➜ nog.fi git:(main)

Generating 1 Tbps of traffic on a commodity hardware using T-Rex

➜ nog.fi git:(intro)

Vlad and how got into this

[>] 15+ years of experience as SysAdmin/SRE/DevOps

[>] side project close to hardware

[>] little to no experience with networking

[>] like to tinker with exotic hardware as a hobby

[>] Switzerland / 25 Gbps internet at home / gear was the bottleneck

[>] talked to Pim Van Pelt about Network performance testing

[>] 1 Tbps router on CPU required load generator

```
➜ nog.fi git:(intro)
```

TLDR;

«I was so preoccupied with whether I could,
I didn't stop to think if I should»

➜ nog.fi git:(part 1: Hardware)

What you need to generate 1 Tbps?

[>] PCIe bandwidth:

[-- >] PCIe Gen3: ~120 Gbps per x16 slot -> 1x100G NIC

[-- >] PCIe Gen4: ~250 Gbps per x16 slot -> 2x100G NIC

[-- >] PCIe Gen5: ~500 Gbps per x16 slot -> 2x200G NIC

➜ nog.fi git:(part 1: Hardware)

What you need to generate 1 Tbps?

[>] PCIe bandwidth

[>] PCIe lanes CPU/MB:

[-- >] Desktop platforms:

[------>] only 28 PCIe Gen5 lanes, less than 600Gbps of total bandwidth (in theory)

[-- >] Server platforms (per CPU):

[------>] 80 PCIe Gen5 for Xeon Sapphire Rapids / Emerald Rapids

[------>] 112 PCIe Gen5 for Xeon-W Sapphire Rapids

[------>] 96 PCIe Gen5 for Epyc 8004

[------>] 128 PCIe Gen5 for Epyc 9004/9005

➜ nog.fi git:(part 1: Hardware)

What you need to generate 1 Tbps?

[>] PCIe bandwidth

[>] PCIe lanes CPU/MB

[>] money:

[-- >] No sponsorship

[-- >] Hardware must be as cheap as possible

➜ nog.fi git:(part 1: Hardware)

What you need to generate 1 Tbps?

[>] PCIe bandwidth

[>] PCIe lanes CPU/MB

[>] money

[>] usable for VPP for next part of the project:

[-->] Intel supports DDIO since Xeon-E5v2

[-->] AMD supports SDCI since Zen 5 (not available yet as of the start of the project)

➜ nog.fi git:(part 1: Hardware)

What you need to generate 1 Tbps?

[>] PCIe bandwidth

[>] PCIe lanes CPU/MB

[>] money

[>] usable for VPP for next part of the project

[>] use small packets, ideally 64b

➜ nog.fi git:(part 1: Hardware)

Idea: Xeon-W Sapphire Rapids

[>] Start low-end:

[-- >] Xeon W5-3435X, 16 cores, 32 threads

[>] Because of the platform:

[-- >] 6 x16 PCIe Gen5 slots, 1x8 PCIe Gen5

With Gen4 NICs that is 1.3 Tbps of theoretical performance

➜ nog.fi git:(part 1: Hardware)

First problems: not all NICs are equal

[>] Claimed performance:

[-->] Mellanox ConnectX 148–268 Mpps*

[-->] Intel NICs 112–228 Mpps*

[-->] Broadcom NICs 106 Mpps**

[>] Availability & price on a used market varies

---

* – depending on generation and exact model
** – information about latest generation of HW was not available

➜ nog.fi git:(part 1: Hardware)

NICs. Conclusion

[>] Start with mix of ConnectX-5, 6 and 7s

[>] Start simple, with loopback tests per NIC

[>] Investigate performance of cheap Bluefield-2 (MBF345A-VENOT)

[>] Investigate how HyperThreading affects performance

Use T-Rex — what is it?

- [>] Realistic traffic generator
- [>] Uses DPDK under the hood
- [>] Fast — claims «up to 200 Gb/sec with one server»
- [>] Uses ScaPy to generate payload

➜ nog.fi git:(part 2: Tests & Software)

