WiFi Lab

Alessandro Mulassano s330263, Manuel Firrera s329226, Iacopo Epinot s333998
Politecnico di Torino

1 INTRODUCTION

The main goal of this laboratory is to perform some tests with TCP and UDP, using IEEE 802.11 and 802.3 protocols, compute the related goodput on the results, and perform some analysis. We'll delve into understanding how to assess the effectiveness of a LAN/WLAN and explore methods to evaluate it. With Commercial off the shelf (COTS) we created a setup to get the expected goodput by running some tests and comparing them afterward with our predictions.

2 ENVIRONMENT SETUP

For this lab experience, we used a set of hardware components and software that should represent the average setup of a domestic LAN.

- AP: Sercom RHG3006;
  - 802.3 interface: 3xLAN 1Gbps max;
  - 802.11 interface: Broadcom 802.11ax ready 4.8 Gbps max;
- Host1: HP Pavilion Gaming 16 2020;
  - 802.3 interface: Realtek GbE 1Gbps max;
  - 802.11 interface: Realtek RTL8822CE 802.11ac 867 Mbps max;
- Host2: Lenovo Ideapad L340 Gaming;
  - 802.3 interface: Realtek GbE 1Gbps;
  - 802.11 interface: Intel AX210 5.3 Gbps max;
- Cat5 and Cat5e UTP cables;

The following plots were generated using Wireshark, a powerful network capture and analysis tool. The built-in DHCP server assigned IPs in the following way:

- Both WiFi: 192.168.1.89 server, 1.5 client;
- Both Ethernet: 1.236 server, 1.21 client;
- Mixed mode: 1.236 server, 1.16 client;

3 TOOLS AND THEORY

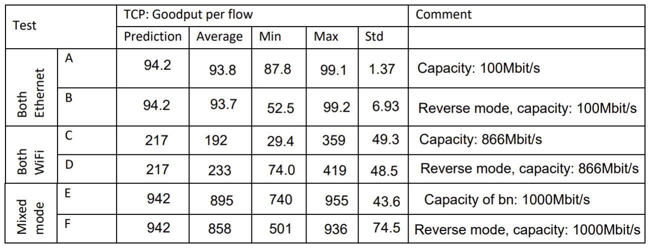
The **Goodput** measures the useful data transfer rate over a network, excluding all protocol overheads and retransmissions. It represents the amount of actual data successfully delivered from the source to the destination per unit of time, typically measured in bits per second (bps).
To obtain the expected goodput, we multiply the capacity at the physical layer by the **efficiency**, which is the ratio of useful data at the application layer over the total data. The efficiency itself is a measure of useful data over the total data sent on the link. This ratio is directly proportional to the number of encapsulations, transmission modes, and protocols used to transmit data. To get the expected goodput over TCP and UDP we used *iperf3*. It's a tool used for bandwidth benchmarking, it measures network transmission capacity between two nodes providing detailed reports, including transmission rates, jitter, and packet loss. Moreover, it performs tests in both directions simultaneously (full-duplex).

4 ANALYZED SCENARIOS

In this report we are analyzing three scenarios, either using UDP or TCP L4 protocols:

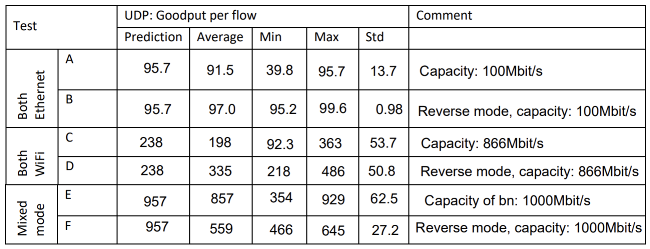
- Client and server using both ethernet, A: regular mode, B: reverse mode;
- Client and server using both WIFI, C: regular mode, D: reverse mode;
- Client using WIFI and server using ethernet, E: regular mode, F: reverse mode.

Then for every scenario, we computed 10 different captures, calculating afterward the average, the minimum, the maximum, and the standard deviation of the bitrates with a custom bash script and comparing them with the goodput predictions as can be seen in the following figure.



