Technical Report

Title: Experimental Evaluation of MiniTelemetry: A Lightweight UDP-Based IoT Telemetry Protocol

Team :

ID	Name	
22P0285	Nader Mohamed Elfeel	
22P0124	Mohamed Ibrahim Ghoneem	
22P0128	Zakria Alaa Eisa	
22P0127	Mohamed Magdy	
22P0126	Mahmoud Nashaat	
22P0089	Mohmed Ahmed Mahdi	

1. Introduction

The rapid growth of IoT systems has increased the demand for efficient, lightweight telemetry protocols capable of operating under unreliable network conditions. Many existing solutions, such as MQTT or CoAP, introduce non-trivial overhead or depend on TCP-based transport, which may not be suitable for constrained or high-frequency telemetry scenarios.

This report presents the **design, implementation, and experimental evaluation** of **MiniTelemetry**, a custom UDP-based telemetry protocol. The protocol focuses on batching efficiency, low overhead, and resilience to packet loss, duplication, and reordering.

The goal of this evaluation is to quantify the performance and reliability trade-offs of MiniTelemetry under different network conditions and to compare batching strategies against a baseline configuration.

2. Methodology

2.1 System Architecture

The system consists of two main components:

- **Client (IoT Device Emulator):**
  - Periodically generates temperature and humidity readings.
  - Groups readings into batches.
  - Sends telemetry packets using UDP.
  - Implements heartbeat signaling and retransmission for reliability.
- **Server (Telemetry Collector):**
  - Receives packets and parses protocol headers.
  - Detects duplicate packets and sequence gaps.
  - Buffers out-of-order readings and reorders them by timestamp.
  - Logs all telemetry data and performance metrics into CSV files.

The implemented system follows a **client-server UDP telemetry framework** designed to handle multiple devices transmitting sensor readings reliably. Each client device generates temperature and humidity readings in batches and sends them to the server over UDP, with the following process:

1. **Initialization**: Each client starts by sending an `INIT` message (`TYPE=0`) to register itself with the server. The server maintains a device state, tracking the last sequence number received for each device.
2. **Data Transmission**: Sensor readings are grouped into batches (`TYPE=1`). Each packet contains a header (message type, device ID, sequence number, timestamp, batch size) and a payload (temperature, humidity, timestamp for each reading). The server parses the header and payload, detects duplicates, and flags any missing sequences. ACKs are sent immediately to acknowledge received batches, and unacknowledged packets are retransmitted by the client after a timeout.
3. **Heartbeat Mechanism**: To monitor device availability, clients send periodic heartbeat messages (`TYPE=2`). The server tracks the last heartbeat timestamp for each device, generating warnings if a device becomes unresponsive.

4. **Reordering and Logging**: Received readings are temporarily stored in a **reordering buffer** to handle out-of-order arrivals. After a short window (250 ms), readings are flushed in timestamp order to a CSV log. This ensures accurate chronological logging despite UDP's potential packet disorder.
5. **End of Transmission**: When the client finishes the test, it sends an `END` message (`TYPE=4`). The server flushes all remaining buffered readings and calculates **metrics** including bytes per report, packets received, duplicate rate, sequence gaps, and CPU time per report. These metrics are stored in a separate summary CSV.

Overall, the system ensures **reliable batch delivery**, maintains **chronological ordering**, and provides a detailed **performance report** while handling UDP's inherent unreliability using acknowledgments, retransmissions, and heartbeat monitoring.

2.2 Experimental Setup

Hardware / Software Environment:

- OS: Linux (Ubuntu-based)
- Python Version: 3.11
- Transport: UDP over loopback interface (`lo`)
- Packet Capture: `tcpdump` / Wireshark

Protocol Parameters:

- Reporting interval: 2 seconds
- Heartbeat interval: 5 seconds
- Retransmission timeout (RTO): 3 seconds
- Reordering window: 250 ms
- Test duration: 60 seconds

2.3 Experimental Scenarios

Three main scenarios were evaluated:

1. **Baseline (No Impairment):**
    a. No artificial loss or delay.

    b. Used as reference for performance and reliability.
2. **Packet Loss Scenario:**
    a. 5% packet loss introduced using Linux `netem`.
3. **Delay and Jitter Scenario:**
    a. 100 ms delay with 10 ms jitter.

Baseline Comparison:

- Batch size = 1 (single reading per packet)
- Batch size > 1 (multiple readings per packet)

2.4 Metrics Collected

The following metrics were computed and logged:

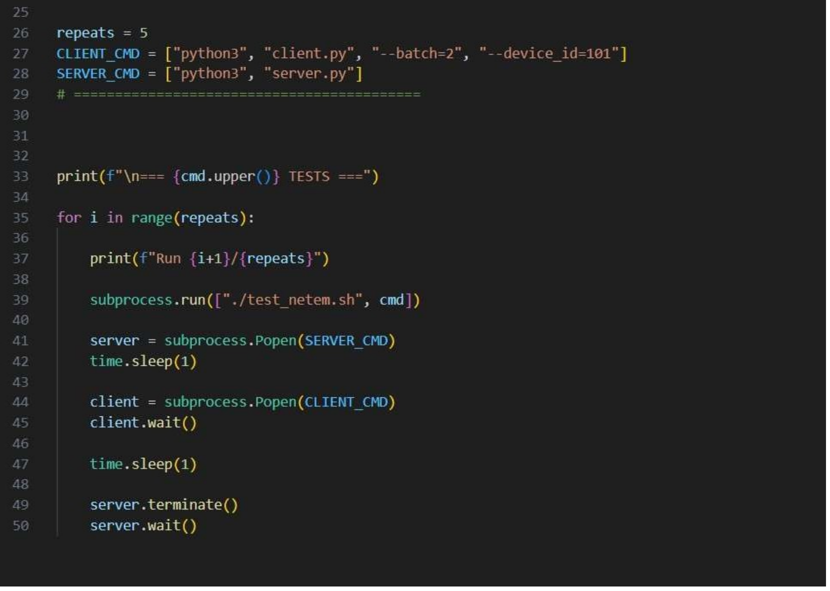
- **Bytes per Report:**
  Measures protocol overhead efficiency.
- **Packets Received:**
  Total number of UDP packets received by the server.
- **Duplicate Rate:**
  Ratio of duplicate packets to total packets.
- **Sequence Gap Count:**
  Number of missing sequence numbers detected.
- **CPU Time per Report:**
  Processing overhead on the server per telemetry reading.

All metrics were stored in CSV files for offline analysis.

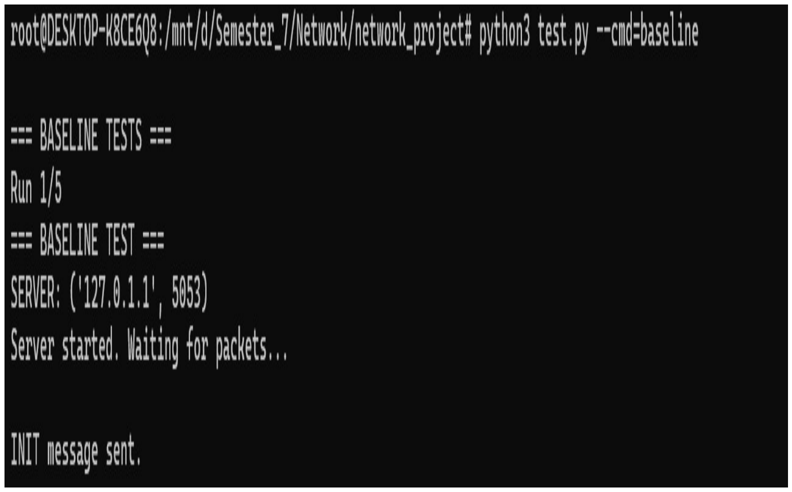
3. Results and Plots

Baseline :