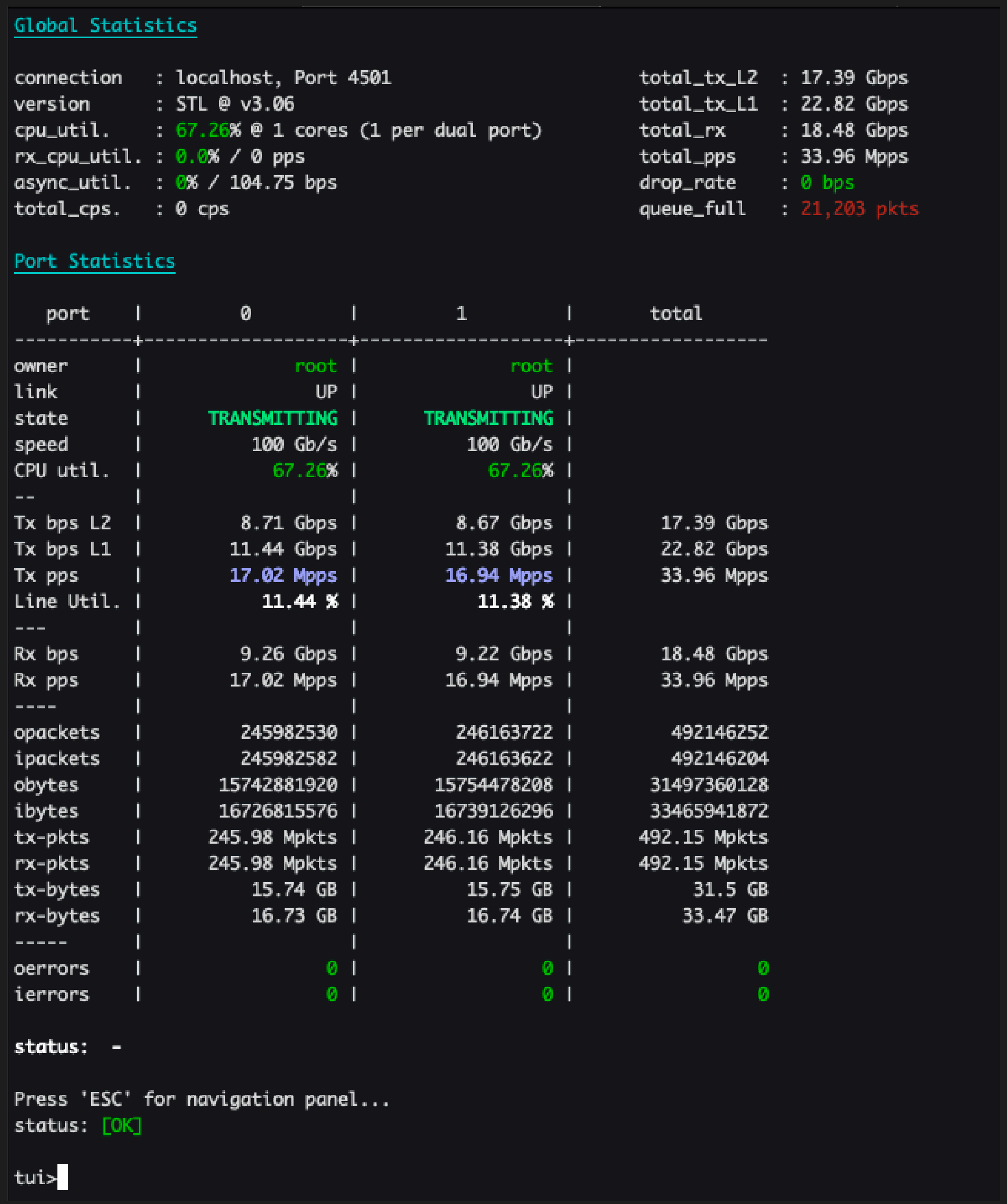
Problem 1: collecting data

[>] T-rex's TUI is nice, but doesn't scale and requires QoL patches

[>] Stateless GUI is heavy

[>] trex-loadtest-viz — nice, but not real-time

Solution: write my own simplistic prometheus exporter



Writing prometheus exporter

T-Rex have some documentation, but not all that's required:

[>] Documentation is not complete (no information what is returned)

[>] get_stats API requires write access, while TUI works in read-only mode

[>] TUI is large and do a lot of stuff

Solution: read carefully what TUI do and experiment. Iterate quickly and early.

→ nog.fi git:(part 2: Tests & Software)

Visualizing in Grafana

[>] Took few hours to write a PoC

[>] TUI uses non-documented API or direct field access to do some of the work

[>] Have rough edges — requires restart to reconnect to T-Rex

Code of exporter is on GitHub

→ nog.fi git:(part 2: Tests & Software)

Problem 2

Example bench.py generator is not fast — no caching

Solution: implement your own

```
Port Statistics

   port    |         0         |         1         |       total
-----------+-------------------+-------------------+------------------
owner      |              root |              root |
link       |                UP |                UP |
state      |      TRANSMITTING |      TRANSMITTING |
speed      |          100 Gb/s |          100 Gb/s |
CPU util.  |            99.12% |            99.12% |
--         |                   |                   |
Tx bps L2  |         3.19 Gbps |          3.2 Gbps |         6.39 Gbps
Tx bps L1  |         4.19 Gbps |          4.2 Gbps |         8.38 Gbps
Tx pps     |         6.23 Mpps |         6.24 Mpps |        12.48 Mpps
Line Util. |            4.19 % |             4.2 % |
---        |                   |                   |
Rx bps     |         3.19 Gbps |          3.2 Gbps |         6.39 Gbps
Rx pps     |         6.23 Mpps |         6.24 Mpps |        12.48 Mpps
```

Heavily based on Pim's talk @ FOSDEM 2024 and Talk about Scapy @ FOSDEM 2024

Problem 2

T-Rex allows to write your own generator. It's simpler than it looks [examples 1, 2].

All the logic — 18 lines of code

Features:

[>] Cache the packets

[>] Uses just UDP

[>] Tries to randomise IP and Port

ConnectX-5 & 6

Specs of those cards are the same: 2x100G, PCIe Gen4.

They should perform the same, right? — Nope!

