Tiny Telemetry v1

Team Members:

Youssef Moustafa Elsayed	22P0047
Mostafa Fouad	22P0077
Nour Eldeen Ahmed Rehab	22P0040
Abdelrahman Mostafa	22P0053
Mohamed Yasser Khallaf	22P0245
Ibrahim Shaker Hammad	22P0056

Table of Contents

1. Introduction

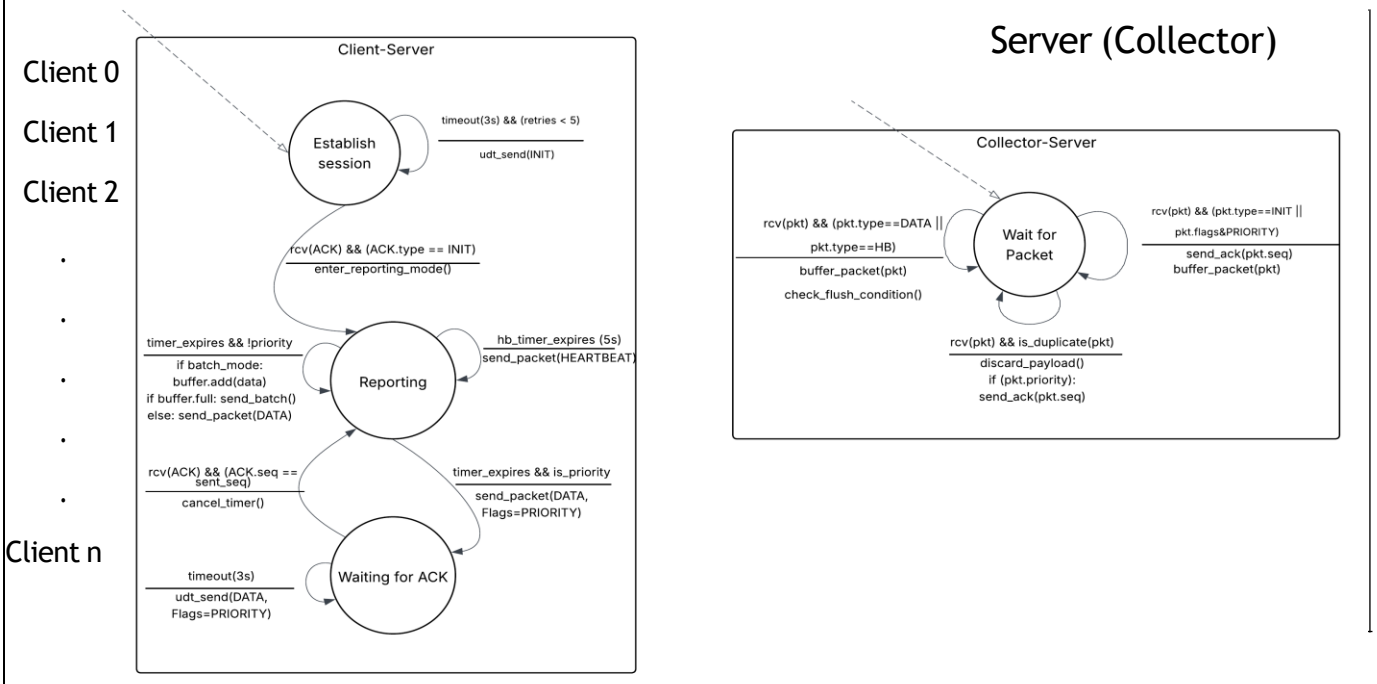
This document specifies a lightweight IoT Telemetry Protocol (ITP) designed for sensor devices that periodically send temperature and humidity data to a central collector (server) using UDP. The goal is to provide a simple, connectionless, low-overhead transport for periodic sensor readings while maintaining reliability metrics such as sequence tracking and duplicate suppression.

Existing protocols like MQTT and CoAP provide reliable messaging but add connection overhead, acknowledgments, and session management unsuitable for small embedded IoT nodes. This protocol focuses on *baseline data collection over UDP*, trading delivery guarantees for simplicity and low latency.
It is intended for controlled environments such as lab simulations or small-scale IoT deployments where occasional packet loss is acceptable.

2. Protocol Architecture

The enhanced protocol architecture adopts a stateful telemetry model overlaid on the connectionless UDP transport, where the server functions as a context-aware coordinator for multiple concurrent clients. The server enforces a logical session lifecycle, requiring an explicit handshake to establish a trusted communication channel before data processing begins. In this capacity, the server maintains distinct state vectors for each connected device, enabling it to track sequence progression, reorder out-of-sequence packets via a jitter buffer, and validate data integrity before permanent storage. Additionally, the server employs a passive liveness mechanism to govern session duration, automatically reclaiming resources from unresponsive clients while guaranteeing the delivery of critical priority events through selective application-layer acknowledgments.

