

AIN SHAMS UNIVERSITY
FACULTY OF ENGINEERING
Fall 2025



CSE361: Computer Networking

AUDP-X: Lightweight UDP-Based IoT Telemetry Protocol

Team 30:

Mohamed Ahmed Abdelhamid

22P0287

Nouran Mokhtar Elsayed

22P0254

Abdallah Belal Momen

22P0036

Habeba Adel Sallam

22P0259

Mohamed Awad Shawki

22P0240

Ahmad Sherif ElBaz

2201075

The AUDP-X IoT Telemetry System is designed to provide efficient and lightweight communication between constrained sensor devices and a central data collector. It uses a custom UDP-based application-layer protocol, sequence numbering, timestamping, and session-aware control messages to transmit sensor readings reliably under variable network conditions. This report outlines the system architecture, protocol design, and experimental evaluation, and explains how the different components work together to achieve loss-tolerant, low-overhead, and reliable telemetry data collection.

Contents

1. Introduction.....	5
2. System Overview and Architecture.....	5
2.1 System Components	5
2.2 Communication Model.....	6
3. Protocol Design	6
3.1 Protocol Overview	6
3.2 Message Types.....	6
3.3 Header Format.....	7
3.4 Payload and Batching.....	7
4. Implementation Details	8
4.1 Client Implementation	8
4.1.1 Initialization & Handshake.....	8
4.1.2 Header Packing.....	9
4.1.3 Data Transmission	9
4.1.4 Heartbeat Mechanism.....	11
4.1.5 Session Termination.....	11
4.2 Server Implementation.....	11
4.2.2 Header Parsing & Validation.....	12
4.2.3 Initialization & Handshake	12
4.2.4 Heartbeat Handling.....	13
4.2.5 Data Packet Processing	13
4.2.6 Session Termination.....	15
4.3 Logging	16
5. Experimental Setup and Methodology	16
5.1 Test Environment.....	16
5.2 Test Scenarios	16
5.3 Metrics Collected	16
6. Results and Analysis	17
7. Discussion	17
8. Limitations and Future Work.....	17

9. Conclusion 17

10. References 18

1. Introduction

The rapid growth of Internet of Things (IoT) systems has created a strong demand for lightweight and efficient communication protocols that can operate reliably over constrained and unreliable networks. In our project, we focus on these constraints and design a protocol that is practical, simple, and efficient for real-world sensor deployments. Many IoT devices, such as temperature, humidity, or voltage sensors, are resource-limited in terms of processing power, memory, energy consumption, and bandwidth availability. In such environments, traditional transport protocols that provide strong reliability guarantees often introduce unnecessary overhead and complexity.

In this project, we present AUDP-X, a custom application-layer telemetry protocol built on top of UDP. The protocol is specifically designed for periodic sensor reporting scenarios where occasional packet loss is acceptable and simplicity, low overhead, and energy efficiency are prioritized. Instead of attempting to recover lost packets, AUDP-X focuses on detecting loss, suppressing duplicates, and enabling accurate offline analysis.

Our goal in this project is to design, implement, and experimentally evaluate a compact UDP-based telemetry protocol that satisfies the functional and operational requirements defined in Project 1. We evaluate the behavior of the protocol under baseline conditions as well as under packet loss and network delay, and analyze its performance using quantitative metrics.

2. System Overview and Architecture

Our system follows a simple client–server architecture consisting of multiple sensor devices (clients) and a single central collector (server). Communication is unidirectional for telemetry data, with optional acknowledgments used for control and liveness purposes.

2.1 System Components

Sensor (Client):

Each sensor periodically generates telemetry readings and sends them to the server using UDP. The client is responsible for assigning sequence numbers, attaching timestamps, and switching between DATA and HEARTBEAT messages depending on activity.

Collector (Server):

The server listens on a fixed UDP port and maintains per-device state. It processes incoming packets, suppresses duplicates, detects missing sequence numbers, and logs all relevant metadata for analysis.

