Mini-RFC: ECHOP v1

(Efficient Compact Header Optimization Protocol)

Compact UDP IoT Telemetry Protocol

Presented by:

22P0150 Abdelrahman Mostafa Ali Eldin

22P0162 Habiba Sherif Abdelaziz

22P0210 Mohamed Ashraf Mohamed

22P0112 Roger Sherif Selim Salama

22P0021 Rowaida Emad Khalaf

22P0139 Somaya Ahmad Galal

Presented to:

Dr. Ayman Mohamed Bahaa Eldin

Eng. Rafik Tamer Magdy

1. Introduction

ECHOP v1 is a lightweight, sessionless application-layer protocol designed for constrained IoT sensors to periodically transmit telemetry data (e.g., temperature, humidity, voltage) to a central collector over UDP.

The protocol is designed to operate in bandwidth-constrained environments where TCP overhead is undesirable. It features a compact 10-byte fixed header, a "Smart Compression" payload encoding scheme to dynamically optimize for integer vs. floating-point precision, and a custom lightweight stream cipher for basic confidentiality. While UDP is inherently unreliable, ECHOP v1 implements a receiver-driven reliability mechanism using Negative Acknowledgments (NACKs) to recover lost packets without the overhead of per-packet ACKs.

1.1 Assumptions and Constraints

- **Transport:** UDP (User Datagram Protocol).
- **MTU:** Designed for payloads under 200 bytes (application layer).
- **Loss Tolerance:** The protocol must detect gaps and recover from up to 5% random packet loss.
- **Timestamps:** Devices are assumed to have a synchronized clock (UNIX epoch) for meaningful timestamp analysis.

2. Protocol Architecture

2.1 Entities

2.1.1 Client (Sensor Node)

The client is responsible for:

- Reading sensor data and buffering it into batches.
- Applying "Smart Compression" (int32 vs float64) to reduce payload size.
- Encrypting the payload using a session-based XOR stream cipher.
- Sending MSG_INIT on startup and MSG_DATA periodically.
- Maintaining a history of sent packets to fulfill retransmission requests (NACKs).
- Sending HEART_BEAT messages during idle periods to maintain liveness.

2.1.2 Server (Collector)

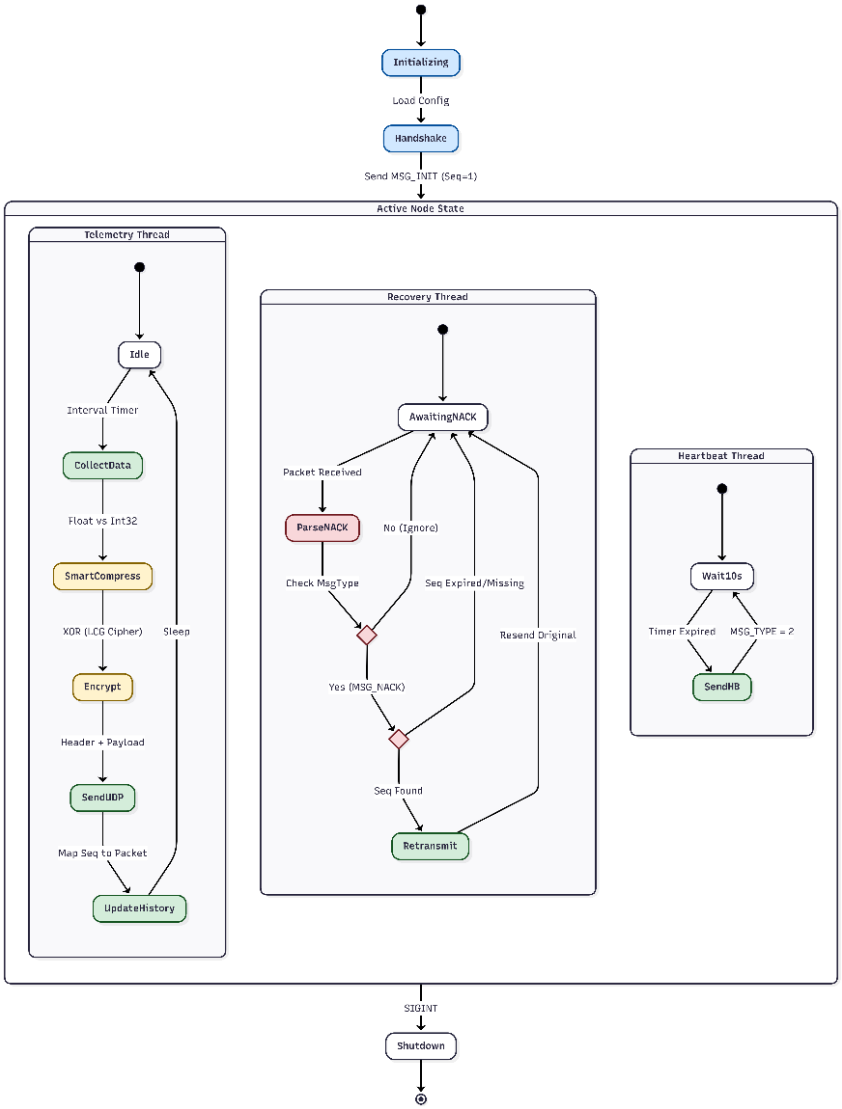
The server is responsible for:

- Listening for incoming UDP datagrams.
- Validating the 1-byte checksum.
- Tracking sequence numbers per DeviceID to detect gaps or duplicates.
- Scheduling and sending NACK_MSG requests to clients when gaps are detected.
- Reordering packets based on device timestamps (using a jitter buffer) before final processing.
- Logging metrics (loss rate, latency, throughput) to CSV.

2.2 Finite State Machine

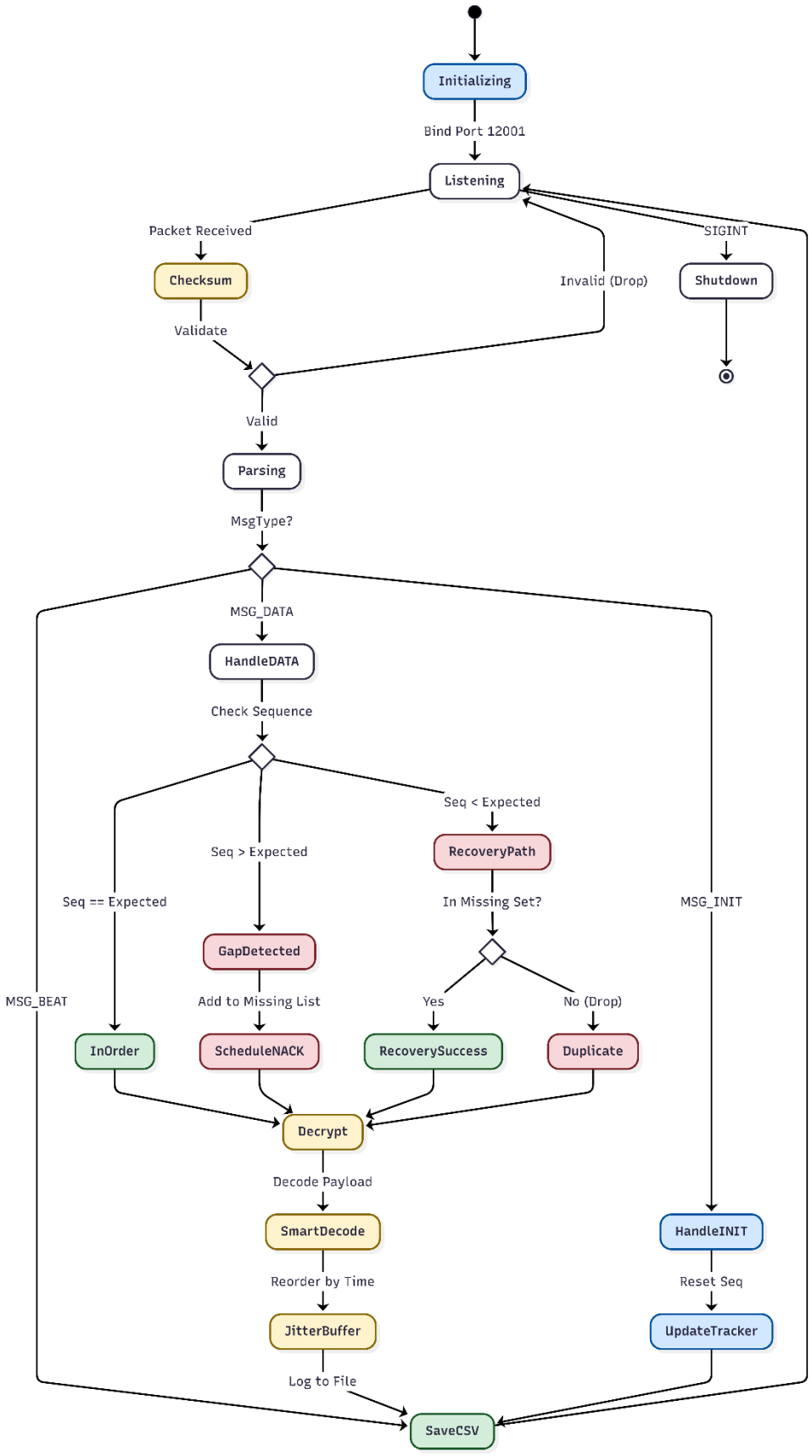
2.2.1 Client FSM

- **Initialization:** Load config → Send MSG_INIT.
- **Active:** Loop → Read Sensors → Send MSG_DATA → Sleep.
- **Error Handling:** Listen thread receives NACK → Retransmit from History.