Test		TCP: Goodput per flow					Comment
		Prediction	Average	Min	Max	Std	
Both Ethernet	A	94.2	93.8	87.8	99.1	1.37	Capacity: 100Mbit/s
	B	94.2	93.7	52.5	99.2	6.93	Reverse mode, capacity: 100Mbit/s
Both WiFi	C	217	192	29.4	359	49.3	Capacity: 866Mbit/s
	D	217	233	74.0	419	48.5	Reverse mode, capacity: 866Mbit/s
Mixed mode	E	942	895	740	955	43.6	Capacity of bn: 1000Mbit/s
	F	942	858	501	936	74.5	Reverse mode, capacity: 1000Mbit/s

Figure 1: TCP: Goodput per flow



Test		UDP: Goodput per flow					Comment
		Prediction	Average	Min	Max	Std	
Both Ethernet	A	95.7	91.5	39.8	95.7	13.7	Capacity: 100Mbit/s
	B	95.7	97.0	95.2	99.6	0.98	Reverse mode, capacity: 100Mbit/s
Both WiFi	C	238	198	92.3	363	53.7	Capacity: 866Mbit/s
	D	238	335	218	486	50.8	Reverse mode, capacity: 866Mbit/s
Mixed mode	E	957	857	354	929	62.5	Capacity of bn: 1000Mbit/s
	F	957	559	466	645	27.2	Reverse mode, capacity: 1000Mbit/s

Figure 2: UDP: Goodput per flow

For the ethernet connections, we used a capacity link of 100 Mbit/s, so the expected goodput with TCP is 94.2, while for UDP is 95.7. For WiFi, the expected goodput is the product of half the capacity and the efficiency, but this last one results to be 0.5 for TCP and 0.55 for UDP. So by using a capacity of 866 Mbit/s the expected goodput for TCP is around 217 Mbit/s, while for UDP is around 238 Mbit/s.
Regarding the case of the mixed approach using both WIFI and ethernet, the expected goodput will be taken from the one whose capacity is lower, since it becomes a bottleneck. In our case the bottleneck was the ethernet since the maximum capacity was 1000 Mbit/s, while the WIFI was more than 1600 Mbit/s, using WIFI 6E standard.

5 RESULTS

In this section, we will analyze the results of each scenario.
The first scenario is about the client and server using ethernet, first with TCP and then with UDP. As we can see in the table (Figure 1, tests A, B), the results seem to be consistent with the expectations, both in normal mode and reverse mode, indeed the averages are very close with the predictions.

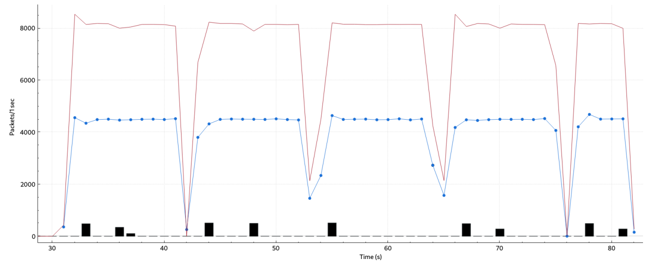


Figure 3: I/O client

In the figure 3, blue dots represent the packets sent, and the red line represents the packets received. The plot is generated using the server-captured packets, so we can easily see that the number of packets sent consists of cumulative ACKs. Consequently, this number is much lower than the actual number of received packets. Additionally, we can notice 10 seconds-wide bridges caused by the repeating tests performed by iperf3. There aren't relevant TCP errors, as we can see in the black histogram.

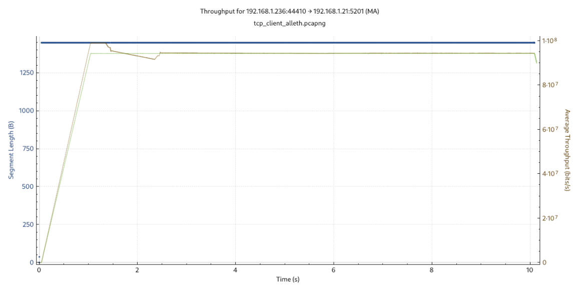
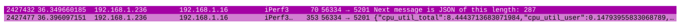


