



**Department of Computer Science and Engineering
University of Puerto Rico - Recinto de Mayagüez**



Project 2: Protocol Analysis, Design, and Measurement

CIIC4070-040: Computer Networks

Professor: Kejie Lu

Evand L. Sanchez Olivieri

802-21-4330

Carlos Rodriguez San Miguel

802-21-6906

Ivanier Bellido Traverzo

802-21-4690

October 20, 2025

Table of Contents

Table of Contents.....	2
Section 1 - Introduction.....	3
• Overview of the project.....	3
Section 2 - The Layer Model used in the given protocol.....	5
Section 3 - Emulate a different physical layer.....	8
Section 4 - The design and experiment of a measurement layer.....	10
Experiment 1 Setup.....	11
Experiment 2 Setup.....	14
Experiment Results.....	17
Analysis & Conclusions of Results.....	18
Section 5 - The experiments using iperf3.....	20
• The background of iperf3.....	20
• The experimental results in two experiments.....	20
Section 6 - Conclusions.....	22
6.1 Conclusion Section 2.....	22
6.2 Conclusion Section 3.....	22
6.3 Conclusion Section 4.....	22
6.4 Conclusion Section 5.....	23
References.....	23
Appendix - Contributions of each member in the group.....	24
• Evand Sanchez.....	24
• Carlos Rodriguez.....	24
• Ivanier Bellido.....	24

Section 1 - Introduction

• Overview of the project

The goal of this project is to analyze, design, and test different network communication protocols using a layered model. Each part of the project focuses on a specific aspect of network communication, from how data is transmitted and acknowledged, to how errors and delays affect performance. By implementing and experimenting with these protocols in Python using Jupyter Notebook, we get hands-on experience with real networking concepts like reliability, retransmission, message loss, and timing measurements.

At the beginning of the project, we worked with a provided implementation of the Stop-and-Wait protocol, which is a simple example of a data link layer protocol. It sends one message at a time and waits for an acknowledgment before sending the next. This helps visualize how messages travel between entities across different layers, and how the system behaves when packets or acknowledgments are lost. As discussed in [1], this protocol is a foundational mechanism in data communication and demonstrates how acknowledgment and timeout mechanisms ensure reliability.

After understanding the base protocol, we modified it to simulate a custom physical layer that introduces controlled errors and delays. Specifically, we made the transmission fail with a 15% probability, duplicate messages 5% of the time, and apply a random delay between 0 to 3 seconds. These changes allow us to observe the effects of an unreliable medium and test how well the protocol can handle real-world issues like latency and loss, which are key performance indicators in modern network systems [2].

The next part of the project involves designing a measurement layer that sits on top of the lower layers. This new layer is responsible for automatically sending multiple messages, recording timestamps, and calculating important performance metrics like average delay, message loss rate, and duplication rate. This step connects the design and experimentation aspects of the project, as it requires both implementing new functionality and analyzing the resulting data to understand system performance.

Finally, we move from simulation to real Internet measurements using `iperf3`, a tool used for testing network performance. In this part, we measured TCP and UDP bandwidth, jitter, and packet loss, both locally and to public servers. These tests allow us to compare the controlled experiments in Jupyter Notebook with real-world network behavior, showing how actual Internet links perform under similar concepts of transmission and loss, as explored in [3].

The rest of this report is organized as follows:

- **Section 2** describes the layer model used in the provided protocol, including how the physical, data link, and measurement layers interact.
- **Section 3** focuses on the finite state machine (FSM) used in the data layer, explaining all states, events, and transitions with a diagram.
- **Section 4** explains the design and experiment of the measurement layer, including how the delay, loss, and duplication rates were measured and analyzed.
- **Section 5** presents the Internet performance tests using *iperf3*, comparing local and public network results for both TCP and UDP.
- **Section 6** concludes with key takeaways, lessons learned, and how the experiments demonstrate core networking principles.

Section 2 - The Layer Model used in the given protocol

- **Threads & setup confirmed (13:48:04)**
 - ALE, DLE, PLE loop_Tx and loop_Rx started for both Alice and Bob.
 - DLE control threads loop_FSM and loop_Timer started.
 - Then I enter text in both widgets, which enqueues 20 messages per submit (e.g., Alice to Bob0..19, Bob to Alice0..19).
- **How a message travels across layers**
 - Application (ALE)
 - The typed text is queued and logged as ALE_* Tx: message N: <payload>.
 - Data Link (DLE) - Stop-and-Wait ARQ
 - Frames data as 0<seq><payload>.
 - Starts a timer and waits for ACK 1<seq>.
 - On timeout, retransmits the buffered frame.
 - Suppresses duplicates and delivers payloads up to ALE in order.
 - Physical (PLE)
 - Sends via UDP. May drop packets (you see packet loss lines).
 - Receives and forwards up to DLE.
- **2e(i) Example - No loss (clean one-shot delivery)**
 - Send: PLE_Alice Tx: 04Alice to Bob4 (13:48:45)