[>] NIC vendors doesn't explain well the difference between NICs

[>] With ConnectX you always can send more than receive

ConnectX-5 & 6

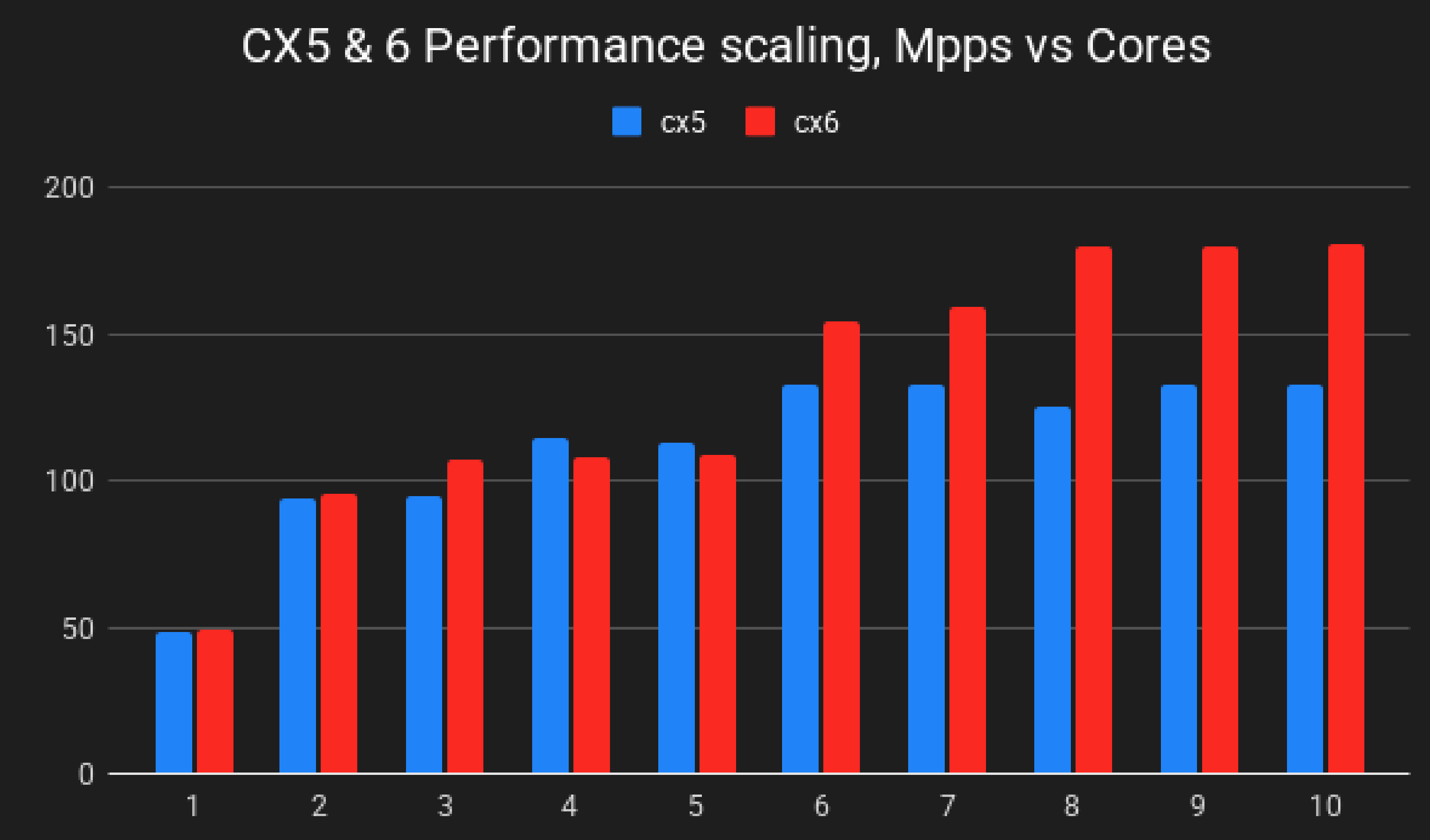
[>] Per port performance:

[-- >] CX5 capped at ~132 Mpps

[-- >] CX6 capped at ~180 Mpps

[>] Until approx. 5 cores per NIC, there is no significant difference

[>] Prioritize CX6 over CX5

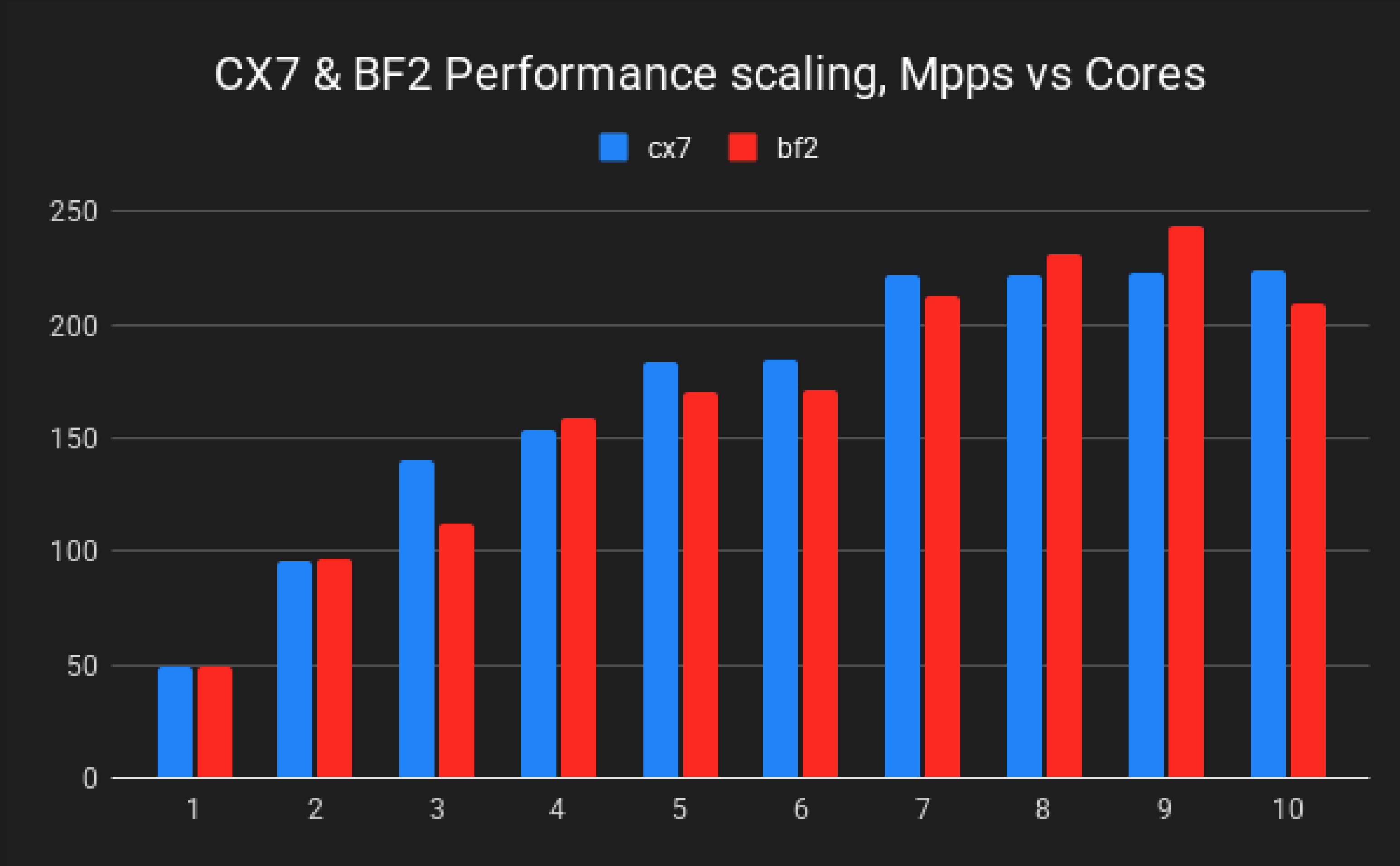


➜ nog.fi git:(part 2: Tests & Software)

ConnectX-7 & BF2

Note: BF2 is 1x200

- [>] ConnectX-7 per-slot is better
- [>] Price-wise BF2 is unbeatable



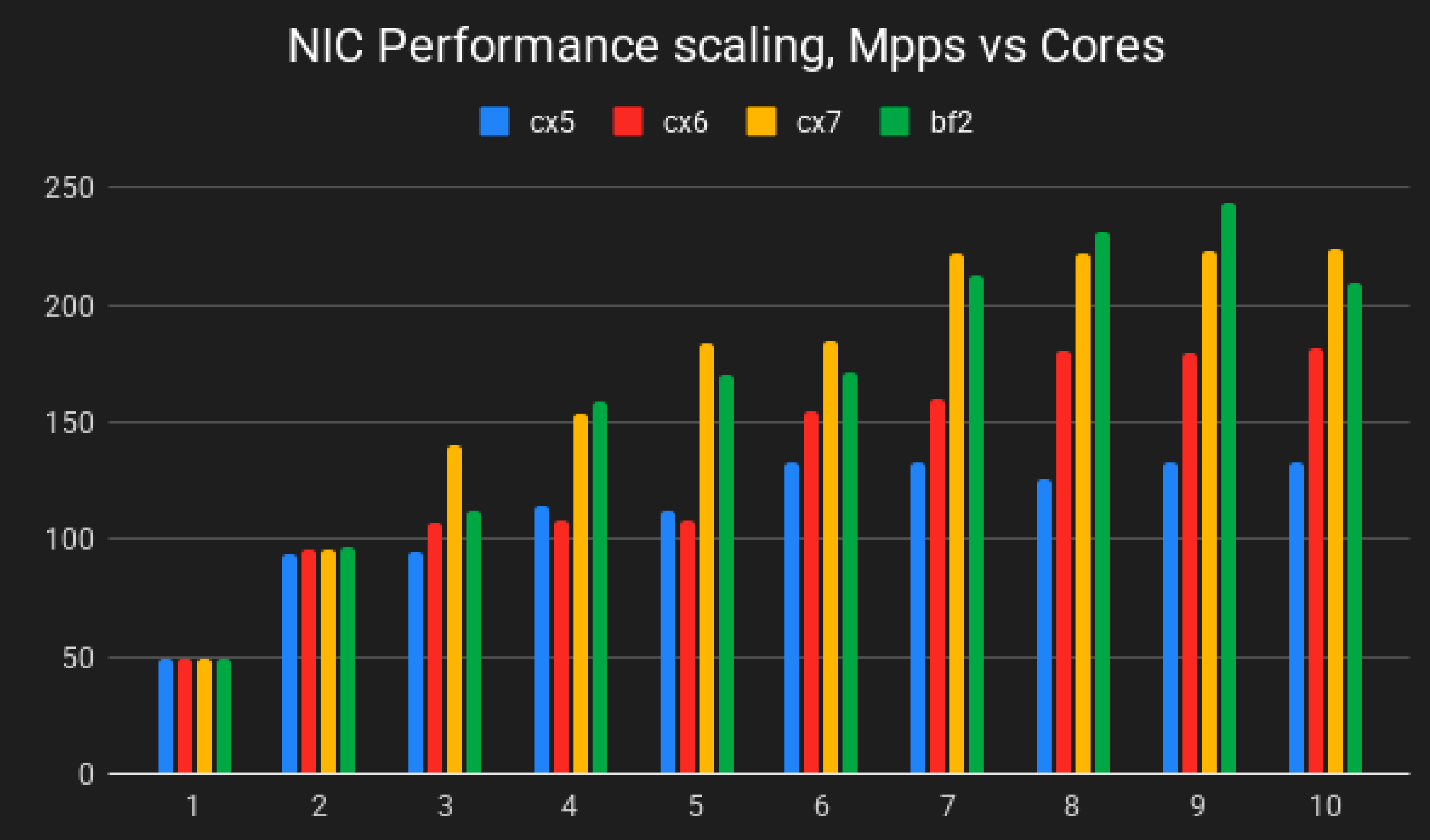
➜ nog.fi git:(part 2: Tests & Software)

Conclusion

Ideally get CX7s, if they are cheap.

If not — BF2 and CX6 are second best.

CX5 is not worth it unless very cheap.