Running baseline for 5 times



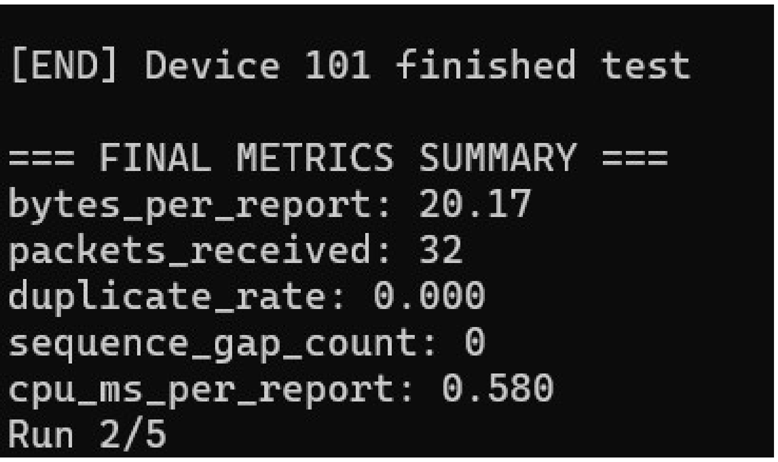
```
repeats = 5
CLIENT_CMD = ["python3", "client.py", "--batch=2", "--device_id=101"]
SERVER_CMD = ["python3", "server.py"]
# ========================================



print(f"\n=== {cmd.upper()} TESTS ===")

for i in range(repeats):

    print(f"Run {i+1}/{repeats}")

    subprocess.run(["./test_netem.sh", cmd])

    server = subprocess.Popen(SERVER_CMD)
    time.sleep(1)

    client = subprocess.Popen(CLIENT_CMD)
    client.wait()

    time.sleep(1)

    server.terminate()
    server.wait()
```



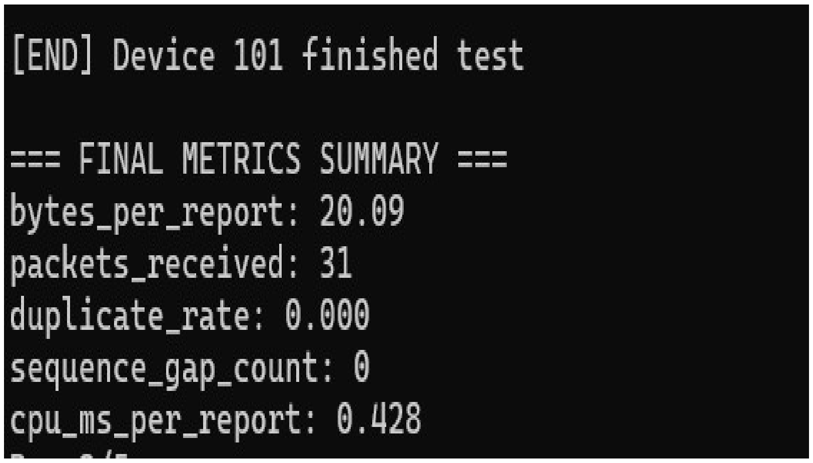
```
root@DESKTOP-K8CE6Q8:/mnt/d/Semester_7/Network/network_project# python3 test.py --cmd=baseline

=== BASELINE TESTS ===
Run 1/5
=== BASELINE TEST ===
SERVER: ('127.0.1.1', 5053)
Server started. Waiting for packets...

INIT message sent.
```

Results : (1)

(2)



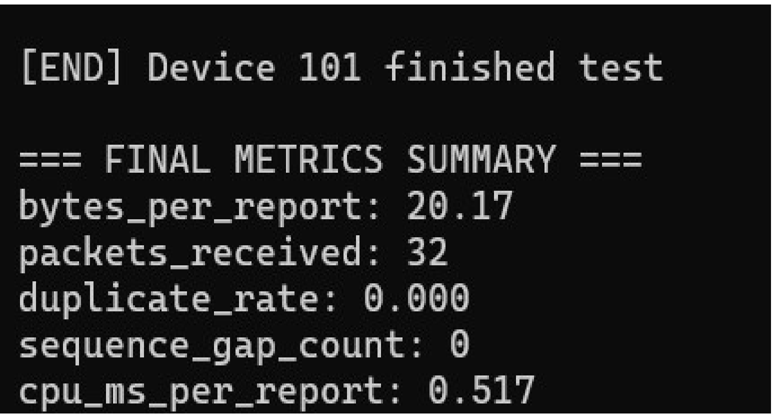
```
[END] Device 101 finished test

=== FINAL METRICS SUMMARY ===
bytes_per_report: 20.17
packets_received: 32
duplicate_rate: 0.000
sequence_gap_count: 0
cpu_ms_per_report: 0.580
Run 2/5
```



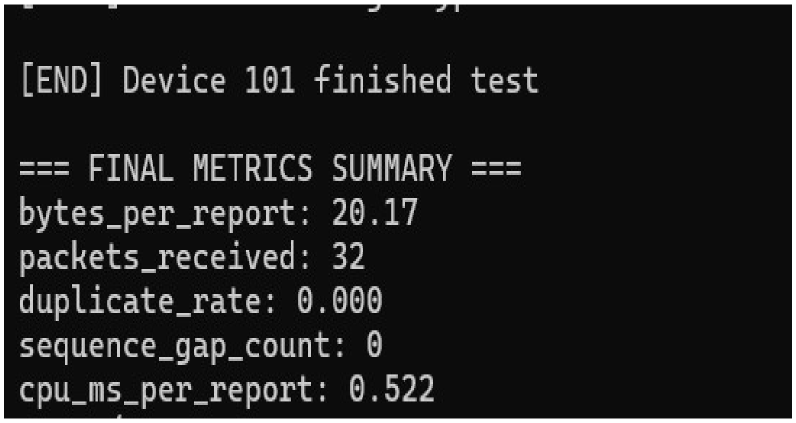
```
[END] Device 101 finished test

=== FINAL METRICS SUMMARY ===
bytes_per_report: 20.09
packets_received: 31
duplicate_rate: 0.000
sequence_gap_count: 0
cpu_ms_per_report: 0.428
```

(3)

(4)

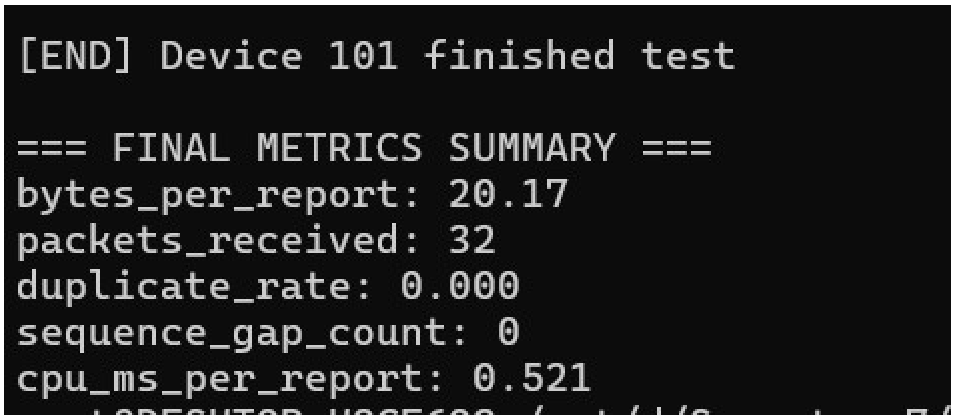


```
[END] Device 101 finished test

=== FINAL METRICS SUMMARY ===
bytes_per_report: 20.17
packets_received: 32
duplicate_rate: 0.000
sequence_gap_count: 0
cpu_ms_per_report: 0.517
```

