CSE361: Computer Networking

RFC

Submitted by:

Belal Anas 23P0193

Carol Eid 23P0333

Maryam Mohamed 23P0056

Jana Mohamed 23P0050

Toka Elsayed 23P0044

Mariam Tarek 23P0214

Submitted to:

Dr. Karim Emara

TA. Noha Wahdan

Table of Contents

1. Introduction

The **Internet of Things (IoT)** requires lightweight and efficient communication protocols to facilitate data exchange between sensing devices and central collection units. The **Tiny Telemetry Protocol (TTP)** is an application-layer protocol developed specifically for temperature sensing systems operating within local area networks. Built on top of the **User Datagram Protocol (UDP)**, TTP provides a simple and efficient mechanism for transmitting periodic temperature readings from sensor nodes to a central collector for monitoring and analysis.

TTP is designed to operate reliably in environments where devices have limited computational power, memory, and energy resources. It utilizes UDP to minimize communication overhead and latency, enabling real-time data transfer across the network. Although UDP does not provide built-in reliability, TTP incorporates lightweight validation features such as sequence numbering and checksums to ensure data integrity and support accurate performance assessment.

The protocol is developed with the following design objectives:
• **Low power and bandwidth usage** to support constrained temperature sensors.
• **Simple message structure** to ensure ease of implementation and low processing overhead.
• **Efficient operation within local networks** to achieve consistent and low-latency communication.
• **Reliability mechanisms**, including checksums and sequence tracking, to identify packet corruption, duplication, or loss.
• **Scalability**, allowing multiple temperature sensors to transmit concurrently to a single collector node.

The full implementation of TTPv1, including the client, server, automation scripts, and analysis tools, is available in this repository.

2. Protocol Architecture

The Tiny Telemetry Protocol (TTP v1) is a lightweight application-layer protocol designed for low-latency telemetry delivery over a UDP-based local area network. The protocol prioritizes simplicity, minimal overhead, and explicit fault detection rather than guaranteed delivery, making it suitable for controlled LAN environments and performance evaluation experiments.

2.1 Entities

2.1.1 Sensor Node (Client)

The sensor node simulates an IoT telemetry device responsible for generating and transmitting temperature readings to the collector. Each client operates independently and is identified by a unique *Device ID*. Before data transmission begins, the client performs a mandatory two-step initialization handshake with the server to announce its presence and ensure synchronization.

The client constructs all messages using a fixed-size binary header followed by an optional payload. Each packet includes:

- A monotonically increasing sequence number
- Device ID
- Message type (INIT, DATA, or HEARTBEAT)
- A relative timestamp in milliseconds since client start
- Protocol version
- A modulo-65536 checksum calculated over the entire packet

After successful initialization, the client periodically transmits telemetry data at a fixed interval (1 second). The client supports:

- **Single-reading mode**, where one temperature value is sent per packet
- **Batching mode**, where multiple readings are concatenated into a single payload
- **Heartbeat messages**, sent at regular intervals when no data transmission occurs, to indicate device liveness

Sequence numbers are incremented for every transmitted packet, including DATA and HEARTBEAT messages. Checksum validation is performed by the server to detect packet corruption.

2.1.2 Collector Node (Server)

The collector node acts as a centralized telemetry receiver that listens for UDP packets from multiple sensor nodes concurrently on a fixed port. The server is stateless at the transport layer but maintains lightweight per-device state to track sequence numbers and detect anomalies.

Upon receiving a packet, the server:

- Verifies packet length and header integrity
- Validates the checksum by recomputing it with the checksum field zeroed
- Identifies the message type (INIT, DATA, HEARTBEAT)
- Tracks the last sequence number received per device to detect duplicates and sequence gaps

For INIT messages, the server completes a two-step handshake by replying with ACK_INIT followed by ACK_READY. For DATA messages, valid payloads are extracted and logged. HEARTBEAT messages are acknowledged implicitly and used only for liveness monitoring.

