IoTStream V1 - Mini-RFC

Contents

1. Introduction

The goal of this project is to design a lightweight IoT telemetry protocol that allows small, resource-constrained sensors to send periodic readings such as temperature, humidity or voltage to a central collector. Normal protocols like HTTP may be too heavy for simple sensors, so IoTStream v1 aims to provide a compact, efficient, and loss-tolerant alternative that uses UDP.

This protocol is intended for environments where:

- Devices have limited processing power and memory.
- Network bandwidth is low or unreliable.
- Small data updates are sent periodically.

By using UDP and a compact binary header, the protocol minimizes overhead and ensures faster data delivery even under mild packet loss.

1.1. Use Case Overview

In a typical setup, multiple sensor nodes (clients) periodically measure temperature, humidity or voltage and transmit the readings to a collector server on the same local network or over the Internet. Each sensor identifies itself with a unique Device ID and attaches metadata such as sequence number, timestamp, and message type to every transmission.

Example:

1. The sensor starts and sends an INIT message to announce itself.
2. It then periodically sends DATA messages containing readings.
3. When no new data is available, it sends a HEARTBEAT message to show it's still alive.

The collector receives these messages, logs them, and monitors data continuity.

1.2. Design Goals

IoTStream v1 is designed to be:

- **Lightweight** — total UDP payload ≤ 200 bytes.
- **Loss-tolerant** — no retransmission, but detect missing packets.
- **Compact header** ≤ 12 bytes.

- **Support batched data** (multiple readings per packet).
- **Provide timestamped readings** for reordering and analysis.

1.3. System Assumptions and Constraints

Parameter	Description	Value / Limit
Transport Layer	Protocol used for message delivery	UDP
Max UDP Payload	Application data limit per packet	≤ 200 bytes
Header Size	Binary header for metadata	≤ 12 bytes
Reporting Intervals	Frequency of sensor reports	1s, 5s, 30s
Message Types	Types supported by the protocol	INIT, DATA, HEARTBEAT
Loss Tolerance	Network reliability requirement	Must handle 5% random loss

Table 1: System parameters and constraints

1.4. Design Justification

IoTStream V1 takes a minimalist approach compared to established protocols like MQTT-SN. While MQTT-SN provides publish-subscribe messaging with quality of service guarantees and bidirectional communication, our protocol eliminates these features to achieve maximum simplicity and minimal overhead. MQTT-SN requires connection establishment, topic registration, and acknowledgment mechanisms that add complexity and bandwidth consumption. Our unidirectional design with fixed message types removes the need for topic management entirely.

The important difference lies in our acceptance of packet loss as a design feature rather than a problem to solve. MQTT-SN implements three quality of service levels including assured delivery with retransmission, which introduces state management overhead and latency. IoTStream V1 achieves below 1 millisecond processing times and 32-byte total packet sizes by omitting reliability mechanisms entirely. For high-frequency telemetry where individual readings have limited value and the next update arrives within seconds, this tradeoff favors throughput and simplicity over guaranteed delivery.

Our 12-byte fixed header contrasts with MQTT-SN's variable-length headers that accommodate topic names and message identifiers for routing flexibility. This makes IoTStream V1 unsuitable for dynamic topic-based routing but ideal for scenarios where each device reports a predefined data type to a single collector. The protocol serves applications that prioritize minimal resource consumption and accept occasional data gaps over the routing flexibility and delivery guarantees that MQTT-SN provides.

2. Protocol Architecture

This section describes the structure of the IoTStream v1 protocol, including the entities that will communicate with each other, the flow of messages between them, and the finite-state machine (FSM) that defines their states and behavior.

2.1. Entities

The software architecture determines how the pieces of the application will interact with other. We use the client-server architecture with the following entities:

1. **IoT Sensors (client)**
   - They run the client process that takes measurements and periodically sends data to a central collector (server)
   - No per-packet retransmission
2. **Central Collector (server)**
   - It runs the server process
   - It collects and processes all the data from all the sensors.
   - Per-device state is maintained.

The protocol is an application-layer protocol designed to be lightweight, efficient, and robust, operating directly over UDP.

UDP is a connectionless (no handshaking required) transport protocol that offers minimal services, and unreliable data transfer.

Because we rely on UDP, the protocol implements features that enable us to still maintain state per each device.

2.2. Sequence Flow

The message exchange only flows from the sender to receiver. The sensor initiates all communication, sending packets based on a configured `reporting_interval`.