2.2 Communication Model

The protocol is session-aware but connectionless. A lightweight initialization handshake is used to establish session state, after which the client continuously sends telemetry without waiting for acknowledgments. This design minimizes latency and avoids head-of-line blocking.

Overall communication flow:

Sensor → INIT → Server → ACK

Sensor → DATA / HEARTBEAT → Server (best-effort)

Sensor → END → Server

This architecture aligns well with IoT telemetry use cases where data freshness is more important than guaranteed delivery.

3. Protocol Design

3.1 Protocol Overview

The protocol is named AUDP-X (Application-layer UDP eXtended), version 1. It is designed with the following goals:

- Minimal header overhead
- Loss tolerance without retransmission
- Simple session management
- Support for batching and extensibility

3.2 Message Types

AUDP-X defines five message types:

- **INIT:** Initializes a session between client and server
- **DATA:** Carries sensor telemetry readings
- **HEARTBEAT:** Indicates device liveness during idle periods
- **ACK:** Acknowledges control and heartbeat messages
- **END:** Terminates a session

DATA messages are sent using a fire-and-forget model, while INIT and HEARTBEAT messages are acknowledged to maintain session awareness.

3.3 Header Format

All messages share a compact 13-byte binary header consisting of the following fields:

Field	Size	Description
Version + MsgType	1 byte	Protocol version (4 bits) and message type (4 bits)
Device ID	2 bytes	Unique identifier for the sensor
Sequence Number	2 bytes	Monotonic packet counter
Timestamp	8 bytes	Client-side timestamp (Unix time)

The compact header design ensures low overhead while still providing sufficient metadata for analysis and ordering.

3.4 Payload and Batching

Payloads are encoded in JSON for readability and flexibility. DATA messages include a batch array that may contain one or more sensor readings. In the current implementation, the batch size is fixed to one reading per packet to minimize latency and simplify loss analysis.

Batching support is included in the protocol design to allow future optimization when network conditions are stable.

4. Implementation Details

We implemented the protocol in Python 3 using standard UDP sockets. The implementation is fully cross-platform and was tested on both Windows and Linux systems.

4.1 Client Implementation

The sensor client is implemented in Python using UDP sockets. Each client generates a unique device identifier derived from the process ID and communicates with the collector using the AUDP-X protocol header and message formats.

4.1.1 Initialization & Handshake

```
def send_and_wait_ack(sock, packed_msg, expect_seq=None, expect_type=None,
timeout=ACK_TIMEOUT):
    tries = 0
    while tries <= MAX_RETRIES:
        try:
            sock.sendto(packed_msg, (SERVER_IP, SERVER_PORT))
            sock.settimeout(timeout * (2 ** tries))
            data, _ = sock.recvfrom(4096)

            if len(data) < HDR_LEN:
                tries += 1
                continue

            v, t, did, seq_r, ts = unpack_header(data)
            if expect_type is not None and t != expect_type:
                tries += 1
                continue
            if expect_seq is not None and seq_r != expect_seq:
                tries += 1
                continue

            payload_bytes = data[HDR_LEN:]
            payload = None
            if payload_bytes:
```

```

        try:
            payload = json.loads(payload_bytes.decode())
        except Exception:
            payload = None
        return True, (v, t, did, seq_r, ts, payload)
    except socket.timeout:
        tries += 1
        if tries > MAX_RETRIES:
            return False, None
        backoff = BASE_BACKOFF * (2 ** (tries - 1)) * (0.8 + 0.4 *
random.random())
        time.sleep(backoff)
    except Exception:
        return False, None
    return False, None

```

At startup, the client constructs an INIT message containing protocol metadata and sends it to the server. A lightweight handshake is performed by waiting for an ACK message with a configurable timeout. If no acknowledgment is received, the client terminates execution, ensuring that no telemetry data is sent without an active session.

