.0.100 -s 192.168.0.33 -j ACCEPT
sudo iptables -A OUTPUT -s 192.168.0.33 -d 192.168.0.100 -j ACCEPT
sudo tc class add dev eth0 parent 1: classid 1:12 htb rate 100mbit
sudo tc class add dev eth0 parent 1: classid 1:15 htb rate 100mbit
sudo tc filter add dev eth0 protocol ip parent 1:0 prio 1
        u32 match ip dst 192.168.0.33 match ip src 192.168.0.100 flowid 1:12
sudo tc filter add dev eth0 protocol ip parent 1:0 prio 1
        u32 match ip dst 192.168.0.100 match ip src 192.168.0.33 flowid 1:15
sudo tc qdisc add dev eth0 parent 1:12 handle 12: tbf rate 2000kbit burst 2500 latency 100ms
sudo tc qdisc add dev eth0 parent 1:15 handle 15: tbf rate 2500kbit burst 3125 latency 100ms
sudo tc qdisc add dev eth0 parent 15:2 handle 18 netem delay 4ms
# IFB+netem at ingress to impl. loss
sudo modprobe ifb numifbs=1
sudo ip link set dev ifb0 up
sudo tc qdisc del dev eth0 ingress
sudo tc qdisc del dev ifb0 root
sudo tc qdisc add dev eth0 ingress
sudo tc filter add dev eth0 parent ffff: protocol ip u32 match u32 0 0
        flowid 1:1 action mirred egress redirect dev ifb0
sudo tc qdisc add dev ifb0 root handle 2: htb default 20
sudo tc class add dev ifb0 parent 2: classid 2:20 htb rate 100mbit
sudo tc class add dev ifb0 parent 2: classid 2:21 htb rate 100mbit
sudo tc filter add dev ifb0 protocol ip parent 2:0 prio 1
        u32 match ip dst 192.168.0.33 match ip src 192.168.0.100 flowid 2:21
sudo tc qdisc add dev ifb0 parent 2:21 handle 21 netem loss random 10%
sudo nfd-stop
sleep 5
sudo nfd-start
sleep 5
sudo nfdc register /best-route/100 udp://192.168.0.100 -c 1
sudo nfdc set-strategy /best-route/ /localhost/nfd/strategy/best-route
```

2 iptables

In order to simulate packet loss on the link, the chains INPUT and FORWARD respectively OUTPUT and FORWARD must be subjected to packet loss.

iptables - chain traversal order: [8]

- Incoming packets destined for the local system: **PREROUTING -> INPUT**
- Incoming packets destined for another host: **PREROUTING -> FORWARD -> POSTROUTING**
- Locally generated packets: **OUTPUT -> POSTROUTING**

INPUT, OUTPUT, FORWARD ... sets of packets traversing the chains INPUT, OUTPUT, FORWARD
TOTAL_INCOMING = INPUT $\cup$ FORWARD
TOTAL_OUTGOING = OUTPUT $\cup$ FORWARD
INPUT $\cap$ FORWARD = $\emptyset$ = OUTPUT $\cap$ FORWARD

```
man iptables
man iptables-extensions
```

2.1 iptables - loss at egress

Here, the TOTAL_OUTGOING - traffic is subjected to packet loss.

This variant harbours an annoyance, namely the application of the DROP-target in the OUTPUT-chain causes

the immediate notification of the associated application (EPERM - permission denied). For more information regarding this, see [9], [10], [11], [12].

2.1.1 Example, inspired by [7]

```
# add rules to output / forward chains
sudo iptables -A OUTPUT -m statistic --mode random --probability 0.03 -j DROP
sudo iptables -A FORWARD -m statistic --mode random --probability 0.03 -j DROP
sudo iptables -L
```

2.2 iptables - loss at ingress

Here, the TOTAL_INCOMING traffic is subjected to packet loss.

2.2.1 Example, inspired by [7]

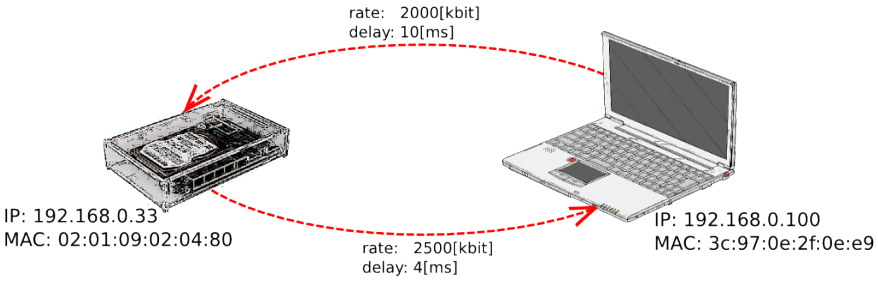
```
iptables -A INPUT -s <SOURCE_IP_ADDRESS> -m statistic --mode random --probability 0.1 -j DROP

