Politecnico di Torino

Internet Performance and Troubleshooting Lab

Performance Measurements

Report 3 **Group 9**

Giuseppe Bruno s319892
Michela Salvadori s302544
Salvatore Giarracca s314701

December 10, 2023

1 Introduction

1.1 Environment setup

We set up two hosts, **H1** (the client) with **10.0.9.2** address and **H2** (the server) with **10.0.9.3** address. In the following tests, the data flow is always from **H1** to **H2**.
At the beginning, using the **ethtool** command, the **NIC** is configured as follow: **full duplex**, with a transmission speed of **2.5 Gbit/s** and an MTU set to **1500B**

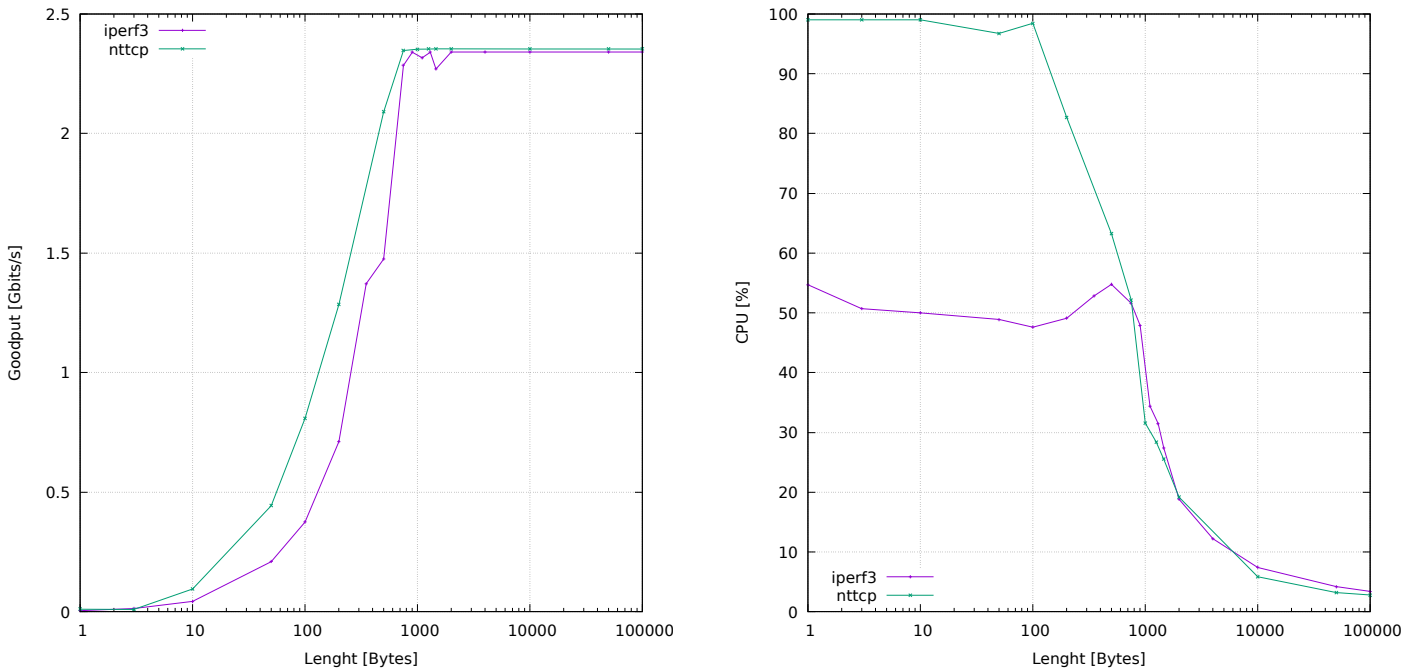
2 IPERF3 vs NTTCP

2.1 Goodput

From the theoretical point of view, the max goodput recheable through a **TCP** connection (in a **full duplex** link) is equal to:

$$\frac{Mss}{Mss + 20 + 20 + 38} \cdot Link\ capacity \tag{1}$$

In this case MSS is equal to **1460** so the final result is **2.37 Gbits/s**.



