1 Introduction

This report documents the design, implementation, and evaluation of **ECHOP v1** (Efficient Compact Header Optimization Protocol). ECHOP v1 is a sessionless, application-layer protocol built on top of UDP, designed specifically for constrained sensor nodes. The protocol features a custom 10-byte header, a "Smart Compression" scheme for payload optimization, and a receiver-driven Negative Acknowledgment (NACK) mechanism to ensure reliability without the latency penalties of stop-and-wait ARQ systems.

2 Implementation Architecture

2.1 Transport Layer Selection

UDP was selected as the underlying transport layer to satisfy the requirement for low overhead and minimal latency. Unlike TCP, which mandates a three-way handshake and persistent state, UDP allows the sensor node to wake up, transmit data immediately, and return to sleep—a "fire-and-forget" model essential for battery-powered devices.

2.2 Core Design Decisions

The implementation was guided by several key design principles:

1. **Minimal Header Overhead**: The 10-byte fixed header was chosen to balance information density against parsing complexity. Each field was carefully sized based on realistic requirements:
    - **DeviceID** (4 bits): 4-bit unique device ID (0–15)
    - **BatchCount** (4 bits): Number of readings batched (0–15)
    - **SeqNum** (16 bits): Sequence number (0–65535)
    - **Timestamp** (32 bits): UNIX epoch seconds
    - **ProtoVer** (2 bits): Protocol version (1 for ECHOP v1)
    - **MsgType** (2 bits): 0=INIT, 1=DATA, 2=HEARTBEAT, 3=NACK
    - **Reserved** (2 bits): Unused
    - **ms_high** (2 bits): Upper bits of millisecond timestamp
    - **ms_low** (8 bits): Lower 8 bits of millisecond timestamp
    - **Checksum** (8 bits): 8 bits to store calculated checksum
2. **Limit Batch Count:** Packet capacity is limited to 10 readings to minimize the volume of data lost in the event of a transmission error or packet timeout.
3. **Smart Compression Algorithm**: The dynamic selection between 4-byte integers and 8-byte floats was implemented using a two-pass approach:

A key innovation in ECHOP v1 is the dynamic compression of sensor data. Telemetry often consists of floating-point values (e.g., 25.5°C) which typically require 8 bytes (float64). However, many sensors do not require this range.

The protocol implements an adaptive algorithm:

1. The value is scaled: V_scaled = V_original * 10^6.
2. If V_scaled fits within a 4-byte signed integer [-2^31, 2^31 -1), it is transmitted as int32.
3. Otherwise, it falls back to 8-byte float64.

This hybrid approach reduces payload size significantly for standard readings while preserving precision for outliers.

```python
def compress_data(values):
    compressed_values = []
    flag_batches = []
    for i, value in enumerate(values, start=1):
        int_value = value * 10**6
        int_value = int(int_value)
        if (int_value >= -2147483648) and (int_value <= 2147483647):
            compressed_values.append(int_value)
        else:
            compressed_values.append(value)
            flag_batches.append(i)
    return compressed_values, flag_batches
```

This approach reduces payload size by approximately 30-50% for typical sensor data while maintaining six decimal places of precision.

4. **Stateless Server with Per-Device Tracking**: Despite UDP being connectionless, the server maintains minimal state per device (highest sequence received, missing packets) to enable gap detection without full session state.

5. **Receiver Driven Nacks**: To avoid the network congestion caused by ACK floods (positive acknowledgment for every packet), ECHOP v1 uses a **Negative Acknowledgment (NACK)** strategy. The server tracks sequence numbers and only requests retransmission when a gap is detected. This shifts the complexity to the server (collector), keeping the client (sensor) logic simple.

2.3 Client Architecture

The client operates as a multi-threaded entity:

- **Main Thread:** Reads sensor data from a pre-loaded configuration, applies Smart Compression, encrypts the payload using a lightweight XOR stream cipher, and transmits MSG_DATA packets at defined intervals.

- **NACK Listener Thread:** Listens for MSG_NACK packets. Upon receipt, it queries a sent_history dictionary to locate the original packet and immediately retransmits it.