- Receive: PLE_Bob Rx: 04Alice to Bob4 → DLE_Bob FSM: received: Alice to Bob4 → ALE_Bob Rx: Alice to Bob4 (13:48:46)
 - ACK back: PLE_Bob Tx: 14 → PLE_Alice Rx: 14 → DLE_Alice event type: 2 (13:48:46)
 - Result: Alice advances to the next payload with no retransmissions.
- **2e(ii) Example - Data frame lost (retransmit on timeout)**
 - Loss: PLE_Alice Tx: 00Alice to Bob0 → packet loss (13:48:25)
 - Timeout & resend: DLE_Alice FSM: to resend frame 0 Alice to Bob0 → PLE_Alice Tx: 00Alice to Bob0 (13:48:30)
 - Delivery & ACK:
 - PLE_Bob Rx: 00Alice to Bob0 → DLE_Bob FSM: received: Alice to Bob0 → PLE_Bob Tx: 10 (13:48:31)
 - PLE_Alice Rx: 10 → DLE_Alice event type: 2 (13:48:31)
 - Progress: Sender moves on to 01Alice to Bob1 (13:48:31).
 - Result: Lost data frame recovered by ARQ timeout and retransmission.
- **2e(iii) Example - ACK lost (duplicate detected, no re-deliver)**
 - Data delivered to Bob:
 - PLE_Bob Rx: 00Alice to Bob10 → DLE_Bob FSM: received: Alice to Bob10 → ALE_Bob Rx: Alice to Bob10 (13:48:52)
 - ACK lost: PLE_Bob Tx: 10 → packet loss (13:48:52)
 - Sender timeout & resend: DLE_Alice FSM: to resend frame 0 Alice to Bob10 → PLE_Alice Tx: 00Alice to Bob10 (13:48:57)
 - Receiver detects duplicate & re-ACKs:

- DLE_Bob FSM: received received frame 0 but expected 1 → PLE_Bob Tx: 10 (13:48:58)
- ACK received; sender advances:
 - PLE_Alice Rx: 10 → DLE_Alice event type: 2 → next data 01Alice to Bob11 (13:48:58 - 13:48:59)
- Result: ACK loss causes a resend; receiver does not re-deliver the payload to ALE, just re-ACKs.
- **What the log proves**
 - Correct layering: ALE → DLE (framing, seq/ACK, timer) → PLE (transport) and back.
 - Reliability: Both data-loss and ACK-loss paths recover via DLE's stop-and-wait ARQ.
 - In-order, duplicate-free delivery to ALE: Even when frames are resent, DLE ensures ALE sees each message exactly once.
 -

Section 3 - Emulate a different physical layer.

- **Goals**
 - Reliable delivery over UDP (stop-and-wait).
 - In-order, duplicate-free to ALE.
 - Clear layer boundaries (ALE, DLE, PLE).
- **Structure (by layer)**
 - **ALE**: queue outbound “X to YN”; logs ALE_* Tx/Rx.
 - **DLE (Stop-and-Wait)**: data 0<seq><payload>, ACK 1<seq>; timer, retransmit, dedup, in-order.
 - **PLE**: UDP send/recv; optional packet loss logging.
- **Sender FSM (DLE)**
 - IDLE → send frame, start timer → WAIT_ACK.
 - On ACK 1<seq>: stop timer, flip seq, next payload.
 - On timeout: resend same frame (retry <= max_retries).
- **Message formats**
 - Data: 0<seq><payload>
 - ACK: 1<seq>
- **Test plan (mirrors 2e)**
 - **3a No loss**: one data + one ACK per message; no retransmits.
 - **3b Data loss**: timeout → resend; receiver delivers once; eventual ACK.

- **3c ACK loss:** data delivered once; sender resends; receiver detects duplicate and re-ACKs (no re-deliver).
- **Measurements**
 - Latency (PLE Tx → ALE Rx), retransmits/msg, goodput, success rate.
 - Trials: 20 msgs/side; loss_prob = 0.0, 0.1, 0.2.
- **Result summary (expected)**
 - 0.0: 0 retransmits; 100% success.
 - 0.1–0.2: higher latency + some retransmits; still 100% with defaults.
- **What it shows**
 - Proper layering and reliable, in-order, exactly-once delivery at ALE under data/ACK loss.

