Academic Year 2023/2024

Internet Performance and Troubleshooting Lab

Politecnico di Torino

Report III

Performances with 2.5 Gb/s line-cards

Group 3

Brendon Mendicino (s317639)

Alessandro Ciullo (s310023)

Davide Colaiacomo (s313372)

1 Network Configuration

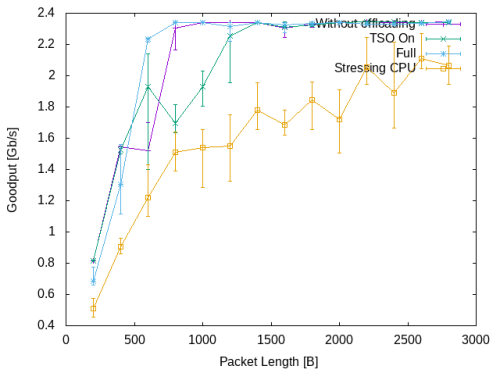
The network configuration used during the experiments is the following:

Figure 1.1: Network Topology

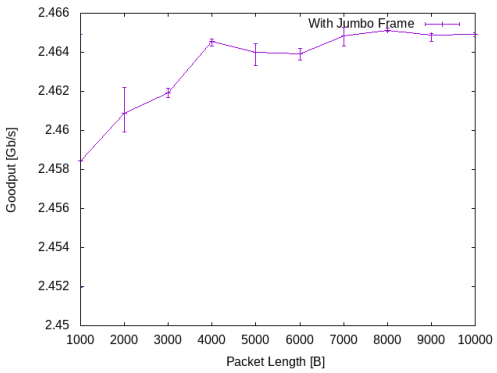
Host name	**IP address**	Role
H1	10.0.3.1/25	Server
H3	10.0.3.3/25	Client