The following is an example of a typical sequence flow:

1. Sensor: sensor turns on
2. Sensor: `reporting_interval` is set
3. Sensor: sensor reads data
4. Sensor: `reporting_interval` expires
5. Sensor: determine if `DATA` or `HEARTBEAT` is needed and construct a packet (including other header fields)
6. Sensor: send packet over the UDP transport layer and increment sequence number
7. Collector: receive packet and extract header and payload
8. Collector: check the device ID and compare last sequence number to current sequence number
   - If duplicate, set `duplicate_flag`
   - If gap detected, set `gap_flag`
9. Collector: log all received data to a CSV file into the correct fields

2.3. Finite-State Machine (FSM)

The finite-state machines for the sensor (sender) and the collector (receiver) describe the states that they can be in, and the required actions based on certain events.

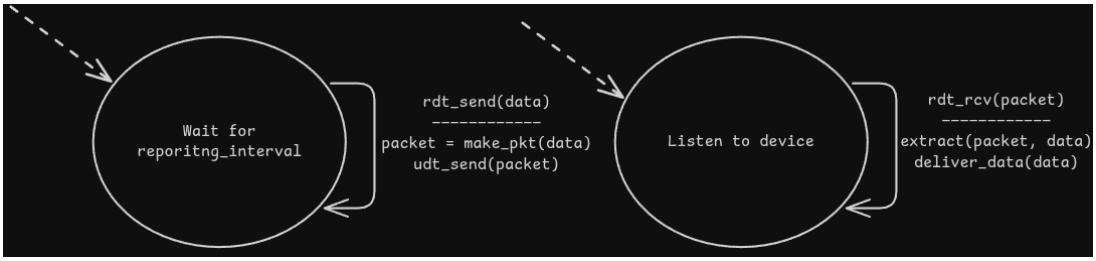

Figure 1: Finite-State Machine diagram for sensor and collector

Both the sensor and collector exist in a single state, that of waiting. The sensor waits until the `reporting_interval` expires, then sends the data to the transport layer where a packet is made. If there was no sensor readings, then the message type is set to `HEARTBEAT`, but if there was sensor readings, the message type is `DATA`. The packet is then sent over UDP.

As for the collector, it waits until it receives a packet from the transport layer. Once received, it extracts the header and payload and sends it to the application layer. Duplication and sequence gaps are checked and the data is logged to maintain state.

2.4. Sessionless Telemetry vs. Sessionful File Transfer

The IoTStream v1 is a sessionless and loss-tolerant protocol. We opt for a sessionless protocol because it aligns with our constraints while optimizing efficiency and minimizing connection overhead. This is appropriate for periodic telemetry, unlike file transfer where a sessionful protocol would be more appropriate. Here is a simple comparison between our protocol and a sessionful protocol like HTTP:

	IoTStream V1	**HTTP**
Transport Layer Protocol	UDP	TCP
Connection State	Connectionless	Connection-oriented
Data Integrity	Loss-tolerant	Reliable data transfer
Overhead	Low	High
Use case Example	Periodic telemetry	File transfer

Table 2: Protocol comparison

3. Message Formats

Each IoTStream v1 packet begins with a fixed 12-byte binary header.

Field	**Size (bytes)**	**Description**
Version	1	Protocol version
MsgType	1	Determine between `INIT`, `DATA`, and `HEARTBEAT`
Device ID	2	Unique ID assigned to each sensor
Sequence Number	2	Increments with each packet from the same device (used for loss detection)
Timestamp	4	UNIX time in seconds when the packet is sent
Batching flag	1	Determines number of sensor readings in packet
Checksum	1	8-bit checksum for header integrity

Table 3: Header format

3.1. Payload Format

Each DATA packet can carry five readings from a single sensor.

Field	**Type**	**Size (bytes)**	**Description**
Reading 1	`float`	4	First recorded measurement
Reading 2	`float`	4	Second recorded measurement
Reading 3	`float`	4	Third recorded measurement
Reading 4	`float`	4	Fourth recorded measurement
Reading 5	`float`	4	Fifth recorded measurement

Table 4: Payload format

Total Payload Size: 5 × 4 = 20 bytes

Total Packet Size: Header (12 bytes) + Payload (20 bytes) = 32 bytes

3.2. Encoding Format

The header is encoded using the Python struct format string: `!BBHHIBB`

```
struct.pack('!BBHHIBB', version, msgtype, device_id, seq_num, timestamp, batching_flag, checksum)
```

Field	Byte Offset	Size (bytes)	Type	Example Representation
Version	0	1	Unsigned char	B
MsgType	1	1	Unsigned char	B
Device ID	2	2	Unsigned short	H
Sequence Number	4	2	Unsigned short	H
Timestamp	6	4	Unsigned int	I
Batching flag	10	1	Unsigned char	B
Checksum	11	1	Unsigned char	B

Table 5: Encoding format specification

4. Communication Procedures

4.1. Session Start

This protocol uses UDP. This protocol is sessionless and hence there is no handshake.