2.1. Finite State Machine (FSM)



3. Message Formats

In any communication protocol, defining a consistent and well-structured message format is essential for reliable data exchange. Proper formatting allows both the sender and receiver to correctly interpret the transmitted data, ensuring synchronization and reducing ambiguity. By separating the header (which contains metadata such as version, message type, device ID, and sequence number) from the payload (which carries the actual sensor readings like temperature and humidity), the protocol achieves modularity and extensibility. This structure also simplifies parsing, error detection, and future upgrades without breaking backward compatibility.

Tiny Telemetry v1 Header table (total 12 bytes)

Header Field	Size (bytes)	Symbol	Symbol Meaning	Description
Version + MsgType	1	B	Unsigned char	protocol version & INIT or DATA OR HEARTBEAT OR ACK
DeviceID	2	H	Unsigned short	Unique ID for each sensor
SeqNum	2	H	Unsigned short	Order of sent packets (0 for INIT usually)
Timestamp	4	I	Unsigned int	Current Unix timestamps (seconds)
Flags	1	B	Unsigned char	Operational flags (Batch flag + Priority flag)
Checksum	1	H	Unsigned short	16-bit One's Complement Checksum (Header + Payload)

Telemetry Payload (4 bytes)

Payload Field	Symbol	Symbol Meaning	Size	Size (bits)	Description
Temperature	h	Signed short	2	16	protocol version (version 1 for now)
Humidity	H	Unsigned short	2	16	INIT or DATA (for our version 1 now)

Total Message Size: 12 (header) + 4 (payload) = 16 bytes

4.Communication Procedures

This section defines the Hybrid Reliability model. The protocol mixes connectionless streaming for standard data with connection-oriented confirmed delivery for critical events.

4.1. Session Establishment

1. **INIT Request:** The Client sends a Type 0 packet. It enters a blocking wait state (INIT_TIMEOUT = 3.0s).
2. **Server Response:** Upon receiving a valid INIT, the Server immediately replies with a **Type 3 ACK** containing the same Sequence Number.
3. **Retries:** If the Client does not receive an ACK within the timeout, it retries up to **5 times** (MAX_INIT_RETRIES). If all fail, the Client terminates.

4.2. Data Transmission Modes

- **Standard Mode (Best Effort):** For small data changes, the Client sends Type 1 packets and *does not* wait for an ACK. This maximizes throughput and minimizes battery usage.
- **Priority Mode (Reliable):** If the sensor detects a large sudden change, it sets the Priority Flag (bit 2).
    - The Server must ACK this packet.
    - The Client waits for the ACK and retransmits, if necessary (Stop-and-Wait logic), ensuring critical alerts are never lost.
- **Batch Mode (High Efficiency):** To reduce CPU interruptions and header overhead, the Client can buffer readings. When the buffer fills (e.g., 5 readings), it flushes them in a single Type 1 packet with the Batch Flag (bit 0) set

4.3. Keep Alive Mechanism

To maintain visibility during idle periods or to verify connectivity separate from data flow:

- The Client tracks the time since the last heartbeat.
- If Time_Elapsed >= 5 seconds, the Client sends a HEARTBEAT (Type 2) packet.
- This ensures the Server logs the device's presence even if data packets are not being sent.

5. Reliability & Performance Features

To mitigate the inherent unreliability of UDP while preserving the bandwidth efficiency required for IoT, the protocol employs a Hybrid Reliability Architecture. This design combines selective application-layer acknowledgments (ARQ) for critical control packets with robust server-side reconstruction logic for the standard high-volume telemetry streams.

5.1. Buffering and Reordering

Mechanism: Incoming packets are held in a packet_buffer list rather than being written to disk immediately.

5.2. Duplicate Suppression

Action: If true, the packet is flagged as a DUPLICATE and ignored. This prevents replayed packets from corrupting the data logs.

5.3. Gap Detection

The Server tracks the last_seq processed for each device.

5.4. Data integrity (Checksum)

To detect corruption over the wire (for example: bit flips), every packet includes a 16-bit Internet Checksum (One's Complement sum).

5.5. Selective Reliability

The protocol avoids the overhead of TCP by only acknowledging critical data.

5.6. Batching Optimization

Batching improves Channel Utilization by reducing the ratio of Header bytes to Payload bytes

6. Experimental Evaluation Plan

6.1. The test.py Automation Suite

ensure reproducible verification of the protocol's new features, a custom Python test runner (test.py) was developed. This script acts as an orchestrator that integrates three components:

- **Traffic Shaping (Netem):** It automatically injects Linux kernel-level impairments.
- **Process Management:** It spawns the Server and multiple Client instances (subprocess.Popen) in the background, manages their PIDs, and terminates them after Run Duration.
- **Packet Capture (Scapy):** It launches a background thread using scapy to sniff UDP traffic on port 5000. It saves the session to a timestamped .pcap file (For Example: Packet_Loss_5pct_2023.pcap) for analysis in Wireshark.