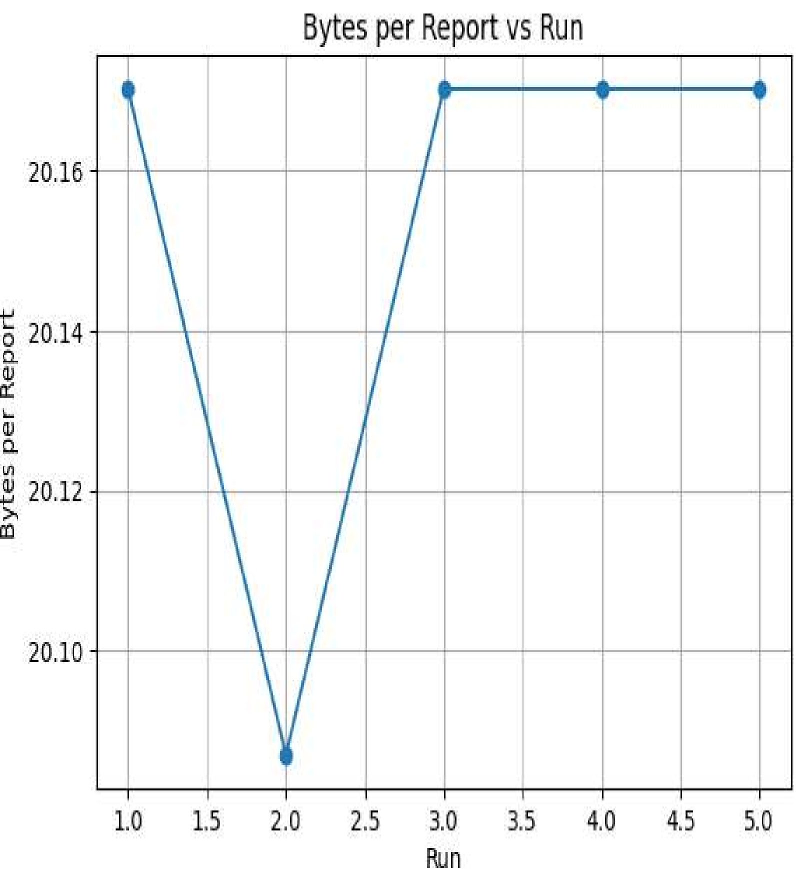


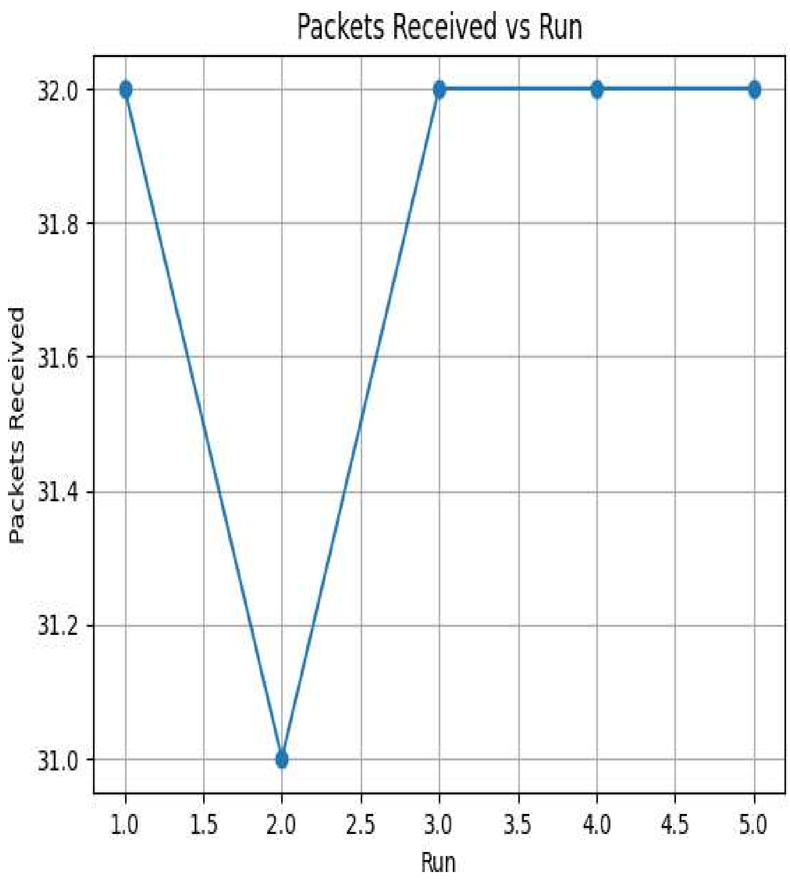
```
[END] Device 101 finished test

=== FINAL METRICS SUMMARY ===
bytes_per_report: 20.17
packets_received: 32
duplicate_rate: 0.000
sequence_gap_count: 0
cpu_ms_per_report: 0.522
```

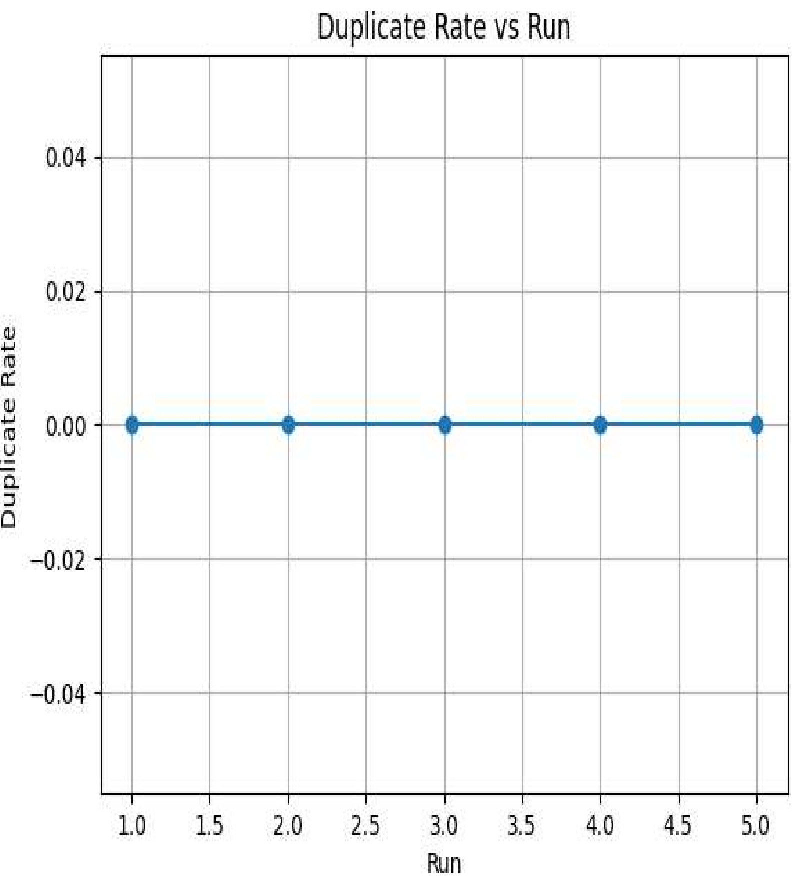
(5)

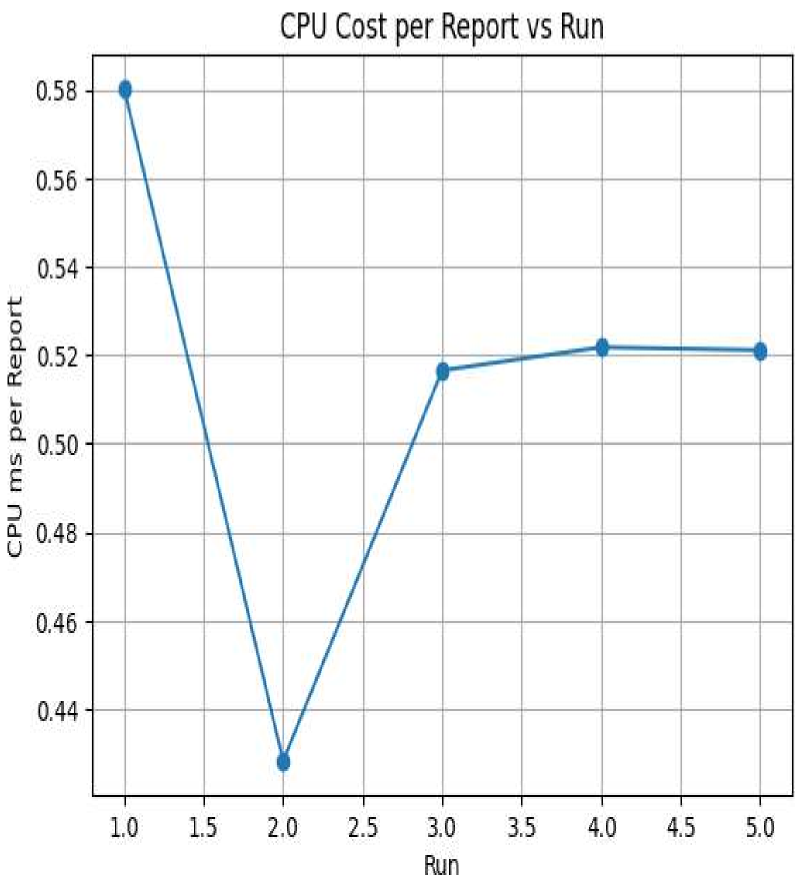


```
[END] Device 101 finished test

=== FINAL METRICS SUMMARY ===
bytes_per_report: 20.17
packets_received: 32
duplicate_rate: 0.000
sequence_gap_count: 0
cpu_ms_per_report: 0.521
```

