RFC: Compact Telemetry Protocol (CTP)

Version 1.0

Prepared By:

Ahmed Mohamed Al Amin (22P0137)
Mohamed Yehia Zakaria (22P0064)
Jana Hany Elshafie (22P0235)
Jana Sameh Samir (22P0237)
Seif Elhusseiny (22P0215)
Youssef Tarek Kamal (22P0236)

Delivered To:

Dr. Ayman Bahaa Eldin
Dr. Karim Emara

Faculty of Engineering, Ain Shams University
Computer Engineering and Software Systems Department
CSE361 — Computer Networking

Fall 2025

Transport Protocol	UDP
Default Port	12000
Byte Order	Big Endian (Network)
Status	Experimental

1 Introduction

The **Compact Telemetry Protocol (CTP)** is a lightweight, binary application-layer protocol designed for constrained IoT sensors operating over UDP. To address bandwidth limitations and battery constraints, CTP eliminates traditional TCP connection overhead, using a minimal 11-byte bit-packed header to maximize efficiency and support high-frequency telemetry. The protocol adopts a strict "fire-and-forget" model suitable for idempotent sensor data (e.g., temperature, humidity, voltage). It does not support retransmission (ARQ); instead, the Server detects gaps in sequence numbers without attempting recovery.

Despite its simplicity, CTP preserves observability and data integrity through the following architectural constraints:

- **Latency Handling:** UDP jitter is managed entirely at the application layer using a timestamp-driven Server Reordering Buffer, preventing head-of-line blocking and ensuring in-order processing.
- **Data Representation:** All multi-byte fields **MUST** be encoded in Network Byte Order (Big Endian). Timestamps use 32-bit integers relative to a Custom Epoch (Dec 1, 2025) to preserve millisecond-level precision while minimizing header size.
- **Integrity:** Every packet includes a mandatory 16-bit checksum, enabling the Server to detect transmission corruption and discard invalid data.

2 Protocol Architecture

2.1 Entities & Operational Modes

CTP follows a unidirectional **Push Model** in which the Sensor (Client) continuously transmits telemetry to a Collector (Server). The Client operates in one of two modes:

1. **Direct Mode:** The Client sends a packet immediately after each sensor reading to minimize latency.
2. **Batching Mode:** The Client aggregates N readings into a local buffer and transmits them as a single payload once the buffer is full ($N \leq 16$), maximizing throughput.

2.2 Reliability & Liveness

Liveness Monitoring: If the Client generates no data for `HEARTBEAT_INTERVAL` (5 s), it sends a `HEARTBEAT` packet. The Server marks a device `OFFLINE` if no packets are received for `LIVENESS_TIMEOUT` (10 s).

Jitter Handling: The Server employs a **Reordering Buffer**. Packets are queued upon arrival and sorted by timestamp rather than processed immediately. The buffer is flushed to application logic when its size exceeds `FLUSH_THRESHOLD` (20 packets), ensuring correct temporal ordering under UDP jitter.

2.3 Finite State Machines (FSM)

Client FSM

The Client alternates between reading sensor data and managing network transmission.

```
      +-----------+
      |   IDLE    | <-------------------------------------+
      +-----------+                                       |
        |       |                                         |
(Timer) |       | (Silence > 5s)                          |
        v       v                                         |
  +----------+  +-----------+   (Send Packet)             |
  |  GATHER  |  | HEARTBEAT | ----------------------------+
  +----------+  +-----------+
        |
        v (Buffer Full?)
  +----------+   (Send Packet)
  |   SEND   | -------------------------------------------+
  +----------+
```

Server FSM

The Server manages packet validation, jitter buffering, and device state tracking.

```
     +-----------+
     | LISTENING | <-----------------------------------------+
     +-----------+                                           |
        |     |                                              |
 (Recv) |     | (Timeout > 10s)                              |
        v     v                                              |
   +----------+      +---------+                             |
   | VALIDATE | ---> | OFFLINE |                             |
   +----------+      +---------+                             |
        | (Checksum OK)                                      |
        v                                                    |
   +----------+                                              |
   | BUFFERING| (Insert into Sort Queue)                     |
   +----------+                                              |
        |                                                    |
        v (Queue Size > Threshold?)                          |
   +----------+      +-----+      +-----+                    |
   | PROCESS  | ---> | LOG | ---> | CSV |                    |
   +----------+      +-----+      +-----+                    |
        |                                                    |
        +----------------------------------------------------+
```