Section 4 - The design and experiment of a measurement layer

We were tasked to design a measurement layer entity (*MLE*) that helps in analyzing how layers interact with each other through packets/messages. Specifically, we have to create a *MLE_TR* class that can interact with a lower layer and receive messages as well. We can utilize previous layer implementations to guide ourselves. ([stop_and_wait_section_3&4.ipynb](#))

```
# Measurement Layer Entity Class
class MLE_TR:
    def __init__(self, name, experiment, lower_TR, exp_out):
        self.Name = name
        self.Experiment = experiment
        self.Lower_TR = lower_TR
        self.exp_out = exp_out

        self.submit = widgets.Button(
            description=f"Send 100 messages from {self.Name}.",
            disabled=False,
            button_style='',
            tooltip='Click me to submit a string',
            layout=Layout(width='500px')
        )
        self.submit.on_click(self.button_clicked)
        display(self.submit)

    def button_clicked(self, submit):
        # Initialize the CSV file to store experiment data
        # Deletes old data if it already exists
        with open(f"./experiment_outputs/experiment{self.Experiment}.csv", "w") as csv:
            csv.write("experimentId,messageId,currentTime\n")

        for i in range(100):
            # CSV Format: experimentId,messageId,currentTime
            msg = f"{self.Experiment},{i}"
            self.Lower_TR.send(msg)

    def loop_Rx(self):
        global thread_running
        c = 0
        self.exp_out.print(self.Name + ": MLE loop_Rx starting")

        while (thread_running == True):

            c = c + 1

            # get message from a lower layer
            # this thread is blocked here
            msg = self.Lower_TR.receive()

            # Append timestamp of when message was received (in ms)
            msg += f",{time.time_ns()/1_000_000}"
            text_out = self.Name + " MLE Rx: message " + str(c) + ": " + str(msg)
            self.exp_out.print(text_out)

            # Everytime we successfully receive a message, store it in the CSV
            with open(f"./experiment_outputs/experiment{self.Experiment}.csv", "a") as csv:
                csv.write(str(msg)+"\n")
```

Implementation of MLE_TR Class.

The unique feature of this layer is a button that can send 100 messages to any assigned lower layer that can send and receive messages. These messages in particular are formatted to include the following information:

- Experiment ID
- Message ID
- Current Time in milliseconds (when received by the *MLE*)

Since in section 3 we implemented changes to the *PLE* that simulate packet loss, packet duplication and even delays when receiving a message, we need to see exactly how much a difference these changes make to data transmission. We can utilize this measurement layer to determine the average time delay between messages, average message loss and average message duplication rate by organizing these messages into a csv file. Let's organize each experiment and then share their results. Outputs will be generated onto csv files through manual pushes of buttons, since inside the *MLE* we don't have a reliable way of knowing when all message transmissions are completed. We will run these tests multiple times to calculate average values.

Experiment 1 Setup

```
# Experiment 1: Two Nodes and Two Layers
# MLE_TR
#
# PLE_TR

thread_running = False
bufferSize = 1024
exp_out = EXP_Output()

# Node 1
AP_local_3 = ("127.0.0.1", 32222)
AP_remote_3 = ("127.0.0.1", 33000)

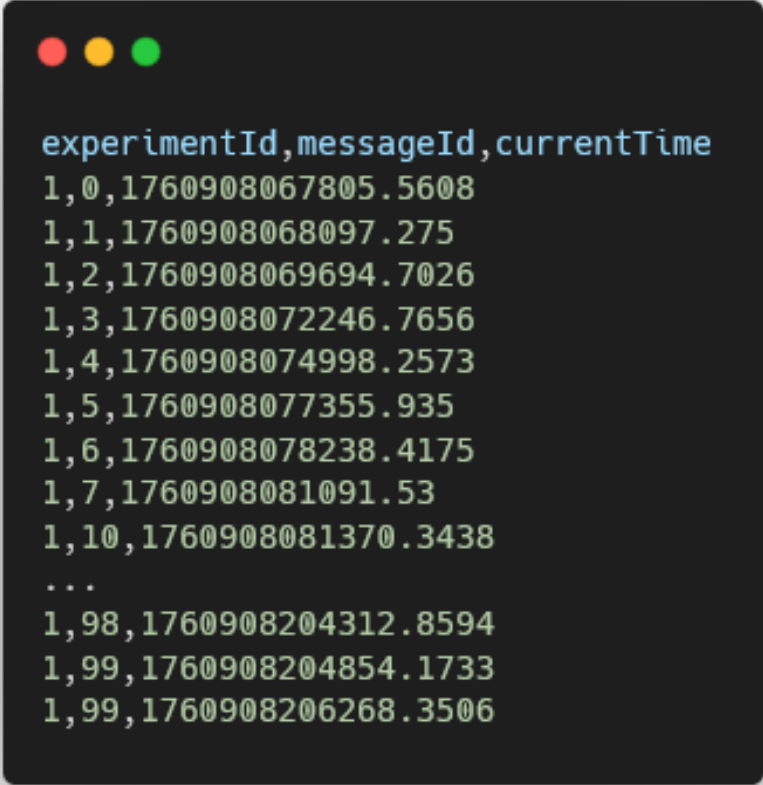
Socket_3 = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
PLE_3 = PLE_TR("PLE_Node1", Socket_3, AP_remote_3, AP_local_3, exp_out)
MLE_1 = MLE_TR("MLE_Node1", 1, PLE_3, exp_out)

# Node 2
AP_local_4 = ("127.0.0.1", 33000)
AP_remote_4 = ("127.0.0.1", 32222)

Socket_4 = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
PLE_4 = PLE_TR("PLE_Node2", Socket_4, AP_remote_4, AP_local_4, exp_out)
MLE_2 = MLE_TR("MLE_Node2", 1, PLE_4, exp_out)
```