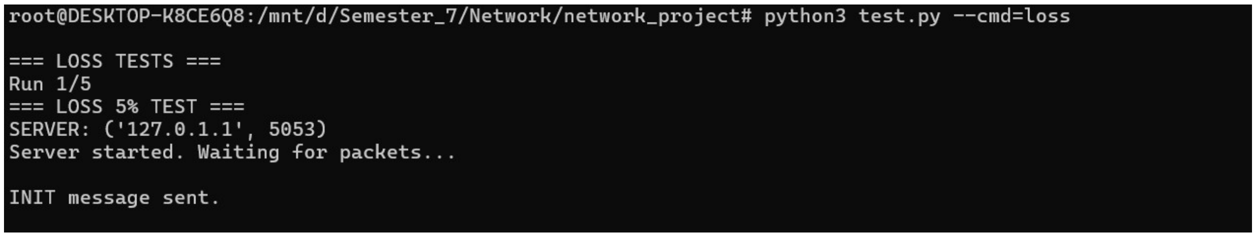

Bytes per Report vs Run
Bytes per Report
20.16
20.14
20.12
20.10
1.0
1.5
2.0
2.5
3.0
3.5
4.0
4.5
5.0
Run


Packets Received vs Run
Packets Received
32.0
31.8
31.6
31.4
31.2
31.0
1.0
1.5
2.0
2.5
3.0
3.5
4.0
4.5
5.0
Run


Duplicate Rate vs Run
Duplicate Rate
0.04
0.02
0.00
−0.02
−0.04
1.0
1.5
2.0
2.5
3.0
3.5
4.0
4.5
5.0
Run


CPU Cost per Report vs Run
CPU ms per Report
0.58
0.56
0.54
0.52
0.50
0.48
0.46
0.44
1.0
1.5
2.0
2.5
3.0
3.5
4.0
4.5
5.0
Run

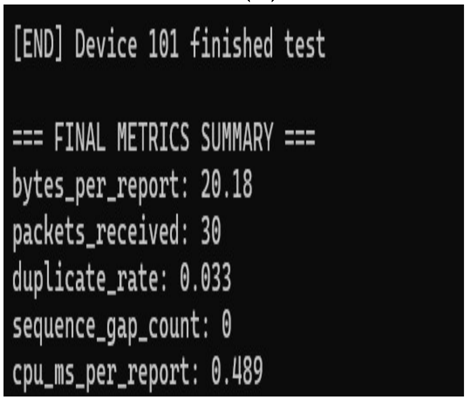
5% Loss :



```
root@DESKTOP-K8CE6Q8:/mnt/d/Semester_7/Network/network_project# python3 test.py --cmd=loss

=== LOSS TESTS ===
Run 1/5
=== LOSS 5% TEST ===
SERVER: ('127.0.1.1', 5053)
Server started. Waiting for packets...

INIT message sent.
```

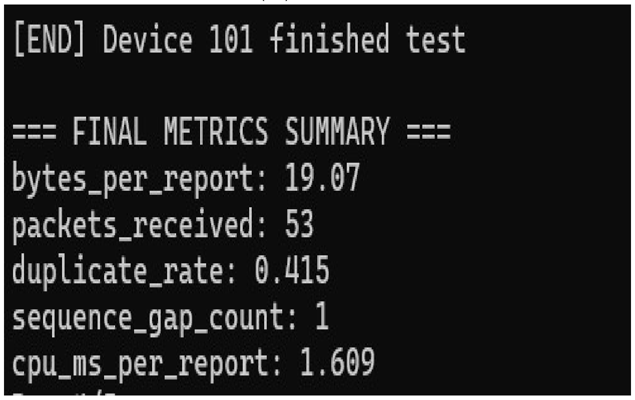
Results :

(1)



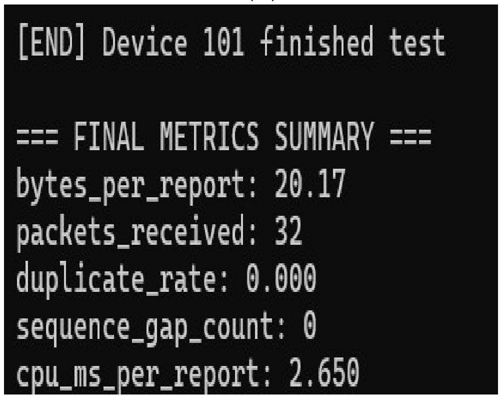
```
[END] Device 101 finished test

=== FINAL METRICS SUMMARY ===
bytes_per_report: 20.18
packets_received: 30
duplicate_rate: 0.033
sequence_gap_count: 0
cpu_ms_per_report: 0.489
```

(2)



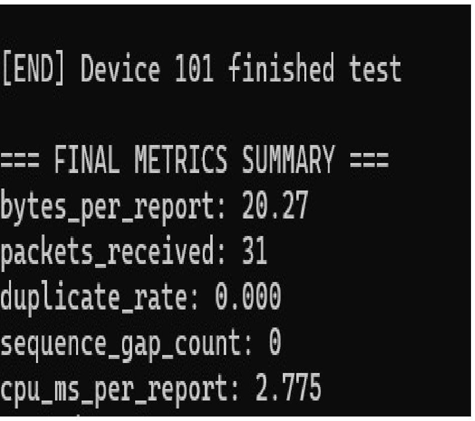
```
[END] Device 101 finished test

=== FINAL METRICS SUMMARY ===
bytes_per_report: 19.07
packets_received: 53
duplicate_rate: 0.415
sequence_gap_count: 1
cpu_ms_per_report: 1.609
```

(3)



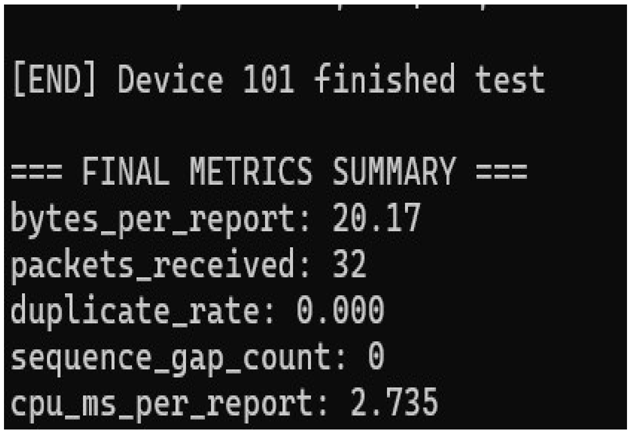
```
[END] Device 101 finished test

=== FINAL METRICS SUMMARY ===
bytes_per_report: 20.17
packets_received: 32
duplicate_rate: 0.000
sequence_gap_count: 0
cpu_ms_per_report: 2.650
```

(4)