Table 1.1: Net hosts

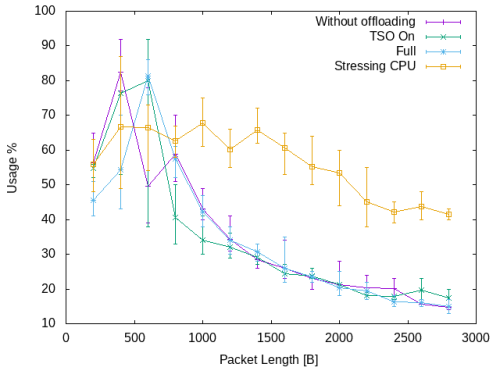
2 Experiment Setup



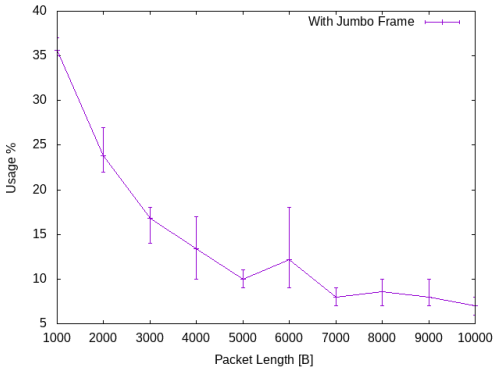
(a) Good-put against Payload Length



(b) Good-put with Jumbo Frames enabled

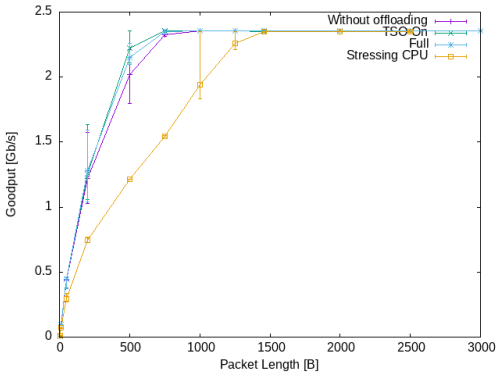


(c) CPU Usage %

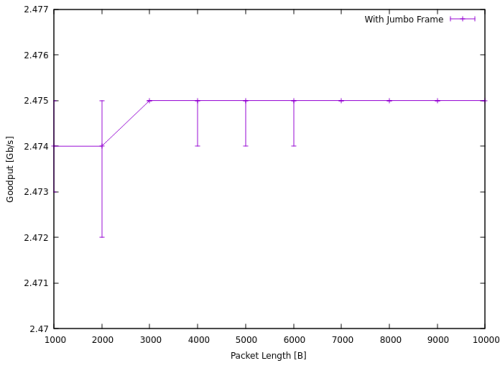


(d) CPU Usage % with Jumbo

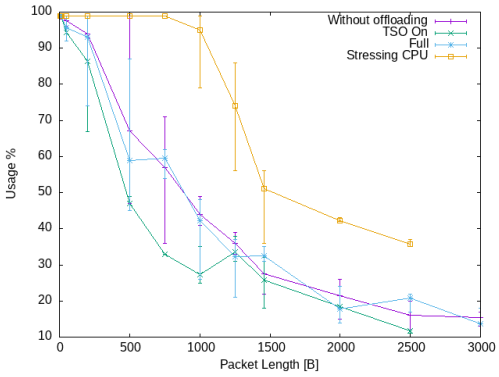
Figure 2.1: `iperf3` capture (On the Sender)



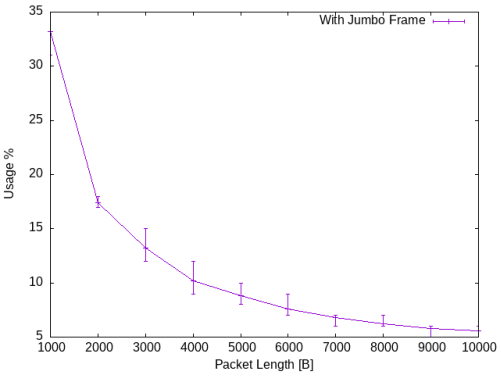
(a) Good-put against Payload Length



(b) Good-put with Jumbo Frames enabled



(c) CPU Usage %



(d) CPU Usage % with Jumbo

Figure 2.2: `nttcp` capture (On the Sender)

We performed a series of experiments to test how the different sizes of packet length impacted the **Good-put** in a TCP communication; this was done in conjunction with the offloading capabilities of the Network Card used for this experiment, which was *Realtek 2.5Gb*. We conducted 5 kinds of experiment with each program to do **speed tests**: `iperf3` and `nttcp`. The various configurations were all tested on the Sender (sends data), on which the measurements were taken:

- **No Offloading Enabled.**
- **TSO Offloading Enabled.**
- **Every Offloading Capability Enabled.**
- **Every Offloading Capability Enabled with the System Under Stress.**
- **Every Offloading Capability Enabled with the Jumbo Frame Enabled.**

2.1 Comparison between `nttcp` and `iperf3`

After performing various tests and plotting the data retrieved, using both `iperf3` and `nttcp`, we noticed that the graphical trends generally resemble each other, but with some noticeable differences:

- by looking at the **CPU usage %** graphs, it can be noted that the CPU usage is averagely much higher using `nttcp`; in fact, we can see that, when the packet size is low, the CPU's busyness is over 90%, especially in the **Stressing CPU** case. This could lead to think that `nttcp` implements a rower *sending-loop*, while `iperf3` provides a more sophisticated one, with some level of optimization in the way it sends the buffers to the server.

- by looking at the **Good-put against Payload Length** graphs, it is possible to see that the graphical progress is more stable when using `nttcp`: the boundaries are clearer and it actually takes the shape of a monotonic increasing curve, which is as expected when increasing the packet size; it cannot be said the same for the experiment with `iperf3`, as it can be observed that the curves have many more oscillations than the ones of `nttcp` and the boundaries are more uncertain.
- by looking again at the **Good-put against Payload Length** graphs, using `nttcp`, the good-put establishes itself at around 2.3 Gb/s with block size from about 750 bytes on (with the exception of the **Stressing CPU** case, whose block size to get the same behavior is from about 1500 bytes on). On the other hand, using `iperf3`, this trend is replicated only when all offloading capabilities are disabled or enabled; if the TSO is the only offloading capability enabled, `iperf3` struggles more to reach the saturation point and, in case of **Stressing CPU**, a stability in the good-put is actually never reached for the duration of the experiment, even for block size over 2500 bytes.
- by looking at the **Good-put with Jumbo Frames enabled** graphs, it can be seen that `nttcp` reaches the saturation with packet length equal to 3000 bytes, while `iperf3` touches the saturation point at 4000 bytes packet length and then struggles to maintain its stability.