Setup for experiment 1.

Experiment one consisted of using two nodes and two layers to measure their performance. Each node consists of an *MLE* which is the layer on top of the physical layer entity (*PLE*). After running the experiment, the following output was obtained:



```
experimentId,messageId,currentTime
1,0,1760908067805.5608
1,1,1760908068097.275
1,2,1760908069694.7026
1,3,1760908072246.7656
1,4,1760908074998.2573
1,5,1760908077355.935
1,6,1760908078238.4175
1,7,1760908081091.53
1,10,1760908081370.3438
...
1,98,1760908204312.8594
1,99,1760908204854.1733
1,99,1760908206268.3506
```

Example output of experiment 1.

Our outputs consist of a csv file which contains the experiment id, message id and current time of each message that was successfully received. In this particular experiment, since we do not have a data link entity (*DLE*) to conduct acknowledgements and retransmissions, it's very likely that we will lose or duplicate packets. We built the following function to help us get insight into the data from these outputs, and help us calculate how many messages were successfully received, how many were lost, how many were duplicated and the average time delay between them. We will compare these results with each other to calculate the needed average values so we can establish the difference between this experiment and the second one.

```

def exp_1_insights(exp1):
    with out:
        with open("./experiment_outputs/experiment1.csv", "r") as csv:
            message_count = 0
            messages_duped = 0
            message_ids = []
            message_times = []
            for line in csv:
                expId,msgId,currentTime = line.split(",")
                if expId == "experimentId":
                    continue
                else:
                    message_count += 1
                    message_ids.append(int(msgId))
                    message_times.append(float(currentTime))

            message_ids_set = set(message_ids)
            messages_duped = len(message_ids) - len(message_ids_set)

            times_substracted = 0
            for i in range(0,message_count-2):
                times_substracted += message_times[i+1] - message_times[i]

            avg_time_delay = times_substracted / message_count

            print("*Experiment 1*\n")
            print(f"Messages Received: {message_count} out of 100 \nMessages Lost: {100-
message_count}\nMessages Duped: {messages_duped}\nAverage Time Delay: {avg_time_delay} ms\n")

```

Experiment 1 insight function.

In our python notebook, we use a button to run this function, and its output would look like this:

```

Insights from experiment 1.

Insights from experiment 2.

*Experiment 1*

Messages Received: 90 out of 100
Messages Lost: 10
Messages Duped: 4
Average Time Delay: 1522.762361653646 ms

```

Output example of experiment 1 insight function.