# add rules to input / forward chains
sudo iptables -A INPUT -m statistic --mode random --probability 0.03 -j DROP
sudo iptables -A FORWARD -m statistic --mode random --probability 0.03 -j DROP
sudo iptables -L
```

2.2.2 network.sh - Beispiel

```
#!/bin/sh
sudo iptables --flush
sudo iptables -P INPUT DROP
sudo iptables -P FORWARD DROP
sudo iptables -P OUTPUT DROP
sudo tc qdisc del dev eth0 root
sudo tc qdisc add dev eth0 root handle 1: htb default 10
sudo tc class add dev eth0 parent 1: classid 1:10 htb rate 100mbit
#sudo iptables -A INPUT -d 192.168.0.33 -s 192.168.0.100 -j ACCEPT
#sudo iptables -A FORWARD -d 192.168.0.33 -s 192.168.0.100 -j ACCEPT
sudo iptables -A FORWARD -d 192.168.0.100 -s 192.168.0.33 -j ACCEPT
sudo iptables -A OUTPUT -s 192.168.0.33 -d 192.168.0.100 -j ACCEPT
sudo tc class add dev eth0 parent 1: classid 1:12 htb rate 100mbit
sudo tc class add dev eth0 parent 1: classid 1:15 htb rate 100mbit
sudo tc filter add dev eth0 protocol ip parent 1:0 prio 1
        u32 match ip dst 192.168.0.33 match ip src 192.168.0.100 flowid 1:12
sudo tc filter add dev eth0 protocol ip parent 1:0 prio 1
        u32 match ip dst 192.168.0.100 match ip src 192.168.0.33 flowid 1:15
sudo tc qdisc add dev eth0 parent 1:12 handle 12: tbf rate 2000kbit burst 2500 latency 100ms
sudo tc qdisc add dev eth0 parent 1:15 handle 15: tbf rate 2500kbit burst 3125 latency 100ms
sudo tc qdisc add dev eth0 parent 15:2 handle 18 netem delay 4ms
# iptables rules for loss at ingress (ACCEPT with p = (1 - probability of loss))
sudo iptables -A INPUT -d 192.168.0.33 -s 192.168.0.100
        -m statistic --mode random --probability 0.9 -j ACCEPT
sudo iptables -A FORWARD -d 192.168.0.33 -s 192.168.0.100
        -m statistic --mode random --probability 0.9 -j ACCEPT
sudo nfd-stop
sleep 5
sudo nfd-start
sleep 5
sudo nfdc register /best-route/100 udp://192.168.0.100 -c 1
sudo nfdc set-strategy /best-route/ /localhost/nfd/strategy/best-route
```

3 Comparison/Measurements

3.1 Measurement-setup




```
# BPI-R1 acting as consumer, Laptop acting as producer.
sudo consumer -p /best-route/100/4 -r 100 -t 10
producer -p /best-route/100 -s 2048
```

At the moment, only two directly connected nodes are examined (loss on the up/down - connection between them). The chosen forwarding-strategy (best-route) does not react to loss/perceived congestion, it keeps forwarding at the same rate over the same links. Longer paths between two nodes in the network are expected to generally result in higher loss (e.g. IP-fragmentation).

```
# checking using iperf (UDP, 1470 byte datagrams)
iperf -s -u

iperf -c 192.168.0.100 -u -t 60 -b 2500000   (von *.33 aus)
bzw.
iperf -c 192.168.0.33 -u -t 60 -b 2000000  (von *.100 aus)