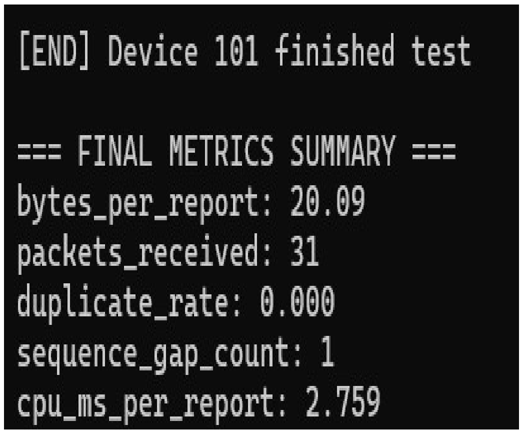
```
[END] Device 101 finished test

=== FINAL METRICS SUMMARY ===
bytes_per_report: 20.27
packets_received: 31
duplicate_rate: 0.000
sequence_gap_count: 0
cpu_ms_per_report: 2.775
```

(5)

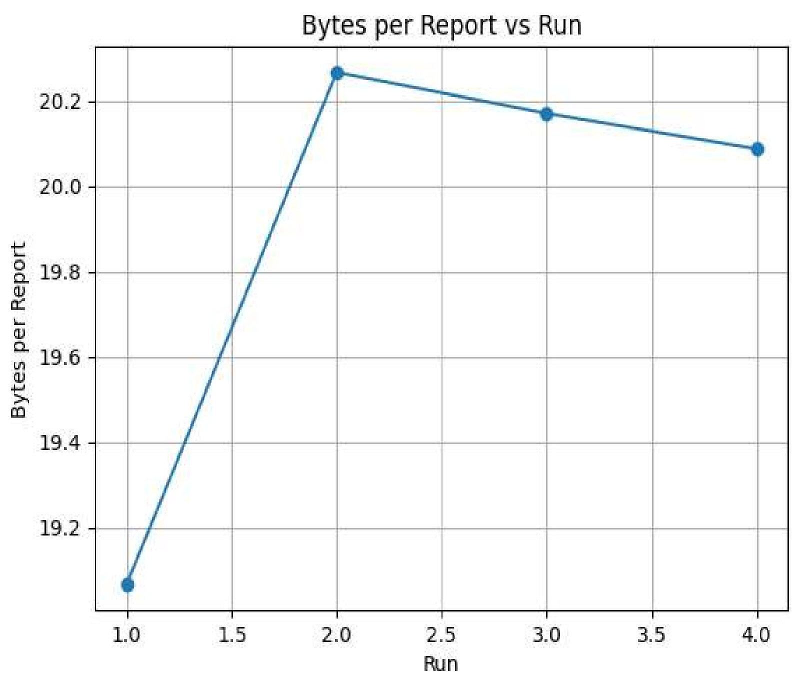


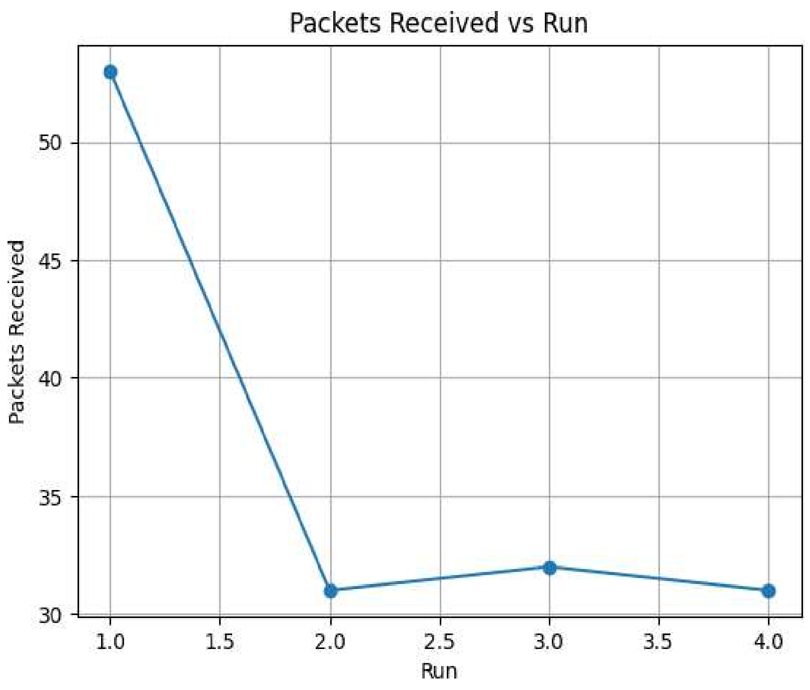
```
[END] Device 101 finished test

=== FINAL METRICS SUMMARY ===
bytes_per_report: 20.17
packets_received: 32
duplicate_rate: 0.000
sequence_gap_count: 0
cpu_ms_per_report: 2.735
```

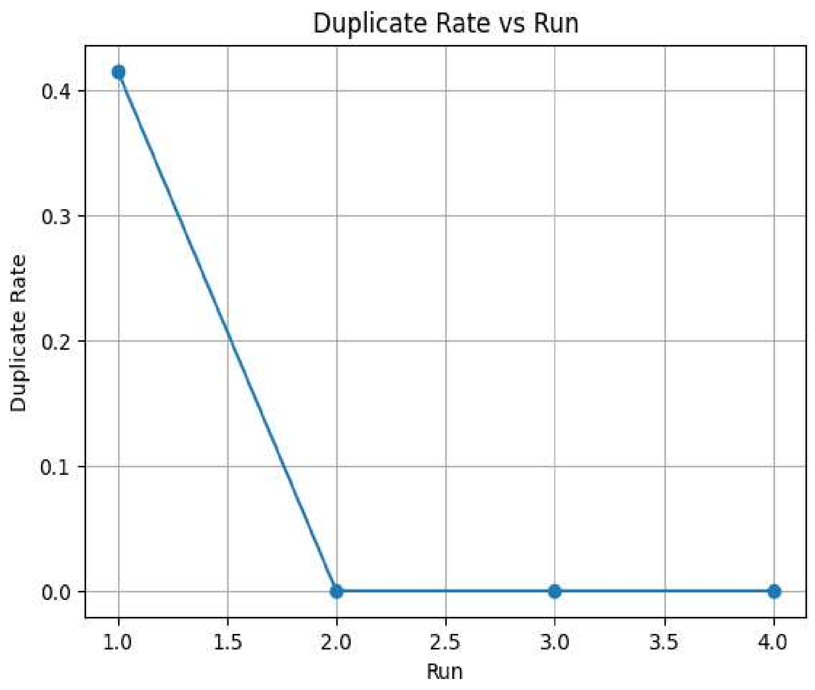
(6)