2.2 Theoretical Limits of the Good-put

We can now take a rough calculation of what is the **efficiency** of this exchange of TCP messages. We can estimate, for the maximum efficiency, that TCP is exchanging data using the MSS and, by taking into account the additional overhead, we can calculate the efficiency η_{TCP}.

$$\eta_{TCP} = \frac{1448_{MSS}}{1448_{MSS} + 32_{TCP_H} + 20_{IP_H} + 38_{ETH_H}} = 94\%$$

If we compute the theoretical maximum **good-put** on the $2.5Gb/s$ network card we get: $\eta_{TCP} \cdot 2.5Gb/s = 2.35Gb/s$; as a matter of facts, we get as the maximum value of the **Good-put** from `nttcp` and `iperf3` at about $2.35Gb/s$ and $2.34Gb/s$ respectively, for which, in both cases, all the offloading capabilities were enabled.

If we enable **jumbo frames** on the network card, we can have a MTU of 9000 Bytes. By doing so, we do not increase the speed of the network card but we increase the **efficiency**.

$$\eta_{TCP,Jumbo} = \frac{8948_{MSS}}{8948_{MSS} + 32_{TCP_H} + 20_{IP_H} + 38_{ETH_H}} = 99\%$$

We can compute the maximum theoretical **Good-put** with **Jumbo Frames** enabled, which is $\eta_{TCP,Jumbo} \cdot 2.5Gb/s = 2.476Gb/s$. In the graphs, we observe that, by increasing the packet size up to 10000 bytes, we get a good-put of about $2.475Gb/s$ and $2.465Gb/s$ for `nttcp` and `iperf3` respectively, for which, in both cases, all the offloading capabilities were enabled.

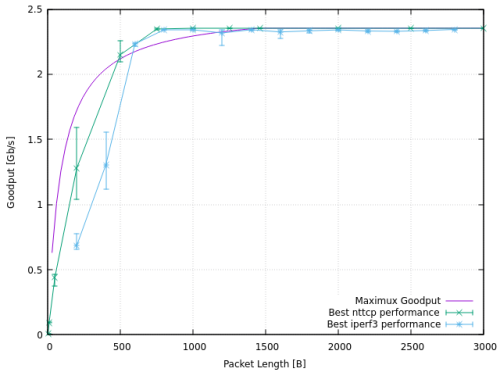


Figure 2.3: Maximum Good-put (No Jumbo)

2.3 Impact of the block size on the Good-put

The first thing we can observe from the graph is, in every analyzed case, that an increase in block size leads to an improvement in `Good-put`, but further considerations must be done.

As shown in figure 2.3, it's possible to see the two best scenarios for both `nttcp` and `iperf3` plotted against the theoretical maximum good-put curve and what immediately catches the eye is that the actual curves are significantly different from the theoretical curve. For small `Block size`, in fact, we have

a sub-optimal `Good-put`; an explanation for this phenomenon
can be extrapolated by looking at the **CPU usage %**: with small blocks we can notice that the CPU usage is very high, due to `iperf3` and `nttcp` creating in loop many small buffers and then invoking the system call `write()` to send the data through the **socket**. The majority of the time is spent creating these small buffers and invoking a **system call**, the latter being a particularly complex operation from a computational point of view.

At around 800 bytes in `Block size`, we observe an even stranger phenomenon: the real `Good-put` curve surpasses the `theoretical maximum good-put`, which is supposed to be a non surmountable limit. In reality, using **Wireshark**, we observed that for most of the packets' payload length was bigger than the `Block size` set in the running application for `block size` smaller than 1448 bytes. Due to the way TCP segmentation is implemented, the kernel (or the Network Interface when TSO is ON) groups multiple blocks to achieve better efficiency, resulting in increased `Good-put`, while our theoretical curve doesn't address this behavior. Not knowing how the kernel is implemented, we can theorize as another plausible reason for this behavior that, when the application make a `write()` system call on a socket, the kernel will maintain an **inner buffer** that will dump on the socket only when it is not busy; as a consequence, other calls to the `write()` system call might add more data to that specific buffer before it is actually passed to the underlying layer. As a side note, this behavior is also noticeable in the figure 2.2b, where the **good-put** already starts saturating at 3000 bytes, this should not be possible, in fact that Good-put value can only be achieved at MSS, but as said above, the Kernel performs some optimizations.