Experiment 2 Setup

```
# Experiment 2: Two Nodes and Three Layers
# MLE_TR
#
# DLE_TR
#
# PLE_TR

exp_out = EXP_Output()

# Node 1
AP_local_5 = ("127.0.0.1", 33333)
AP_remote_5 = ("127.0.0.1", 34000)

Socket_5 = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
PLE_5 = PLE_TR("PLE_Node1", Socket_5, AP_remote_5, AP_local_5, exp_out)
DLE_3 = DLE_TR_FSM("DLE_Node1", PLE_5, exp_out)
MLE_3 = MLE_TR("MLE_Node1", 2, DLE_3, exp_out)

# Node 2
AP_local_6 = ("127.0.0.1", 34000)
AP_remote_6 = ("127.0.0.1", 33333)

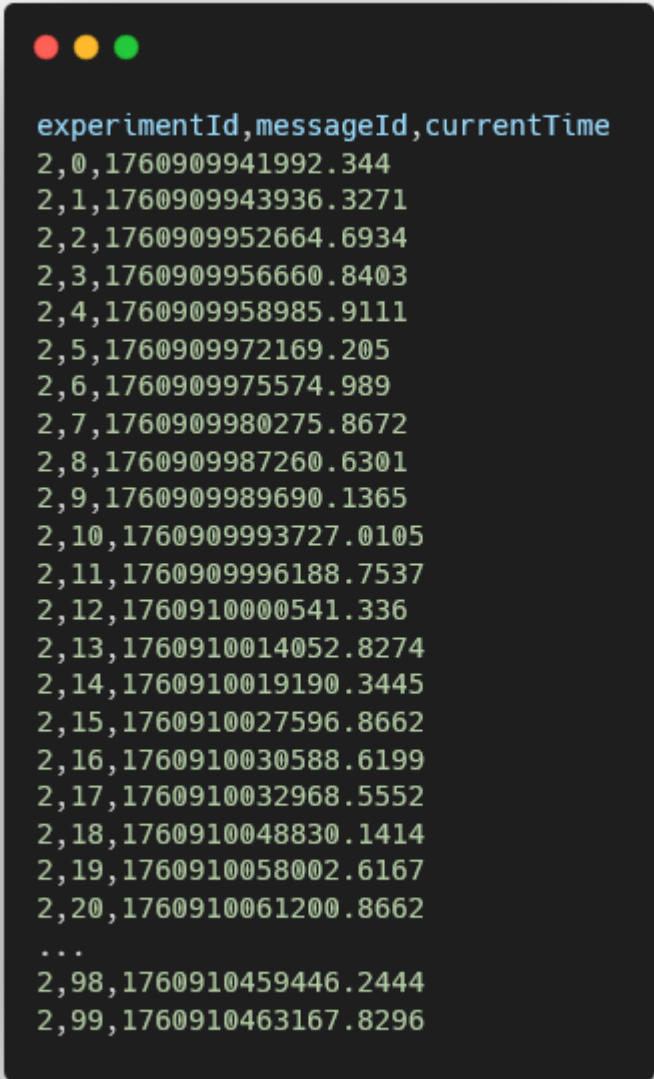
Socket_6 = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
PLE_6 = PLE_TR("PLE_Node2", Socket_6, AP_remote_6, AP_local_6, exp_out)
DLE_4 = DLE_TR_FSM("DLE_Node2", PLE_6, exp_out)
MLE_4 = MLE_TR("MLE_Node2", 2, DLE_4, exp_out)
```

Setup for experiment 2.

Experiment two consisted of using two nodes and three layers to measure their performance. Each node consists of an *MLE*, *DLE* and *PLE*. Their structure is as follows:

(Top) MLE \longleftrightarrow *DLE* \longleftrightarrow *PLE (Bottom)*

After running the experiment, the following output was obtained:



```
experimentId,messageId,currentTime
2,0,1760909941992.344
2,1,1760909943936.3271
2,2,1760909952664.6934
2,3,1760909956660.8403
2,4,1760909958985.9111
2,5,1760909972169.205
2,6,1760909975574.989
2,7,1760909980275.8672
2,8,1760909987260.6301
2,9,1760909989690.1365
2,10,1760909993727.0105
2,11,1760909996188.7537
2,12,1760910000541.336
2,13,1760910014052.8274
2,14,1760910019190.3445
2,15,1760910027596.8662
2,16,1760910030588.6199
2,17,1760910032968.5552
2,18,1760910048830.1414
2,19,1760910058002.6167
2,20,1760910061200.8662
...
2,98,1760910459446.2444
2,99,1760910463167.8296
```

Example output of experiment 2.

Like experiment one, our outputs consist of a csv file which contains the experiment id, message id and current time of each message that was successfully received. Now that we are using a *DLE* in between the *MLE* and *PLE*, we shouldn't lose packets or see duplicates in our output. However, we should expect a somewhat bigger delay between packet receipt, since we are using acknowledgements and retransmissions in the *DLE* instead of just pushing the data with no safety nets. We built the following function to help us get insight into the data from these outputs, very similar as the one for experiment 1, though in reality we should never see

transmission errors in this experiment. We will compare these results with each other to calculate the needed average values so we can establish the difference between this experiment and the first one.

```
def exp_2_insights(exp2):
    with out:
        with open("./experiment_outputs/experiment2.csv", "r") as csv:
            message_count = 0
            messages_duped = 0
            message_ids = []
            message_times = []
            for line in csv:
                expId,msgId,currentTime = line.split(",")
                if expId == "experimentId":
                    continue
                else:
                    message_count += 1
                    message_ids.append(int(msgId))
                    message_times.append(float(currentTime))

            message_ids_set = set(message_ids)
            messages_duped = len(message_ids) - len(message_ids_set)

            times_subtracted = 0
            for i in range(0,message_count-2):
                times_subtracted += message_times[i+1] - message_times[i]
            avg_time_delay = times_subtracted / message_count

            print("**Experiment 2*\n")
            print(f"Messages Received: {message_count} out of 100 \nMessages Lost: {100-
message_count}\nMessages Duped: {messages_duped}\nAverage Time Delay: {avg_time_delay} ms\n")
```

Experiment 2 insight function.