# another way via classic ping
sudo ping -i 0.1 -c 10000 192.*.*.*
```

3.2 Measurements

3.2.1 netem:(10% loss on egress, ip-matching Filter, 2048k data-size)

	Interest-Satisfaction-Ratio								
Loss at the consumer-host	0.9	0.912	0.906	0.9	0.9	0.903	0.898	0.897	0.905
Loss at the producer-host	0.79	0.795	0.815	0.823	0.803	0.804	0.788		

	UDP - throughput[Mbits/sec]	
	***.33 -> *.100**	***.100 -> *.33**
no loss	2.43	1.94
loss at *.33	2.24 (92.2%)	1.94
loss at *.100	2.43	1.80 (92.8%)

Effect of the IP-packet loss on the NDN-Interest-Satisfaction-Ratio (ISR):
A interest-packet of the size of around 85 byte fits into one IP-packet.
A data-packet of the size of e.g. 2048 byte requires the underlying UDP-Packet to be transported using two IP-fragments. The loss of one of these two IP-fragments causes the loss of the UPD-packet and with it the loss of the data-packet. (In case of no retransmissions)

Example: 1000 Interests

- Losing 10% of the IP-packets on the egress of the consumer (*.33), causes only 900 Interests (900 IP-packets) to be transmitted to the producer. Because there is no loss happening on the egress of the producer, the producer is able to successfully send 900 data-packets (1800 IP-fragments) back to the consumer. (Interest-Satisfaction-Ratio = 0.9)
  $ISR = (1 - p)$ where $p \in [0, 1]$ ... probability for packet loss

- Losing 10% of the IP-packets on the egress of the producer (*.100) and having no loss at the egress of the consumer initially causes all 1000 interest-packets (1000 IP-packets) to be transmitted to the producer. After their arrival at the producer, he tries to send 1000 data-packets (2000 IP-fragments) back to the consumer, but 10% of the IP-packets get lost before they are sent to the consumer.

  Best case: Having lost 200 IP-fragments, all two IP-fragments of 100 data-packets were lost, which translates to 100 lost data-packets. (Interest-Satisfaction-Ratio = 0.9)

  Worst case: Having lost 200 IP-fragments, all IP-fragments belong to different data-packets, which translates to 200 lost data-packets (Interest-Satisfaction-Ratio = 0.8)
  $(1 - (\#IPFragmente * p)) \leq ISR \leq (1 - p)$ where $p \in [0, 1]$ ... probability of packet loss

Size of NDN data-packet [Byte]	Number of IP-fragments	Best-Case ISR	Worst-Case ISR
1024	1	0.9	0.9
2048	2	0.9	0.8
3072	3	0.9	0.7
4096	3	0.9	0.7
8192	6	0.9	0.4

In reality, IP-packet loss often happens in bursts (loss of multiple successive IP-packets), a modelling of packet loss using a Markov-chain or the Gilber-Elliot loss model is closer to reality than independent packet loss. A unavoidable downside of those models is their non-trivial parametrization, especially in regards to what exactly shall be modelled.

Bursty IP-packet loss causes more associated IP-fragments (e.g. part of same UDP-packet) to be lost, which results in a NDN-Interest-Satisfaction-Ratio closer to the "Best case".

3.2.2 netem:(10% loss on ingress, ip-matching Filter, 2048k data-size)

	Interest-Satisfaction-Ratio							
Loss at the consumer-host	?	?	?	?	?	?	?	?
Loss at the producer-host	0.907	0.902	0.892	0.906	0.912	0.885	0.91	0.892

	UDP - throughput[Mbits/sec]	
	***.33 -> *.100**	***.100 -> *.33**
no loss	2.43	1.94
loss at *.100	2.18 (89.7%)	1.94

3.2.3 iptables:(10% loss on ingress, ip-matching Filter, 2048k data-size)

	Interest-Satisfaction-Ratio							
Loss at the consumer-host	0.904	0.908	0.903	0.897	0.894	0.918	0.880	0.901
Loss at the producer-host	0.907	0.919	0.894	0.903	0.882	0.903	0.903	0.915

	UDP - throughput[Mbits/sec]	
	***.33 -> *.100**	***.100 -> *.33**
no loss	2.43	1.94
loss at *.33	2.43	1.75 (90.2%)
loss at *.100	2.19 (90.1%)	1.94

3.3 Comparison

Using netem instead of iptables would allow for a more flexible emulation of packet loss (different kinds of loss).

Emulating packet loss on ingress would allow to leave the egress-side unchanged, furthermore, this approach would make the packet loss emulation appear closer to reality for all protocols (especially TCP).