From this point onward, as the `Block size` increase, the actual curves consistently tend more closely towards the theoretical curve because the TCP segmentation is limited by the MTU size.

2.4 CPU time for the tests

We used, as a measure of `CPU-load`, the percentage of CPU used during the processing of an experiment; in order to retrieve this value, we used `/usr/bin/time` followed by the `nttcp/iperf3` command. One detail that catches the eye is that the CPU usage of `nttcp` is always higher than the CPU usage of `iperf3`; this could be due to the fact that `iperf3` has a `pselect()` system call between the `write()` system calls (we observed that using `strace`). As stated in the Documentation, this system call monitors the state of a File Descriptor, and returns when its state is ready for an operation to be performed on; in our case, the `pselect()` waits for the socket to be ready, and then the process writes on it, allowing to reduce the workload on the CPU.

3 Analysis of the offloading capabilities

3.1 Driver's offloading capabilities

The driver offers a variety of offloading capabilities. These are:

- **Receiving/Transmission Offloading (RX/TX)**: this capability allows the NIC of the receiver/-transmitter to perform the validation of the checksum field for the `TCP` header, while the `IP` checksum is always validated through software, because, when it is built, it is already in cache, so it is not expensive to sum it, as stated in the Linux Documentation.
- **TCP Segmentation Offloading (TSO)**: this capability allows the usage of the NIC card to segment large chunks of data into smaller TCP segments, also adding a TCP, an IP and an Ethernet header, in order to reduce the CPU overhead. If this capability is disabled, then the CPU has the burden to segment the data before the individual packets will be sent.
- **Large Receive Offload (LRO)**: this capability allows to reassemble small packets into larger buffers as they are received, reducing the number that the CPU has to process.
- **Scatter-gather (SG)**: this capability allows to pass around small buffers that make up for a big logical one, instead of passing directly the large packet; this allows a quicker and more efficient data processing .

- **Generic Segmentation/Receive Offloading (GSO/GRO)**: this is an alternative to the solution proposed by the TSO capability, which means to handle, through a specific software implementation, the segmentation when the NIC is not capable of doing so. It allows upper layer applications to process large packets and the segmentation is delayed as much as possible, reducing per-packet overhead.

3.2 Impact of the offloading capabilities

Observing the graphs 2.1c and 2.2c, we can perceive how different network interface offloading settings can generate small but not unnoticeable changes in the CPU load.

3.2.1 Difference between Nttcp and Iperf3

First of all we observe slightly different behavior between the `Iperf3` graphs and the `Nttcp` graphs. Starting from Nttcp, we can notice a way more aggressive implementation: the CPU load when payload is short reaches nearly 100% without any stress from outside sources. This overhead seems to be caused by an uncontrolled usage of the syscall `write` and the following context switch between user and kernel mode: in fact, this overhead seems to become increasingly negligible as the length of the packets grows and the biggest part of the CPU load becomes the generation of the payload and with it the differences with `Iperf3` decrease.

By using `strace`, we spot that in `Iperf3` the calls to the `write` system-call are interleaved with calls to the `pselect`, another system call used to monitor file descriptor and the possible action that can be performed to the corresponding file to avoid CPU waste.

Even if not relevant to this experiment, by using `strace` we also observe that the reason why the `nttcp` receiver has more system calls than the sender is the presence of a continuous call to a non-blocking `read` syscall, without first checking that the socket isn't empty.

3.2.2 Offloading configuration differences

Once the differences between the two applications are overcome, we can focus on the differences between the different configuration we can adopt on the Network Interface. Starting by disabling all the offloading, we can observe that the curve representing this scenario is, in average, above the other curves of the *non-stressed cluster*. This curve differs significantly from the others between `Block size` of 500 and 1200 bytes, where there's more than a 20% load difference with the `TSO on curve`. When the `Block size` is big, the overhead caused by the TCP/IP stack's headers to add becomes negligible and with it the differences with the other curves.