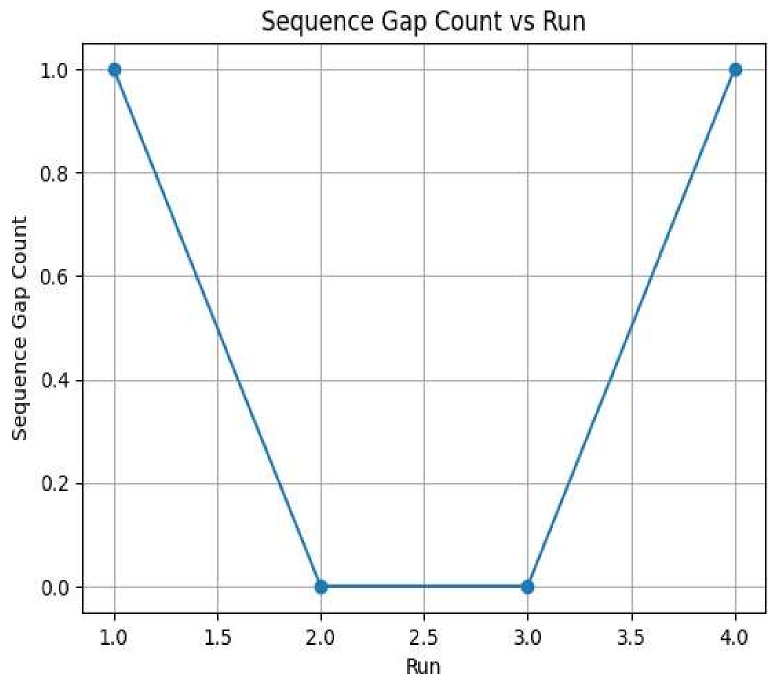


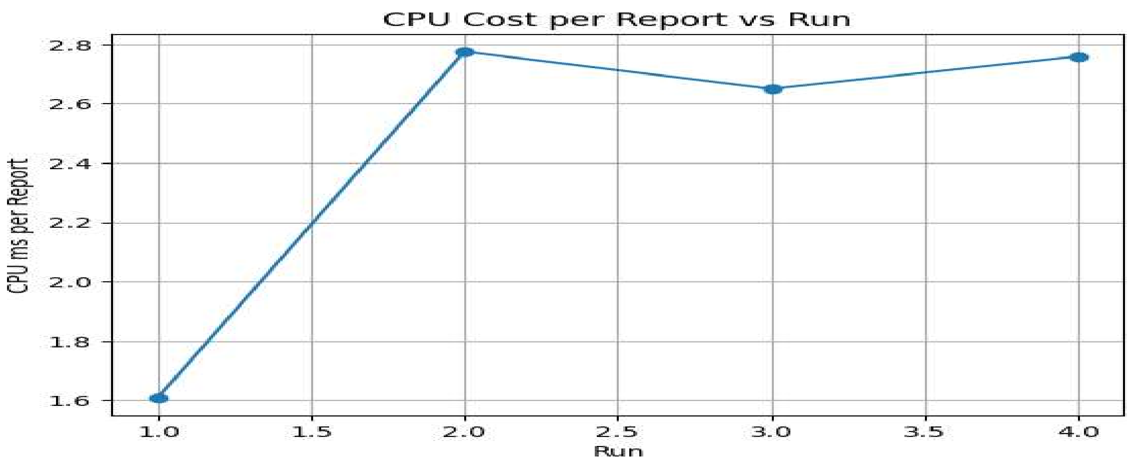
```
[END] Device 101 finished test

=== FINAL METRICS SUMMARY ===
bytes_per_report: 20.09
packets_received: 31
duplicate_rate: 0.000
sequence_gap_count: 1
cpu_ms_per_report: 2.759
```


Bytes per Report vs Run
Bytes per Report
Run

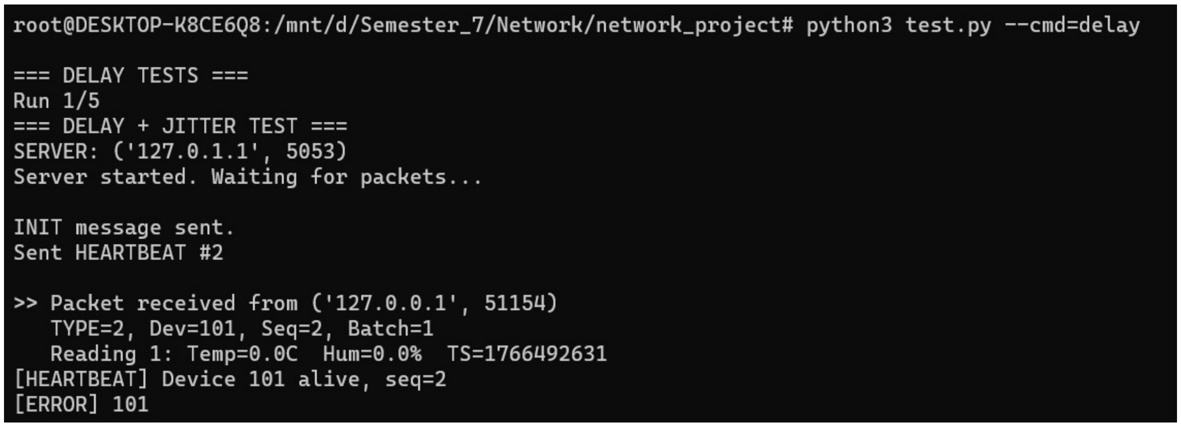

Packets Received vs Run
Packets Received
Run


Duplicate Rate vs Run
Duplicate Rate
Run


Sequence Gap Count vs Run
Sequence Gap Count
Run

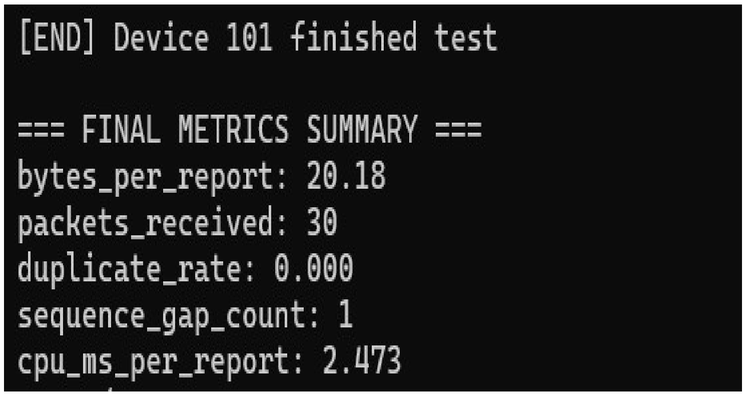

CPU Cost per Report vs Run
CPU ms per Report
Run

Delay 100ms :



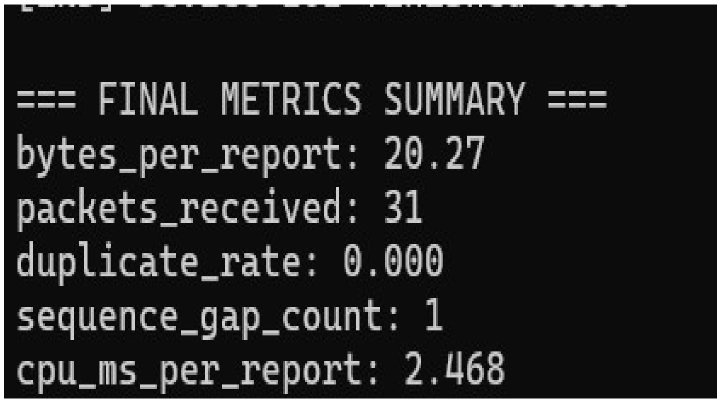
```
root@DESKTOP-K8CE6Q8:/mnt/d/Semester_7/Network/network_project# python3 test.py --cmd=delay

=== DELAY TESTS ===
Run 1/5
=== DELAY + JITTER TEST ===
SERVER: ('127.0.1.1', 5053)
Server started. Waiting for packets...

INIT message sent.
Sent HEARTBEAT #2

>> Packet received from ('127.0.0.1', 51154)
   TYPE=2, Dev=101, Seq=2, Batch=1
   Reading 1: Temp=0.0C  Hum=0.0%  TS=1766492631
[HEARTBEAT] Device 101 alive, seq=2
[ERROR] 101
```

Results : (1)



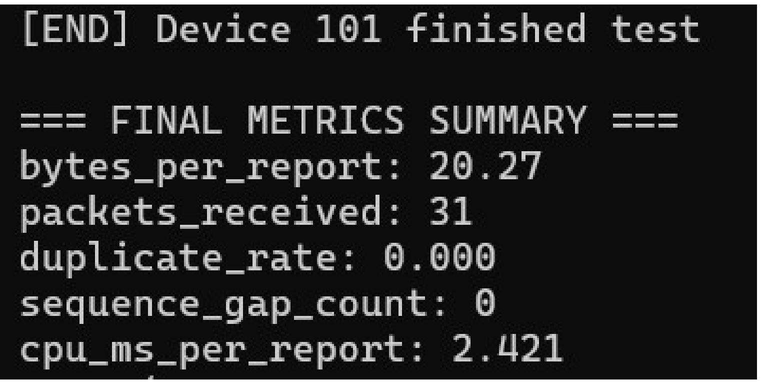
```
[END] Device 101 finished test

=== FINAL METRICS SUMMARY ===
bytes_per_report: 20.18
packets_received: 30
duplicate_rate: 0.000
sequence_gap_count: 1
cpu_ms_per_report: 2.473
```

(2)