2.2.2 Server FSM

- **Listening:** Await UDP packet.
- **Processing:** Parse Header → Verify Checksum → Decrypt.
- **State Update:**
  - Seq == Expected: Accept.
  - Seq > Expected: Gap detected → Schedule NACK.
  - Seq < Expected: Duplicate detected → Drop.


Initializing
Bind Port 12001
Listening
Packet Received
Checksum
Validate
Invalid (Drop)
SIGINT
Shutdown
Valid
Parsing
MsgType?
MSG_DATA
HandleDATA
Check Sequence
Seq == Expected
Seq > Expected
Seq < Expected
MSG_INIT
MSG_BEAT
InOrder
GapDetected
Add to Missing List
ScheduleNACK
RecoveryPath
In Missing Set?
Yes
No (Drop)
RecoverySuccess
Duplicate
Decrypt
Decode Payload
SmartDecode
Reorder by Time
JitterBuffer
Log to File
HandleINIT
Reset Seq
UpdateTracker
SaveCSV

3. Message Formats

All multi-byte fields are encoded in Big-Endian (Network Byte Order). The header is exactly **10 bytes**.

3.1 Header Structure

Byte Offset	Field	Size (Bits)	Description
0	DeviceID	4	Unique Device Identifier (0-15).
0	BatchCount	4	Number of readings in batch (0-15).
1-2	SeqNum	16	Monotonically increasing sequence number.
3-6	Timestamp	32	UNIX Epoch timestamp (seconds).
7	ProtoVer	2	Protocol Version (Fixed: 1).
7	MsgType	2	Message Type (0-3).
7	Reserved	2	Reserved (0).
7	ms_high	2	Upper 2 bits of millisecond counter.
8	ms_low	8	Lower 8 bits of millisecond counter.
9	Checksum	8	ASCII Sum Modulo 256 of Header + Payload.

3.1.1 Struct Packing Format

struct.pack('!B H I B B B', [DeviceID + BatchCount], SeqNum, Timestamp, [ProtoVer + MsgType + ms_high], ms_low, Checksum)

3.2 Message Types

Value	Name	Description	Payload Content
0	MSG_INIT	Session Start	Empty**.** **BatchCount holds Unit Code.**
1	MSG_DATA	Telemetry	Encrypted, Smart-Compressed sensor values.
2	HEART_BEAT	Liveness	Empty.

3	NACK_MSG	Retransmission	ASCII String: "DeviceID:MissingSeq".

3.3 Smart Payload Compression

To save bandwidth, floating-point sensor readings are analyzed before transmission:

1. Value is scaled by 10^6.
2. If the result fits in a 4-byte signed integer, it is sent as int32.
3. If it overflows, it is sent as an 8-byte double (float64).
4. **Payload Format: [FlagCount (1Byte)] + [FlagIndices (N Bytes)] + [Data Bytes].**
   Flags indicate which indices in the batch required 8-byte precision.

3.4 Encryption

Confidentiality is provided by a lightweight XOR Stream Cipher.

- Key Generation: An LCG (Linear Congruential Generator) is seeded with (DeviceID << 16) ^ SeqNum ^ Secret.
- Process: The keystream is XORed with the payload bytes.
- Properties: Symmetric (Encryption == Decryption) and stateless.

4. Communication Procedures

4.1 Session Initialization

- Client starts and reads configuration.
- Client sends MSG_INIT with Seq=1.
- Server receives INIT, initializes tracking for that DeviceID, and expects Seq=2 next.

4.2 Data Transmission

- Client buffers N readings.
- Client compresses and encrypts the payload.
- Client constructs the header with current Seq and Timestamp.
- Client computes the 1-byte checksum over Header + Payload.

- Packet is sent. Client stores packet in sent_history map.

4.3 Error Recovery (NACK)

1. **Detection:** Server receives Seq=5 but expected Seq=3. Gaps [3, 4] are recorded.
2. **Scheduling:** Server schedules a NACK task with a delay (e.g., 1000ms) to account for packet reordering.
3. **Request:** If the gap persists, Server sends NACK_MSG containing "DeviceID:3".
4. **Retransmission:** Client receives NACK, looks up Seq=3 in history, and re-sends the exact original packet.