In one of the other test we enabled all the offloading and in the other one we only enabled the `TSO` and found out that the latter, even if counter-intuitively, has slightly better performance. This could be caused by the fact that we only use TCP in this experience and a more generic set of offloads isn't needed, causing a loss of performance.

3.3 Analysis with Jumbo frames

This experiment aims to observe the behavior of the network and its properties in terms of `Good-put` and `CPU-load` when jumbo frames and all offloads are enabled.

3.3.1 Good-put with Jumbo frame

The first thing we can observe is the greater `Good-put` that nearly saturates the capacity of the channel, even with short length payload, with a value of $2.458 Gb/s$ (in the `iperf3` case), greater than the maximum `Good-put` reached with a normal MTU of 1500. This is a direct consequence of a better `efficiency` compared to a standard MTU: the `TSO and GSO` collect a lot more data before adding the packet's header, consequentially increasing the `Good-put`. The `Good-put` continues to slightly increase in a directly proportional manner to the `Block size`. In Figure 3.1, we can see the different graphs compared to each other (when all the Offloading Capabilities are enabled): it's noticeable that, even when the packet size is set at

around 1000 Bytes, the efficiency of the Jumbo packets is much greater then the one without them. If the kernel or the NIC decides to group more data, when the process passes its blocks to the underlying layers, then it will probably send them in packets a lot larger than 1000 Bytes.

3.3.2 CPU load with Jumbo frames

The `CPU-load` in this experience is exactly the same as the corresponding one (Full offload) with a 1500 bytes MTU. The only cause of the `CPU-load` is the generation of the payload, that has no differences in the two scenarios, while all the TCP/IP stack headers are managed by the Network Interface. We would probably observe a decrease in the `CPU-load` with the Network Interface offloading disabled because the headers would be built at the expense of CPU resources and less headers are needed with a bigger MTU.

In this experiment we went way further with the `Block size` and, as expected, the `CPU-load` continues to decrease in an inversely proportional manner relative to it.

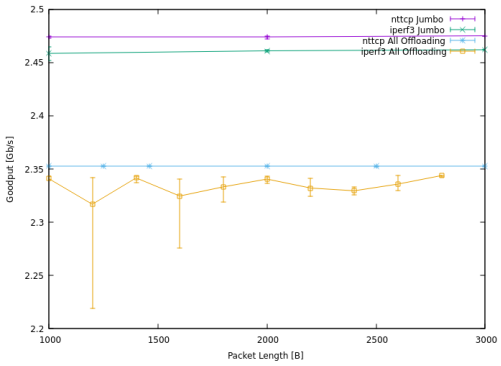


Figure 3.1: Comparing Jumbo to Non-Jumbo Tests

4 Performances with busy CPU

We run `stress` application to test the network behavior when the transmitter is overloaded. The first thing we observe is a significant loss in term of `Good-put` and also a significant increase in the CPU usage of both the application.

4.1 CPU-load

The most relevant thing we can observe is the increase in the CPU-load: the scheduler need to context switch a lot more between the applications when the system is stressed, causing a significant increase in the CPU usage of our applications. While, apparently, both `Nttcp` and `iperf` show the same general behavior, upon closer inspection differences can be noticed. Contrary to what was seen earlier `nttcp` has a lower system load compared to `iperf3` (beside for small block size where is still 100%). Paradoxically, this could be due to the more aggressive approach nttcp uses: by leaving to the scheduler less opportunities to switch to another thread, the context-switch overhead decreases.

4.2 Good-put

What we have just seen is also reflected in the `Good-put`: while nttcp, even if at higher block sizes, is able to saturate the channel capacity, iperf3 fails to do so at any block size. In any case, the general trend is towards a lower good-put compared to a non-overloaded system.

4.3 Test reliability consideration

The result we obtained are also not very reliable because iperf3 sends data until 10 seconds are passed while nttcp sends a constant quantity of bytes. In an overloaded system the quantity of bytes sent in a fixed quantity of time and the time needed to send a fixed quantity of bytes is a lot more unpredictable, causing a greater variance over the experiment.