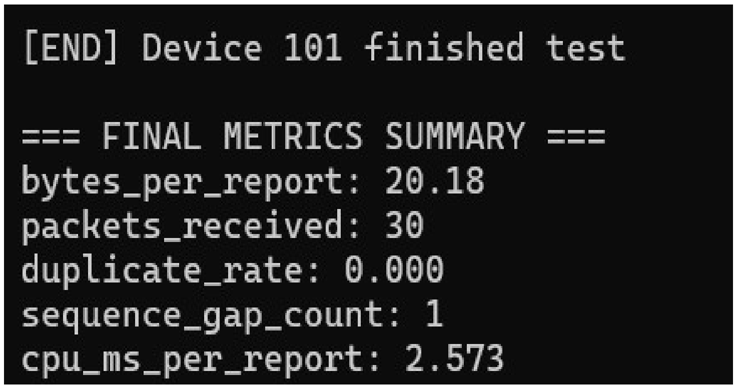
```
=== FINAL METRICS SUMMARY ===
bytes_per_report: 20.27
packets_received: 31
duplicate_rate: 0.000
sequence_gap_count: 1
cpu_ms_per_report: 2.468
```

(3)



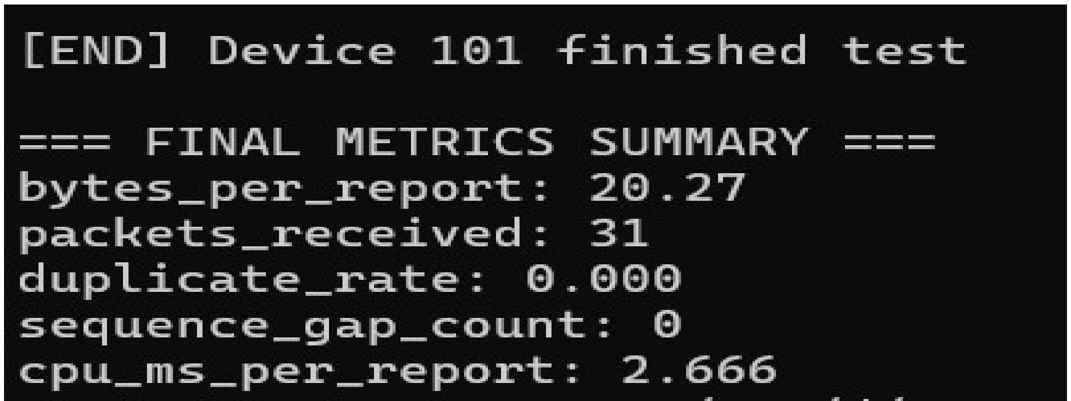
```
[END] Device 101 finished test

=== FINAL METRICS SUMMARY ===
bytes_per_report: 20.27
packets_received: 31
duplicate_rate: 0.000
sequence_gap_count: 0
cpu_ms_per_report: 2.421
```

(4)

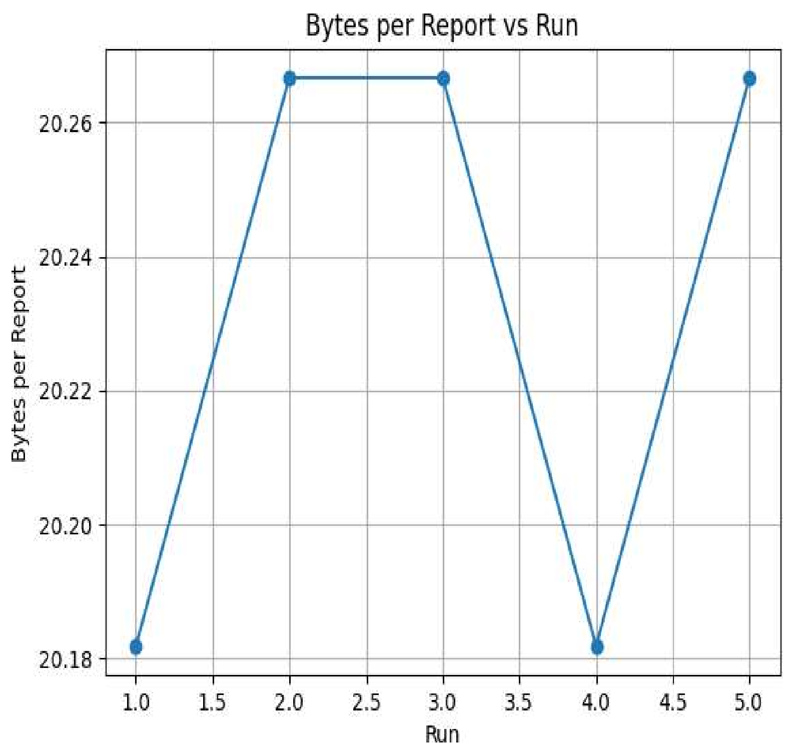
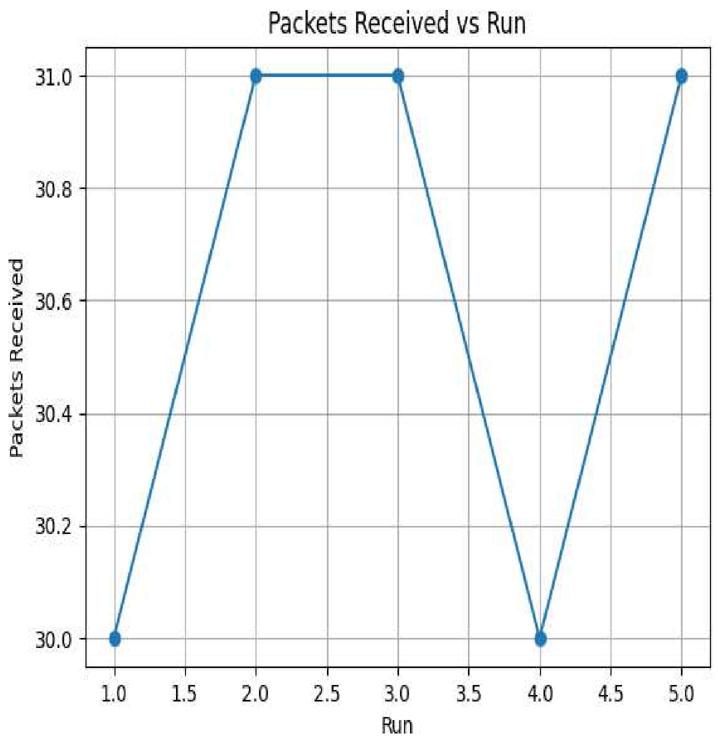
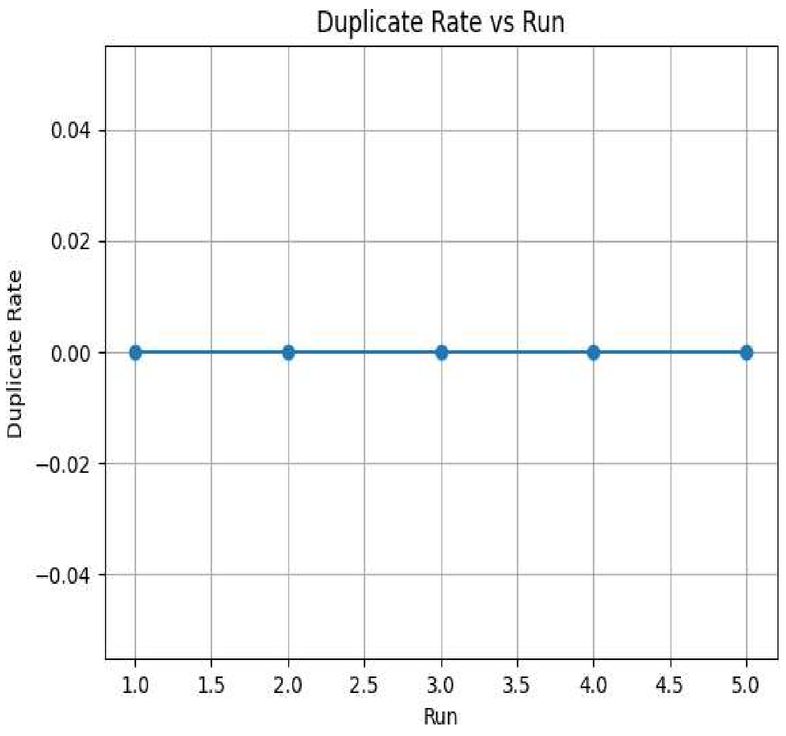
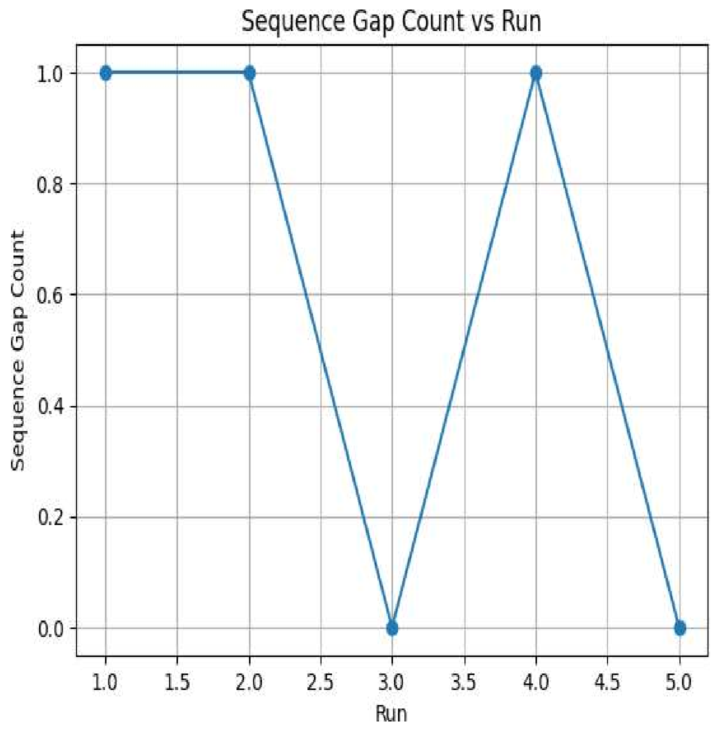
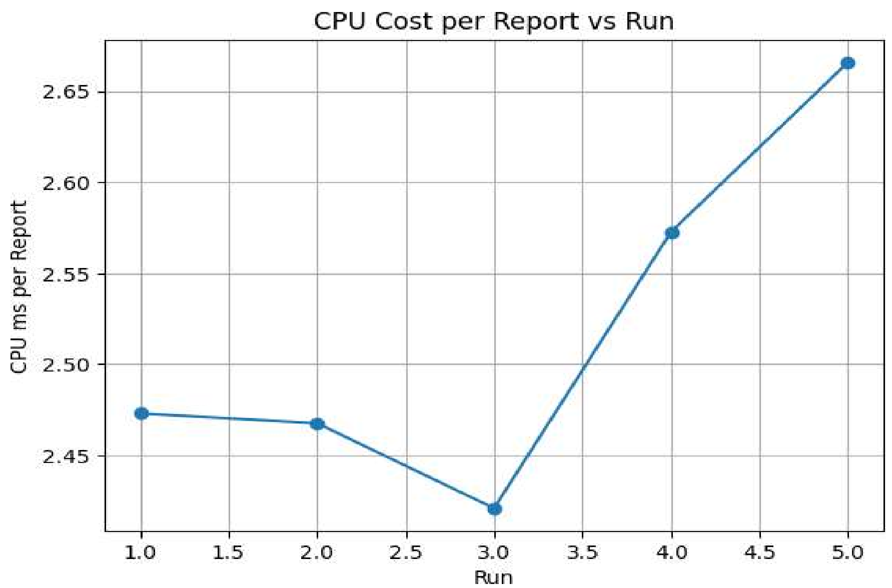


```
[END] Device 101 finished test

=== FINAL METRICS SUMMARY ===
bytes_per_report: 20.18
packets_received: 30
duplicate_rate: 0.000
sequence_gap_count: 1
cpu_ms_per_report: 2.573
```