Server start:

- A socket is created and assigned port 5005.
- Enter receive loop and listen for incoming packets on all network interfaces.

Client boot:

1. A client socket is created without binding (port assigned by OS).
2. Send an `INIT` packet with sequence number 0 to announce the device.

4.2. Normal Data Exchange

Client enters periodic reporting phase. During each reporting interval a timestamp is generated, and the packet type is selected randomly. `data` type is selected with probability 80%, and `heartbeat` with probability 20%. `seq_num` is then incremented by 1. For the `data` packet, reading are collected and header added. For `HEARTBEAT` packet, the client sends only the header with no payload and `batching_flag=0`. A simple checksum is calculated by summing all header bytes and taking the result modulo 256. Finally, the packet is sent to the server.

Packet Type	Message Type	Probability	Payload	Purpose
`DATA`	1	80%	Batch of sensor readings	Measurement reading transmission
HEARTBEAT	2	20%	Empty	Device is still alive

Table 6: Message type distribution

When the server receives the packet, it records the time of arrival and starts a CPU timer to measure processing time. Then, it unpacks the packet with the format `!BBHHIBB`, and validates the packet size. The server then write a CSV row and logs the contained data for later analysis.

Example DATA packet exchange:

1. Client waits 943ms (1000ms interval with negative jitter)
2. Client generates five readings: [29.37, 23.53, 22.61, 22.04, 22.99]
3. Client encodes readings as 20 bytes of IEEE 754 floats
4. Client constructs header with device ID 1001, sequence 1, timestamp 245642102, message type 0x01, batch flag 5
5. Client calculates checksum over header bytes, yielding 0xf2
6. Client transmits 32-byte packet (12-byte header plus 20-byte payload)
7. Server receives packet at 1765477200.758614
8. Server verifies sequence 1 not in `seen_seqs` set for device 1001
9. Server adds sequence 1 to `seen_seqs` set
10. Server compares sequence 1 against highest_seq 0, detects no gap
11. Server updates highest_seq to 1
12. Server logs entry with `duplicate_flag=0`, `gap_flag=0`, `cpu_ms_per_report=0.0208`

4.3. Error Recovery

Error recovery is simple and lightweight which is suitable. This is a loss-tolerant protocol. There is no acknowledgments, retransmissions, or error correction. Error recovery is handled by the collector.

Packet loss is detected using sequence numbers, but not corrected by retransmission. If the server detects a gap in the sequence numbers, it sets the `gap_flag` in the CSV log.

Duplicate packets are identified by checking whether the sequence number already exists in the device's `seen_seqs` set. Duplicates receive `duplicate_flag=1` and are not processed further, preventing metric corruption. The `seen_seqs` set is capped at 2000 entries per device to prevent memory exhaustion during long-running sessions.

Malformed packets that are too small or fail header unpacking are silently discarded. The server continues processing valid packets from other devices, prioritizing availability over strict validation. Thread locks protect device state modifications to prevent race conditions when multiple packets arrive concurrently.

Example gap detection sequence:

1. Server successfully receives and logs sequence 5 from device 1001
2. Server updates `highest_seq` to 5
3. Network drops packets with sequences 6 and 7
4. Client transmits sequence 8 with timestamp 245648943
5. Server receives sequence 8 at 1765477207.600495
6. Server checks: sequence 8 not in `seen_seqs` set (new packet)
7. Server compares: 8 exceeds `highest_seq` 5 by more than 1
8. Server sets `gap_flag=1` in log entry
9. Server adds sequence 8 to `seen_seqs` and updates `highest_seq` to 8
10. Post-analysis reveals gap count of 2 (sequences 6 and 7 missing)

4.4. Shutdown

The client operates continuously until manually interrupted via `Ctrl+C`, at which point it exits the transmission loop and closes the UDP socket. The server continues listening indefinitely and maintains device state in memory until the server process itself terminates. When the server terminates, also via `Ctrl+C`, it exits the listening loop, and closes the UDP socket.