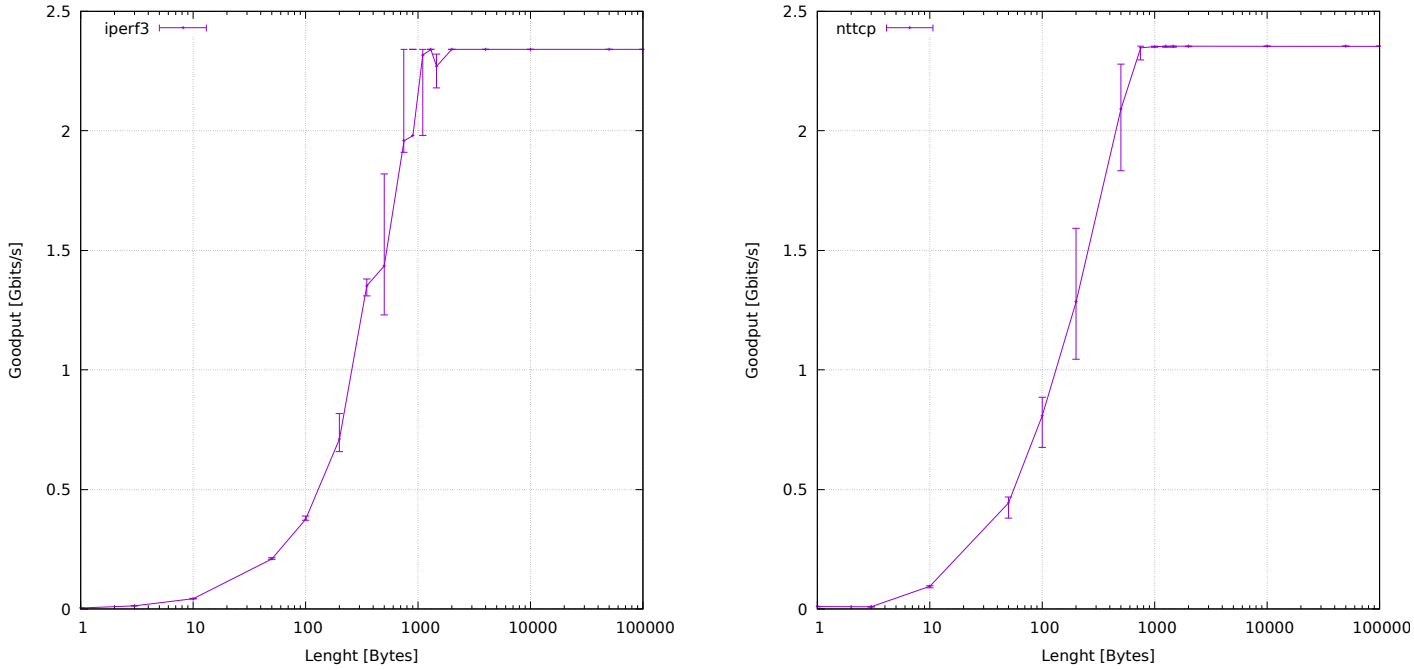
(a) Goodput Nttcp vs Iperf3

(b) %CPU Nttcp vs Iperf3

Figure 1: Comparison - TSO off

In Figure 1a, the goodput obtained from Nttcp and Iperf3 is shown. In case of Nttcp, we observed that the variance around the mean tends to be higher, so each sample is the average over 20 attempts, while in iperf3 is only 10.
Is possible to notice that the two tools behalves differently: in particular the goodput measured with Nttcp is always higher than Iperf3 until a proper value of L is reached (**750B in this case**). After that both exhibit a asymptotic behavior, where the difference between the two is very slight and the green one si always over the violet one.
However the peak for Iperf3 is **2.340 Gbits/s**, while Nttcp is **2.353 Gbit/s**, approaching more closely to the previously calculated theoretical value.

