Mini-RFC

1. Introduction

Telemetry protocol is a lightweight telemetry protocol that we designed to let small IoT sensors send data to a central collector using UDP.
It's meant for simple, real-time applications such as temperature or humidity reporting, where packets are small, sent frequently, and occasional packet loss is acceptable.

Most existing IoT protocols, like MQTT or HTTP, rely on TCP, which guarantees delivery but adds a lot of overhead — extra headers, connection setup, and retransmissions that waste time and energy.
In contrast, it focuses on speed, simplicity, and low power use.

Instead of trying to guarantee delivery, it lets the collector detect if a packet was lost or duplicated.
This trade-off keeps the system light and perfect for low-power or resource-limited sensors.

2. Protocol Architecture

It follows a simple client–server design.

- The Sensor (Client) simulates a small IoT device that measures something (like temperature).
  It sends messages at regular intervals and uses a unique device ID so the collector can tell which device sent which packet.
- The Collector (Server) listens for UDP packets, checks if they belong to the protocol, and records them in a CSV log file.
  It keeps a small internal "memory" for each device so it can detect if a packet arrived twice (a duplicate) or if one went missing (a gap).

The protocol doesn't require an open connection. Every message stands alone and includes everything the collector needs to understand it.

Operation Flow

1. Initialization:
   When a sensor starts, it sends an INIT message to announce itself.
   The collector replies with an INIT_ACK to confirm that it's ready.
2. Data Transmission:
   After that, the sensor sends regular DATA messages that carry readings (like temperature values).
   These packets are sent every few seconds, depending on the configuration.
3. Heartbeat:
   If the sensor has no new data to send, it sends a HEARTBEAT message just to say "I'm still here."
4. Logging and Monitoring:
   The collector logs every packet it receives into a CSV file that includes:
5. device_id, seq, timestamp, arrival_time, duplicate_flag, gap_flag

This makes it easy to later analyze how reliable the communication was.

Key Design Choices

- Transport Layer: UDP
- Reliability: Detection only
- Header Size: 12 bytes fixed
- Payload Limit: 200 bytes maximum per message
- Reordering: The collector can buffer out-of-order packets for about one second before writing them to the log
- Supported Format: Float32 readings

This simple structure allows efficient communication while still making it possible to measure performance and packet behavior accurately.

3. Message Formats

Header Structure

Every packet begins with a **12-byte header** that identifies the sender, the message type, and basic timing information.

Field	Size	Description
MAGIC	1 byte	A constant value (0x54) used to confirm the packet belongs to Telemetry protocol.
VER_TYPE	1 byte	Combines the protocol version (upper 4 bits) and message type (lower 4 bits).
DEVICE_ID	2 bytes	Unique ID for the sending device.
SEQ	4 bytes	Sequence number that increases with each new message.
TIMESTAMP	4 bytes	Time (in seconds) since the device started running.

Message Types

Code	Type	Direction	Purpose
0x0	**INIT**	Sensor → Collector	Announces the sensor and starts the session.
0x1	**INIT_ACK**	Collector → Sensor	Confirms initialization.
0x2	**DATA**	Sensor → Collector	Sends one or more sensor readings.

Code	Type	Direction	Purpose
0x3	**HEARTBEAT**	Sensor → Collector	Indicates the sensor is still alive when no new data exists.

DATA Payload Format

After the 12-byte header, a DATA message carries one or more small "reading blocks." Each block contains:

Field	Size	Description
Sensor ID	1 byte	Identifies which sensor the reading came from (e.g., 1 = temperature).
Format	1 byte	Data type (0x01 = float32).
Value	4 bytes	The measurement value in IEEE 754 float format.

Multiple readings can be batched in one packet, as long as the total size does not exceed **200 bytes** (header + payload).

Example Message

Below is an example of a single **DATA** packet from a temperature sensor:

Field	Example Value	Explanation
MAGIC	0x54	Identifies this as a packet.
VER_TYPE	0x12	Version 1, Message Type = DATA (2).
DEVICE_ID	100	Sensor ID = 100.
SEQ	5	Fifth packet sent by this sensor.
TIMESTAMP	60 s	Sent 60 seconds after startup.
PAYLOAD	Sensor 1, Format = float32, Value = 20.45 °C	The actual reading.

