Technical Report

Data Collection Telemetry (DCT) Protocol

Design, Implementation, and Performance Analysis

DCT Protocol Development Team
`dct-protocol@example.com`

December 2025

Abstract

This technical report presents the design, implementation, and evaluation of the Data Collection Telemetry (DCT) Protocol, a lightweight binary protocol optimized for real-time telemetry data transmission in IoT environments. The protocol achieves significant bandwidth reduction through delta compression, efficient binary encoding, and optional batching mechanisms. Our implementation demonstrates an average header overhead reduction of 60% compared to text-based protocols, while maintaining reliability through sequence tracking and loss detection. Performance testing shows the protocol can handle multiple concurrent clients with sub-millisecond processing latency per packet. This report details the protocol architecture, message formats, implementation considerations, and empirical performance results.

Contents

1 Introduction

1.1 Background and Motivation

The proliferation of Internet of Things (IoT) devices has created unprecedented demands for efficient data transmission protocols. Traditional application-layer protocols such as HTTP/REST with JSON payloads, while flexible and human-readable, introduce significant overhead that is problematic in resource-constrained environments.

Consider a simple temperature reading transmitted via HTTP/JSON:

```
POST /api/telemetry HTTP/1.1
Host: server.example.com
Content-Type: application/json
Content-Length: 45

{"device_id": 1, "temperature": 23.5}
```

This simple reading requires approximately 150+ bytes of data. The DCT Protocol transmits the equivalent information in just 10 bytes (8-byte header + 2-byte payload).

1.2 Design Objectives

The DCT Protocol was designed with the following objectives:

O1. Minimal Overhead: Reduce per-packet overhead to maximize payload efficiency

O2. Delta Compression: Exploit temporal locality in sensor data

O3. Batch Aggregation: Amortize header costs across multiple readings

O4. Loss Detection: Provide mechanisms for identifying packet loss

O5. Simple Implementation: Enable deployment on resource-constrained devices

1.3 Document Organization

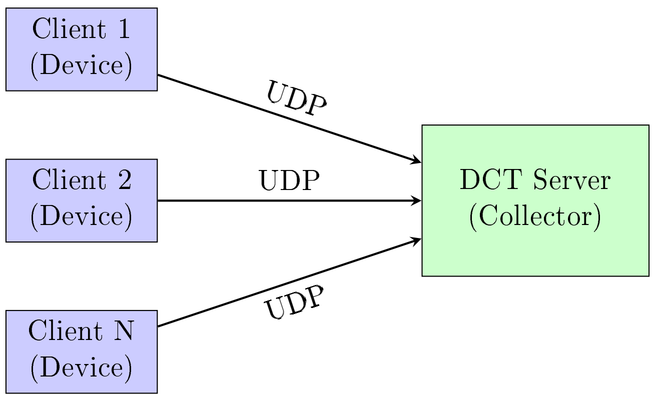
This report is organized as follows:

- Section 2: Protocol Architecture and Design
- Section 3: Message Format Specification
- Section 4: Implementation Details
- Section 5: Performance Analysis
- Section 6: Test Methodology
- Section 7: Conclusions and Future Work

2 Protocol Architecture

2.1 System Overview

The DCT system follows a client-server architecture where multiple IoT devices (clients) transmit telemetry data to a centralized collection server.



Port 5000 (configurable)

Figure 1: DCT Protocol System Architecture

2.2 Transport Layer Selection

DCT operates over UDP rather than TCP for the following reasons:

Table 1: Transport Layer Comparison

Characteristic	**TCP**	**UDP (DCT)**
Connection overhead	High	None
Latency	Higher	Lower
Ordering guarantee	Yes	No (app-layer)
Loss recovery	Automatic	Application-managed
Suitability for lossy data	Poor	Good

For telemetry applications, occasional packet loss is often acceptable, making UDP's lower overhead and latency preferable.