Packet loss on ingress causes the utilization of the physical link to stay the same as there would be no packet loss. (Packets get dropped after arriving)

Packet loss on egress reduces the utilization of the physical link. (Packets get dropped before departure, no retransmissions).

3.3.1 Netem + packet loss on egress

The implementation of packet loss on egress using netem causes the fewest changes to the codebase.

The approach meets the requirements to emulate packet loss in the current use-case (NDN-overlay using UDP, no 100% correct behaviour in combination with TCP necessary).

3.3.2 Netem + packet loss on ingress

Because this approach involves packet loss to happen outside of the sending host, it has the potential to look very "natural" / to come closer to reality regarding the transport-protocols (especially TCP). (TCP problem, mentioned in 1.1)

The implementation of this approach to emulate packet loss causes the most changes to the codebase. Not only the creation of the network.sh - files gets more complicated than in other approaches, the necessary kernel module (ifb) is currently not present in our Pi-images. So the kernel would have to be rebuild and re-distributed.

3.3.3 Iptables + packet loss on egress

There is no silent drop for IP-packets on the egress (OUTPUT-chain), the respective UDP-application gets notified immediately. (2.1) Regardless of what this UDP-application decides to do with that information, this approach does not qualify as a transparent implementation of packet loss.

3.3.4 Iptables + packet loss on ingress

The implementation of packet loss using iptables on ingress does not involve many changes, due to the fact that this kind of operation is very close to the common tasks of iptables.

In the conducted tests/measurements (two directly linked hosts, no forwarding), IP-packets were lost/dropped always after being reconstructed in the target-host. (IP-packets arriving on their target-host get reconstructed before they show up for the INPUT-chain).

3.4 Choice according to current knowledge

Iptables vs. netem: Netem is chosen, because it allows for a more varied emulation of packet-loss and is closer to the network-layer (egress and ingress, [13]).

Netem ingress vs egress: For a emulation of packet loss close to reality (regarding protocols), a "middleman" - node which handles the loss is recommended, without such, the emulation of packet loss on the ingress is the second best choice.

If this universally usable emulation of packet loss is worth the comparatively high effort, packet loss on ingress should be preferred.

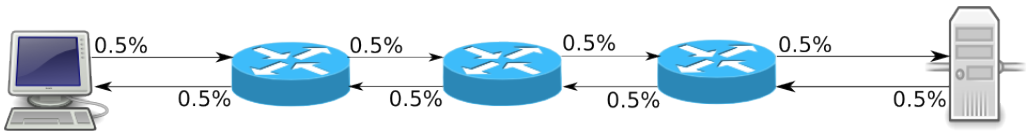
If the correct emulation of packet loss for connection-less transmission-protocols is sufficient, the emulation of packet loss on the egress should be preferred.

Decision (Juli 2016): Implementation of packet loss on egress using netem.
Additionally, future versions of the provided Pi-images should include support for the kernel module ifb.

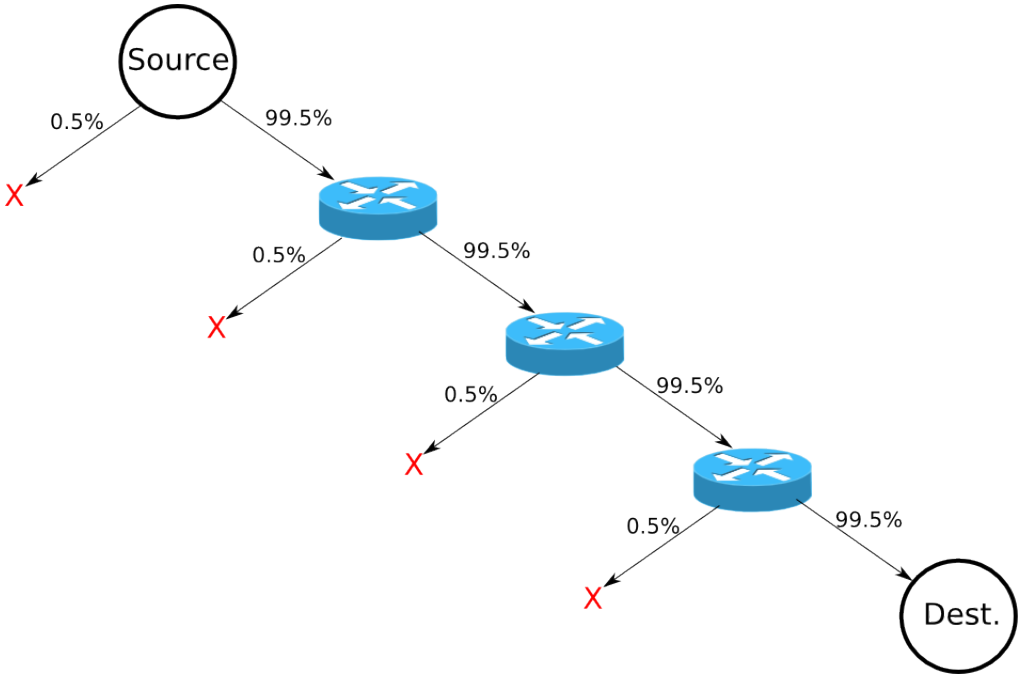
4 Post-implementation-measurement of the impact of loss in networks with multiple nodes in between source and destination

4.1 Setup

All nodes are BananaPi-R1, 0.5% packet loss on egress using netem.

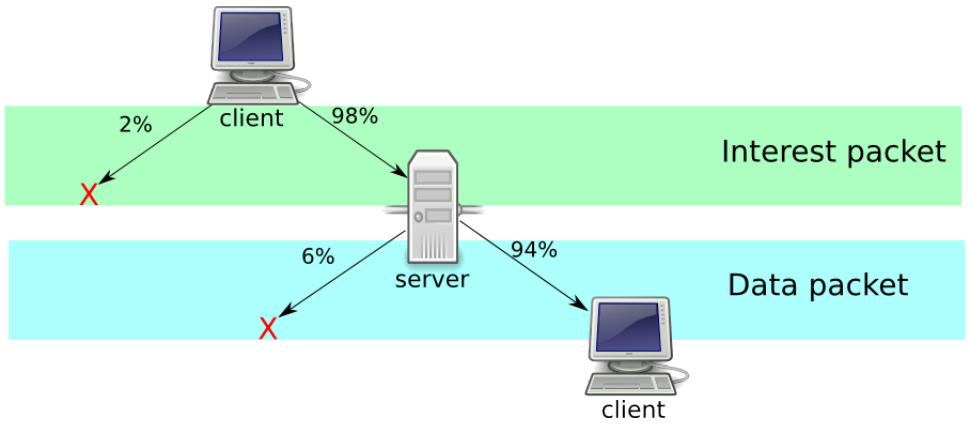


4.2 One way traversal through the network:



$p_{reachDestination} = 0.995^4 = 0.98$
$p_{lost} = 1 - 0.98 = 0.02$

4.3 Two way traversal of the network (Interest –> Producer, Data –> Consumer)



$p_{IPpacketLost} = p_{interestLost} = 2\%$
$p_{interestSuccess} = 1 - p_{interestLost}$

The Data packet consists of three IP-fragments. (data-size = 4096 byte)
If just one IP-fragment is lost, the whole Data packet is lost.
$p_{dataPacketSuccess} = 0.98^3 = 0.941192$
$p_{dataPacketLost} = 1 - p_{dataPacketSuccess} = 0.058808$

$p_{successfulTrip} = p_{interestSuccess} * p_{dataPacketSuccess} = 0.9225 = 92.2\%$
$p_{fail} = 1 - p_{successfulTrip} = 0.0775 = 7.8\%$

Expected Interest-Satisfaction-Ratio: 0.922

4.4 Measured Interest-Satisfaction-Ratio in 4 runs