2.4 Sequence Diagram (Normal Flow)

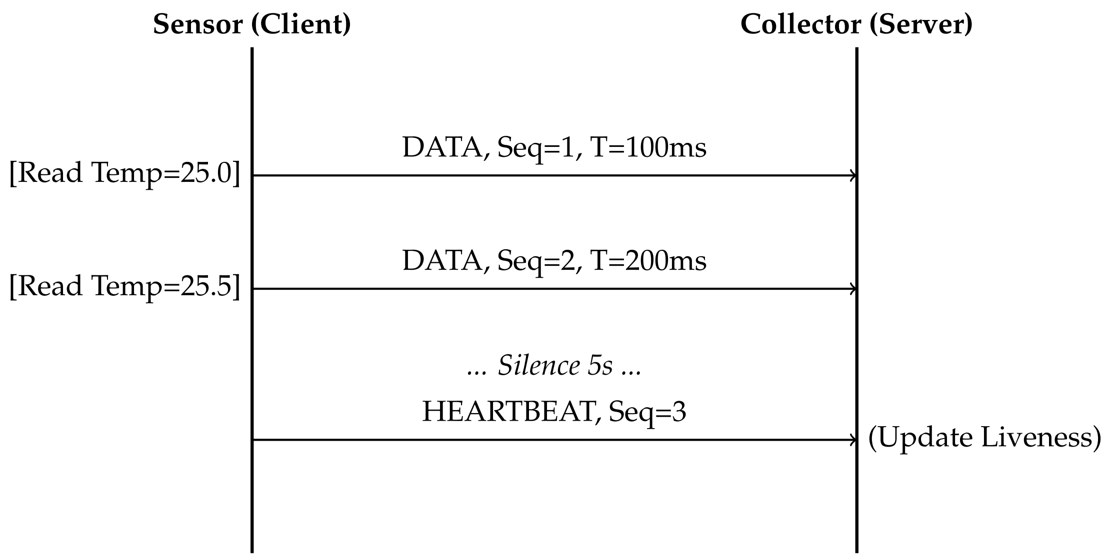

Figure 1: Typical communication flow showing data transmission and heartbeat.

3 Message Formats

3.1 General Encoding Rules

- **Byte Order:** All multi-byte integers MUST be transmitted in Network Byte Order (Big-Endian). In Python `struct` syntax, this is denoted by the `!` prefix.
- **Alignment:** Fields are packed tightly without padding bytes.
- **Total Header Overhead:** 11 Bytes (9 Byte Fixed Header + 2 Byte Checksum). This strictly satisfies the requirement of ≤ 12 bytes.

3.2 Fixed Header Structure

Every packet (whether `DATA` or `HEARTBEAT`) begins with the following 11-byte sequence.

Offset	Field Name	Size (Bits)	Type	Description
0	Device ID	16	`uint16`	Unique identifier for the sensor node.
2	Seq Num	16	`uint16`	Monotonic counter. Wraps at 65,535.
4	Timestamp	32	`uint32`	Milliseconds since Custom Epoch (see 3.3).
8	Ver / Type	8	`uint8`	Bit-Packed: [Ver: 4 bits | Type: 4 bits].
9	Checksum	16	`uint16`	Additive sum of (Header + Payload).

Python Struct Format: `!H H I B H`
Note: The first four fields are packed via `!HHIB`*, and the checksum* `!H` *is appended after being calculated over the specific payload.*

3.3 Field Definitions & Handling Logic

3.3.1 The "Ver / Type" Packed Byte (Offset 8)

To optimize space, the Protocol Version and Message Type are combined into a single byte using bitwise operations.

```
Bit:  7   6   5   4   3   2   1   0
    +---+---+---+---+---+---+---+---+
    |   Version     |   Msg Type    |
    +---+---+---+---+---+---+---+---+
```

- **Encoding Logic:** `packed_byte = (Version « 4) | (MsgType & 0x0F)`
- **Decoding Logic:**
  - `Version = (packed_byte » 4) & 0x0F`
  - `MsgType = packed_byte & 0x0F`

Supported Message Types:

- `0x01` = **DATA** (Contains Sensor Readings)
- `0x02` = **HEARTBEAT** (Empty Payload, Keep-alive only)

3.3.2 Timestamp Compression (Offset 4)

Standard Unix timestamps (seconds since 1970) do not provide enough precision for jitter analysis, but sending full 64-bit milliseconds is wasteful.

Optimization Strategy:

- **Custom Epoch:** We define a custom epoch `TIMESTAMP_OFFSET = 1764547200` (Corresponding to Dec 1, 2025).
- **Truncation:** The timestamp is calculated as `(CurrentTime - Offset) * 1000`.
- **Wrapping:** The result is masked to 32 bits (`& 0xFFFFFFFF`).

This allows us to track millisecond-level precision within a 4-byte field, which is critical for the `latency_ms` and `jitter_ms` calculations in the Server.

3.3.3 Checksum (Offset 9)

The 2-byte checksum provides basic integrity verification.