2.2 Tools reliability comparison



(a) Iperf3 (b) Nttcp

Figure 2: Sample variance - TSO off

Without knowing the implementation of both tools, it is not easy to understand which one is more reliable. However we can make some considerations by plotting the **maxGoodput** and the **minGoodput** registered when values have been sampled (figure 2). With Iperf3 we take 10 samples saving the **minSpeed**, **avgSpeed** and **maxSpeed**. The same for Nttcp but in this case the number of samples is 20. By performing the sum of the range of each sample and dividing it by the number of tests, we obtained that the "variance" is equal to **7.62 Mb/s** for Iperf3 and **24.08Mb/s** for Nttcp. As a consequence, the measure of reliability depends on what we want to achieve: if the goal of the experiment is to have a more stable result with less sample outliers, it is better to use Iperf3; on the contrary, if the goal of the experiment consist in reaching better speed in terms of goodput, then it is better to use nttcp, having in mind that it is always better chose a properly number of samples for each examined value of L.

2.3 How block size L impact the results?

In both tools, increasing the buffer length (**L**) necessitates an increase in the number (**N**). In Iperf3, N is implicitly calculated by the application, whereas for Nttcp, specific values were chosen to remain within the duration range of 10 seconds.
The goodput is proportional to the size of L; it is noticeable that with the same samples, nttcp has a value almost double compared to the other. A relevant observation, at the software level, the goodput value (for large L) is certainly higher than the previously calculated limit, but the hardware (**link capacity**) acts as a **bottleneck**, preventing to reach even better performances.

2.4 CPU consumption

In Figure 1b a fundamental difference in CPU utilization is shown, Nttcp uses it significantly more.

```
  PID TTY          TIME CMD
 1995 pts/0    00:00:00 nttcp
 2050 pts/0    00:00:01 nttcp
```

(a) Processes by nttcp

```
  PID TTY          TIME CMD
 2055 pts/0    00:00:25 iperf3
```

(b) Processes by Iperf3

Figure 3: Processes generated

The figure above shows the number of process generated on the client for each tool, that's help to understand why Nttcp is more aggressive on the CPU usage. There are structural differences in how the software are implemented.
At the beginning, the buffer length is very small, resulting in a high number of buffers, and consequently, the **CPU** performs **many read and write** operations in both tools. **As the length increases**, the number of buffers tends to **decrease** and the **load** with it. Both tend to influence the CPU similarly once high values of L are reached. As indicated by the goodput, Nttcp operates faster, leading the CPU to free up earlier, while Iperf3 exhibits a slight delay for the same buffer sizes. However, in both plots, a very similar slope can be observed once saturation is reached.