6.2. Test Scenarios

Scenario	Objective
Baseline	Verify that the Server can multiplex traffic from multiple devices
Packet Loss	Ensures the Server detects gaps in standard data streams while the Client successfully detects timeouts and retransmits critical Priority packets
High Jitter	Force severe out-of-order delivery to verify that the Server's Jitter Buffer correctly sorts data by timestamp before logging
Packet Duplication	Validate Duplicate Suppression, which discards payloads to protect data integrity but must still re-send ACKs for Priority packets to maintain session liveness
Batch Mode	Ensures the Client buffers 5 readings into a single payload and the Server correctly unpacks them, resulting in reduced packet overhead

6.3. Metrics

Each Test records:

- **Packets_received (successfully)**
- **Bytes_per_report**
- **Duplicate_rate**
- **Sequence_gap_count**
- **Cpu_ms_per_report**

7. Automation & Reproducibility

To ensure consistent execution, two automation scripts were developed:
test.py (Impairment Runner): A Python wrapper that automates the setup of tc-netem rules, launches the server and clients, and captures traffic using scapy for scenarios involving packet loss and jitter.

Verified Scenarios

The framework includes pre-defined scenarios to validate the specific mechanisms added in Phase 2:
Scenario 1 (Packet Loss 5%): Validates that INIT_RETRY logic works when handshake packets are dropped

7.1. Example Use Case Walkthrough

This Part is where we provide evidence that our protocol handles loss correctly in different ways

Test Case 1 (Packet Loss 5%)

Proof 1: Priority Retry Logic (Reliability)
Packet #23 (a Priority packet) was lost or its ACK was lost.
The client timed out and re-sent it. The server saw it again and re-ACKed it.
CLIENT LOG:
|client | seq=23 | client id=0 | PRIORITY_SEND attempt=1/5
|client | seq=23 | client id=0 | ACK_TIMEOUT attempt=1 <--- Timeout
| client | seq=23 | client id=0 | ACK_RETRY wait=1s <--- ACK Retry
| client | seq=23 | client id=0 | PRIORITY_SEND attempt=2/5
| client | seq=23 | client id=0 | ACK_RECEIVED <--- ReACKING Successful!
SERVER LOG:
| server | seq=23 | client id=0 | PRIORITY_DUPLICATE ACK_RESENT <--- Server handles retry safely

1	1	0	23	1.77E+09	2	2797	19.8	45.1	1.77E+09

Since it appeared in the csv file after recording is done means it is REACKED

Proof 2: Gap Detection (Normal Data Loss)
Packet #10 (a Normal packet) was lost. Since it's not priority, the client *didn't* retry. The server detected the hole later.
The server log shows a jump from Seq 9 to Seq 11.
CLIENT LOG:
| client | seq=10 | client id=0 | SENT temp=19.3C
| client | seq=11 | client id=0 | SENT temp=19.5C

SERVER LOG:
| server | seq=9 | client id=0 | ACK_SENT
| server | seq=11 | client id=0 | DATA_RECEIVED temp=19.5C <--- #10 is missing!
| server | seq=11 | client id=0 | SEQ_GAP expected=10 got=11 missing=1 <--- Detected!

1	1	0	9	1.77E+09	2	10231	19.3	44.6	1.77E+09
1	1	0	11	1.77E+09	0	8951	19.5	44.6	1.77E+09

It shows a detected gap. The sequence jumps from 9 to 11, indicating Packet #10 was lost and correctly identified as a gap.

Proof 3: Heartbeat Liveness
Even though 5% loss is active, heartbeats are getting through to keep the connection alive.
SERVER LOG:
|| server | seq=0 | client id=0 | HEARTBEAT_RECEIVED at=1765547644.99
…
| server | seq=0 | client id=0 | HEARTBEAT_RECEIVED at=1765547660.00

Test Name	packets_received	bytes_per_report	duplicate_rate	sequence_gap_count	cpu_ms_per_report
Packet Loss 5%	65	16	0.0152	2	0.4615

Here is an Automated metrics summary generated by the test framework, verifying the 5% packet loss scenario

8. Limitations & Future Work

8.1. Limitations

- **No Encryption**
  - Data is sent in plaintext
  - Vulnerable to spoofing or interception
- **Single-Threaded Server**
  - Performance degrades with many clients
- **Simple Checksum**
  - Not cryptographically secure

8.2. Future Work

- **Add DTLS or TLS-over-UDP**
  - Secure communication
- **Adaptive Rate Control**
  - Adjust send rate based on loss/delay
- **Multithreaded or Async Server**
  - Improve scalability
- **Persistent Storage (Database)**
  - Replace CSV with SQL/NoSQL

9. Reference

- RFC 768 – User Datagram Protocol (UDP)
- Tanenbaum & Wetherall, *Computer Networks*, Pearson
- Kurose & Ross, *Computer Networking: A Top-Down Approach*
- Python socket, struct, csv
- **Scapy** – Packet capture & analysis
- **Linux tc / netem**