Figure 4: Throughput/Goodput

In Figure 4, the blue line represents the segment size, which is constant for the entire duration of the test. The 1500 Byte length is equal to the L2 frame size, more precisely wireshark shows us a length of 1448 Bytes because $MSS = 1500 - 20(IP) - 20(TCP) - 12 = 1460 Bytes$ and 12 bytes are options analyzed separately by the software. The throughput and goodput exhibits a gradual increase, reaching a peak before experiencing a slight decrease and stabilizing. The initial behaviour can be attributed to congestion control techniques, the slow start in which the congestion windows grow exponentially, while the fluctuations are a result of retransmissions and lost packets.
Regarding the case in which we used UDP, Figure 2 tests A and B, most of the values were similar to the predictions, but a case diverged: the minimum in the normal mode. This can be caused by potential packet loss due to full buffers or other external factors. UDP uses a "Best effort" approach, so packets can be lost without the senders or receivers being notified, as UDP does not include any ARQ mechanisms. Additionally, UDP packets can arrive out of order. Unlike TCP, UDP does not have a mechanism to reorder packets, which can cause problems if the order of data is important.
The Wireshark-generated plots did not reveal any significant anomalies. However, it is noteworthy that the *iperf3* output highlighted substantial loss due to a capacity mismatch between the peers. This discrepancy arose from the Cat5 cable, which is rated for a maximum bandwidth of 100 Mbps, while the 5e cable is capable of handling ten times more.
Consequently, the loss amounts to appoximately 90%.

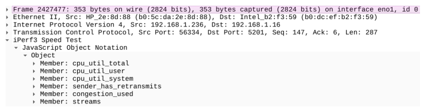
```
Interval       Transfer     Bitrate        Lost/Total
0.00-10.00 s 1.11 GBytes 955 Mbits/s  0/824790 (0%) send
0.00-10.26 s 115 MBytes 93.9 Mbits/s  741680/824782 (90%) rec
```

Iperf3 is able to track UDP datagrams by appending a small trailer containing a sequence number, this can be seen in the packet captures, which are marked with the *iperf3* label in the protocol column.



```
2427432 36.349660185  192.168.1.236         192.168.1.16          iPerf3      70 56334 → 5201 Next message is JSON of this length: 287
2427477 36.396097151  192.168.1.236         192.168.1.16          iPerf3…    353 56334 → 5201 {"cpu_util_total":8.4443713683071984,"cpu_util_user":0.14793955833068789,…
```

Figure 5: JSON packet



```
▸ Frame 2427477: 353 bytes on wire (2824 bits), 353 bytes captured (2824 bits) on interface eno1, id 0
▸ Ethernet II, Src: HP_2e:8d:88 (b0:5c:da:2e:8d:88), Dst: Intel_b2:f3:59 (b0:dc:ef:b2:f3:59)
▸ Internet Protocol Version 4, Src: 192.168.1.236, Dst: 192.168.1.16
▸ Transmission Control Protocol, Src Port: 56334, Dst Port: 5201, Seq: 147, Ack: 6, Len: 287
▾ iPerf3 Speed Test
  ▾ JavaScript Object Notation
    ▾ Object
      ▸ Member: cpu_util_total
      ▸ Member: cpu_util_user
      ▸ Member: cpu_util_system
      ▸ Member: sender_has_retransmits
      ▸ Member: congestion_used
      ▸ Member: streams
```

Figure 6: JSON utility info

Three times for each test session, the client and the server notify each other about the impending transmission of a JSON object containing comprehensive statistics related to the CPU, retransmissions, congestion, and other relevant metrics. Subsequently, the sender closes the connection after the test.
The second scenario, C and D, is about the client and server using both the 802.11 protocol. We can notice that the results differ slightly between the server side and the client side in TCP mode. This is because the client side sends packets at almost the full capacity of the link, while the server side shows the bitrate at the application level, so the expected bitrate is lower because some packets are still being processed.