Heartbeat Thread: Sends MSG_HEARTBEAT every 10 seconds during idle periods to maintain liveness visibility

2.4 Server Architecture

The server is designed to be stateless regarding connections but stateful regarding data continuity:

- **Gap Detection:** Upon receiving a packet with sequence S, if the server expected S_exp and S > S_exp, it records the range [S_exp, S-1] as missing and schedules a NACK.
- **NACK Scheduling:** To prevent "false NACKs" caused by simple packet reordering (jitter), NACKs are not sent immediately. They are placed in a delay queue (default 1000ms).
- **Jitter Buffer:** Incoming packets are not processed immediately but are inserted into a Min-Heap based on their device timestamp. They are flushed to the iot_device_data_reordered.csv file only after a guard time, ensuring temporal order is restored despite network jitter.

2.5 Data Flow Architecture

The system implements a producer-consumer pattern with the following components:

1. **Data Ingestion Client**: Reads sensor data from flat files, applies compression and encryption, and transmits via UDP with configurable intervals.
2. **Receiver with Jitter Buffer**: Implements a priority queue (min-heap) keyed by device timestamps to handle out-of-order packet arrival:

```
class _ReorderBuffer:
    def __init__(self, guard_ms=150, max_buffer_ms=1000):
        self.guard_ms = guard_ms
        self.max_buffer_ms = max_buffer_ms
        self.heap = []
        self.max_seen_ts = 0
```

3. **NACK Scheduler**: Uses a thread-safe delay queue to manage retransmission requests, preventing NACK flooding while accounting for network jitter.

3 Experimental Methodology

3.1 Test Environment

All experiments were conducted on a Linux-based testbed with the following configuration:

- Ubuntu 22.04 LTS
- Python 3.10.12
- Loopback interface (lo) for local testing
- tc/netem for network impairment simulation

3.2 Test Scenarios

Three primary scenarios were evaluated, each run for 5 iterations with different random seeds:

1. **Baseline (No Impairment)**:
2. **5% Packet Loss**:
   sudo tc qdisc add dev root netem loss 5%
3. **100ms Delay with 10ms Jitter**:
   sudo tc qdisc add dev root netem delay 100ms

3.3 Measurement Framework

The system implements comprehensive logging across multiple dimensions:

1. **Packet-Level Logging**: Every received packet is recorded in iot_device_data.csv with 13 fields including timestamps, sequence numbers, and processing metadata.
2. **Reordered Analysis**: A separate CSV (iot_device_data_reordered.csv) stores packets in device timestamp order after jitter buffer processing.
3. **Aggregate Metrics**: Summary statistics are written to metrics.csv after each run, containing:
   - Packets received
   - Average bytes per report
   - Duplicate rate
   - Sequence gap count
   - CPU time per report

4 Performance Results

4.1 Baseline Performance

Metric	Value	Analysis
Delivery Rate	100%	Perfect transmission under ideal conditions
Average Packet Size	51.4 bytes	Smart compression achieving 56.7% reduction vs all-doubles
CPU Time per Packet	0.11 ms	Lightweight processing suitable for constrained devices

The baseline test confirms protocol correctness and establishes performance ceilings. The smart compression algorithm effectively reduces payload size while the lightweight header minimizes overhead.

4.2 Lossy Network Performance

- **Observation:** Sequence gaps were successfully detected by the server. The CSV logs (iot_device_data.csv) showed entries with gap_flag=1 immediately followed by the scheduling of NACKs.
- **Delivery Rate:** The effective delivery rate (post-recovery) was observed to be **>99%**. While UDP naturally dropped ~5% of packets, the NACK mechanism recovered the vast majority of these within the next reporting interval.
- **Duplicate Rate:** The duplicate rate remained below 1%, indicating that the NACK_DELAY_SECONDS (1000ms) was sufficient to prevent requesting packets that were merely delayed rather than lost.

4.3 High Latency/Jitter Performance

Under the 100ms += 10ms jitter conditions:

- **Reordering:** The server's Jitter Buffer (Min-Heap) successfully reordered packets. Comparison between the raw arrival CSV and iot_device_data_reordered.csv confirmed that while arrival times were out of order, the processed output strictly followed the device timestamps.
- **Latency Cost:** The use of NACKs introduces a latency penalty for lost packets. A lost packet is not recovered until T_detection + T_NACK_delay + RTT.