5 Appendix

5.1 Script to generate iperf3 graphs

```
#!/bin/python3

import os
import json
import argparse
from subprocess import Popen, PIPE

parser = argparse.ArgumentParser()
parser.add_argument('-r', '--retrans', action='store', type=int, default=5, help='number of retransmissions')
parser.add_argument('-s', '--step', action='store', type=int, default=300, help='number of steps to reach 3000 bytes')
parser.add_argument('--interface', action='store', type=str, default='eth0', help='interface to modify the offloading')
parser.add_argument('-S', '--server', action='store', type=str, default='10.0.3.1', help='server to connect')
parser.add_argument('-i', '--interactive', action='store_true', help='run gnuplot in interactive mode')
parser.add_argument('-t', '--test', choices=['tso_on', 'full', 'no_off', 'stressed', 'jumbo', 'jumbo_long'], nargs='+', help='test for different modes')
parser.add_argument('-k', '--keepold', action='store_true', help='set if you want to don\'t override old files')
parser.add_argument('-j', '--jumbo', action='store_true', help='do the jumbo test')
args = parser.parse_args()

RETRANS = args.retrans
INTERFACE = args.interface
STEP = args.step
SERVER = args.server
INTERACTIVE = args.interactive
KEEP_OLD = args.keepold
JUMBO = args.jumbo
TEST = args.test if args.test is not None else []

def get_receiver(proc):
    out = proc.stdout.read().decode()
    out = json.loads(out)
    return out["end"]["streams"][0]["receiver"]

def get_cpu(proc):
    out = proc.stderr.read().decode()
    return int(out.replace("%", ""))

def show_res(receiver):
    print(receiver)
    print(receiver["bytes"])

def write_out(output, packet_len, bit_s, bit_min, bit_max, cpu, cpu_min, cpu_max, out_name):
    print(f"len:{packet_len} bit_s:{bit_s} min:{bit_min} max:{bit_max} cpu:{cpu} cpu_min:{cpu_min} cpu_max:{cpu_max}")
    line = f"{packet_len} {bit_s} {bit_min} {bit_max} {cpu} {cpu_min} {cpu_max}\n"
    output.write(line)
    #os.system(f"echo {packet_len} {bit_s} {bit_min} {bit_max} {cpu} {cpu_min} {cpu_max}>> {out_name}")

def run_test(out_name, jumbo = False):
    stop = 3000 if not jumbo else 10001

    with open(out_name, "w") as output:
        for packet_len in range(STEP, stop, STEP):
            bit_s = []
            cpu_t = []
```

```
            for _ in range(RETRANS):
                with Popen(["/usr/bin/time", "-f", "%P", "iperf3", "-J", "-l", str(packet_len), "-c", SERVER], stdout=PIPE, stderr=PIPE) as proc:
                    # Convert them in Gb/s
                    bit_s.append(get_receiver(proc)["bits_per_second"] / 1e9)
                    cpu_t.append(get_cpu(proc))

            bit_min = min(bit_s)
            bit_max = max(bit_s)
            bit_s = sum(bit_s) / RETRANS

            cpu_min = min(cpu_t)
            cpu_max = max(cpu_t)
            cpu = sum(cpu_t) / RETRANS

            write_out(output, packet_len, bit_s, bit_min, bit_max, cpu, cpu_min, cpu_max, out_name)


if 'tso_on' in TEST:
    print('tso on')
    os.system(f"sudo ethtool -K {INTERFACE} rx on tx on sg on tso on gso off gro off")
    run_test('tso_on')


if 'full' in TEST:
    print("full")
    os.system(f"sudo ethtool -K {INTERFACE} rx on tx on sg on tso on gso on gro on")
    run_test("full")


if 'no_off' in TEST:
    print("no offloading")
    os.system(f"sudo ethtool -K {INTERFACE} rx off tx off sg off tso on gso off gro off")
    run_test('no_off')


if 'stressed' in TEST:
    print("cpu stressed")
    os.system(f"sudo ethtool -K {INTERFACE} rx on tx on sg on tso on gso on gro on")
    proc = Popen("stress --cpu 8 --io 4 --vm 2 --vm-bytes 128M --timeout 1000s".split())
    run_test("stressed")
    proc.kill()

if 'jumbo' in TEST:
    print('jumbo frame')
    os.system(f"sudo ethtool -K {INTERFACE} rx on tx on sg on tso on gso on gro on")
    os.system(f"sudo ifconfig {INTERFACE} mtu 9000")
    run_test('jumbo', True)
    os.system(f"sudo ifconfig {INTERFACE} mtu 1500")

# Gnuplot section
persist = '-persist' if INTERACTIVE else ''

def output(out):
    return '' if INTERACTIVE else '''set term png
set output "{0}"
'''.format(out)

gnuplot_cmd = '''gnuplot {0} <<EOF
{1}
set xlabel "Packet Length [B]"
set ylabel "{2}"
plot "no_off" using 1:{3} with errorline title "Without offloading", \
    "tso_on" using 1:{3} with errorline title "TSO On", \
    "full" using 1:{3} with errorline title "Full", \
```

```
    "stressed" using 1:{3} with errorline title "Stressing CPU"
EOF
'''

# Goodput
os.system(gnuplot_cmd.format(persist, output("throughput.png"), "Goodput [Gb/s]", "2:3:4"))

# CPU
os.system(gnuplot_cmd.format(persist, output("cpu.png"), "Usage %", "5:6:7"))

# JUMBO
gnuplot_cmd = '''gnuplot {0} <<EOF
{1}
set xlabel "Packet Length [B]"
set ylabel "{2}"
plot "jumbo" using 1:{3} with errorline title "With Jumbo Frame"
EOF
'''

# Goodput
os.system(gnuplot_cmd.format(persist, output("throughput_j.png"), "Goodput [Gb/s]", "2:3:4")
    )

# CPU
os.system(gnuplot_cmd.format(persist, output("cpu_j.png"), "Usage %", "5:6:7"))
```