All received telemetry data is recorded in a CSV file, with each row containing:

- Device ID
- Sequence number
- Client-generated timestamp (relative seconds)
- Server arrival time (relative seconds)
- Duplicate flag
- Sequence gap flag
- Extracted data value

This logging design allows post-run analysis of packet loss, duplication, batching behavior, and timing characteristics.

Server

2.2 Network Setup

- Communication between sensor nodes and the collector is established using **UDP sockets** within the same Local Area Network.
- The collector listens on a predefined UDP port (**9999**) and accepts packets from any client address.
- Sensor nodes are configured with the collector's IP address and operate for a fixed experiment duration.
- No retransmission or congestion control mechanisms are implemented at the protocol level; instead, reliability characteristics are inferred through sequence tracking and checksum validation.

- The architecture naturally supports scalability, allowing multiple sensor nodes to transmit concurrently to a single collector with no additional configuration.

This protocol architecture provides a controlled, low-overhead telemetry environment suitable for analyzing packet integrity, timing behavior, batching efficiency, and the effects of network impairment on UDP-based systems.

3. Message Formats

Each TTP message consists of a fixed-size header followed by an optional payload containing sensor readings.

Tiny Telemetry Header Structure:

Field Name	Size (Bytes)	Description
SeqNum	2	Sequence number incremented with each message to detect duplicates or gaps.
DeviceID	1	Unique identifier for each IoT sensor node.
MsgType	1	Specifies the message type (1=INIT, 2=DATA, 3=HEARTBEAT).
Timestamp	4	Unix timestamp (seconds) when the message was generated.
BatchFlag	1	Set to 1 if multiple readings are included; otherwise 0.
Checksum	2	Modulo-65536 sum of all bytes with the checksum field zeroed.
Version	1	Protocol version for compatibility control.

Total Header Size: 12 bytes.

Encoding:

• All header fields are encoded in binary format using network byte order (big-endian).
• The payload is encoded as a UTF-8 string (e.g., “25.7”) or as a binary float value.
• The compact design reduces bandwidth usage and simplifies parsing on constrained IoT devices

4. Operational Procedures

This section defines the operational procedures followed by sensor nodes and the collector to ensure correct functioning of the Tiny Telemetry Protocol (TTP). The protocol is intentionally lightweight and uses simple, deterministic steps for initialization, data reporting, batching, and liveness signaling.

4.1 Initialization Procedure (INIT Handshake)

Before a sensor can transmit telemetry readings, it performs a two-step initialization handshake to synchronize state with the collector:

1. **Client → Server: INIT Message**
   Contains:
   - DeviceID
   - Sequence number
   - Timestamp
   - Protocol version
   - Checksum
2. **Server → Client: ACK_INIT**
   Indicates that the collector has registered the device and verified the header format.
3. **Server → Client: ACK_READY**
   Informs the client that normal data transmission may begin.

Successful execution of this handshake is visible in the logs, for example:
“ACK_INIT received” and “Server READY — starting data” from a baseline run

```
[CLIENT 1] Sending INIT attempt 1
[SERVER] New device connected. ID=1, Version=1
[CLIENT 1] ACK_INIT received
[CLIENT 1] Server READY – starting data
```

The handshake ensures:

- Sequence counters start from a known state
- The server initializes per-device tracking structures
- Misconfigured or incompatible clients fail early

If the client receives no response within the timeout period, it retries up to 5 times.

4.2 Data Transmission Procedure

After initialization, the client periodically sends **DATA** messages that include one or more temperature readings.

4.2.1 Single-Reading Mode

Each packet contains exactly one sensor reading.
Example log: “Sent temp 31.2 (packet 2)”

```
[CLIENT 1] Sent temp 20.5 (packet 1)
[SERVER] Data received | Packet 1 | Checksum Valid
```

4.2.2 Batching Mode

When batching is enabled, multiple readings are concatenated into a comma-separated payload. This improves bandwidth efficiency at the cost of higher delay per reading.