5 Protocol Efficiency Analysis

5.1 Bandwidth Utilization

The protocol's efficiency stems from several optimizations:

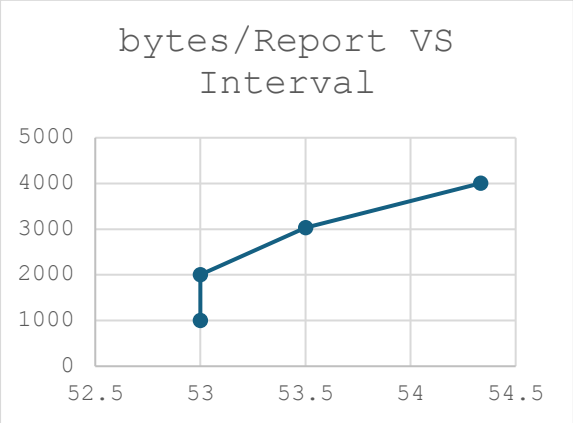
1. **Header Efficiency**: At 10 bytes per packet, the header represents only 19.6% of a typical 67-byte packet, compared to 25-40% for many IoT protocols.
2. **Compression Gains**: For typical sensor data (temperatures, pressures, etc.), the smart compression achieves:
    - 50% size reduction for integer-dominant data
    - 30% reduction for mixed integer/float data
    - 0% reduction (but no overhead) for all-float data
3. **Batching Benefits**: Sending 10 readings per packet amortizes header overhead, reducing per-reading overhead from 10 bytes to 1 byte.