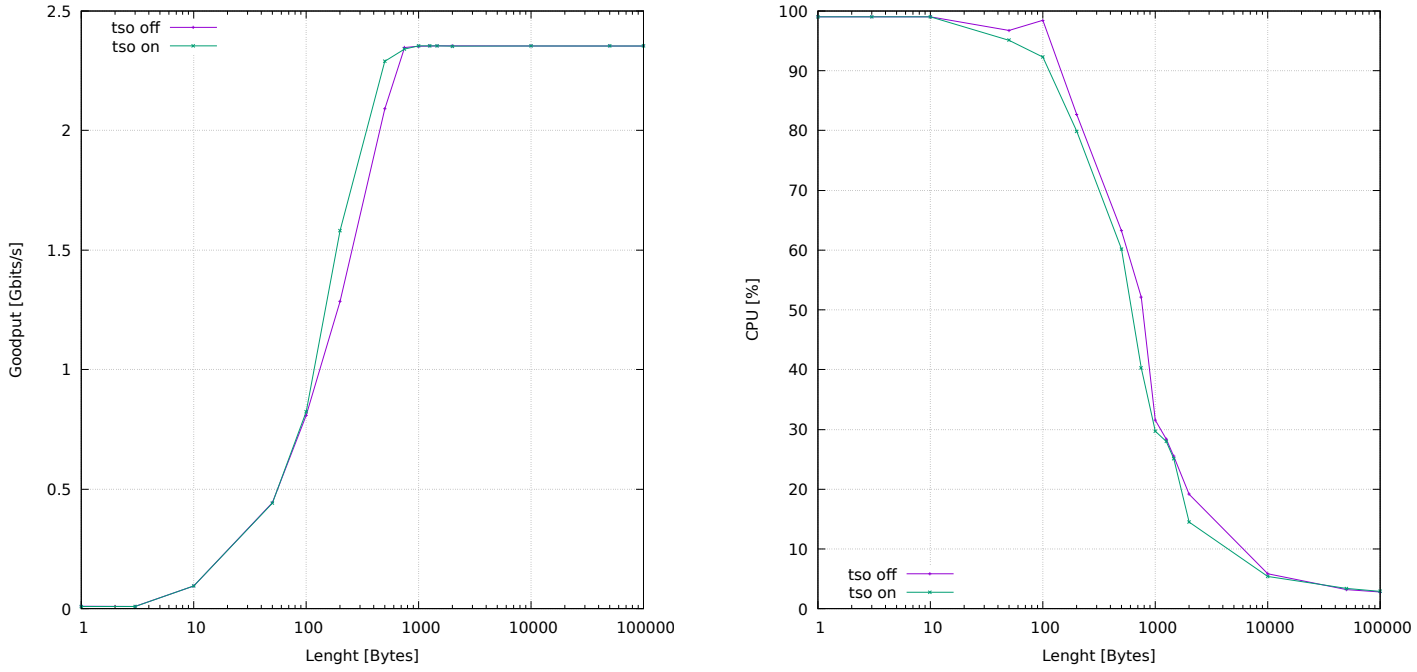
3 Offloading Capabilities

3.1 Evaluating TSO capability

The offloading capabilities are specialized tasks performed by the NIC itself instead of being performed by the host cpu that aim to reduce the cpu load of the system. The Realtek NIC used in these experiments offers a very complete set of those features. Since we used performance measurement tools that works on top of TCP, we activated only few offloading capabilities:

- **sg**: scatter-gather
- **tso**: tcp segmentation offload
- **gso**: generic segmentation offload
- **gro**: generic receive offload

The **sg** refers to a feature that allows the NIC to efficiently handle non-contiguous blocks of memory during the transmission and reception of network packets. It is a dependency of tso (cannot be enabled without sg). The **tso** is a feature designed to offload the task of segmenting large block of data into TCP segments.The **gso** (sender side) and **gro** (receiver side) are very similar to tso but for "generic" network packets. In the following plots (figure 4) it is possible to appreciate the small benefits of having **tso** enabled when running the experiment with nttcp. In particular the goodput slope slightly increases (figure 4a) while the CPU consumption is always below the case of having **tso** disabled.