4.1.2 Header Packing

```

def pack_header(version, msgtype, device_id, seq, timestamp):
    header_byte = ((version & 0xF) << 4) | (msgtype & 0xF)
    return struct.pack(HDR_FMT, header_byte, device_id & 0xFFFF, seq & 0xFFFF,
timestamp)

```

All packets use a compact 13-byte binary header. The header is packed using Python's `struct` module in network byte order, combining the protocol version and message type into a single byte to reduce overhead.

4.1.3 Data Transmission

```

while seq <= NUM_MESSAGES:
    current_time = time.time()

    if current_time - last_heartbeat_time >= HEARTBEAT_INTERVAL:
        ts_str = get_detailed_ts(current_time)
        print(f"[{ts_str}] HEARTBEAT SENT :: DEVICE {device_id}")

```

```

        hb_hdr = pack_header(version, MSG_HEARTBEAT, device_id, 0,
current_time)
        hb_packet = hb_hdr + b''
        send_best_effort(sock, hb_packet)
        last_heartbeat_time = current_time

    readings = []
    for b in range(BATCH_SIZE):
        reading = {
            "reading_id": seq + b,
            "value": round(random.uniform(20.0, 30.0), 2),
            "unit": "C"
        }
        readings.append(reading)
    payload_obj = {
        "batch": readings
    }
    payload_bytes = json.dumps(payload_obj).encode()

    hdr = pack_header(version, MSG_DATA, device_id, seq, time.time())
    packet = hdr + payload_bytes

    ts_str = get_detailed_ts(time.time())

    if send_best_effort(sock, packet):
        print(f"[{ts_str}] DATA SENT OK :: DEVICE {device_id} :: SEQ {seq}")
        seq += BATCH_SIZE
    else:
        print(f"[{ts_str}] DATA SEND FAIL :: DEVICE {device_id} :: SEQ {seq}
:: LOCAL SOCKET ERROR")
        seq += BATCH_SIZE

    time.sleep(1)

```

During normal operation, the client periodically generates sensor readings and sends DATA messages using a best-effort transmission model. Each packet includes a monotonically increasing sequence number and a client-side timestamp. No retransmission is performed for DATA packets, in accordance with the loss-tolerant design of the protocol.

4.1.4 Heartbeat Mechanism

```
if current_time - last_heartbeat_time >= HEARTBEAT_INTERVAL:
    ts_str = get_detailed_ts(current_time)
    print(f"[{ts_str}] HEARTBEAT SENT :: DEVICE {device_id}")

    hb_hdr = pack_header(version, MSG_HEARTBEAT, device_id, 0,
current_time)
    hb_packet = hb_hdr + b''
    send_best_effort(sock, hb_packet)
    last_heartbeat_time = current_time
```

When no data is transmitted for a predefined interval, the client sends HEARTBEAT messages to indicate device liveness. These messages allow the server to maintain session state during idle periods without introducing additional data traffic.

4.1.5 Session Termination

```
end_hdr = pack_header(version, MSG_END, device_id, 0, time.time())
end_packet = end_hdr + b''
send_best_effort(sock, end_packet)
sock.close()
```

After completing data transmission, the client sends an END message and closes the socket gracefully, allowing the server to finalize logging and session state.

4.2 Server Implementation

The collector server is implemented in Python using UDP sockets and is responsible for receiving, processing, and logging telemetry data from multiple sensor devices. The server maintains per-device session state to support duplicate suppression, gap detection, heartbeat monitoring, and experimental analysis.

4.2.1 Session Management

```
sessions = defaultdict(lambda: {
    "addr": None,
    "received_seqs": set(),
    "last_seq": 0,
```

```

    "arrival_times": {},
    "recv_buffer": deque(),
    "last_hb": 0,
    "last_batch_size": 1
})

```

The server maintains a session record for each connected device, indexed by device ID. For each session, the server stores the last received sequence number, a set of received sequence numbers for duplicate detection, the most recent heartbeat time, and the last observed batch size. This state allows the server to process packets independently for multiple devices.