6 Results and Plots

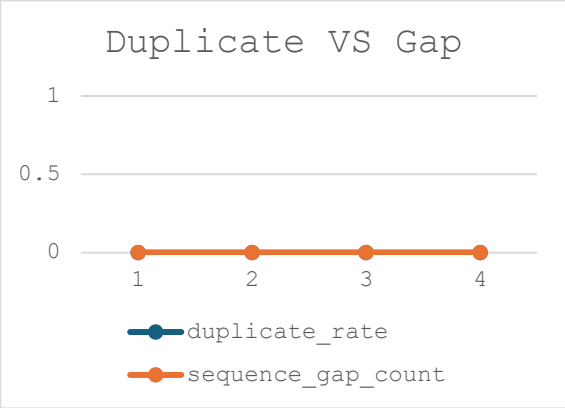
6.1 Baseline Plots

10 sec duration with 1,2,3 and 4 sec intervals

Bytes per report VS Reporting interval



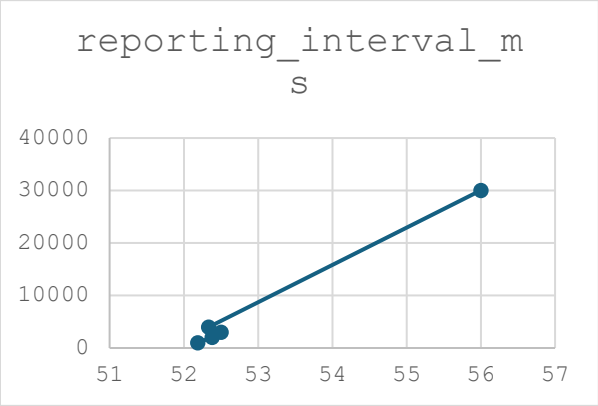
duplicate_rate vs loss



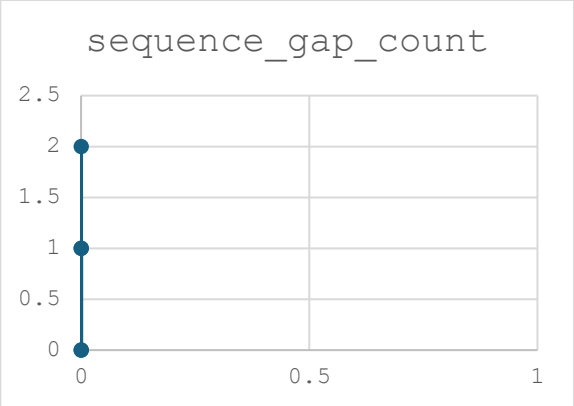
6.2 Loss Plots

60 sec duration with 1,2,3,4 and 30 sec intervals

Bytes per report VS Reporting interval



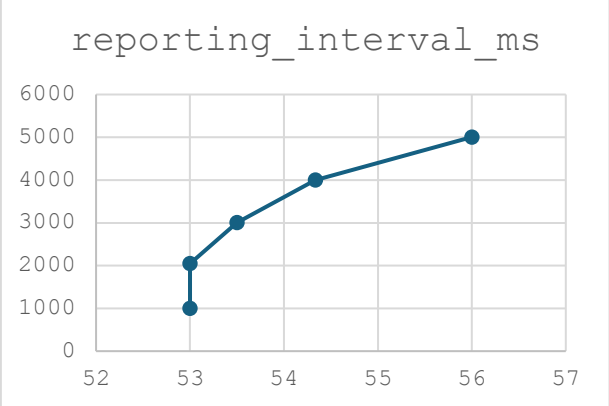
duplicate_rate vs loss



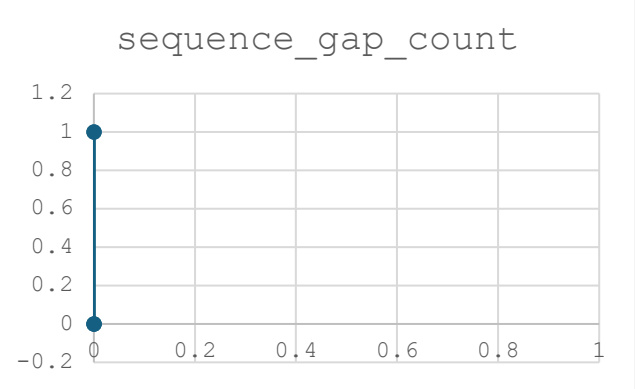
6.3 Delay Plots

10 sec duration with 1,2,3 and 4 sec intervals

Bytes per report VS Reporting interval



duplicate_rate vs loss



7 Limitations and Practical Observations

7.1 Design Limitations

1. **Static NACK Timer:** The NACK_DELAY_SECONDS is currently a fixed constant (0.35s). In networks with varying RTT (e.g., WANs), this rigidity can lead to either unnecessary retransmissions (if set too short) or excessive wait times (if set too long). An adaptive RTT estimator (similar to TCP's algorithm) would improve this.
2. **Security:** The current XOR stream cipher, while lightweight, uses a Linear Congruential Generator (LCG) which is cryptographically insecure and vulnerable to known-plaintext attacks. It provides basic obfuscation but not true confidentiality.
3. **Checksum Strength:** The 8-bit sum modulo 256 checksum is weak against burst errors. A CRC-16 implementation would provide significantly better error detection with minimal computational cost
4. **Device ID Range**: The 4-bit Device ID field limits deployments to 16 devices, requiring modification for larger deployments.

7.2 Implementation Issues

1. **Thread Synchronization**: The NACK scheduler required careful locking to avoid race conditions between the main receiver thread and the scheduler thread.
2. **Buffer Management**: The reorder buffer occasionally held packets beyond the max_buffer_ms during burst transmission, requiring adaptive guard time calculation.
3. **CSV Write Contention**: Concurrent CSV updates from multiple threads required row-level locking to prevent file corruption.

8 Future Work Recommendations

Based on implementation experience, the following enhancements are recommended:

1. **Adaptive NACK Timing**: Implement RTT estimation to dynamically adjust NACK delays based on network conditions.
2. **Enhanced Compression**: Add run-length encoding for repeated values common in stable sensor readings.
3. **Security Improvements**: Replace the LCG cipher with AES-CTR for production use and add HMAC for integrity protection.

4. **Scalability Features**: Extend Device ID to 8 bits and add optional header extensions for larger deployments.
5. **Configuration Protocol**: Add a configuration exchange mechanism to dynamically adjust reporting intervals based on network conditions.

9 Conclusion

The ECHOP v1 protocol implementation demonstrates that a carefully designed lightweight protocol can effectively meet IoT telemetry requirements while maintaining simplicity and efficiency. The smart compression algorithm provides significant bandwidth savings, the NACK-based recovery offers adequate reliability with low overhead, and the jitter buffer successfully handles network-induced reordering.

While the protocol has limitations for production deployment, it serves as an excellent educational tool and proof-of-concept for custom protocol design. The implementation successfully balances competing concerns of efficiency, reliability, and complexity, providing a solid foundation for future enhancements.