- **Calculation:** `sum(byte_array) & 0xFFFF`
- **Scope:** Calculated over the Initial Header (9 bytes) + The Variable Payload.
- **Validation:** The server re-calculates the sum of received bytes (excluding the checksum field itself) and compares it to the received checksum value. Mismatches result in immediate packet drops.

3.4 Payload Formats

3.4.1 Heartbeat Payload

- **MsgType:** `0x02`
- **Length:** 0 Bytes
- **Content:** None. The header timestamp serves as the “last seen” proof.

3.4.2 Data Payload (Variable)

To support the optional batching requirement universally, all DATA payloads use a Count-Prefixed structure. Even a single reading is treated as a batch of size 1.

- **Structure:**
  - **Byte 0:** Count (`uint8`) - Number of readings (N) in this packet.
  - **Byte 1..M:** Array of Readings ($N \times 12$ bytes).
- **Reading Format (12 Bytes per Reading):** `!f f f` (Temperature, Humidity, Voltage).
- **Constraint:** The total payload size MUST NOT exceed 200 Bytes.

$$1 + (N \times 12) \leq 200 \implies N_{\text{max}} = 16 \text{ readings} \tag{1}$$

4 Communication Procedures

4.1 Session Lifecycle (Stateless)

Initiation: CTP operates using a strictly stateless "fire-and-forget" model with no connection handshake (no SYN/ACK). The Client begins transmitting immediately upon startup. When the Server receives the first valid packet from a new `DeviceID`, it allocates the necessary session state, including a jitter buffer and sequence tracking structures.

Termination: There is no explicit teardown (no FIN). If the Server receives no packets from a device for more than `LIVENESS_TIMEOUT` (10.0 s), that device is marked `OFFLINE`. Any remaining packets in the jitter buffer are immediately flushed and processed to avoid data loss.

4.2 Data Exchange

Transmission: In **Direct Mode**, Clients transmit readings at the configured `REPORTING_INTERVAL` (default: 1.0 s). In **Batching Mode**, readings are accumulated locally and transmitted as a single aggregate payload once the buffer size reaches `BATCH_SIZE`.

Keep-Alive: If the Client has no data to send for more than `HEARTBEAT_INTERVAL` (5.0 s), it sends a `HEARTBEAT` packet to prevent the Server from marking the device as inactive.

4.3 Error Handling & Recovery

Corruption: The Server verifies each packet's 16-bit checksum. Any checksum mismatch results in the packet being silently dropped.

Packet Loss: CTP does not support retransmission (ARQ). Missing sequence numbers are detected by the Server and logged as *Gaps*, providing visibility into packet loss without attempting recovery.

Duplicate Suppression: The Server maintains a history of the last 500 processed sequence numbers. If a packet arrives with a sequence number that already exists in this history, it is classified as a duplicate and its payload is ignored.

5 Reliability & Performance Features

5.1 Loss Tolerance Strategy

CTP follows a **loss-tolerant by design** philosophy. It intentionally does *not* implement Automatic Repeat Request (ARQ) or retransmission mechanisms. **Rationale:** In real-time telemetry systems, fresh data is more valuable than delayed reliability. Retransmitting stale sensor values wastes bandwidth that should instead carry current readings.

Gap Reporting: The Server detects packet loss by monitoring discontinuities in the sequence number:

$$Seq_{\text{current}} - Seq_{\text{last}} > 1$$

Any such gap is logged for later network-health diagnostics.

5.2 Jitter Compensation (Windowing)

Because UDP provides no ordering guarantees, the Server implements a **Reordering Window**. **Mechanism:** Incoming packets are not processed immediately. They are inserted into a Sort Queue and ordered by their timestamp.

Flush Threshold: To avoid unbounded buffering, the queue is flushed only when:

- its size exceeds `FLUSH_THRESHOLD` (default: 20 packets), or
- the session terminates.

This ensures the application layer receives events in correct temporal order even when packets arrive out of order.

5.3 Duplicate Suppression

To prevent double-counting caused by duplicated UDP deliveries or link-layer retries, the Server maintains a sliding history of the last 500 processed sequence numbers.

Rejection Logic: If an incoming `SeqNum` already exists in this history, the packet is classified as a **Duplicate** and its payload is discarded.

5.4 Timeout Selection

Timer values were chosen to balance bandwidth efficiency, responsiveness, and NAT stability:

- **Heartbeat Interval (5.0 s):** Low transmission overhead ($< 1\%$ duty cycle), while keeping NAT bindings alive.
- **Liveness Timeout (10.0 s):** Defined as:

  $$2 \times \text{HEARTBEAT\_INTERVAL}$$

  This allows the Server to tolerate a single lost heartbeat without incorrectly marking the device as `OFFLINE`.