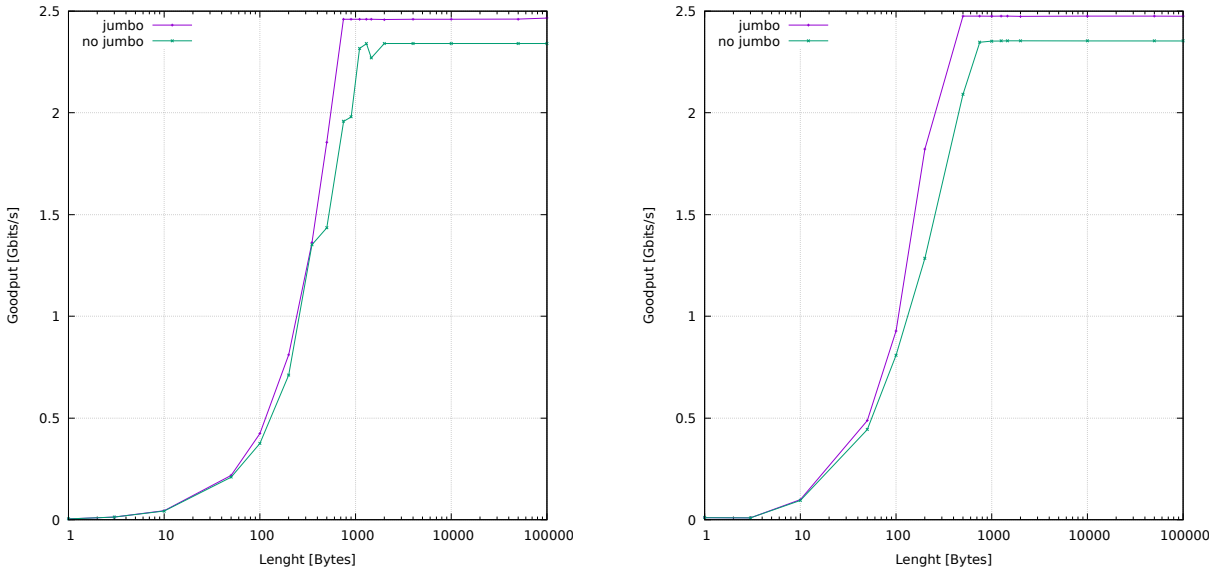
(a) Goodput

(b) CPU Consumption

Figure 4: Nttcp

3.2 Jumbo vs no Jumbo

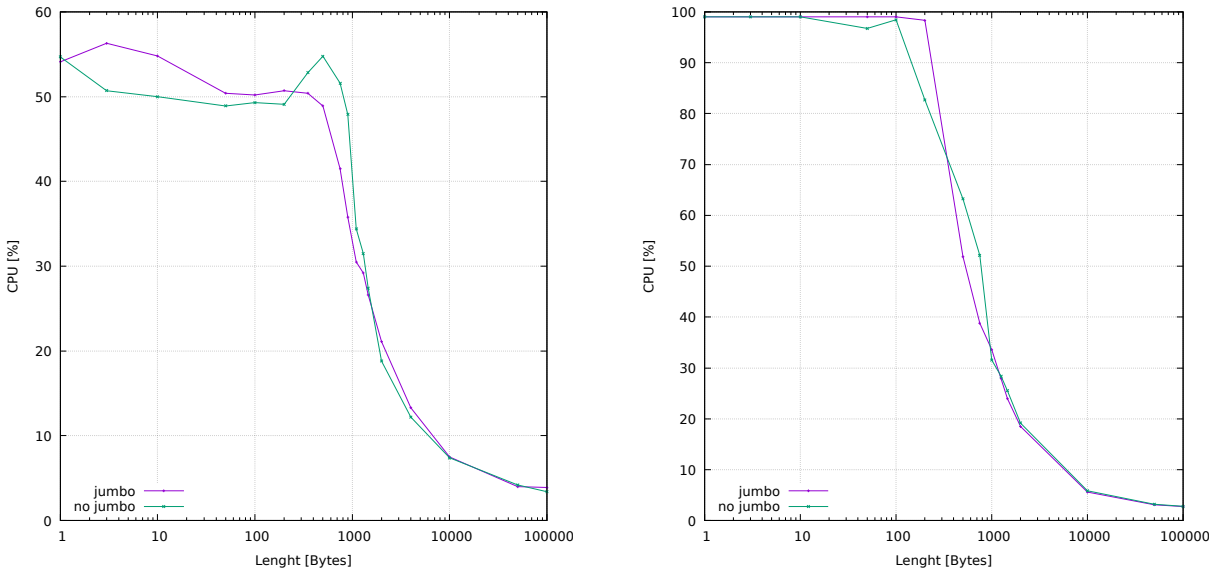
The Jumbo frame has been active setting the **MTU** to **9000B** on both hosts.