Interpretation
After running for one minute, sensor #100 sends its fifth packet containing a single temperature reading of 20.45 °C.
The collector logs this packet and can later tell, from the sequence numbers, if any packets were lost or repeated.

4. Communication Procedures

4.1 Initialization Procedure (INIT / INIT_ACK)

The initialization exchange allows the collector to establish per-device state before processing data.

Sensor → Collector (INIT)

- Sent once when the sensor starts.
- Contains capability string (ASCII).
- Sequence number increments normally.

Collector → Sensor (INIT_ACK)

Upon receiving INIT:

1. Collector resets or creates device state:
    - last_seq
    - seen_seqs
    - reorder_buffer
    - dup_count, gap_count
2. Collector responds with INIT_ACK including:
    - Same device ID
    - Same sequence number
    - Fresh timestamp

If INIT or INIT_ACK is lost, no retransmission occurs. The sensor begins sending data regardless.

4.2 DATA Packet Procedure

DATA packets carry one or more sensor readings. Each reading contains:

- sensor_id (1 byte)
- format (1 byte)
- Value (float32 or int16)

Collector Behavior

For each DATA packet:

1. **Duplicate Detection**
    - If seq ∈ seen_seqs, packet is marked duplicate and logged with duplicate_flag=1.
2. **Gap Detection**
    - If seq != last_seq + 1 and last_seq is not None, the packet is marked with gap_flag=1.
3. **Reordering Buffering**
    - All non-duplicate DATA packets are inserted into a per-device reorder buffer with metadata:
        - seq
        - timestamp
        - arrival_time
        - gap flag
4. **Reordering Flush Conditions**
   Packets flush from buffer to CSV when:
    - Their timestamp is older than the newest by ≥ 1 second
    - Or buffer size > 64 packets
    - Or a HEARTBEAT triggers a forced flush

This ensures correct ordering even under delay or jitter.

4.3 HEARTBEAT Procedure

Heartbeats are sent only when the sensor has no data to report.

Sensor → Collector (HEARTBEAT)

- Same header format as DATA, without payload.
- Indicates device liveness.

Collector Behavior

- Logs heartbeats directly to CSV.
- Forces flush of reorder buffer to avoid waiting for more DATA.
- No retransmission or ACKs are involved.

Lost heartbeats are naturally tolerated — subsequent packets update liveness.

5. Reliability & Error Handling

Using a **detection-based reliability model**:

- No retransmissions
- No ACKs for DATA or HEARTBEAT
- Reliability is achieved through inference (gaps, duplicates, timestamps)

5.1 Duplicate Detection

The collector maintains a seen_seqs set per device.
Duplicate packets are logged but not reprocessed.

5.2 Gap Detection

If seq != last_seq + 1, a loss gap is recorded.
Gaps are logged in CSV for later statistical analysis.

5.3 Reordering Handling

A reorder buffer (max 64 entries) prevents incorrect timestamp ordering.
Flush occurs when window expires or when forced by heartbeat.

5.4 Timestamp-based Ordering

Packets are sorted by (timestamp, seq) to maintain consistent ordering under network impairment.

6. Experimental Evaluation Plan

6.1 Test Setup

- A collector running on localhost (UDP port 9999).
- A sensor transmitting at 1-second intervals.
- Logs recorded to telemetry_log.csv.
- PCAP traces collected via tcpdump or Wireshark.
- Network impairments injected using Linux netem.

6.2 Baseline Test (No Impairment)

- Run collector + sensor for 60 seconds.
- Confirm:
    - No duplicates
    - No gaps
    - Flush behavior consistent
    - CSV ordered by timestamp

6.3 Packet Loss Tests

Test Scenarios

Scenario	Netem Command	Purpose / Expected Behavior
Baseline	none	≥ 99 % of packets received (1 s interval, 60 s run). Seqs in order.
Loss 5 %	sudo tc qdisc add dev lo root netem loss 5%	Collector detects gaps; duplicate rate ≤ 1 %. No crash.
Delay + Jitter (100 ± 10 ms)	sudo tc qdisc add dev lo root netem delay 100ms 10ms	Collector reorders correctly; buffer drains gracefully.

- Each test runs for 60 s and is repeated five times with different random seeds.
- Metrics (median / min / max) are computed across runs.
- Automation
- runner.py starts collector + sensor automatically.
- Additional bash/python scripts run all netem scenarios, capture .pcap traces, and save resulting telemetry_log.csv files in /results/.