→ nog.fi git:(part 2: Tests & Software)

Performance scaling

When CPU util close to 100% — there is huge difference between target vs observed pps:

```
INFO:SimplePacketTest::do_test:Updating rate to 246000kpps
INFO:SimplePacketTest::do_test:Done
INFO:SimplePacketTest:Packets injected from [0]: 915,990,911
INFO:SimplePacketTest:Packets injected from [1]: 916,002,260
INFO:SimplePacketTest:packets lost from [0] --> [1]:   -13,924 pkts
INFO:SimplePacketTest:packets lost from [1] --> [0]:   13,928 pkts
INFO:SimplePacketTest:packet rate [0] --> [1]:   183,199,316.7 pkts/s
INFO:SimplePacketTest:packet rate delta vs previous [0] --> [1]:   1,634,972.099999994 pkts/s
```

➜ nog.fi git:(part 3:)

Hyperthreading

Have 16 cores / ideally need 30 or even more / have 32 threads

Why not to test HT?

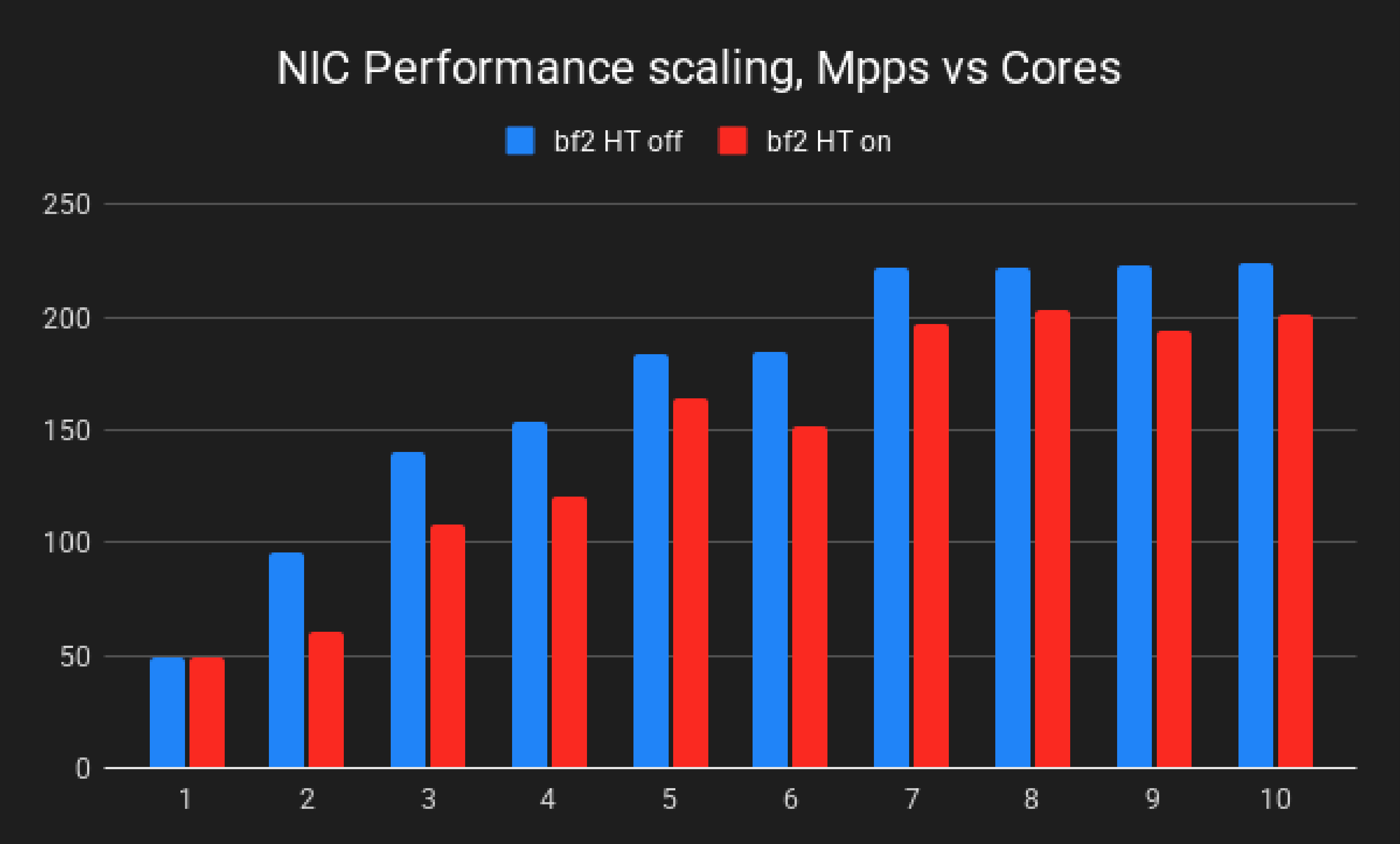
➜ nog.fi git:(part 3: Hyperthreading)

Conclusion

HT on Sapphire Rapids is about 0.4 of a normal core.

Less predictable performance overall.

It is better than nothing and not so harmful as people think.



→ nog.fi git:(part 4: Testing generation capability)

Setup:

- [>] Xeon W5-3435X
- [>] 2xConnectX-7
- [>] 4xConnectX-6
- [>] 5 threads per NIC

➜ nog.fi git:(part 4: Testing generation capability)

Attempt 1

Global Statistics

connection	: localhost, Port 4501	total_tx_L2	: 616.04 Gbps
version	: STL @ v3.06	total_tx_L1	: 712.3 Gbps
cpu_util.	: 82.08% @ 30 cores (5 per dual port)	total_rx	: 628.49 Gbps
rx_cpu_util.	: 0.0% / 0 pps	total_pps	: 601.61 Mpps
async_util.	: 0% / 469.35 bps	drop_rate	: 0 bps
total_cps.	: 0 cps	queue_full	: 595,160,947 pkts

128b packets,
just slightly above 600 Gbps L2.

Doesn't scale beyond that.