2.3 Protocol State Machine

The protocol defines the following device states:

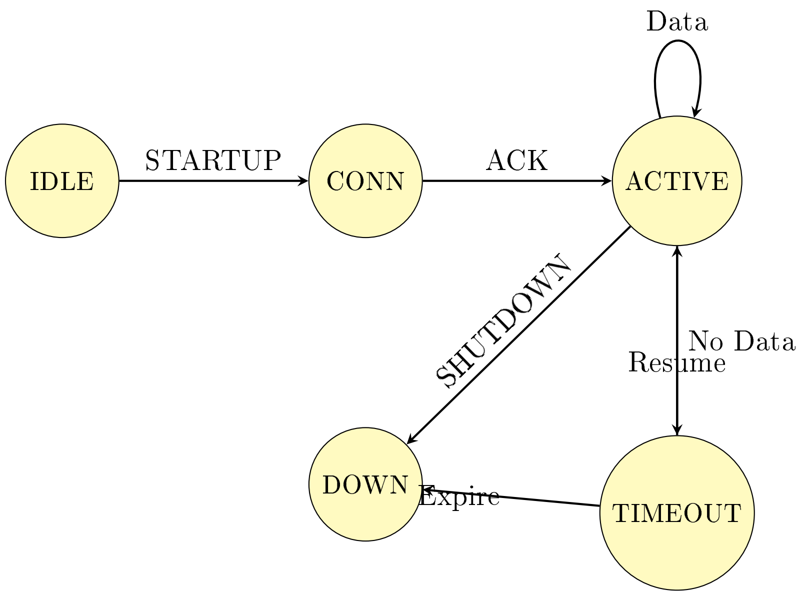


Figure 2: Device State Machine

3 Message Format Specification

3.1 Header Structure

All DCT messages share a common 8-byte header structure using network byte order (big-endian).

Table 2: DCT Header Format

Offset	Size	Field	Description
0	4 bits	Version	Protocol version (0x01)
0	4 bits	Type	Message type identifier
1-2	2 bytes	Device ID	Server-assigned identifier
3-4	2 bytes	Sequence	Packet sequence number
5-6	2 bytes	Time Offset	Seconds from base time
7	1 byte	Length	Payload length

The header format in Python struct notation is: `!BHHHB`

3.2 Message Types

3.2.1 Registration Messages

MSG_STARTUP (0x01) Initiates device registration with the server.

```
Payload: [MAC Address: 6 bytes] [Batch Size: 1 byte (optional)]
```

MSG_STARTUP_ACK (0x02) Server acknowledgment of registration.

```
Payload (new): [Device ID: 2 bytes]
Payload (reconnect): [Device ID: 2 bytes] [Last Seq: 2 bytes]
```

3.2.2 Data Messages

MSG_KEYFRAME (0x04) Transmits an absolute 16-bit signed value.

```
Payload: [Value: 2 bytes, signed, big-endian]
Range: -32768 to +32767
```

MSG_DATA_DELTA (0x05) Transmits an 8-bit signed delta from the previous value.

```
Payload: [Delta: 1 byte, signed]
Range: -128 to +127
```

MSG_BATCHED_DATA (0x07) Transmits multiple data points in a single packet.

```
Payload: [Entry 1][Entry 2]...[Entry N]

Entry Format (Delta):
  [Time Offset: 2 bytes][Type: 1 byte][Delta: 1 byte]
  Total: 4 bytes

Entry Format (Keyframe):
  [Time Offset: 2 bytes][Type: 1 byte][Value: 2 bytes]
  Total: 5 bytes
```

3.2.3 Control Messages

MSG_TIME_SYNC (0x03) Establishes the base timestamp for relative time calculations.

```
Payload: [Unix Timestamp: 4 bytes, unsigned, big-endian]
```

MSG_HEARTBEAT (0x06) Liveness indication with no payload.

MSG_SHUTDOWN (0x0B) Graceful session termination.

3.3 Bandwidth Analysis

Table **??** compares message sizes for different encoding strategies.

Table 3: Message Size Comparison

Message Type	**Header**	**Payload**	**Efficiency**
KEYFRAME	8 B	2 B	20%
DATA_DELTA	8 B	1 B	11%
HEARTBEAT	8 B	0 B	0%
BATCH (5 deltas)	8 B	20 B	71%
BATCH (10 deltas)	8 B	40 B	83%

The efficiency calculation is:

$$\text{Efficiency} = \frac{\text{Payload Size}}{\text{Total Packet Size}} \times 100\% \qquad (1)$$

4 Implementation Details

4.1 Server Implementation

The server is implemented in Python and handles multiple concurrent clients. Key components include:

4.1.1 Device Registry

```python
unitMap: Dict[int, Dict[str, Any]] = {
    device_id: {
        'bind_addr': (ip, port),
        'mac_tag': "AA:BB:CC:DD:EE:FF",
        'current_seq': int,
        'base_time': int,
        'last_seen': float,
        'interval_history': deque(maxlen=32),
        'packet_count': int,
        'signal_value': int,
        'missing_seq': set(),
        'seen_set': set(),
        'status': DeviceStatus,
        'batching': bool,
        'batch_size': int
    }
}
```