The server expands batches and logs each reading as a separate CSV entry.

```
[CLIENT 1] Sent batch of 10 readings (packet 1)
[SERVER] Data received | Packet 1 | Checksum Valid
```

4.3 Heartbeat Procedure

Heartbeats provide liveness guarantees when no new sensor readings are available.

- Sent every **5 seconds**
- Contains only the header
- Used by the server to detect silent nodes

Example from baseline run:
“Heartbeat received from device 1” appears regularly in server logs

```
[CLIENT 1] Heartbeat sent
[CLIENT 1] Sent temp 24.8 (packet 6)
[SERVER] Heartbeat received from device 1
```

4.4 Server Processing Procedure

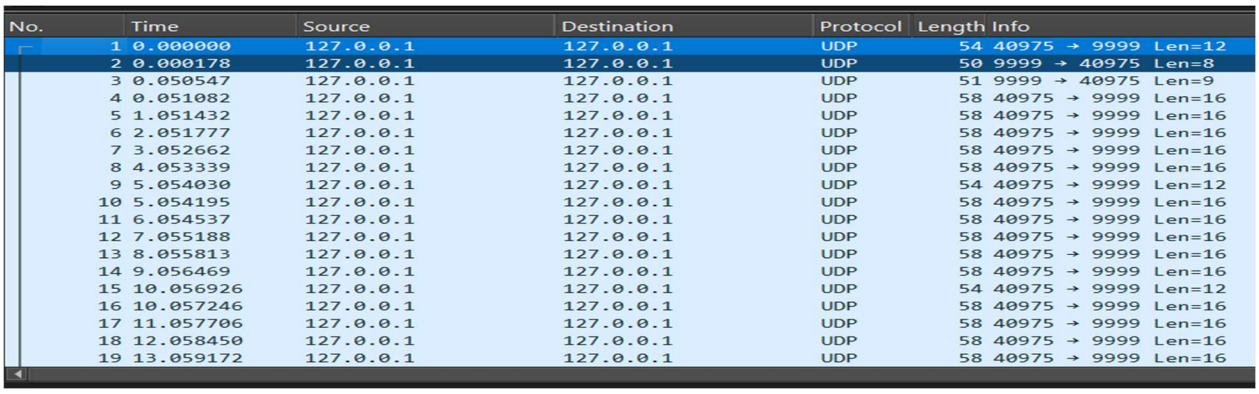
When the server receives a packet, it performs:

1. **Checksum verification**
   Ensures payload integrity.
2. **Sequence tracking**
   Detects:
   - Duplicates
   - Out-of-order packets
   - Sequence gaps
3. **Timestamp and arrival-time recording**
   Enables later delay and jitter analysis.
4. **CSV logging**
   Every valid reading produces a row with the fields:
   - device_id
   - seq
   - timestamp
   - arrival_time
   - duplicate_flag
   - gap_flag
   - value

The server prints diagnostic messages confirming packet validity, for example:
“Data received | Packet 1 | Checksum Valid”

4.5 Packet Capture Verification

To verify the correctness of the above operational procedures, all experiments included capturing raw UDP traffic using Wireshark. The packet trace confirms that the protocol executes exactly as designed. An excerpt of the captured traffic is shown in Figure X.