```
PRODUCER_APP = "producer"
PRODUCER_PARAMETER = "-s 4096 -f 300"

CONSUMER_APP = "consumer"
CONSUMER_PARAMETER = "-r 30 -t 1800 -l 1000"

Retransmissions disabled
```

run	ISR (best-route)	ISR (broadcast)	ISR (saf)
1	0.924519	0.92263	0.690593
2	0.922537	0.923537	0.723833
3	0.924333	0.923426	0.720019
4	0.921519	0.923815	0.711444

The measured loss was roughly equal to the estimation for those Forwarding-Strategies, which do not concern themselves with congestion control.
SAF notices the packet loss (less reliable interface), and sends less traffic over the lossy interface resulting in a lower ISR in this particular network-topology.

5 Open questions

- What is the exact impact of netem egress packet loss on TCP - connections.
- How high is the additional computational load introduced by the usage of the ifb-kernel-module in order to realize packet loss using netem on ingress?

6 Appendix A - suggested changes on the emulation-scripts

Implementation of the simplest form of loss using netem to drop packets on egress.

6.1 rand_network.py

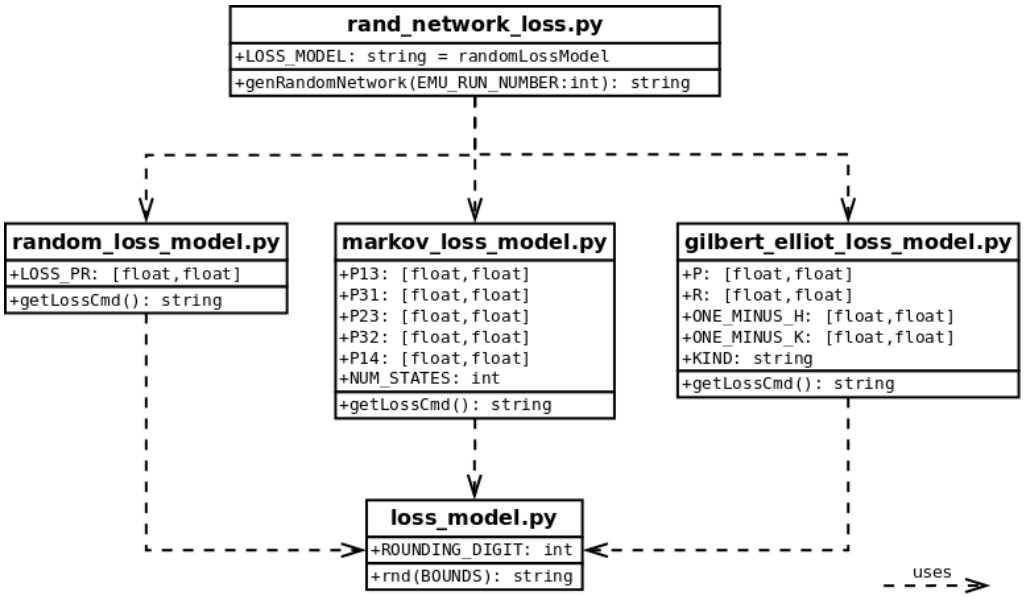
The Support of multiple loss models requires a hightened flexibility by rand_network.py.

The plan is to load and use a predefined packet loss model while generating the network-topology file. The loss model returns the loss-command-string to configure the netem-instances on the nodes. The actual parameters for netem, making up the loss-command are randomly selected out of a predefined range.

By loading/including a loss model into rand_network.py after setting a seed for the random number generator, the seed persists when randomly selecting the actual parameter-values, making them too dependent on the current EMU_RUN_NUMBER.

Concerning the network-topology-files, rand_network.py is backwards compatible, older network-topology-files containing no mention of loss can be used with no problem. In fact, if rand_network.py is configured to use no loss model (LOSS_MODEL=""), the resulting topology-files match the "old" format.

LOSS_MODEL	Effect
""	No packet loss on the links of the network-topology (same format as "old" network-topology files).
"random_loss_model"	Packet loss on the links of the network-topology, realized by a independent packet loss probability.
"markov_loss_model"	Packet loss on the links of the network-topology, realized by a Markov-chain.
"gilbert_elliot_loss_model"	Packet loss on the links of the network-topology, realized by a Gilbert-Elliot loss model.



Info: Sometimes, "/n" was used instead of "\n", in such cases, "\n" is meant.