4.2.2 Header Parsing & Validation

```

def unpack_header(raw):
    if len(raw) < HDR_LEN:
        raise ValueError("Packet too short for header")
    header = struct.unpack(HDR_FMT, raw[:HDR_LEN])
    header_byte, device_id, seq, timestamp = header
    version = (header_byte >> 4) & 0xF
    msgtype = header_byte & 0xF
    return version, msgtype, device_id, seq, timestamp

```

Upon receiving a packet, the server unpacks the fixed-size 13-byte AUDP-X header to extract the protocol version, message type, device ID, sequence number, and client-side timestamp. Packets that are too short or malformed are discarded to ensure robustness.

4.2.3 Initialization & Handshake

```

    if msgtype == MSG_INIT:
        # Clear state and confirm handshake (ACK)
        sessions[device_id]["received_seqs"] = {0}
        sessions[device_id]["last_seq"] = 0

        ack = pack_ack(version, device_id, seq, MSG_ACK)
        sock.sendto(ack, pkt_addr)

```

```

        print(f"[{arrival_ts_str}] [Server] INIT from device
{device_id} seq={seq}. ACK sent.")
        continue

```

When an INIT message is received, the server initializes or resets the session state for the corresponding device and responds with an ACK message. This lightweight handshake confirms that the server is ready to receive telemetry data.

4.2.4 Heartbeat Handling

```

if msgtype == MSG_HEARTBEAT:
    # just update time and ACK
    sessions[device_id]["last_hb"] = arrival_time

    ack = pack_ack(version, device_id, seq, MSG_ACK)
    sock.sendto(ack, pkt_addr)
    print(f"[{arrival_ts_str}] [Server] HEARTBEAT from device
{device_id}. ACK sent.")
    continue

```

HEARTBEAT messages are used to indicate device liveness during idle periods. Upon receiving a heartbeat, the server updates the device's last activity timestamp and responds with an ACK, allowing the session to remain active without transmitting data.

4.2.5 Data Packet Processing

```

if msgtype == MSG_DATA:

    is_dup = seq in sessions[device_id]["received_seqs"]
    packet_gap_flag = False

    # 1. Duplicate check and suppression
    if is_dup:
        # Log it as a duplicate, then ignore the payload.
        row = [device_id, seq, ts, arrival_time, 1, 0,
payload_len]

        write_packet_log_row(row)

        print(f"[{arrival_ts_str}] [Server] Duplicate DATA from
device {device_id} seq={seq}. Ignoring.")

```

```

        ack = pack_ack(version, device_id, seq, MSG_ACK)
        sock.sendto(ack, pkt_addr)
        continue

    # 2. Sequence Gap Detection
    expected_next_seq = sessions[device_id]["last_seq"] +
sessions[device_id]["last_batch_size"]

    if seq > expected_next_seq:
        packet_gap_flag = True

    # Update state
    sessions[device_id]["received_seqs"].add(seq)

    # 3. Payload processing
    try:
        payload = raw_pkt[HDR_LEN:]
        payload_obj = json.loads(payload.decode('utf-8'))
        readings = payload_obj.get("batch", [])
        batch_size = len(readings)

        # Internal Batch Gap Check
        batch_gap_info = get_batch_gap_info(readings, seq)

        sessions[device_id]["last_seq"] =
max(sessions[device_id]["last_seq"], seq)
        sessions[device_id]["last_batch_size"] = batch_size if
batch_size > 0 else 1

    except json.JSONDecodeError:
        print(f"[{arrival_ts_str}] [Server] JSON decode fail from
device {device_id} seq={seq}. Payload ignored.")
        batch_size = 0
        batch_gap_info = (False, [])
        sessions[device_id]["last_batch_size"] = 1

    # Log the packet
    row = [device_id, seq, ts, arrival_time, int(is_dup),
int(packet_gap_flag), payload_len]
    write_packet_log_row(row)

    log_output = f"[{arrival_ts_str}] DATA RECEIVED :: DEVICE
{device_id} :: SEQ {seq} :: BATCH {batch_size} :: SIZE {payload_len} :: CLIENT TS
{ts_str}"