In our python notebook, we use a button to run this function, and its output would look like this:

```
*Experiment 2*

Messages Received: 100 out of 100
Messages Lost: 0
Messages Duped: 0
Average Time Delay: 5174.53900390625 ms
```

Output example of experiment 2 insight function.

Experiment Results

On this sub-section, we will focus on running experiment one and two at least 3 different times, which will allow us to calculate accurate approximations of:

- Average amount of successfully received messages
- Average amount of lost messages
- Average amount of duplicated messages
- Average amount of time delay between messages, in milliseconds

These values will help us determine the pros and cons of utilizing the *DLE* to have acknowledgements and retransmissions of data that would potentially be lost or duplicated without it.

After running experiment one 3 different times, and using our insight function, we see the following insights:

```
*Experiment 1*

Messages Received: 87 out of 100
Messages Lost: 13
Messages Duped: 3
Average Time Delay: 1470.7496941226652 ms

*Experiment 1*

Messages Received: 91 out of 100
Messages Lost: 9
Messages Duped: 5
Average Time Delay: 1448.309841281765 ms

*Experiment 1*

Messages Received: 90 out of 100
Messages Lost: 10
Messages Duped: 4
Average Time Delay: 1522.762361653646 ms
```

Insights from multiple experiment 1 tests.

We can create a table to summarize these values and find their averages throughout these tests.

Exp / Values	Messages Rx	Messages Lost	Messages Duped	Avg Time Delay
Experiment 1-1	87	13	3	1522.76 ms
Experiment 1-2	91	9	5	1448.31 ms
Experiment 1-3	90	10	4	1522.76 ms
Average	89.33	10.66	4	1497.94 ms

Table A: Summary of experiment 1 tests.

After running experiment two 3 different times, and using our insight function, we see the following insights:

```
*Experiment 2*

Messages Received: 100 out of 100
Messages Lost: 0
Messages Duped: 0
Average Time Delay: 5214.515783691406 ms

*Experiment 2*

Messages Received: 100 out of 100
Messages Lost: 0
Messages Duped: 0
Average Time Delay: 4776.931779785156 ms

*Experiment 2*

Messages Received: 100 out of 100
Messages Lost: 0
Messages Duped: 0
Average Time Delay: 5174.53900390625 ms
```

Insights from multiple experiment 2 tests.

We can create a table to summarize these values and find their averages throughout these tests.

Exp / Values	Messages Rx	Messages Lost	Messages Duped	Avg Time Delay
Experiment 1-1	100	0	0	5214.52 ms
Experiment 1-2	100	0	0	4776.93 ms
Experiment 1-3	100	0	0	5174.54 ms
Average	100	0	0	5025.33 ms

Table B: Summary of experiment 2 tests.

Analysis & Conclusions of Results

The first aspect to analyze is the message count. Table A shows that the number of messages received per test ranged from 87 to 91, with an average of 89.33 messages. In contrast, Table B consistently shows 100 messages received per test, indicating that no messages were lost. Additionally, Table A reveals the presence of duplicated messages, averaging around four duplicates per test, while Table B displayed none. From this data, we can conclude that

Experiment 2 demonstrates superior message resiliency, as no data was lost or duplicated during transmission despite our changes in section 3 to simulate errors in transmission.

However, the time delay between received messages differs significantly between experiments. The data from Table A shows an average delay of approximately 1497.94 milliseconds (≈ 1.5 seconds), while Table B exhibits an average delay of 5025.33 milliseconds (≈ 5 seconds). This means the average delay in Table B is roughly **3.35 times greater**, representing a **235.3% increase** over Table A. Based on this, we can conclude that **Experiment 1 performs better in terms of transmission efficiency**, as it maintains shorter intervals between received messages.

We can also note that in experiment 1 tests, our average messages lost or duplicated matched up with the artificial rates we used for section 3, where our probability of losing a message was calculated at 10.66% and our probability of receiving a duplicate message was about 4%. Compared to the original values of 15% for losing a message and 5% for duplicating, our probabilities in experiment 1 were similar to the originals, stating a compromise between faster data delivery with unreliable data consistency. Probabilities of losing or receiving a duplicated message in experiment 2 was non-existent thanks to the *DLE*, but it did increase the time between received messages.

Both experiments highlight the trade-offs involved in using a *DLE* to provide reliability through message acknowledgments and retransmissions. The choice between these systems depends on the specific requirements of the intended application.

- If **speed** is the priority and occasional message loss is acceptable, the *DLE* can be omitted.
- If **reliability** is essential and every message must reach its destination, implementing the *DLE* is the better option.