Port Statistics

port	0	1	2	3	4	5	6	7	8	9	10	11	total
owner	root	root	root	root	root	root	root	root	root	root	root	root	
link	UP	UP	UP	UP	UP	UP	UP	UP	UP	UP	UP	UP	
state	TRANSMITTING	TRANSMITTING	TRANSMITTING	TRANSMITTING	TRANSMITTING	TRANSMITTING	TRANSMITTING	TRANSMITTING	TRANSMITTING	TRANSMITTING	TRANSMITTING	TRANSMITTING	
speed	100 Gb/s	100 Gb/s	100 Gb/s	100 Gb/s	100 Gb/s	100 Gb/s	200 Gb/s	200 Gb/s	200 Gb/s	200 Gb/s	100 Gb/s	100 Gb/s	
CPU util.	89.31%	89.31%	88.68%	88.68%	89.01%	89.01%	61.61%	61.61%	76.81%	76.81%	87.06%	87.06%	
--													
Tx bps L2	51.49 Gbps	45.38 Gbps	50.53 Gbps	50.7 Gbps	50.87 Gbps	50.68 Gbps	54.14 Gbps	54.08 Gbps	54.32 Gbps	54.37 Gbps	49.69 Gbps	49.88 Gbps	616.14 Gbps
Tx bps L1	59.53 Gbps	52.47 Gbps	58.43 Gbps	58.62 Gbps	58.82 Gbps	58.6 Gbps	62.59 Gbps	62.53 Gbps	62.81 Gbps	62.87 Gbps	57.46 Gbps	57.67 Gbps	712.41 Gbps
Tx pps	50.28 Mpps	44.32 Mpps	49.35 Mpps	49.51 Mpps	49.68 Mpps	49.5 Mpps	52.87 Mpps	52.82 Mpps	53.04 Mpps	53.1 Mpps	48.53 Mpps	48.71 Mpps	601.7 Mpps
Line Util.	**59.53 %**	**52.47 %**	**58.43 %**	**58.62 %**	**58.82 %**	**58.6 %**	**31.3 %**	**31.27 %**	**31.4 %**	**31.43 %**	**57.46 %**	**57.67 %**	

Rx bps	46.91 Gbps	52.96 Gbps	52.11 Gbps	52.29 Gbps	52.46 Gbps	52.27 Gbps	54.14 Gbps	54.08 Gbps	54.32 Gbps	54.37 Gbps	51.25 Gbps	51.43 Gbps	628.59 Gbps
Rx pps	44.42 Mpps	50.15 Mpps	49.35 Mpps	49.51 Mpps	49.68 Mpps	49.5 Mpps	52.87 Mpps	52.82 Mpps	53.04 Mpps	53.1 Mpps	48.53 Mpps	48.71 Mpps	601.68 Mpps

➜ nog.fi git:(part 4: Testing generation capability)

Attempt 1

> Recompile T-Rex with Intel's OneAPI Compiler, tuning BIOS, overclocking CPU.
>
> 256b packets, manual core assignments — highest result is 850 Gpbs L2.

```
Global Statistics

connection   : localhost, Port 4501                  total_tx_L2  : 850.18 Gbps
version      : STL @ v3.06                           total_tx_L1  : 916.6 Gbps
cpu_util.    : 98.63% @ 30 cores (6 per dual port)   total_rx     : 801.55 Gbps
rx_cpu_util. : 0.0% / 0 pps                          total_pps    : 415.12 Mpps
async_util.  : 0% / 240.92 bps                       drop_rate    : 0 bps
total_cps.   : 0 cps                                 queue_full   : 249,983,506 pkts
```

➜ nog.fi git:(part 4: Testing generation capability)

Attempt 1. Analysis

- [>] Single card works fine in this machine.
- [>] When you add traffic — performance drastically drops.
- [>] Performance drops on a NIC that doesn't share cores or threads.

➜ nog.fi git:(part 4: Testing generation capability)

Attempt 1. Analysis

Port Statistics

port	0	1	2	3
owner	root	root	root	root
link	UP	UP	UP	UP
state	TRANSMITTING	TRANSMITTING	TRANSMITTING	TRANSMITTING
speed	100 Gb/s	100 Gb/s	200 Gb/s	200 Gb/s
CPU util.	98.53%	98.53%	98.74%	98.74%
--				
Tx bps L2	81.93 Gbps	81.75 Gbps	90.54 Gbps	90.38 Gbps
Tx bps L1	88.33 Gbps	88.14 Gbps	97.61 Gbps	97.44 Gbps
Tx pps	40.01 Mpps	39.92 Mpps	44.21 Mpps	44.13 Mpps
Line Util.	**88.33 %**	**88.14 %**	**48.8 %**	**48.72 %**

Rx bps	73.69 Gbps	73.53 Gbps	90.54 Gbps	90.38 Gbps
Rx pps	35.43 Mpps	35.35 Mpps	44.21 Mpps	44.13 Mpps

Attempt 1. Analysis

[>] Potential culprit: Sub-NUMA clustering.

[>] Modern server CPUs are non-uniform.