```

```

        if packet_gap_flag:
            log_output += " :: MISSING PACKET BEFORE THIS"
        if batch_gap_info[0]:
            log_output += f" :: BATCH MISSING IDS: {'',
''.join(map(str, batch_gap_info[1]))}"
        print(log_output)

        ack = pack_ack(version, device_id, seq, MSG_ACK)
        sock.sendto(ack, pkt_addr)
        continue

```

For DATA messages, the server performs several checks:

- **Duplicate detection:** Packets with previously received sequence numbers are identified and ignored.
- **Sequence gap detection:** The server checks whether the received sequence number matches the expected next sequence based on the last received batch size.
- **Batch validation:** The payload is parsed as JSON, and the batch is inspected for missing reading identifiers within the batch itself.

This directly matches:

- received_seqs
- last_seq
- last_batch_size
- get_batch_gap_info()

4.2.6 Session Termination

```

except KeyboardInterrupt:
    print("\nSHUTDOWN REQUESTED. PROCESSING LOGS...")
    analyze_log_and_sort()
finally:
    sock.close()

```

When the server is terminated, it processes all logged packet data and generates a sorted analysis file. This file is used to evaluate protocol performance metrics such as latency, packet loss, duplication, and reordering.

4.3 Logging

Each received packet is logged with the following fields:

```
device_id, seq, timestamp, arrival_time, duplicate_flag, gap_flag, payload_len
```

These logs are later used for metric computation and plotting.

5. Experimental Setup and Methodology

5.1 Test Environment

We conducted our experiments using a Windows-based client and a Linux-based server connected through a local network. Network impairments were introduced using Clumsy (Windows) and packet captures were collected using tcpdump on the server.

5.2 Test Scenarios

Three scenarios were evaluated:

- **Baseline:** No network impairment
- **Loss 5%:** Random packet loss
- **Delay:** Fixed 100 ms delay

Each scenario was executed five times to ensure statistical validity.

5.3 Metrics Collected

- Packets received
- Duplicate rate
- Sequence gap rate

- Latency (min/median/max)
- Bytes per report

6. Results and Analysis

The baseline scenario achieved 100% packet delivery with zero duplicates and no sequence gaps. Under 5% packet loss, the server correctly detected missing packets while maintaining zero duplicate rate. The delay scenario showed increased latency but no system instability.

Batching efficiency analysis showed that header overhead is significant for small payloads, motivating adaptive batching as a future enhancement.

All the test scenarios and results are clearly demonstrated in our Mini-RFC document.

7. Discussion

Our results demonstrate that AUDP-X successfully meets all project acceptance criteria. The protocol remains stable under loss and delay while maintaining accurate detection of missing packets. The decision to avoid retransmission simplifies the design and aligns with IoT telemetry requirements.

8. Limitations and Future Work

Current limitations include lack of encryption, authentication, retransmission, and adaptive batching. Future work includes adding selective reliability, compression, security mechanisms, and multi-collector support.

9. Conclusion

In this project, we presented AUDP-X, a lightweight and loss-tolerant UDP-based telemetry protocol for IoT environments. Through careful protocol design and experimental evaluation, the

protocol was shown to perform reliably under varying network conditions while maintaining minimal overhead.

AUDP-X demonstrates that simple, purpose-built application-layer protocols can effectively support IoT telemetry use cases without relying on heavyweight transport mechanisms.

10. References

- RFC 768: User Datagram Protocol
- RFC 8259: JSON Data Interchange Format
- RFC 7252: Constrained Application Protocol
- Python struct module documentation
- Clumsy network conditioner
- Linux tc and netem documentation