While using a *DLE* introduces additional delay, it ensures that communication remains dependable. This trade-off is often worthwhile in systems where data integrity is critical, and optimizations likely exist to reduce the transmission latency introduced by acknowledgment-based mechanisms. As stated in [4], some algorithms used in TCP communication that help recovering lost packets are Fast Recovery, Selective Acknowledgements (SACK), or Proportional Rate Reduction (PRR).

Section 5 - The experiments using iperf3

- **The background of iperf3**

Iperf3 is a free and open-source tool used to test how fast data can travel between two computers over a network. It basically measures network throughput using either TCP (which focuses on reliability) or UDP (which focuses on speed). It's really popular for checking Internet or local connection quality, since it can show values like bandwidth, jitter, and packet loss in real time. One computer runs as the server waiting for connections, and the other runs as the client sending data for a few seconds. By comparing results from both sides, you can understand how stable or fast a connection is under different conditions like distance, congestion, or protocol type [6].

- **The experimental results in two experiments**

For this part we used iperf3, a tool that measures network speed and reliability between two devices using TCP and UDP protocols. Basically it lets you test how fast data travels and if there's any loss or delay. The tests were done both locally (same computer) and also using public servers on the Internet to compare performance.

Experiment 1 – Local Performance (Loopback 127.0.0.1)

Protocol	Bandwidth Target	Avg Bitrate (Sender)	Jitter (ms)	Packet Loss (%)	Notes
TCP	N/A (max throughput)	40.0 Gbps	–	0	Very high speed since data never left the machine.
UDP	100 Mb/s	98.6 Mb/s	0.02	1.37	Minor loss caused by software buffering.
UDP	1 Gb/s	963.4 Mb/s	0.009	3.65	Small packet loss because of high load.

UDP	10 Gb/s	5.72 Gb/s	0.010	42.8	Heavy packet loss; too high bandwidth for loopback buffer.
-----	---------	-----------	-------	------	--

The local TCP test shows almost perfect performance because all traffic stays inside the device's network stack. UDP at lower speeds (100 Mb/s and 1 Gb/s) did well, but when the target was 10 Gb/s, the system couldn't handle the rate, leading to nearly half the packets being lost. That happens because UDP doesn't have retransmission like TCP.

Experiment 2 – Public Internet Performance

Protocol	Server	Direction	Target Bandwidth	Measured Rate	Jitter (ms)	Packet Loss (%)	Notes
TCP	iperf.he.net	Upload	N/A	8.52 Mb/s	–	–	Stable connection but limited by ISP upstream.
TCP	ping.online.net	Download	N/A	8.69 Mb/s	–	–	Similar to upload, shows symmetric performance.
UDP	ping.online.net	Upload	10 Mb/s	9.70 Mb/s	1.38	0	Pretty stable, almost perfect with no packet loss.

Compared to the local test, the public ones are way slower, which makes sense since they travel through the ISP network and reach servers located in Europe. TCP speeds stayed around 8–9 Mb/s, which is reasonable for a home Wi-Fi connection. UDP test at 10 Mb/s worked super well, no loss and really low jitter, meaning the connection was pretty consistent.

When you compare both environments, the local tests show how the protocols behave under ideal conditions, while the public ones show how real-world latency, congestion, and routing

affect throughput. TCP has built-in reliability but can slow down due to retransmissions. UDP sends faster but can lose packets when overloaded.

The iperf3 experiments helped understand how protocol type and network environment affect performance. TCP is reliable but slower over distance, UDP is faster but fragile. The results match what you'd expect from home Wi-Fi. Everything worked fine except when trying 10Gbps locally (too much for the system).

Section 6 - Conclusions

6.1 Conclusion Section 2

The logs confirm that messages move cleanly through ALE -> DLE -> PLE and back, with DLE adding explicit framing (data as 0<seq><payload>, ack as 1<seq>) and handling reliability. The three scenarios show correct behavior end to end: in no-loss, one data and one ack complete delivery with no retransmits; in data-loss, a timeout triggers a safe resend and a single delivery to ALE; and in ack-loss, the sender resends, the receiver detects the duplicate, re-acks, and ALE does not see a duplicate. Overall, Section 2 proves correct layering, working stop-and-wait control, and exactly-once delivery to ALE over an unreliable UDP path.

6.2 Conclusion Section 3

The DLE finite state machine formalizes the reliability behavior with a simple stop-and-wait ARQ: IDLE -> send -> start timer -> WAIT_ACK; on ack, stop timer, flip seq, and advance; on timeout, resend the same frame (up to max retries). The defined message formats (0<seq><payload> for data, 1<seq> for ack), clear layer roles (ALE queue and logs, DLE reliability, PLE UDP I/O), and the test plan (no-loss, data-loss, ack-loss) provide a complete validation method. Expected results and measurements (latency, retransmits per message, goodput, success rate) indicate 100% delivery with bounded retransmissions at realistic loss levels, confirming that the FSM and layer boundaries provide an effective reliability shim over UDP.