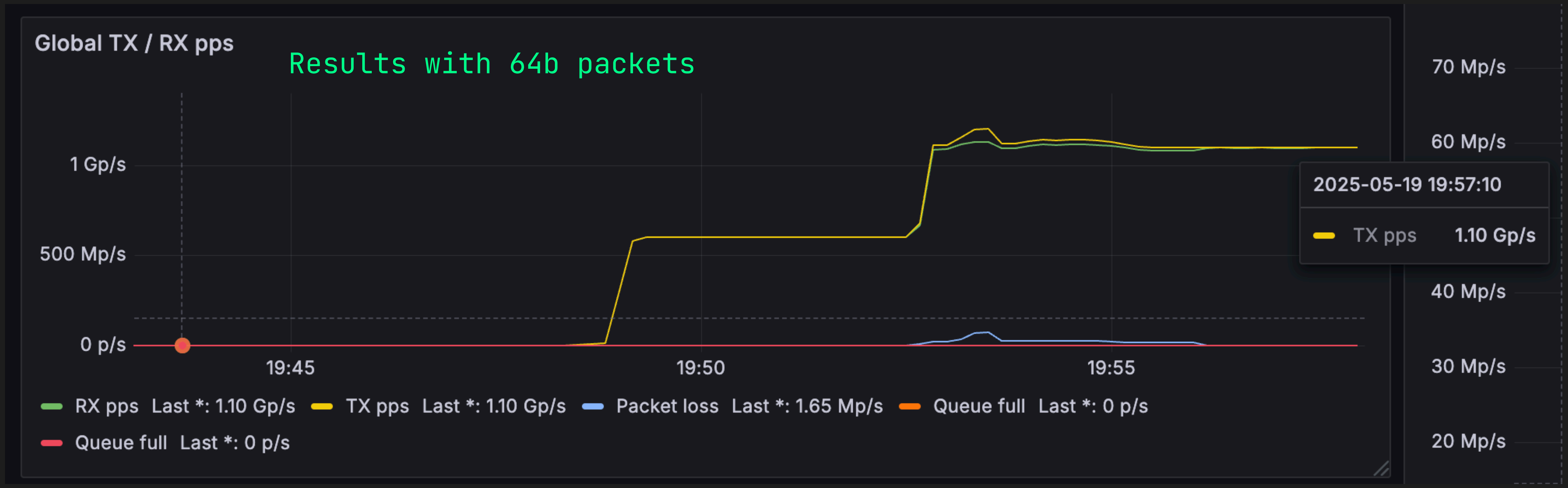
[>] W5-3435 consists of 4 clusters, 4 cores each.

[>] Communication between clusters is not ideal, when all PCIe Root Complexes are used.

➜ nog.fi git:(part 4: Testing generation capability)

Attempt 2

> Add second machine. Same Xeon W5-3435X, swap all NICs for BF2s 1x200G



➜ nog.fi git:(part 4: Testing generation capability)

Results

- [>] 128b packets — 1.06Gpps & 1Tbps
- [>] Required 2 machines
- [>] Could replace with 1 machine, but with 2x cores
- [>] Power consumption: 870W and 630W
- [>] Slight difference — different settings

➜ nog.fi git:(part 5:)

Conclusions

[>] You can build a load generator on a relatively tight budget:

W5-3435X + MB + RAM costed ~3.5k EUR

[>] It is possible to get one relatively small machine to do ~0.8 Tbps,

4x T-Rex claims one machine can.

[>] You should allocate 1 real core per each 35 Mpps you generate

[>] HyperThreading helps, but not as much, adds ~10 Mpps.

[>] Vendor's compiler might help when you really need that extra 5–10% speed

➜ nog.fi git:(part 5:)

Plans for the future

- [>] Investigate cross-effects of different NICs and Sub-NUMA
- [>] Bluefield-2 have an ARM CPU, might be able to do 100G from just the standalone NIC
- [>] Do more extensive tests & report for the VPP (see Pim's talk at DENOG16)
- [>] Test AMD Epyc machine as a load generator
- [>] Try to fit load generation into single one machine

➜ nog.fi git:(questions)

Thank you for your time.

➜ nog.fi git:(outro)

Contacts

[>] linkedin: vladsmirnov

[>] github: Civil

[>] email: civil.over@gmail.com

[>] discord/telegram: @Civiloid

➜ nog.fi git:(bonus)

Power consumption

```
corsairpsu-hid-3-2
Adapter: HID adapter
v_in:         230.00 V
v_out +12v:    11.98 V  (crit min =  +8.41 V, crit max = +15.59 V)
v_out +5v:      5.06 V  (crit min =  +3.50 V, crit max =  +6.50 V)
v_out +3.3v:    3.34 V  (crit min =  +2.31 V, crit max =  +4.30 V)
psu fan:       424 RPM
vrm temp:      +57.8°C  (crit = +70.0°C)
case temp:     +54.8°C  (crit = +70.0°C)
power total: 630.00 W
power +12v:  604.00 W
power +5v:    17.00 W
power +3.3v:   9.00 W
curr +12v:    50.50 A  (crit max = +168.75 A)
curr +5v:      3.44 A  (crit max = +40.00 A)
curr +3.3v:    2.75 A  (crit max = +40.00 A)
```

➜ nog.fi git:(bonus)

Power consumption

```
corsairpsu-hid-3-1
Adapter: HID adapter
v_in:          230.00 V
v_out +12v:     11.92 V  (crit min =  +8.41 V, crit max = +15.59 V)
v_out +5v:       5.03 V  (crit min =  +3.50 V, crit max =  +6.50 V)
v_out +3.3v:     3.33 V  (crit min =  +2.31 V, crit max =  +4.30 V)
psu fan:        472 RPM
vrm temp:       +61.5°C  (crit = +70.0°C)
case temp:      +48.0°C  (crit = +70.0°C)
power total:   866.00 W
power +12v:    842.00 W
power +5v:      17.00 W
power +3.3v:     8.00 W
curr +12v:      70.75 A  (crit max = +168.75 A)
curr +5v:        3.44 A  (crit max = +40.00 A)
curr +3.3v:      2.50 A  (crit max = +40.00 A)
```

➜ nog.fi git:(bonus: Full test results, Attempt 1)

Global Statistics

```
connection   : localhost, Port 4501                  total_tx_L2  : 850.18 Gbps
version      : STL @ v3.06                           total_tx_L1  : 916.6 Gbps
cpu_util.    : 98.63% @ 30 cores (6 per dual port)   total_rx     : 801.55 Gbps
rx_cpu_util. : 0.0% / 0 pps                          total_pps    : 415.12 Mpps
async_util.  : 0% / 240.92 bps                       drop_rate    : 0 bps
total_cps.   : 0 cps                                 queue_full   : 249,983,506 pkts
```