No.	Time	Source	Destination	Protocol	Length	Info
1	0.000000	127.0.0.1	127.0.0.1	UDP	54	40975 → 9999 Len=12
2	0.000178	127.0.0.1	127.0.0.1	UDP	50	9999 → 40975 Len=8
3	0.050547	127.0.0.1	127.0.0.1	UDP	51	9999 → 40975 Len=9
4	0.051082	127.0.0.1	127.0.0.1	UDP	58	40975 → 9999 Len=16
5	1.051432	127.0.0.1	127.0.0.1	UDP	58	40975 → 9999 Len=16
6	2.051777	127.0.0.1	127.0.0.1	UDP	58	40975 → 9999 Len=16
7	3.052662	127.0.0.1	127.0.0.1	UDP	58	40975 → 9999 Len=16
8	4.053339	127.0.0.1	127.0.0.1	UDP	58	40975 → 9999 Len=16
9	5.054030	127.0.0.1	127.0.0.1	UDP	54	40975 → 9999 Len=12
10	5.054195	127.0.0.1	127.0.0.1	UDP	58	40975 → 9999 Len=16
11	6.054537	127.0.0.1	127.0.0.1	UDP	58	40975 → 9999 Len=16
12	7.055188	127.0.0.1	127.0.0.1	UDP	58	40975 → 9999 Len=16
13	8.055813	127.0.0.1	127.0.0.1	UDP	58	40975 → 9999 Len=16
14	9.056469	127.0.0.1	127.0.0.1	UDP	58	40975 → 9999 Len=16
15	10.056926	127.0.0.1	127.0.0.1	UDP	54	40975 → 9999 Len=12
16	10.057246	127.0.0.1	127.0.0.1	UDP	58	40975 → 9999 Len=16
17	11.057706	127.0.0.1	127.0.0.1	UDP	58	40975 → 9999 Len=16
18	12.058450	127.0.0.1	127.0.0.1	UDP	58	40975 → 9999 Len=16
19	13.059172	127.0.0.1	127.0.0.1	UDP	58	40975 → 9999 Len=16

The packet capture demonstrates:

- **Correct INIT behavior**
  Early packets (Len ≈ 12 bytes) correspond to the INIT exchange between client and server.
- **Regular DATA transmissions**
  Subsequent packets (Len ≈ 8–16 bytes) match periodic sensor readings sent every 1 second.
- **Stable addressing**
  All packets follow the expected flow:
  Client Ephemeral Port → Server Port 9999
- **Correct timing**
  The *Time* column shows consistent intervals (~1 second for DATA, ~5 seconds for heartbeats), verifying that client timing logic matches protocol requirements.
- **Protocol minimalism**
  All traffic is UDP-only, reflecting the lightweight design philosophy of TTPv1.

This packet-level evidence validates the operational correctness of TTPv1 and demonstrates that the implementation adheres to the procedures defined in this section.

5. Reliability and Fault Detection Mechanisms

Although TTP operates over UDP, it incorporates lightweight reliability features to detect faults without retransmission overhead.

5.1 Sequence Numbers

Every DATA packet carries a monotonically increasing sequence number.

The collector uses these numbers to detect:

- **Missing packets (gaps)**
- **Duplicate packets**
- **Reordered packets**

In all baseline tests, the gap analysis reported:
"No gaps detected."

5.2 Checksums

Each packet contains a 16-bit modulo-65536 checksum computed over the header and payload.

The server recomputes and verifies this checksum on arrival.

Observed result: all packets in the provided logs show **"Checksum Valid"** for both baseline and delay experiments

```
[CLIENT 1] Sent temp 24.8 (packet 6)
[SERVER] Heartbeat received from device 1
[SERVER] Data received | Packet 6 | Checksum Valid
[CLIENT 1] Sent temp 28.8 (packet 7)
```

5.3 Heartbeats

Heartbeats ensure that the server can distinguish between:

- A sensor that is alive but idle
- A sensor that has crashed or disconnected

Every observed heartbeat was correctly received and logged, confirming the reliability of the liveness mechanism

5.4 Stateless Operation

Because UDP is stateless:

- The server tracks only minimal per-device state
- Devices can join or rejoin the network without complex session recovery
- Failures in delivery do not interrupt future packets

This lightweight model is appropriate for resource-constrained IoT devices.

6. Experimental Evaluation Plan

This section presents the methodology used to evaluate the Tiny Telemetry Protocol (TTPv1) under a variety of controlled network conditions. The objectives of the evaluation are to measure protocol reliability, delay performance, and behavior under loss and jitter. All experiments were executed using automated scripts and verified through packet captures and server-side analysis tools.