5. Reliability & Performance Features

5.1. Reliability

This protocol is designed to be loss-tolerant and does not implement packet retransmission. This is a unidirectional communication model: the client sends data without expecting any acknowledgment from the server. There is not much point in retransmitting a lost sensor reading at a later time especially given the high reporting frequency.

Lost packets are detected using sequence numbers. The collector compares the new sequence number with the last received sequence number. If there's a gap, then a `gap_flag` is set to log that data was missing. If the sequence number indicates a duplicate packet, a `duplicate_flag` is set, and the payload is ignored.

We do not use stop-and-wait and so there are no timers. All this minimizes overhead and latency and simplifies the implementation. The client just transmits data at fixed intervals with some small random jitter of around plus or minus ten percent to prevent network synchronization effects when multiple devices operate simultaneously.

5.2. Performance

We optimize performance through payload batching. Rather than transmitting a single sensor reading per packet, the client aggregates multiple readings into a single packet and transmitting that. We use a default batch size of five. This allows us reduce the overhead per packet by having one header per five readings instead of just one.

Server-side processing has been optimized for minimal CPU consumption. Each packet is processed in under one millisecond, with the actual processing time logged in the `cpu_ms_per_report` column in CSV. Thread-safe device state management using locks ensures correct operation when handling concurrent packets from multiple devices without introducing significant contention delays. The `seen_seqs` set is limited at two thousand entries per device to bound memory usage during extended operation periods.

This protocol prioritizes simplicity, low latency, and minimal resource consumption over guaranteed delivery. The design is appropriate for applications where occasional data loss is acceptable and where the overhead of reliability mechanisms would be disproportionate to the value of individual data points.

6. Experimental Evaluation Plan

Specify baselines, metrics, measurement methods, and how to simulate network conditions. Include exact netem commands for Linux and alternatives for other OSes. Provide scripts to automate runs and collect results.

6.1. Test Scenarios

The evaluation consists of three scenarios using multiple reporting intervals. We checked normal operation at 1-second, 5-second, and 30-second intervals under ideal conditions to see how it performs on different sensor sensor types. The packet loss scenario applies 5% random drop at 1-second intervals to ensure good handling of missing data. The latency scenario is 100ms delay with 10ms jitter at 1-second intervals to verify timestamp handling and gap detection under significant network delays.

6.2. Metrics and Measurement

Average end-to-end latency measures how long packets take to get from client to server by comparing their timestamps, handling 32-bit wraparound by adding 2^{32} when negative differences exceed one billion milliseconds. Packet duplication rate calculates the proportion of duplicate sequence numbers received. Gap count identifies missing sequence numbers after sorting by client timestamp. We also track how much CPU time the server uses per packet. Each scenario runs five independent runs, with results aggregated as minimum, median, and maximum values to characterize typical performance and bounds.

6.3. Network Condition Simulation

On Linux, we use NetEm through the tc utility on the loopback interface. Baseline requires no configuration. For packet loss, we apply `tc qdisc add dev lo root netem loss 5%`. Latency applies `tc qdisc add dev lo root netem delay 100ms 10ms`. Cleanup between scenarios uses `tc qdisc del dev lo root`. Root privileges are required.

On Windows, we use Clumsy with filter `udp and udp.DstPort == 5005`. Users manually configure Drop at 5.0% for loss scenarios or Lag at 100ms for latency scenarios when prompted by the automation script.

Packet capture uses `sudo tcpdump -i lo -U -w output.pcap udp port 5005` on Linux or `tshark -i INTERFACE_NUMBER -f "udp port 5005" -w output.pcap` on Windows after identifying the interface with `tshark -D`.

6.4. Automation and Execution

The `run_experiments.py` script orchestrates all testing by detecting the operating system, configuring network conditions, launching server and client processes, collecting results, and analyzing CSV outputs. The script generates `results_SCENARIO_INTERVALs_runN.csv` files and `trace_SCENARIO_INTERVALs_runN.pcap` captures for each run. Console output displays aggregated statistics tables showing minimum, median, and maximum values across runs. Total execution time approximates 25 minutes for all scenarios. Linux users execute with `sudo python3 run_experiments.py` while Windows users run `python run_experiments.py` and follow prompts for Clumsy configuration.

7. Example Use Case Walkthrough

Provide an end-to-end trace example (timestamps, messages sent/received) for a single session. Include pcap excerpt (as appendix) and explanation.