(5)



```
[END] Device 101 finished test

=== FINAL METRICS SUMMARY ===
bytes_per_report: 20.27
packets_received: 31
duplicate_rate: 0.000
sequence_gap_count: 0
cpu_ms_per_report: 2.666
```


Bytes per Report vs Run
Bytes per Report
Run
20.26
20.24
20.22
20.20
20.18
1.0
1.5
2.0
2.5
3.0
3.5
4.0
4.5
5.0

Packets Received vs Run
Packets Received
Run
31.0
30.8
30.6
30.4
30.2
30.0
1.0
1.5
2.0
2.5
3.0
3.5
4.0
4.5
5.0

Duplicate Rate vs Run
Duplicate Rate
Run
0.04
0.02
0.00
−0.02
−0.04
1.0
1.5
2.0
2.5
3.0
3.5
4.0
4.5
5.0

Sequence Gap Count vs Run
Sequence Gap Count
Run
1.0
0.8
0.6
0.4
0.2
0.0
1.0
1.5
2.0
2.5
3.0
3.5
4.0
4.5
5.0

CPU Cost per Report vs Run
CPU ms per Report
Run
2.65
2.60
2.55
2.50
2.45
1.0
1.5
2.0
2.5
3.0
3.5
4.0
4.5
5.0

3.1 Bytes per Report vs Batch Size

Observation:
Batching significantly reduces the average bytes per telemetry report. When batch size increases, the fixed header overhead is amortized over multiple readings.

Interpretation:
This confirms that the `BatchCnt` mechanism improves bandwidth efficiency, especially in high-frequency reporting scenarios.

3.2 Duplicate Rate under Packet Loss

Observation:
Under 5% packet loss, duplicate packets appear due to retransmissions. However, the duplicate rate remains bounded and manageable.

Interpretation:
The ACK-based retransmission mechanism ensures reliability, while server-side duplicate detection prevents data corruption.

3.3 Sequence Gap Count

Observation:
Sequence gaps increase under packet loss and delay, especially before retransmissions complete.

Interpretation:
Sequence gap detection provides visibility into network reliability issues and validates the usefulness of explicit sequence numbering.

3.4 CPU Time per Report

Observation:
CPU time per report remains low (< few milliseconds) even with batching and reordering enabled.

Interpretation:
The reordering buffer and duplicate detection logic introduce minimal computational overhead, making MiniTelemetry suitable for scalable deployments.

4. Packet Reordering Analysis

MiniTelemetry includes a **timestamp-based reordering buffer** on the server side. Incoming readings are temporarily stored for 250 ms and sorted by payload timestamp before logging.

Effectiveness:

- Correctly handles out-of-order packets caused by network delay or retransmission.
- Prevents time-series corruption in stored telemetry data.

Trade-off:

- Introduces a small latency (≤250 ms) before data is committed.

5. Reliability Analysis

Reliability is achieved through:

- Explicit ACK messages.
- Timeout-based retransmission.
- Duplicate and gap detection.

Unlike TCP, MiniTelemetry avoids connection establishment and congestion control, trading strict reliability for lower overhead and simplicity. Experimental results show that this approach is sufficient for telemetry workloa

6. Limitations

Despite its effectiveness, MiniTelemetry has several limitations:

1. **No Security:**
   a. No authentication, encryption, or integrity protection.
2. **Fixed Retransmission Timeout:**
   a. RTO is static and not adaptive to RTT variations.
3. **One-Way Communication Model:**
   a. Server does not send control or configuration updates.
4. **UDP Constraints:**
   a. Packet size limited by MTU; fragmentation not handled.

7. Future Work

Potential improvements include:

- Adaptive retransmission timeout (RTT estimation).
- Lightweight encryption or DTLS support.
- Adaptive batching based on network conditions.
- Forward Error Correction (FEC) to reduce retransmissions.
- Bidirectional control messages.

8. Conclusion

This report evaluated MiniTelemetry, a lightweight UDP-based telemetry protocol designed for IoT systems. Experimental results demonstrate that batching and reordering significantly improve efficiency and robustness under lossy and delayed networks. While the protocol sacrifices some features of mature standards, it offers a simple, extensible, and effective solution for telemetry-oriented applications.

G. References

1. RFC 768 – User Datagram Protocol (UDP)
2. RFC 6298 – Computing TCP’s Retransmission Timer
3. Linux netem documentation
4. Python `socket` and `struct` documentation