```
import importlib
...
MAX_BANDWIDTH = 4000
CONTENT_POPULARITY = "uniform"
MIN_DELAY = 5
MAX_DELAY = 20
# "" -> no loss, "random_loss_model", "markov_loss_model", "gilbert_elliot_loss_model"
LOSS_MODEL = "randomLossModel"
...
random.seed(EMU_RUN_NUMBER)
```

```
...
# load selected loss model if specified
if LOSS_MODEL:
        lossModel = importlib.import_module(LOSS_MODEL)
...
f.write("#nodes setting (n1,n2,bandwidth in kbits a -> b, bandwidth in kbits a <- b,
        delay a -> b in ms, delay b -> a in ms"
        + (", loss a -> b, loss a <- b" if LOSS_MODEL else "") + ")/n")
...
for edge in g.es:
        f.write( str(edge.source) + "," + str(edge.target) + ","
                 + str(int(random.uniform(MIN_BANDWIDTH, MAX_BANDWIDTH))) + ","
                 + str(int(random.uniform(MIN_BANDWIDTH, MAX_BANDWIDTH))) + ","
                 + str(int(random.uniform(MIN_DELAY, MAX_DELAY))) + ","
                 + str(int(random.uniform(MIN_DELAY, MAX_DELAY))))

        if not LOSS_MODEL:
                f.write("/n")
        else:
                f.write("," + lossModel.getLossCmd() + ","
                + lossModel.getLossCmd() + "/n")

...
```

6.1.1 loss_model.py

```
import random

ROUNDING_DIGIT = 5

def rnd(BOUNDS):
        return str(round(random.uniform(BOUNDS[0], BOUNDS[1]), ROUNDING_DIGIT))
```

6.1.2 random_loss_model.py

```
from loss_model import *

# random PERCENT
LOSS_PR = [0.000, 1] # [%],  min nonzero value =  0.0000000232%

def getLossCmd():
        return " loss random " + rnd(LOSS_PR)
```

6.1.3 markov_loss_model.py

```
from loss_model import *

#  state p13 [ p31 [ p32 [ p23 [ p14]]]] ... state-transition probabilities
P13 = [0.5, 0.5]
P31 = [99.5, 99.5]
P23 = [0,0]
P32 = [0,0]
P14 = [0,0]
NUM_STATES = 2

def getLossCmd():
        if NUM_STATES == 1:
                PROBABILITIES = [rnd(P13)]
        elif NUM_STATES == 2:
                PROBABILITIES = [rnd(P13), rnd(P31)]
        elif NUM_STATES == 3:
                PROBABILITIES = [rnd(P13), rnd(P31),rnd(P23), rnd(P32)]
```

```
        elif NUM_STATES == 4:
                PROBABILITIES = [rnd(P13), rnd(P31),rnd(P23), rnd(P32), rnd(P14)]
        else:
                raise ValueError("The number of states must be 1,2,3 or 4")

        return " loss state " + ' '.join(str(p) for p in PROBABILITIES)
```

6.1.4 gilbert_elliot_loss_model.py

```
from loss_model import *

# Gilbert-Elliot loss model
# p     ... transition probability to bad state (loss probability)
# r     ... transition probability to good state ((1-r) ... burst duration)
# (1-h) ... loss probability in the bad state
# (1-k) ... loss probability in the good state

# gemodel p [ r [ 1-h [ 1-k ]]]
P = [0.28, 0.297]
R = [0.9, 0.967]
ONE_MINUS_H = [0, 0]
ONE_MINUS_K = [0.05, 0.05986]
KIND = "GilbertElliot"

def getLossCmd():
        if KIND == "Bernoulli":
                PROBABILITIES = [rnd(P)]
        elif KIND == "SimpleGilbert":
                PROBABILITIES = [rnd(P), rnd(R)]
        elif KIND == "Gilbert":
                PROBABILITIES = [rnd(P), rnd(R), rnd(ONE_MINUS_H)]
        elif KIND == "GilbertElliot":
                PROBABILITIES = [rnd(P), rnd(R), rnd(ONE_MINUS_H), rnd(ONE_MINUS_K)]
        else:
                raise ValueError("'" + kind + "' : This kind is not supported by this loss model." +
                        "[Bernoulli,SimpleGilbert,Gilbert,GilbertElliot]")

        return " loss gemodel " + ' '.join(str(p) for p in PROBABILITIES)
```

6.1.5 generated_network_top.txt with loss defined by random_loss_model.py