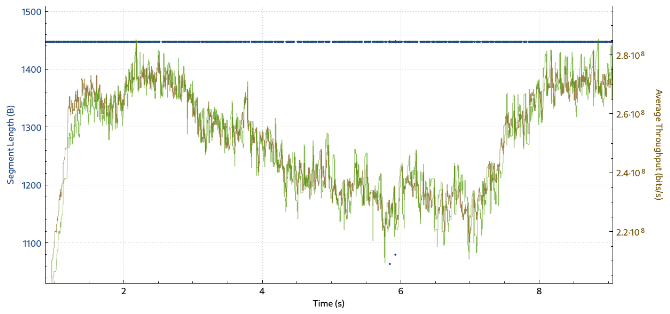


Figure 7: Throughput TCP

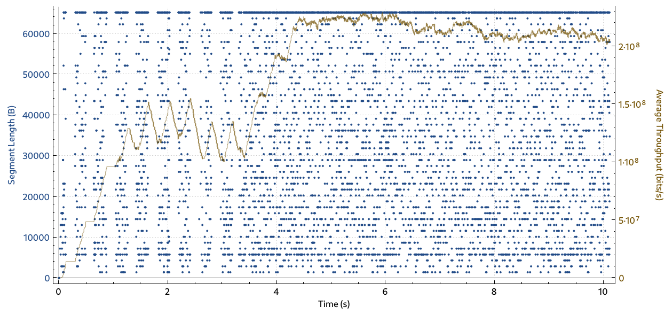


Figure 8: Throughput TCP reverse

In the graph, Figure 7, the packet length holds steady at 1448B while the throughput varies in tandem with the goodput. Notably, there is a marked decrease from 2s to 8s, suggesting that the throughput and goodput may have been approaching the packet length around 2s, indicating a probable onset of congestion and triggering TCP's congestion control.
In the reverse mode depicted in the second graph, it is apparent that the segment length exhibited fluctuations from packet to packet. This anomaly is likely attributable to interference or suboptimal signal quality.
We also noticed an abnormal segment length of 60,000 Bytes. This mechanism is called "offloading" and it is caused by some software configurations in order to delegate the fragmentation task to the NIC.

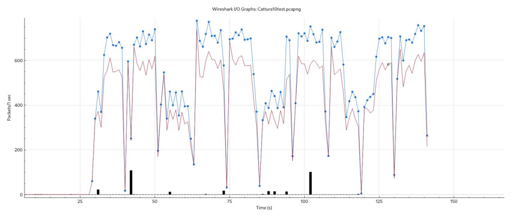


Figure 9: I/O TCP

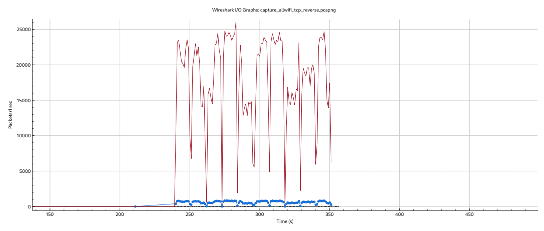


Figure 10: I/O TCP reverse

The client's perspective requires taking into account both graphs. The initial graph illustrates the normal mode, while the subsequent one presents the reverse mode. In the first graph, it is evident that, as previously noted, the cumulative acknowledgment received by the server results in the red line, and is lower than the number of packets sent, represented by the blue line. The downward spikes correspond to the conclusion of one test and the commencement of the next one or a possible loss of packets as the TCP error bar rises. The low number of ACKs in 10 could be caused by power saving mode. When the client awakes it receives the frames and then sends a cumulative ACK, thus lowering the number of packets sent.

In the last scenario, we observed a situation where the client was using a WiFi connection and the server was using an ethernet connection, with the latter being the bottleneck. Upon reviewing Figure 1, it is evident that the average results deviate slightly from the predicted values, showing approximately 5-10% less throughput. Additionally, the standard deviation for the reverse mode is notably high, as illustrated in Figure 12. This indicates a highly variable throughput and goodput, along with significant instability and peaks, characterized by segment lengths of up to 80,000 bytes. In contrast, Figure 11 depicts a more stable connection with consistent segment sizes and a relatively uniform throughput.

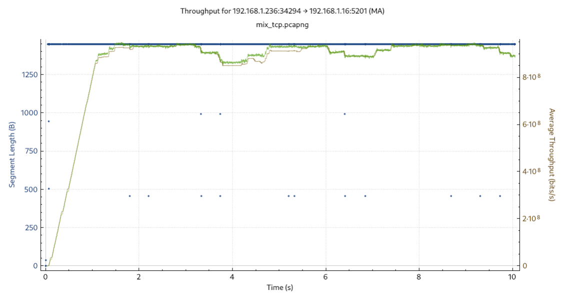


Figure 11: Thrughput mixed TCP normal

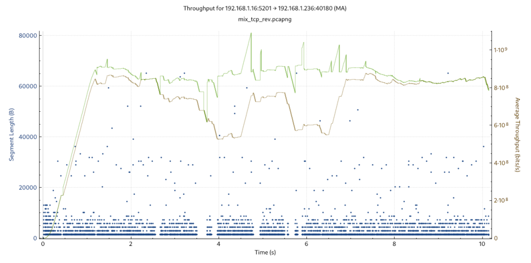


Figure 12: Thrughput mixed TCP reverse

Something else that is interesting to see is the Figure 13 that represents the progressive sequence number, and as expected it's a linear function, since with TCP we don't have packet loss.