5.2 Script to generate nttcp graphs

```
#!/bin/sh
H1=10.0.3.1


/usr/bin/time -f "%P" nttcp -n $1 -l $2 $H1 > data$2.dat 2> cpu$2.dat

Cpu=$(cat cpu$2.dat | tr -d "%")
Tx=$(cat data$2.dat | tail -n 2 | head -n 1 | tr -s " ")
Rx=$(cat data$2.dat | tail -n 1 | head -n 1 | tr -s " ")
Gtx=$(echo $Tx | cut -d " " -f5)
Grx=$(echo $Rx | cut -d " " -f5)

echo $1 $2 >> megalog.log
echo $Tx >> megalog.log
echo $Rx >> megalog.log
echo "" >> megalog.log

echo $Gtx $Grx $Cpu
```

```
#!/bin/python3

import os
import json
import argparse
from subprocess import Popen, PIPE

parser = argparse.ArgumentParser()
parser.add_argument('-r', '--retrans', action='store', type=int, default=5, help='number of
    retransmissions')
parser.add_argument('-s', '--step', action='store', type=int, default=300, help='number of
    steps to reach 3000 bytes')
parser.add_argument('--interface', action='store', type=str, default='eth0', help='interface
     to modify the offloading')
parser.add_argument('-S', '--server', action='store', type=str, default='10.0.3.1', help='
    server to connect')
parser.add_argument('-i', '--interactive', action='store_true', help='run gnuplot in
    interactive mode')
```

```
parser.add_argument('-t', '--test', choices=['tso_on', 'full', 'no_off', 'stressed', 'jumbo', 'jumbo_long'], nargs='+', help='test for different modes')
parser.add_argument('-k', '--keepold', action='store_true', help='set if you want to don\'t override old files')
parser.add_argument('-j', '--jumbo', action='store_true', help='do the jumbo test')
args = parser.parse_args()

RETRANS = args.retrans
INTERFACE = args.interface
STEP = args.step
SERVER = args.server
INTERACTIVE = args.interactive
KEEP_OLD = args.keepold
JUMBO = args.jumbo
TEST = args.test if args.test is not None else []


ls = [1, 3, 10, 50, 100, 200, 500, 750, 1000, 1250, 1448, 2000, 2500, 3000]
ns = [14000000, 13000000, 13000000, 12000000, 11000000, 7000000, 5000000, 4000000, 3000000, 2500000, 2000000, 1500000]

def get_receiver(proc):
    out = proc.stdout.read().decode()
    out = out.split()
    return float(out[1]), float(out[2])

def get_cpu(proc):
    out = proc.stderr.read().decode()
    return int(out.replace("%", ""))

def show_res(receiver):
    print(receiver)
    print(receiver["bytes"])

def write_out(output, packet_len, bit_s, bit_min, bit_max, cpu, cpu_min, cpu_max, out_name):
    print(f"len:{packet_len} bit_s:{bit_s} min:{bit_min} max:{bit_max} cpu:{cpu} cpu_min:{cpu_min} cpu_max:{cpu_max}")
    line = f"{packet_len} {bit_s} {bit_min} {bit_max} {cpu} {cpu_min} {cpu_max}\n"
    output.write(line)
    #os.system(f"echo {packet_len} {bit_s} {bit_min} {bit_max} {cpu} {cpu_min} {cpu_max}>> {out_name}")

def run_test(out_name, jumbo = False):
    stop = 3000 if not jumbo else 10001

    with open(out_name, "w") as output:
        for n, l in zip(ns, ls):
            bit_s = []
            cpu_t = []
            for _ in range(RETRANS):
                with Popen(f'./script.sh {n} {l}'.split(), stdout=PIPE, stderr=PIPE) as proc:
                    # Convert them in Gb/s
                    r = get_receiver(proc)
                    print(r)
                    bit_s.append(r[0] / 1e3)
                    cpu_t.append(r[1])

            bit_min = min(bit_s)
            bit_max = max(bit_s)
            bit_s = sum(bit_s) / RETRANS

            cpu_min = min(cpu_t)
            cpu_max = max(cpu_t)
            cpu = sum(cpu_t) / RETRANS