```
#number of nodes
20
#nodes setting (n1,n2,bandwidth in kbits a -> b, bandwidth in kbits a <- b,
delay a -> b in ms, delay b -> a in ms, loss a -> b, loss a <- b)
1,4,3791,3388,13,17,loss random 0.798,loss random 0.657
2,4,3000,3181,12,8,loss random 0.066,loss random 0.86
3,7,3942,3302,11,17,loss random 0.062,loss random 0.641
4,7,3127,3287,17,5,loss random 0.036,loss random 0.418
...
#properties (Client, Server)
1,3
4,5
6,8
12,8
...
#eof //do not delete this
```

6.2 node_parser.py

```
class Link(object):
        def __init__(self,n1, n2, bw_n1_to_n2, bw_n2_to_n1, delay_n1_to_n2, delay_n2_to_n1,
         loss_n1_to_n2, loss_n2_to_n1):
                self.n1=n1
                self.n2=n2
                self.ip1=""
                self.ip2=""
                self.bw_n1_to_n2 = bw_n1_to_n2
                self.bw_n2_to_n1 = bw_n2_to_n1
                self.delay_n1_to_n2 = delay_n1_to_n2
                self.delay_n2_to_n1 = delay_n2_to_n1
                self.loss_n1_to_n2 = loss_n1_to_n2
                self.loss_n2_to_n1 = loss_n2_to_n1

        def __str__(self):
                return "Link(" + self.n1 + ", " + self.n2 + ", " +str(self.bw_n1_to_n2) + ", " +
                 str(self.bw_n2_to_n1)+ ", " + str(self.delay_n1_to_n2) + ", "
                 + str(self.delay_n2_to_n1) + ", " + str(self.loss_n1_to_n2)
                 + ", " + str(self.loss_n2_to_n1) + ")"
        ...
        if line.startswith("#nodes setting"):
                                line = next(f)
                                while(line[0] != '#'):
                                        tmp_str = line.rstrip('/n').split(',')

                                        if (len(tmp_str) == 6):
                                                link_list.append(Link(tmp_str[0],tmp_str[1],
                                                int(tmp_str[2]), int(tmp_str[3]), int(tmp_str[4]),
                                                 int(tmp_str[5]), "", ""))
                                        elif (len(tmp_str) == 8):
                                                 link_list.append(Link(tmp_str[0],tmp_str[1],
                                                 int(tmp_str[2]), int(tmp_str[3]), int(tmp_str[4]),
                                                 int(tmp_str[5]), tmp_str[6], tmp_str[7]))
                                        else:
                                                print "Parsing Error"
                                                exit(-1)

                line = next(f)
        ...
```

6.3 deploy_network.py

```
...
#delay and loss for n1 to n2
netman_handle = str(handle_offset+(nodes+1)*2+int(n2))
commands[ip1.replace(EMU_PREFIX, MNG_PREFIX)].append("sudo tc qdisc add dev eth0 parent "
        + str(handle_offset+nodes+1+int(n2)) + ":" + str(int(n2)+1) + " handle "
        + netman_handle + " netem delay " + str(link.delay_n1_to_n2) + "ms" +
         str(link.loss_n1_to_n2))
...
#delay and loss for n2 to n1
netman_handle = str(handle_offset+(nodes+1)*2+int(n1))
commands[ip2.replace(EMU_PREFIX, MNG_PREFIX)].append("sudo tc qdisc add dev eth0 parent "
        + str(handle_offset+nodes+1+int(n1)) + ":" + str(int(n1)+1) + " handle "
        + netman_handle + " netem delay " + str(link.delay_n2_to_n1) + "ms" +
         str(link.loss_n2_to_n1))
...
```

6.3.1 Alternative way to disable packet loss

Apart from omitting the command for netem as a whole, setting the loss probability to zero causes netem to setup accordingly.