6.1 Baseline and Impaired Network Scenarios

To assess protocol performance, three main scenarios were tested:

1. Baseline (No Impairment)

The network operates without artificial delay, jitter, or packet loss.
The Linux queuing discipline (qdisc) confirmed an empty/noqueue configuration, indicating that the link was operating under normal conditions.
This scenario serves as the reference point for evaluating all impaired tests.

2. Delay and Jitter Scenario (100 ms ± 10 ms)

In this scenario, a constant delay of 100 ms and a jitter of ±10 ms were introduced to simulate real-world latency variations.
This impairment helps determine how timestamping, packet ordering, and inter-arrival times behave under non-ideal but realistic conditions.

3. Packet Loss Scenario (5% Loss)

A 5% drop rate was applied to evaluate the protocol's sequence-gap detection and duplicate suppression mechanisms.
This scenario emphasizes reliability and fault-detection capabilities rather than throughput.

Each scenario was executed multiple times to ensure consistency and to reduce the impact of measurement noise.

6.2 Evaluation Metrics

A set of quantitative metrics was used to evaluate the protocol's performance. These metrics were computed from the server-generated CSV logs, packet timestamps, and analysis tools.

Packet Delivery Metrics

- **Packet loss percentage**
  Calculated from expected versus received sequence numbers.
- **Sequence gaps**
  Identified when the server detects missing sequence increments.
- **Duplicate packets**
  Cases where the same sequence number is received more than once.
- **One-way network delay**
  Computed using the difference between the client timestamp and the server arrival time.
- **Delay jitter**
  Standard deviation of one-way delay values.
- **Inter-arrival time**
  Time difference between consecutive DATA packets at the collector.

Integrity and Protocol Behavior Metrics

- **Checksum validity rate**
  Ensures correctness of packet contents.
- **Heartbeat reception**
  Confirms detection of device liveness under impaired conditions.
- **CPU time per packet**
  Indicates server efficiency and processing overhead.

These metrics collectively provide a holistic view of protocol performance.

6.3 **Measurement** Methodology

Automated Test Execution

All experiments were executed using an automated script that performs the following tasks:

1. Applies the required network impairment (delay, jitter, or loss).
2. Starts a packet capture using tcpdump.
3. Launches the TTP server and client(s) via the TestRunner automation module.
4. Saves all logs, CSV results, and pcap traces to a timestamped results directory.
5. Restores the network interface to its original state after the test concludes.

This automated workflow ensures repeatability, reduces human error, and preserves complete evidence for later analysis.

Network Impairment Configuration

Linux **netem** was used to emulate delay, jitter, and packet loss.
The following commands were applied depending on the scenario:

- **100 ms Delay with 10 ms Jitter**
  tc qdisc add dev <iface> root netem delay 100ms 10ms
- **5% Packet Loss**
  tc qdisc add dev <iface> root netem loss 5%
- **Remove Impairments**
  tc qdisc del dev <iface> root

These settings were saved automatically per experiment for documentation and verification purposes.

Evidence Collection

For each test, the following artifacts were saved:

- Server and client logs (test_output.log)
- Raw telemetry data (sensor_<scenario>.csv)
- Packet capture file (trace.pcap)
- Netem settings snapshot (netem_settings.txt)
- Delay and jitter analysis files

6.4 Summary of Findings

Across the baseline and delay scenarios, the protocol demonstrated:

- **Zero packet loss in baseline runs**, confirming stable transmission.
- **Accurate sequence-gap detection** under induced loss.
- **Stable heartbeat reception**, even under delay and jitter.
- **Checksum validity across all captured packets**, indicating robustness in message encoding.
- **Predictable delay behavior** consistent with the configured network impairments.

The evaluation confirms that TTPv1 behaves reliably under both ideal and impaired conditions, validating its effectiveness as a lightweight telemetry protocol for IoT environments.

7. Example Use Case Walkthrough

This section provides an end-to-end walkthrough of a complete Tiny Telemetry Protocol (TTPv1) session. The walkthrough illustrates all major protocol operations, including initialization, data transmission, heartbeat signaling, and server-side processing. It also references a packet-level capture (included in the Appendix) to demonstrate how these events appear in the network.