```

```
            write_out(output, packet_len, bit_s, bit_min, bit_max, cpu, cpu_min, cpu_max,
    out_name)


if 'tso_on' in TEST:
    print('tso on')
    os.system(f"sudo ethtool -K {INTERFACE} rx on tx on sg on tso on gso off gro off")
    run_test('tso_on')


if 'full' in TEST:
    print("full")
    os.system(f"sudo ethtool -K {INTERFACE} rx on tx on sg on tso on gso on gro on")
    run_test("full")


if 'no_off' in TEST:
    print("no offloading")
    os.system(f"sudo ethtool -K {INTERFACE} rx off tx off sg off tso on gso off gro off")
    run_test('no_off')


if 'stressed' in TEST:
    print("cpu stressed")
    os.system(f"sudo ethtool -K {INTERFACE} rx on tx on sg on tso on gso on gro on")
    proc = Popen("stress --cpu 8 --io 4 --vm 2 --vm-bytes 128M --timeout 1000s".split())
    run_test("stressed")
    proc.kill()

if 'jumbo' in TEST:
    print('jumbo frame')
    os.system(f"sudo ethtool -K {INTERFACE} rx on tx on sg on tso on gso on gro on")
    os.system(f"sudo ifconfig {INTERFACE} mtu 9000")
    run_test('jumbo', True)
    os.system(f"sudo ifconfig {INTERFACE} mtu 1500")

# Gnuplot section
persist = '-persist' if INTERACTIVE else ''

def output(out):
    return '' if INTERACTIVE else '''set term png
set output "{0}"
'''.format(out)

gnuplot_cmd = '''gnuplot {0} <<EOF
{1}
set xlabel "Packet Length [B]"
set ylabel "{2}"
plot "no_off" using 1:{3} with errorline title "Without offloading", \
    "tso_on" using 1:{3} with errorline title "TSO On", \
    "full" using 1:{3} with errorline title "Full", \
    "stressed" using 1:{3} with errorline title "Stressing CPU"
EOF
'''

# Goodput
os.system(gnuplot_cmd.format(persist, output("throughput.png"), "Goodput [Gb/s]", "2:3:4"))

# CPU
os.system(gnuplot_cmd.format(persist, output("cpu.png"), "Usage %", "5:6:7"))

# JUMBO
gnuplot_cmd = '''gnuplot {0} <<EOF
{1}
set xlabel "Packet Length [B]"
set ylabel "{2}"
```

```
plot "jumbo" using 1:{3} with errorline title "With Jumbo Frame"
EOF
'''

# Goodput
os.system(gnuplot_cmd.format(persist, output("throughput_j.png"), "Goodput [Gb/s]", "2:3:4")
    )

# CPU
os.system(gnuplot_cmd.format(persist, output("cpu_j.png"), "Usage %", "5:6:7"))
```