(a) Iperf3 (b) Nttcp

Figure 5: Goodput - TSO Off

Observing Figure 5, there is an increase in goodput, specifically a higher saturation point.
This value is due to the lower overhead on the transmitted data, fewer header bytes, and more payload bytes. The new theoretical limit can be calculated using the previous formula by setting the **MSS** to **8860**. In this case the final result is actually **2,478 Gbits/s**, the peak for Iperf3 is **2.46 Gbits/s**, while Nttcp is **2.475 Gbit/s**.
As in the standard case, Nttcp is closer to the theoretically calculated value. In both tools, there is a noticeable improvement in goodput. Seems that the **slope** of the graphs and the value of **L** at which saturation is reached are respectively **directly and inversely proportional to the MTU**.



(a) Iperf3 (b) Nttcp

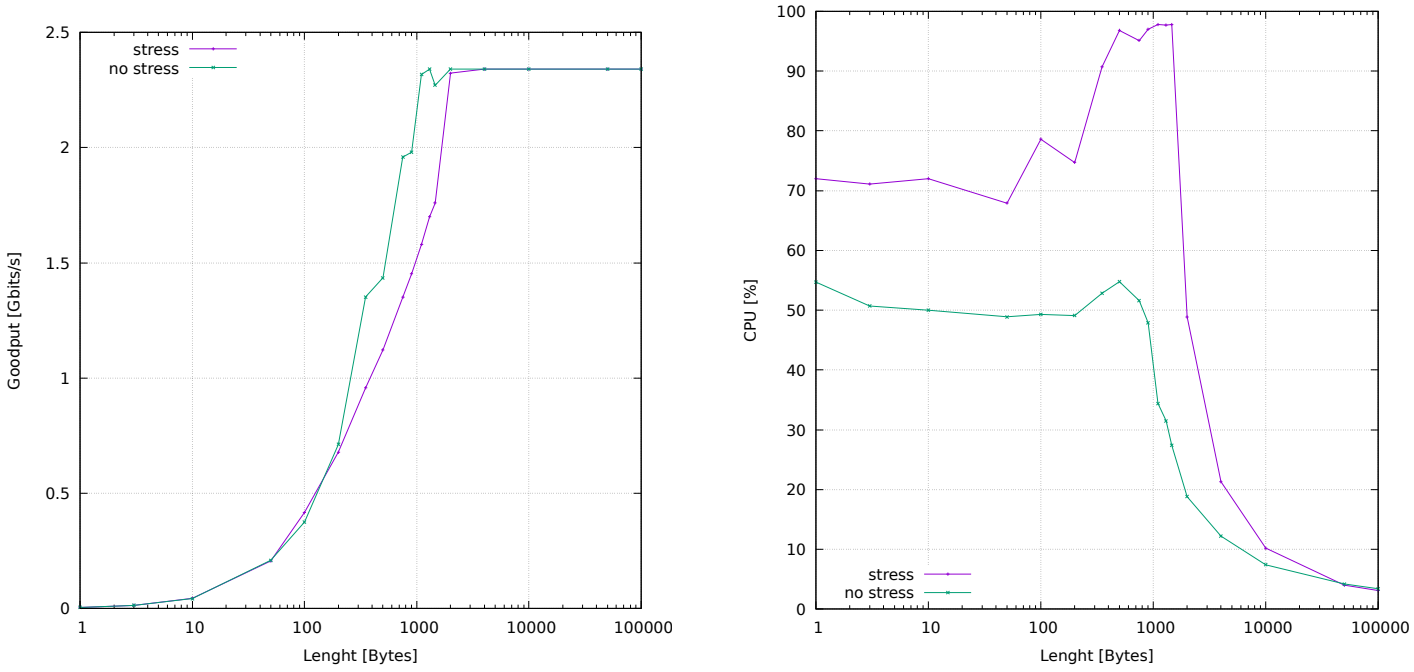
Figure 6: CPU consumption - TSO Off

Using Jumbo frames, it would be expected better CPU performance as the overhead should be reduced. However, as evidenced by Figure 6, there are no significant differences in its utilization. In the Iperf3 graph (with Jumbo frames), it can be observed that when the goodput saturates, CPU consumption decreases drastically.