7.1. Session Overview

This section traces a complete protocol session demonstrating initialization, data transmission, and heartbeat messages. Device 1001 reports to server `127.0.0.1:5005` with one-second intervals and batch size five. The trace spans the first ten seconds, capturing eleven packets that illustrate all message types.

7.2. Message Flow

The client initiates at client timestamp 245641159 (milliseconds) by sending an `INIT` packet (sequence 0, message type 0x00) containing only the 12-byte header. The server receives it at arrival time 1765477199.815354 (Unix timestamp) with approximately 0.656 milliseconds latency.

At client timestamp 245642102, the client transmits the first DATA packet (sequence 1, message type 0x01) containing five temperature readings: 29.37, 23.53, 22.61, 22.04, and 22.99 degrees Celsius, encoded as 20 bytes of IEEE 754 floats with batching flag set to 5. The server receives it at 1765477200.758614 with 0.656 milliseconds latency and 0.0208 milliseconds processing time.

Sequence 2 arrives at server time 1765477201.725278 carrying readings 23.42, 26.75, 26.51, 28.66, and 24.84. At client timestamp 245644003, the client randomly selects a `HEARTBEAT` message (sequence 3, message type 0x02) with no payload beyond the 12-byte header. The server receives it at 1765477202.659554 with similar sub-millisecond latency.

The pattern continues with `DATA` packets at sequences 4 (readings: 25.83, 24.20, 22.63, 26.79, 25.93) and another HEARTBEAT at sequence 5. Sequences 6, 7, and 8 contain `DATA` packets with readings including [21.12, 24.94, 24.09, 29.98, 22.57], [20.83, 26.72, 20.19, 20.23, 23.14], and [21.21, 22.50, 23.01, 23.29, 20.57] respectively. Sequence 9 sends a `HEARTBEAT`, followed by `DATA` at sequence 10 with readings [28.01, 29.91, 25.00, 28.39, 23.09].

All eleven packets arrive in order with no loss or duplication. The `duplicate_flag` remains 0 for all packets, indicating no retransmissions or network duplication occurred. The `gap_flag` remains 0 for all packets, confirming continuous sequence number progression without missing packets.

7.3. Server Processing

The server validates each packet meets the 12-byte minimum, unpacks the header using format string "!BBHHIBB", and maintains per-device state tracking the highest sequence number and previously seen sequences. For each packet, the server checks if the sequence number was seen before (duplicate flag) and whether it exceeds the highest by more than one (gap flag). The server logs seven fields to CSV: `device_id`, seq, timestamp (32-bit client value), `arrival_time` (server Unix timestamp), `duplicate_flag`, `gap_flag`, and `cpu_ms_per_report`.

8. Limitations & Future Work

8.1. Current Limitations

The protocol's reliance on UDP without retransmission makes it unsuitable for applications requiring reliable data transfer. Packet loss is permanent, as the client has no mechanism to detect lost packets or initiate retransmission. Critical sensor data could be lost without detection beyond gap logging. The unidirectional communication model prevents the server from providing feedback to clients.

We don't have security mechanisms. This is a major vulnerability in the current implementation. The protocol lacks authentication which can make it vulnerable to spoofed device identities. The protocol transmits all headers and payloads in plaintext, allowing any actor on the network to inspect sensitive sensor data. Moreover, the system lacks authentication. The server accepts data based solely on a 2-byte `device_id`, making it trivial for malicious entities to spoof device identities and inject false metrics. Without encryption, sensor data can be intercepted in transit, making the protocol inappropriate for sensitive data.

8.2. Future Enhancements

Future versions could implement optional acknowledgment where clients request confirmation for critical packets.

Implement message authentication, encryption, and user management for real deployments.

A GUI application or website could also be build to visualize sensor data and system health in real-time. This could provide real-time graphs of packet arrival rates, sequence gap patterns, and duplicate occurrences.

Moreover, replacing CSV logging with a database enables us to organize the data in order to easily run queries on them efficiently. For example, searching and sorting historical data, calculating averages, totals, and trends over time. It would also provide scalability, supporting hundreds of devices sending data constantly.

9. References

Postel, J. (1980). **User Datagram Protocol** (RFC 768). Internet Engineering Task Force. https://www.rfc-editor.org/rfc/rfc768.html

Python Software Foundation. (n.d.). **Python Standard Library Documentation: socket, struct, subprocess**. Retrieved from https://docs.python.org/3/library/