6 Experimental Evaluation Plan

6.1 Measurement Methodology

Performance evaluation is conducted using an automated Linux-based test suite (`Run.sh`).

Network Emulation: The Linux `netem` module is used to inject controlled latency, jitter, loss, and duplication on the `lo` interface.

Verification: Each experiment captures a `.pcap` trace using `tcpdump` to validate header correctness and on-wire behavior.

Timing: Server-side processing time is measured using `time.perf_counter()` to record per-packet computational cost with microsecond precision.

6.2 Key Metrics

For each received packet, the Server logs the following metrics to a CSV file:

- **Latency (ms):**
$$T_{\text{arrival}} - T_{\text{sent}}$$
based on synchronized clocks or relative timestamps.
- **Jitter (ms):**
$$|Latency_{\text{current}} - Latency_{\text{prev}}|$$
- **Sequence Gaps:** Missing packets inferred from:
$$Seq_{\text{current}} - Seq_{\text{last}} - 1$$
- **Duplicate Rate:** Percentage of packets rejected by the Duplicate Suppression logic.
- **CPU Cost:** Time required to parse, validate, reorder, and log a packet.

6.3 Test Scenarios & Netem Configurations

The following scenarios are executed automatically by `Run.sh` to evaluate key protocol behaviors:

Scenario	Netem Command	Objective
Baseline	none	Establish reference metrics for latency and throughput (1 s reporting interval).
Packet Loss	`loss 5%`	Verify that the Server detects Gaps without interrupting operation.
Heavy Loss	`loss 30%`	Stress-test gap detection logic under severe degradation.
Jitter	`delay 100ms 10ms`	Validate that the Reordering Buffer correctly restores timestamp order.
Duplication	`duplicate 20%`	Confirm that Duplicate Suppression filters redundant packets.
Batching	none	Evaluate throughput efficiency when sending packets containing 5 aggregated readings.

7 Example Use Case Walkthrough

7.1 Narrative Description

Startup ($T = 0$): Device 101 powers on and initializes its sequence counter to 1.

Data Transmission ($T = 1000$ ms): The sensor reads 25.5°C and sends a `DATA` packet with $Seq = 1$.

Packet Loss ($T = 2000$ ms): The sensor reads 25.6°C and sends $Seq = 2$, but the packet is dropped due to simulated network loss (5%).

Gap Detection ($T = 3000$ ms): The sensor transmits $Seq = 3$. The Server receives it, detects that the previous packet was $Seq = 1$, and logs a Gap of 1 packet.

Idle & Heartbeat ($T = 8000$ ms): The sensor becomes idle. After 5 seconds of silence, it sends a `HEARTBEAT` packet with $Seq = 4$ to maintain liveness.

7.2 Trace Log (Server CSV Output)

Example CSV entries corresponding to the scenario:

Device ID	Seq	Msg Type	Gap?	Latency (ms)	Notes
101	1	DATA	0	2.05	Initial Packet. Session Created.
101	3	DATA	1	1.98	Gap Detected (Seq 2 lost).
101	4	HEARTBEAT	0	1.50	Keep-alive received.

8 Limitations & Future Work

8.1 Current Limitations

Security: The protocol lacks authentication and encryption. Device IDs and source IPs can be spoofed, enabling injection of false telemetry.

Scalability: The Server uses a Python dictionary for state tracking. While adequate for a small lab setup (4 clients), it may not scale to thousands of devices due to memory and locking constraints.

Congestion Control: Clients transmit at a fixed `REPORTING_INTERVAL` and do not adjust rate in response to congestion. This risks contributing to network overload.

Clock Synchronization: Latency measurements assume synchronized clocks. Without NTP or PTP, one-way delay calculations in real-world deployments may be inaccurate.

8.2 Future Work

Security Layer: Add HMAC-SHA256 authentication to optional header fields to verify packet origin and prevent spoofing.

Adaptive Rate Limiting: Enhance the Client to dynamically adjust its `REPORTING_INTERVAL` based on observed packet loss (via a new ACK message type).

Dynamic Buffering: Replace the fixed-size Reordering Buffer (20 packets) with a dynamically sized jitter buffer that scales with measured arrival-time variance.

9 References

1. Tanenbaum, A. S., & Wetherall, D. *Computer Networks*, 5th or 6th Edition, Pearson, 2011/2021.

2. Stallings, W. *Data and Computer Communications*, 10th Edition, Pearson, 2014.

3. Kurose, J. F., & Ross, K. W. *Computer Networking: A Top-Down Approach*, 8th Edition, Pearson, 2021.

4. Postel, J. "User Datagram Protocol," RFC 768, August 1980.

5. IEEE Computer Society. *IEEE Standard for Floating-Point Arithmetic*, IEEE 754-2019.