4 Impact of being CPU busy

4.1 Stress test

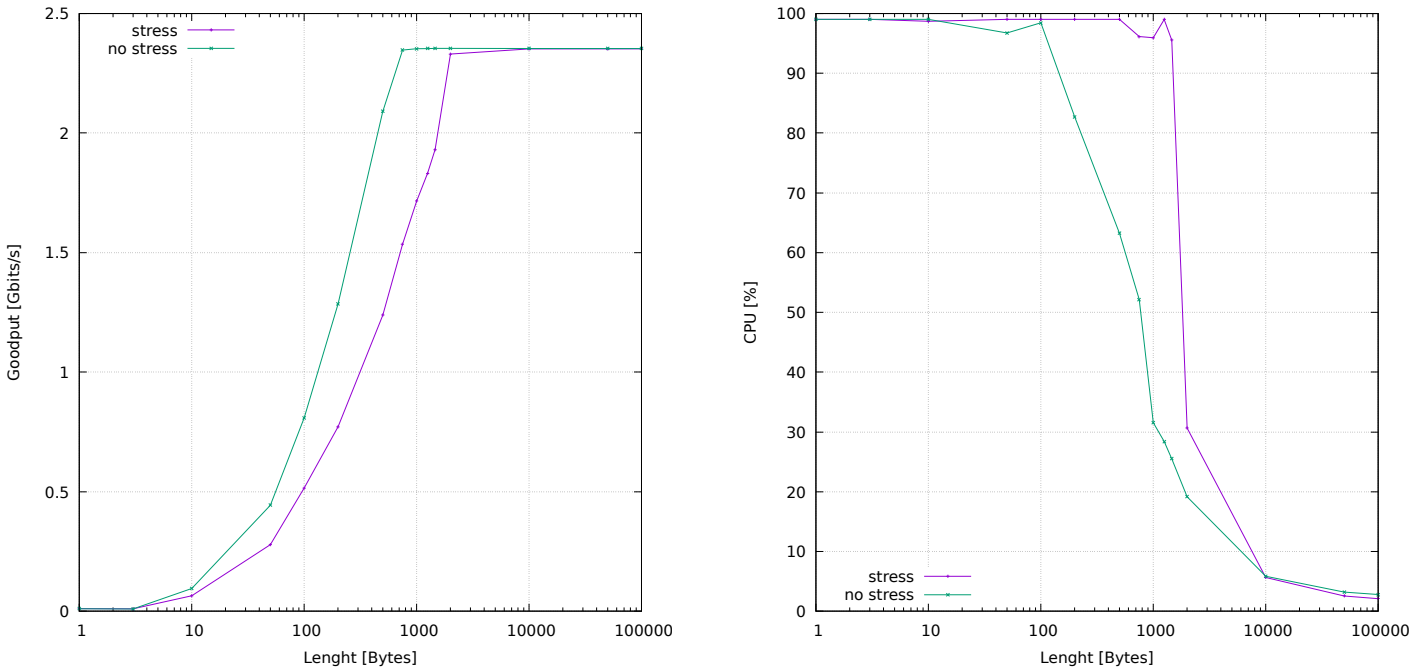
In the following experiment we perform a stress test on the client machine, that have 4 physical cores, using the command `stress -c 8`. It spawns 8 worker thread aiming to overload the cpu utilization. Every offloading capability and jumbo frames have been disabled. In figure 7 are shown the results. It is possible to appreciate the difference in running the test under an heavy workload condition. The slope of the goodput (figure 7a) decreases during the stress test and, as a consequence, the max value of goodput (**2.340 Gb/s**) is reached with greater values of **L**. This means that the cpu has not enough resources to reserve to reassembly TCP segments as fast as it can does when the cores are available.
It can be easily seen in the figure 7b. The cpu consumption under stress test is extremely higher than the one without stress, reaching its peak around 1000 Byte long packets and then decreasing as usual for very large segments.