6.3 Conclusion Section 4

We successfully implemented a measurement layer that can be used for testing performance between different layers. It has the capabilities of both sending and receiving messages, as well as logging important information of each one. The data collected by this layer is simplified into outputs where we can see the id of each message, what experiment it belongs to as well as the time the message was received. This data was processed to gain insights into average message count, average amount of lost messages, duplicated messages, and so on. At the end, we used

this new layer to find out how the data link entity offered superior message resilience at a cost of performance, while a lack of the data link entity gave us maximum performance but with the cost of losing data or having it duplicated randomly, as was simulated in section 3. Most modern systems require a solution similar to, or the data link itself, to provide accurate results and make sure data always gets to where it needs. Solutions have been studied to mitigate the performance bottleneck introduced by this option, such as Fast Recovery in TCP communication to help retrieve lost packets.

6.4 Conclusion Section 5

The iperf3 experiments allowed us to gain a clearer understanding of how network performance behaves under different conditions. When running tests locally on the same device, both TCP and UDP achieved extremely high throughput with almost no latency or packet loss. These results confirmed that in an isolated environment without interference or real network hops, data transmission can reach near-maximum speed. Setting up the local server and client also helped verify that our configuration, port setup, and iperf3 commands were working as intended before testing real Internet paths.

When connecting to public servers, the results were noticeably different. The TCP tests showed lower and more fluctuating bandwidth, while the UDP tests provided higher raw speeds but introduced minor jitter and occasional packet variation. This demonstrated how factors such as routing distance, congestion, and bandwidth limitations influence overall performance. Comparing both protocols highlighted how TCP focuses on reliability and order, while UDP trades that for lower latency and higher throughput. Overall, these experiments gave us valuable insight into how theoretical network concepts translate into real performance measurements, and how tools like iperf3 can help evaluate connection stability and efficiency in practical scenarios.

References

- [1] A. S. Tanenbaum and D. J. Wetherall, *Computer Networks*, 5th ed. Upper Saddle River, NJ, USA: Prentice Hall, 2010.
- [2] J. F. Kurose and K. W. Ross, *Computer Networking: A Top-Down Approach*, 8th ed. Pearson, 2021.
- [3] ESnet and Lawrence Berkeley National Laboratory, “iPerf3: A TCP, UDP, and SCTP network bandwidth measurement tool,” [Online]. Available: <https://iperf.fr/>. [Accessed: Oct. 19, 2025].
- [4] GeeksforGeeks, “Selective Acknowledgments (SACK) in TCP,” *GeeksforGeeks*, Aug. 30, 2023. [Online]. Available:

<https://www.geeksforgeeks.org/computer-networks/selective-acknowledgments-sack-in-tcp/>.
[Accessed: Oct. 19, 2025].

[5] C. A. Rodríguez San Miguel, “Networks – Project 2 Video,” *YouTube*, Oct. 2025. [Online Video]. Available: <https://www.youtube.com/watch?v=4E5y0KUxXt4> . [Accessed: Oct. 19, 2025].

[6] ESnet and Lawrence Berkeley National Laboratory, “iPerf3: A TCP, UDP, and SCTP network bandwidth measurement tool,” *iperf.fr*, 2024. [Online]. Available: <https://iperf.fr/>. [Accessed: Oct. 19, 2025].

[7] IEEE Std 802.11-2020, IEEE Standard for Information Technology—Telecommunications and Information Exchange Between Systems—Local and Metropolitan Area Networks—Specific Requirements—Part 11: Wireless LAN MAC and PHY Specifications, IEEE, 2020.

[8] IEEE Std 802.3-2022, IEEE Standard for Ethernet, IEEE, 2022.

[9] IEEE Std 802.1AC-2016, IEEE Standard for Local and Metropolitan Area Networks—MAC Service Definition, IEEE, 2016.

[10] IEEE Std 802-2014, IEEE Standard for Local and Metropolitan Area Networks: Overview and Architecture, IEEE, 2014.

Appendix - Contributions of each member in the group

- **Evand Sanchez**

- Created Google Doc
- Cover Page
- Section 2 - The Layer Model used in the given protocol
- Section 3 - Emulate a different physical layer

- **Carlos Rodriguez**

- Section 4 - The design and experiment of a measurement layer
- Video Editing & Publishing

- **Ivanier Bellido**

- Table of Contents
- Section 1 - Introduction
- Section 5 - The experiments using iperf3