```
################################
# loss 0% -> no loss-qdisc added
################################
sudo tc qdisc add dev eth0 root netem loss 5%

sudo tc qdisc show
qdisc netem 8002: dev eth0 root refcnt 2 limit 1000 loss 5%

sudo tc qdisc del dev eth0 root

sudo tc qdisc add dev eth0 root netem loss 0%

sudo tc qdisc show
qdisc netem 8003: dev eth0 root refcnt 2 limit 1000
```

7 Appendix B - (Segmentation-Offloading-) features of the BPR1

```
sudo ethtool -k eth0.101

Features for eth0.101:   # eth0.102 has the same features, [fixed] settings cannot be changed
rx-checksumming: on [fixed]
tx-checksumming: off
tx-checksum-ipv4: off
tx-checksum-ip-generic: off
tx-checksum-ipv6: off
tx-checksum-fcoe-crc: off
tx-checksum-sctp: off
scatter-gather: off
tx-scatter-gather: off
tx-scatter-gather-fraglist: off
tcp-segmentation-offload: off
tx-tcp-segmentation: off
tx-tcp-ecn-segmentation: off
tx-tcp6-segmentation: off
udp-fragmentation-offload: off [fixed]
generic-segmentation-offload: off [requested on]
generic-receive-offload: on
large-receive-offload: off [fixed]
rx-vlan-offload: off [fixed]
tx-vlan-offload: off [fixed]
ntuple-filters: off [fixed]
receive-hashing: off [fixed]
highdma: on
rx-vlan-filter: off [fixed]
vlan-challenged: off [fixed]
tx-lockless: on [fixed]
netns-local: off [fixed]
tx-gso-robust: off [fixed]
tx-fcoe-segmentation: off
fcoe-mtu: off
tx-nocache-copy: off
loopback: off [fixed]
rx-fcs: off [fixed]
rx-all: off [fixed]
```

8 Appendix C - suggestions for improvement

- **Use a homogenous testing/measurement-environment**
  Two or more Pis, instead of a Pi and a Laptop, proper gateway-configuration (separation of MGM and SIM network, ...)
- Way more measurement-runs, statistically evaluated.

9 Appendix D - miscellaneous links

Linux Advanced Routing & Traffic Control HOWTO (Bert Hubert, 2013)
`http://www.ime.usp.br/~rbrito/docs/lartc.dbk.psom.pdf`
alternativ:
`http://www.lartc.org/howto/`
speziell:
`http://lartc.org/howto/lartc.qdisc.html`

Queueing in the Linux Network Stack
`http://www.linuxjournal.com/content/queueing-linux-network-stack?page=0,0`

generic segmentation offload (TSO for TCP and UFO for UDP also used)
`http://www.linuxfoundation.org/collaborate/workgroups/networking/gso`

Tc: ingress policing and ifb mirroring
`http://serverfault.com/questions/350023/tc-ingress-policing-and-ifb-mirroring`

netem - kernel module
`http://lxr.free-electrons.com/source/net/sched/sch_netem.c`

ifb - kernel module
`http://lxr.free-electrons.com/source/drivers/net/ifb.c`

traffic-control howto (inofficial?)
`http://linux-ip.net/articles/Traffic-Control-HOWTO/index.html`

Linux Kernel Networking: Implementation and Theory, Rami Rosen, apress, 2013 `http://www.apress.com/9781430261964`

References

[1] Network Emulation with NetEm - Stephen Hemminger
http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.67.1687&rep=rep1&type=pdf

[2] Netem-Basics - Linuxfoundation
http://www.linuxfoundation.org/collaborate/workgroups/networking/netem

[3] TC-manpage for u32-filter
http://man7.org/linux/man-pages/man8/tc-u32.8.html

[4] Intro to queueing in the Linux network-stack - Linuxjournal
https://www.linuxjournal.com/content/queueing-linux-network-stack?page=0,0

[5] IFB-Basics - Linuxfoundation
http://www.linuxfoundation.org/collaborate/workgroups/networking/ifb

[6] IFB-Position in the packetflow
http://unix.stackexchange.com/questions/288959/how-is-the-ifb-device-positioned-in-the-packet-flow-of-the-linux-kernel

[7] Examples for the realization of packet loss using netem and iptables
https://sandilands.info/sgordon/dropping-packets-in-ubuntu-linux-using-tc-and-iptables

[8] Netfilter - Basics
https://www.digitalocean.com/community/tutorials/a-deep-dive-into-iptables-and-netfilter-architecture

[9] Example for EPERM-error-notification if dropping packets in the OUTPUT-chain.
http://www.spinics.net/lists/netfilter/msg53963.html

[10] IPtables: Difference between DROP and STEAL
http://unix.stackexchange.com/questions/108847/what-is-the-difference-between-j-drop-and-j-steal

[11] IPtables: Mailinglist-entry regarding DROP in the OUTPUT-chain
http://www.spinics.net/lists/netfilter/msg42589.html

[12] IPtables: Clarification of DROP in the OUTPUT-chain
http://unix.stackexchange.com/questions/228647/how-can-i-silently-drop-some-outgoing-packets

[13] Linux netfilter packet flow
https://de.wikipedia.org/wiki/Datei:Netfilter-packet-flow.svg

[14] Basic information about Generic-Segmentation-Offload
https://wiki.linuxfoundation.org/networking/gso

[15] http://netgroup.uniroma2.it/twiki/bin/view.cgi/Main/NetemCLG

[16] http://netgroup.uniroma2.it/TR/TR-loss-netem.pdf