4.4 Shutdown

- Client: Finishes interval loop → Closes socket.
- Server: On signal (SIGINT), flushes the reorder buffer to disk and prints the final summary statistics.

5. Reliability and Performance Features

5.1 Checksum

A simple 8-bit checksum (Sum Modulo 256) is used to detect transmission corruption. Packets with invalid checksums are silently discarded, eventually triggering a sequence gap detection.

5.2 Timestamp Reordering

UDP does not guarantee order. The server implements a Jitter Buffer (Min-Heap):

1. Incoming packets are inserted into the heap keyed by DeviceTimestamp.
2. Packets are held for a guard time (e.g., 150ms).
3. Packets are popped from the heap in strict temporal order for analysis (iot_device_data_reordered.csv).

5.3 Batching

By grouping multiple readings (defined by BatchCount) into a single UDP frame, header overhead is amortized, reducing the total bytes transmitted per reading.

6. Experimental Evaluation Plan

The protocol is evaluated using Linux tc-netem to simulate network impairments.

6.1 Scenarios

1. Baseline: Localhost (Loopback), 0% loss, 0ms delay. Validation of functional correctness.
2. Lossy Network: tc qdisc add dev lo root netem loss 5%. Validates NACK recovery logic.
3. High Latency/Jitter: tc qdisc add dev lo root netem delay 100ms 10ms. Validates the server's reordering buffer.

6.2 Metrics

- Packets Received vs. Expected: Delivery Ratio.
- Duplicate Rate: Efficiency of the retransmission logic.
- Sequence Gap Count: Number of unrecovered losses.
- CPU Time per Report: Computational overhead of the Python implementation.

7. Example Use Case Walkthrough

Scenario: Device 1 sends temperature data every 1 second.

1. T=0.0s: Client sends INIT (Seq=1). Server registers Device 1.
2. T=1.0s: Client sends DATA (Seq=2, Payload="25.5C"). Server receives OK.
3. T=2.0s: Client sends DATA (Seq=3). Packet is dropped by network.
4. T=3.0s: Client sends DATA (Seq=4).
5. T=3.01s: Server receives Seq=4. Detects Gap (Missing 3). Schedules NACK.
6. T=3.35s: Server NACK timer expires. Sends NACK_MSG payload "1:3".
7. T=3.36s: Client receives NACK. Resends DATA (Seq=3).
8. T=3.37s: Server receives Seq=3. Gap resolved.
9. T=4.0s: Client sends DATA (Seq=5).

8. Limitations and Future Work

8.1 NACK Scheduling Delay and Network Predictability

A core limitation of the current reliability model is the NACK_DELAY_SECONDS parameter (currently set to 0.35s).

- **Mechanism:** The server does not send a NACK immediately upon detecting a sequence gap. Instead, it schedules a NACK after a 1000ms window to account for network jitter and out-of-order delivery.
- **Tunability:** This parameter is a "best-guess" heuristic. If the network delay/jitter exceeds 1000ms, the protocol may trigger "false NACKs" for packets that were simply delayed, leading to redundant retransmissions. Conversely, on high-speed links, this delay increases recovery latency.
- **Future Work:** Implementing an Adaptive NACK Timer that calculates the Moving Average of the Round Trip Time (RTT) would allow the protocol to predict the optimal wait time dynamically.

8.2 Security and Integrity

1. The LCG-XOR cipher is not cryptographically secure and is vulnerable to known-plaintext attacks. DTLS should be used for production.
2. A 1-byte sum-based checksum is vulnerable to certain multi-byte errors. Upgrading to CRC-16 or CRC-32 is recommended for high-integrity industrial use.

8.3 NACK Flooding

In high-loss environments (>20%), the NACK mechanism may cause congestion. A selective ACK (SACK) approach could be more efficient (requesting multiple missing packets in a single NACK message).

9. References

- https://docs.python.org/3/library/socket.html
- https://man7.org/linux/man-pages/man8/tc-netem.8.html
- https://docklight.de/manual/checksum_specification.html?utm_source=chatgpt.com
- Postel, J. (1980). User Datagram Protocol. RFC 768.
-
- IEEE Computer Society (2008). IEEE Standard for Floating-Point Arithmetic. IEEE 754-2008.
-

- Montenegro, G., et al. (2007). Transmission of IPv6 Packets over IEEE 802.15.4 Networks. RFC 4944.