Port Statistics

port	0	1	2	3	4	5	6	7	8	9	total
owner	root	root	root	root	root	root	root	root	root	root	
link	UP	UP	UP	UP	UP	UP	UP	UP	UP	UP	
state	TRANSMITTING	TRANSMITTING	TRANSMITTING	TRANSMITTING	TRANSMITTING	TRANSMITTING	TRANSMITTING	TRANSMITTING	TRANSMITTING	TRANSMITTING	
speed	100 Gb/s	100 Gb/s	200 Gb/s	200 Gb/s	100 Gb/s	100 Gb/s	100 Gb/s	100 Gb/s	200 Gb/s	200 Gb/s	
CPU util.	98.53%	98.53%	98.74%	98.74%	98.71%	98.71%	98.55%	98.55%	98.61%	98.61%	
--											
Tx bps L2	81.93 Gbps	81.75 Gbps	90.54 Gbps	90.38 Gbps	81.73 Gbps	81.67 Gbps	81.62 Gbps	81.76 Gbps	89.15 Gbps	89.59 Gbps	850.14 Gbps
Tx bps L1	88.33 Gbps	88.14 Gbps	97.61 Gbps	97.44 Gbps	88.12 Gbps	88.06 Gbps	88 Gbps	88.15 Gbps	96.11 Gbps	96.59 Gbps	916.55 Gbps
Tx pps	40.01 Mpps	39.92 Mpps	44.21 Mpps	44.13 Mpps	39.91 Mpps	39.88 Mpps	39.86 Mpps	39.92 Mpps	43.53 Mpps	43.75 Mpps	415.11 Mpps
Line Util.	**88.33 %**	**88.14 %**	**48.8 %**	**48.72 %**	**88.12 %**	**88.06 %**	**88 %**	**88.15 %**	**48.06 %**	**48.29 %**	

Rx bps	73.69 Gbps	73.53 Gbps	90.54 Gbps	90.38 Gbps	73.52 Gbps	73.46 Gbps	73.77 Gbps	73.9 Gbps	89.15 Gbps	89.59 Gbps	801.52 Gbps
Rx pps	35.43 Mpps	35.35 Mpps	44.21 Mpps	44.13 Mpps	35.35 Mpps	35.32 Mpps	35.47 Mpps	35.53 Mpps	43.53 Mpps	43.75 Mpps	388.05 Mpps
----											
opackets	1360986929	1362278804	2065519009	2067051733	1367431365	1368808038	1364586338	1365978545	2040908500	2042470517	16406019778
ipackets	1204154540	1205494066	2065502991	2067066867	1207715372	1208960624	1209833738	1211071216	2040952158	2042423563	15463175135
obytes	348412653824	348743373824	528772866304	529165243648	350062429440	350414857728	349334102528	349690507520	522472576000	522872452352	4199941063168
ibytes	313080180232	313428456992	528768765696	529169117952	314005996552	314329762072	314556771880	314878516160	522483752448	522860432128	3987561752112
tx-pkts	1.36 Gpkts	1.36 Gpkts	2.07 Gpkts	2.07 Gpkts	1.37 Gpkts	1.37 Gpkts	1.36 Gpkts	1.37 Gpkts	2.04 Gpkts	2.04 Gpkts	16.41 Gpkts
rx-pkts	1.2 Gpkts	1.21 Gpkts	2.07 Gpkts	2.07 Gpkts	1.21 Gpkts	1.21 Gpkts	1.21 Gpkts	1.21 Gpkts	2.04 Gpkts	2.04 Gpkts	15.46 Gpkts
tx-bytes	348.41 GB	348.74 GB	528.77 GB	529.17 GB	350.06 GB	350.41 GB	349.33 GB	349.69 GB	522.47 GB	522.87 GB	4.2 TB
rx-bytes	313.08 GB	313.43 GB	528.77 GB	529.17 GB	314.01 GB	314.33 GB	314.56 GB	314.88 GB	522.48 GB	522.86 GB	3.99 TB
-----											
oerrors	0	0	0	0	0	0	0	0	0	0	0
ierrors	0	0	0	0	0	0	0	0	0	0	0

status: |

→ nog.fi git:(bonus: Tests & Software)

Problem 2

```python
def create_stream (self, size, vm, src, dst):
    # Create base packet and pad it to size
    base_pkt = Ether()/IP(src=src, dst=dst)/UDP(sport=self.ports['min'],dport=30,chksum=0)
    pad = max(0, size - len(base_pkt) - 4) * 'x'
    pkt = STLPktBuilder(pkt=base_pkt/pad,
                        vm=vm)

    return STLStream(packet=pkt,
                     mode=STLTXCont(pps=1),
                     isg=0,
                     flow_stats=None)
```

Available on Github

➜ nog.fi git:(bonus: Tests & Software)

Problem 2

```
        src, dst = f'16.0.{int(port_id/2)+offset}.1', f'48.0.{int(port_id/2)+offset}.1'

        vm_var = STLVM()

        vm_var.var(name='ip', min_value=0, max_value=255, size=1, op='random')
        vm_var.var(name='port', min_value=self.ports['min'], max_value=self.ports['max'], size=2, op='random')
        vm_var.write(fv_name='ip', pkt_offset='IP.src', offset_fixup=3)
        vm_var.write(fv_name='ip', pkt_offset='IP.dst', add_val=128, offset_fixup=3)
        vm_var.write(fv_name='port', pkt_offset='UDP.sport')
        vm_var.fix_chksum()
        vm_var.set_cached(args.cache)

        return [self.create_stream(size, vm_var, src, dst)]
```

Available on Github