(a) Goodput

(b) CPU Consumption

Figure 7: Iperf3



(a) Goodput

(b) CPU Consumption

Figure 8: Nttcp

In figure 8b it's possible to observe the Nttcp behavior under stress test. Unlike what happens with Iperf3, the CPU consumption is already to 100%, so stress is noticeable up to values around **1000** of L and it still tends to have a constant trend. The performance improves significantly for values close to the MTU(in this case).
Naturally, there is also an impact on the slope of the goodput (figure 8a): it is significantly slower compared to not stressed one and saturation is reached for values close to the MTU.

A Iperf3 script

```
import subprocess as sp

h2 = "10.0.9.3"

lengths = [1, 3, 10, 50, 100, 200, 350, 500, 750, 900, 1100, 1300, 1460, 2000, 4000, 10000, 50000, 100000
                                                                                                        ]
ntests = 10

with open("iperf3.txt", "w") as fl:
    for i in lengths:
        speed_values = []
        cpu_values = []
        print(f"test {i}: ")
        for j in range(ntests):
            print(f"number {j}")

            sp.run(
                f"/usr/bin/time iperf3 -c {h2} -l {i} &> data",
                shell=True,
                executable="/bin/bash",
            )
            lines = []
            with open("data", "r") as data:
                lines = data.readlines()
                data.close()
            receiver = lines[16]  # receiver

            columns = receiver.split()

            speed = float(columns[6])
            unit = columns[7]

            if unit == "Kbits/sec":
                speed *= 10**-6
            if unit == "Mbits/sec":
                speed *= 10**-3

            speed_values.append(speed)

            cpu_percentage = int(lines[19].split()[3].split("%")[0])
            cpu_values.append(cpu_percentage)

        speed_max = max(speed_values)
        avg_speed = sum(speed_values) / len(speed_values)
        speed_min = min(speed_values)
        cpu_max = max(cpu_values)
        avg_cpu = sum(cpu_values) / len(cpu_values)
        cpu_min = min(cpu_values)
        strng = (
            f"{i} {speed_min} {avg_speed} {speed_max} {cpu_min} {avg_cpu} {cpu_max}\n"
        )
        fl.write(strng)
    fl.close()
```

B Nttcp script

```
import subprocess as sp

h2 = "10.0.9.3"

# l and n properly chosen to perform reliable tests
lengths = [1, 3, 10, 50, 100, 200, 500, 750, 1000, 1250, 1460, 2000, 10000, 50000, 100000] #see 3 10 100
                                                                                            10000 n values
buffers = [
    14000000,
    13000000,
    13000000,
    12000000,
    11500000,
    7200000,
    5000000,
    4000000,
    3000000,
    2500000,
    2100000,
    1500000,
    315000,
    61000,
    30000
]
ntests = 20

with open("nttcp.txt", "w") as fout:
    for i, j in zip(lengths, buffers):
        print(f"Evaluating l={i} n={j}: ")
        speed_values = []
        cpu_values = []
        for k in range(ntests):
            print(f"test {k}")
            sp.run(
                f"/usr/bin/time nttcp -f %9b%8.2rt%8.2ct%12.4rbr -l {i} -n {j} {h2} &> nttcp_data",
                shell=True,
                executable="/bin/bash",
            )
            lines = []
            with open("nttcp_data", "r") as data:
                lines = data.readlines()
                data.close()

            speed = float(lines[1].split()[3])
            speed_values.append(speed)
            cpu_percentage = float(lines[2].split()[3].split("%")[0])
            cpu_values.append(cpu_percentage)

        speedmax = max(speed_values)
        avg_speed = sum(speed_values) / len(speed_values)
        speedmin = min(speed_values)

        cpumax = max(cpu_values)
        avg_cpu = sum(cpu_values) / len(cpu_values)
        cpumin = min(cpu_values)

        strng = (
            f"{i} {j} {speedmin} {avg_speed} {speedmax} {cpumin} {avg_cpu} {cpumax}\n"
        )
        fout.write(strng)

    fout.close()
```