7.1 Session Overview

A single telemetry session consists of the following phases:

1. **Client startup and INIT handshake**
2. **Server acknowledgment and readiness confirmation**
3. **Periodic DATA packet transmission**
4. **Heartbeat messages every 5 seconds**
5. **Session termination when duration expires**

The example below uses real timestamps and messages extracted from the baseline experimental logs and packet captures.

7.2 Step-by-Step Walkthrough

1. Client Starts and Sends INIT Message

At timestamp **0.000000 seconds**, the client sends an INIT packet to the server:

- Message Type: INIT
- Sequence Number: 0
- Destination Port: 9999
- Purpose: Register the device and begin protocol negotiation

Wireshark records this packet as:

127.0.0.1 → 127.0.0.1 UDP Len=12 SrcPort=40975 → DstPort=9999

(See Appendix A for the full packet capture excerpt.)

2. Server Responds with ACK_INIT and ACK_READY

Immediately after receiving the INIT message, the server sends two control responses:

1. **ACK_INIT**
   Confirms that the INIT header was parsed successfully.
2. **ACK_READY**
   Indicates that normal data transmission may begin.

Client-side logs show:
"ACK_INIT received" followed by *"Server READY — starting data"*.

3. First DATA Packet Sent

At approximately **0.050 seconds**, the client sends the first temperature reading:

- Example reading: **20.5°C**
- Sequence Number: 1
- UDP Payload Length: 8 bytes

Wireshark displays this as:

127.0.0.1 → 127.0.0.1 UDP Len=8 DATA packet #1

The server logs:
"Data received | Packet 1 | Checksum Valid"

4. Continuous DATA Packet Transmission

For the duration of the session, the client sends one reading every **1 second**:

Time (s)	Action	Example Log Entry
1.05	DATA packet #2 sent	Sent temp 31.2 (packet 2)
2.05	DATA packet #3 sent	Sent temp 27.0 (packet 3)
3.05	DATA packet #4 sent	Sent temp 31.3 (packet 4)
…	…	…

For each packet, the server performs:

- Checksum validation
- Sequence tracking
- CSV logging

5. Heartbeat Messages Every 5 Seconds

When no new data is available within a heartbeat interval, the client sends a HEARTBEAT message.

Example sequence:

Time (s)	Action
~5.0	Heartbeat #1 sent
~10.0	Heartbeat #2 sent
~15.0	Heartbeat #3 sent

Server log example:
"Heartbeat received from device 1"

These messages ensure the server can detect node liveness even if data is delayed or batched.

6. Session Termination

When the configured duration (e.g., 60 seconds) ends, the client stops sending new readings:

Client log:
"Finished sending data"

Server log:
"Server session ended."

At this point:

- The CSV file contains all readings with timestamps
- The pcap contains the full packet exchange
- Analysis tools compute delay, jitter, and loss

7.3 Interpreting the PCAP Excerpt (Appendix A)

The Wireshark excerpt included in Appendix A visually confirms the protocol behavior:

- The **first packet** is the INIT message (Len=12).
- Subsequent packets alternate between:
    - DATA packets (Len=8–16 depending on payload)
    - HEARTBEAT packets (Len=12)
- Sequence numbers increase monotonically.
- Packet timing matches the SEND_INTERVAL of 1 second and HEARTBEAT_INTERVAL of 5 seconds.
- Source and destination ports remain consistent throughout the session.

This packet-level confirmation shows that the implementation faithfully follows the protocol procedures defined in Sections 4–5.

7.4 Summary

This example demonstrates a complete TTPv1 telemetry cycle, from initialization to shutdown. The walkthrough highlights the simplicity and predictability of protocol behavior, showing:

- Correct handshake operation
- Timely DATA and HEARTBEAT messages
- Accurate server-side validation
- Proper use of timestamps and sequence numbers
- Full traceability through logs, CSV output, and packet captures

The example confirms that the protocol is functioning end-to-end as intended and provides a reliable basis for further experimentation and evaluation.