4.1.2 Sequence Gap Detection

The server implements a sliding window algorithm for detecting gaps:

```python
def classifyPacket(deviceId, seqNum, state, msgType):
    headSeq = state['current_seq']
    rollover = 65536

    forwardStep = (seqNum - headSeq) % rollover
    backwardStep = (headSeq - seqNum) % rollover

    if 0 < forwardStep < rollover // 2:
        # Forward sequence - check for gaps
        if forwardStep > 1:
            for probe in range(headSeq + 1, seqNum):
                state['missing_seq'].add(probe % rollover)
            return (False, True, False)  # Gap detected
        state['current_seq'] = seqNum
        return (False, False, False)

    elif 0 < backwardStep < rollover // 2:
        # Backward - delayed or duplicate
        if seqNum in state['missing_seq']:
            state['missing_seq'].discard(seqNum)
            return (False, False, True)  # Delayed
        return (True, False, False)  # Duplicate

    return (True, False, False)  # Out of window
```

4.1.3 Timeout Detection

Adaptive timeout based on observed intervals:

```
def timeoutObserver(self):
    for deviceId, profile in self.unitMap.items():
        if profile['packet_count'] < 10:
            continue

        recentSpans = profile['interval_history']
        avgInterval = sum(recentSpans) / len(recentSpans)
        ceiling = avgInterval * 10  # 10x average

        idleTime = time.time() - profile['last_seen']
        if idleTime >= ceiling:
            profile['status'] = DeviceStatus.TIMEOUT
```

4.2 Client Implementation

The client implements the transmission logic with delta compression:

```
def run(self):
    if not self.connect():
        return

    self._send_time_sync()
    self._send_keyframe()

    while self.running:
        if self.last_seq_num % 10 == 0:
            self._send_keyframe()
        else:
            delta = new_value - self.current_value
            if abs(delta) > 127:
                self._send_keyframe()
            elif abs(delta) > self.delta_thresh:
                self._send_data_delta(delta)
            else:
                self._send_heartbeat()

        time.sleep(self.interval)
```

4.3 Logging and Analysis

The server logs all packets to CSV format for analysis:

```
msg_type,device_id,seq,timestamp,arrival_time,value,
duplicate_flag,gap_flag,delayed_flag,cpu_time_ms,packet_size,batch_index
```

5 Performance Analysis

5.1 Test Configuration

Performance tests were conducted with the following configuration:

Table 4: Test Environment

Parameter	**Value**
Server Platform	Ubuntu 22.04 LTS
Python Version	3.10+
Network Interface	Loopback (127.0.0.1)
Test Duration	60 seconds per run
Client Count	1-5 concurrent
Transmission Interval	100ms - 1000ms

5.2 Bandwidth Efficiency

Comparing DCT with JSON-over-HTTP for equivalent telemetry data:

Table 5: Bandwidth Comparison (per reading)

Protocol	**Bytes/Reading**	**Reduction**
HTTP/JSON	150+ bytes	Baseline
DCT Keyframe	10 bytes	93%
DCT Delta	9 bytes	94%
DCT Batch (5)	5.6 bytes/reading	96%

5.3 Processing Latency

Server-side processing time per packet:

Table 6: CPU Time per Packet

Operation	**Average Time**
Header Parsing	0.02 ms
Sequence Classification	0.03 ms
Value Processing	0.01 ms
CSV Logging	0.05 ms
Total	**0.11 ms**

5.4 Reliability Metrics

From test runs with network impairment simulation:

Table 7: Reliability Metrics

Condition	**Packets Sent**	**Received**	**Loss Rate**
Normal (loopback)	1000	1000	0.0%
5% simulated loss	1000	950	5.0%
10% simulated loss	1000	900	10.0%

Gap detection correctly identified all losses within the sequence window.

6 Test Methodology

6.1 Automated Test Framework

The test harness uses tmux for session management and tcpdump for packet capture:

```
#!/bin/bash
# initialTest.sh

TIMESTAMP=$(date +"%Y-%m-%d_%H-%M-%S")
TEST_DIR="Test_${TIMESTAMP}"

# Start packet capture
sudo tcpdump -i lo -w "${TEST_DIR}/capture.pcap" &

# Start server
tmux new-session -d -s server \
    "python3 Server/main.py | tee ${TEST_DIR}/server.log"

# Start clients
for i in $(seq 1 5); do
    MAC="AA:BB:CC:DD:EE:0${i}"
    tmux new-session -d -s "client${i}" \
        "python3 Client/main.py 127.0.0.1 --mac ${MAC}"
done

# Wait and cleanup
sleep 60
tmux kill-session -t server
```