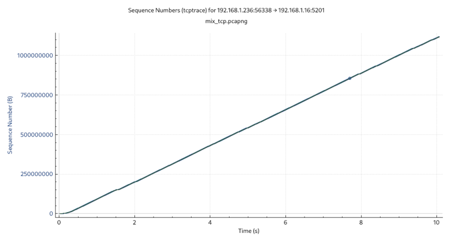


Figure 13: Sequence number TCP normal

In contrast to UDP, as shown in 2, the average in reverse mode is only 58% of what was expected. This is an intriguing case due to the differing conditions during data collection. Other devices were connected to the same WIFI, operating on the same frequencies. Additionally, the client was located in another room, with a door and wall obstructing the signal and significantly reducing the measured goodput by *iperf3*. The comparison of packets/s received in normal and reverse mode revealed that in the normal mode, we had an average of 80000 packets/s (with an average goodput of 857), while in reverse mode, the average was 50000 packets/s (with an average goodput of 559).

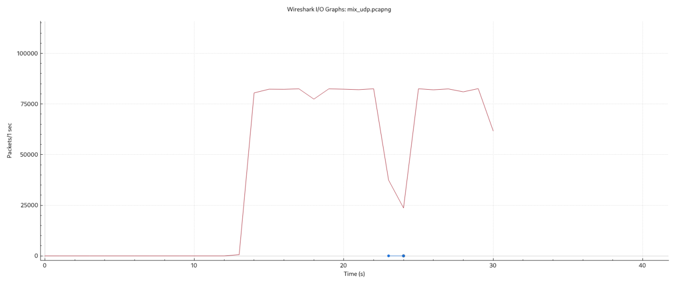


Figure 14: I/O mixed UDP normal

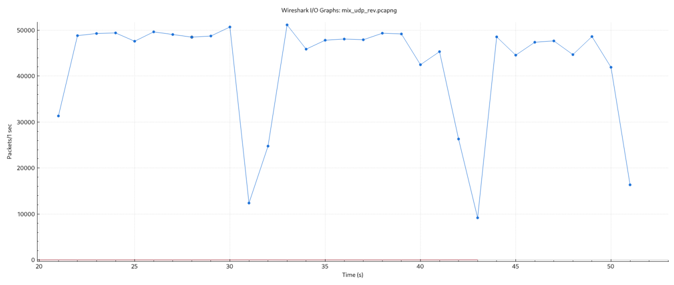


Figure 15: I/O mixed UDP reverse

6 CONCLUSIONS

The experiments show how the transmissions can be influenced by numerous factors in the network stack, from the cables to the elctromagnetic environment in the case of the PHY layer to software limitations and data overhead for the upper layers. Moreover, UDP showed how raw udp data transfer is affected by bottlenecks in the links. Impressive Wifi performances can be achieved surclassing cabled connections with the right hardware.

7 APPENDIX

Here is the script used in the experiments to calculate the average bitrate, minimum, maximum and standard deviation:

```
    #!/bin/bash

# Check argument
if [ "$#" -ne 1 ]; then
    echo "Uso: $0 nome_file.txt"
    exit 1
fi

input_file="$1"
output_file="${input_file%.txt}_analysis.txt"

# Extract bitrate from input file
bitrates=$(grep -oP '\d+(\.\d+)?(?= Mbits/sec)' "$input_file")

min_bitrate=$(echo "$bitrates" | sort -n | head -n 1)
max_bitrate=$(echo "$bitrates" | sort -n | tail -n 1)
mean_bitrate=$(echo "$bitrates" | awk '{sum+=$1} END {print sum/NR}'

# Standard deviation
std_dev_bitrate=$(echo "$bitrates" | awk '
{
    sum += $1;
    sumsq += $1 * $1;
}
END {
    mean = sum / NR;
    variance = sumsq / NR - mean * mean;
    stddev = sqrt(variance);
    print stddev;
}
')

mean_of_means=$(echo "$bitrates" | awk '
{
    sum += $1;
    count++;
    if (count == 10) {
        meansum += sum / count;
        mean_count++;
        sum = 0;
        count = 0;
    }
}
END {
    if (mean_count > 0) {
        print meansum / mean_count;
    } else {
        print "N/A";
    }
}
')

{
    echo "Minimum Bitrate: $min_bitrate Mbits/sec"
    echo "Maximum Bitrate: $max_bitrate Mbits/sec"
    echo "Average Bitrate: $mean_bitrate Mbits/sec"
   echo "Standard Deviation: $std_dev_bitrate Mbits/sec"
    echo "Mean of means: $mean_of_means Mbits/sec"
} > "$output_file"

echo "Analysis complete. Saving results in $output_file"
```