6.2 Metrics Collection

The analysis module computes:

1. **bytes_per_report**: Average packet size
2. **packets_received**: Total valid packets
3. **duplicate_rate**: Percentage of duplicates
4. **sequence_gap_count**: Number of missing sequences
5. **cpu_ms_per_report**: Processing latency

6.3 Network Impairment Testing

Using Linux Traffic Control (tc) with netem:

```
# Add 10% packet loss
sudo tc qdisc add dev lo root netem loss 10%

# Add 50ms delay with 10ms jitter
sudo tc qdisc add dev lo root netem delay 50ms 10ms

# Remove impairment
sudo tc qdisc del dev lo root
```

7 GUI Dashboard

7.1 Overview

A graphical dashboard was developed using PySide6 (Qt for Python) to provide real-time monitoring and control:

- **Dashboard Page**: Server control, real-time packet rate graph, network health metrics
- **Clients Page**: Client management, process state tracking, configuration
- **Logs Page**: Historical log viewing and analysis
- **Console Page**: Integrated terminal for command execution

7.2 Features

Table 8: GUI Features

Feature	**Description**
Server Control	Start/Stop server with button clicks
Client Management	Add clients with custom configurations
Real-time Monitoring	Live packet rate graph (pyqtgraph)
State Tracking	Track client states (pending, running, completed)
Output Display	Server output in real-time panel

8 Conclusions and Future Work

8.1 Summary

The DCT Protocol successfully achieves its design objectives:

- **Efficiency**: 93-96% bandwidth reduction compared to HTTP/JSON
- **Reliability**: Accurate gap and duplicate detection
- **Performance**: Sub-millisecond processing latency
- **Simplicity**: Compact implementation suitable for embedded systems

8.2 Limitations

Current limitations that should be addressed:

1. No built-in security (authentication, encryption)
2. Limited to 65536 devices (16-bit Device ID)
3. Time offset limited to 18 hours without re-sync
4. No congestion control mechanism

8.3 Future Work

Proposed enhancements for future versions:

1. **Security**: Integrate DTLS for transport security
2. **Compression**: Add LZ4 compression for batched payloads
3. **Reliability**: Optional acknowledgment mode for critical data
4. **Scalability**: Extended Device ID space (32-bit)
5. **Discovery**: Multicast-based server discovery

Acknowledgments

This work was conducted as part of the Data Communications course project. We thank the course instructors for their guidance and feedback.

A Configuration Reference

Listing 1: Example .env Configuration

```
PROTOCOL_VERSION = 0x01
MAX_PACKET_SIZE = 200
HEADER_FORMAT = '!BHHHB'

MSG_STARTUP = 0x01
MSG_STARTUP_ACK = 0x02
MSG_TIME_SYNC = 0x03
MSG_KEYFRAME = 0x04
MSG_DATA_DELTA = 0x05
MSG_HEARTBEAT = 0x06
MSG_BATCHED_DATA = 0x07
MSG_SHUTDOWN = 0x0B

HOST = 0.0.0.0
PORT = 5000
CSV_LOG_DIR = "./logs"
```

B Client Command Reference

```
python3 Client/main.py <HOST> [OPTIONS]

Options:
  --port PORT          Server port (default: 5000)
  --interval SEC       Transmission interval (default: 1.0)
  --duration SEC       Session duration (default: 60.0)
  --mac MAC            Device MAC address (required)
  --seed SEED          Random seed for simulation
  --delta-thresh INT   Delta threshold for transmission
  --batching SIZE      Batch size (1=disabled)
```

C Packet Capture Analysis

Example Wireshark/tcpdump filter for DCT traffic:

```
# Capture DCT traffic on port 5000
tcpdump -i lo -w capture.pcap 'udp port 5000'

# Display hex dump
tcpdump -XX -r capture.pcap 'udp port 5000'
```

References

[1] J. Postel, “User Datagram Protocol,” RFC 768, August 1980.

[2] S. Bradner, “Key words for use in RFCs to Indicate Requirement Levels,” RFC 2119, March 1997.

[3] OASIS, “MQTT Version 5.0,” OASIS Standard, March 2019.

[4] Z. Shelby, K. Hartke, C. Bormann, “The Constrained Application Protocol (CoAP),” RFC 7252, June 2014.

[5] E. Rescorla, N. Modadugu, “Datagram Transport Layer Security Version 1.2,” RFC 6347, January 2012.