Sabancı University

Faculty of Engineering and Natural Sciences

CS408 – Computer Networks

Environmental Monitoring System via Edge-Enabled Drone Infrastructure

Final Project Report

Team Members:

Efe Yağız San - 22521

Mete Kerem Berk - 30933

Kaan Karahan - 30715

Instructor:

0>zgür Erçetin

Submission Date:

18 May 2025

This project uses a drone as an edge computing node to simulate an environmental monitoring system. It exhibits battery management, real-time data aggregation, anomaly detection, GUI-based visualization, TCP-based communication, and dynamic sensor connection monitoring. The system consists of:

Sensor Nodes (Clients)

Simulated devices called sensor nodes produce environmental data, like humidity and temperature, on a regular basis. Each of the several sensor clients—such as sensor_1, sensor_2, and sensor_3—operates as a TCP client. These nodes send data in JSON format at regular intervals (e.g., every 2 seconds) and establish a connection with the Drone's TCP server using predefined IP and port parameters. They log transmission events and connection statuses in the console but do not have a graphical user interface. Every few seconds, they automatically try to reconnect if the connection is lost.

Drone – Mobile Edge Server (TCP Server and Client)

The drone serves as a processing unit in between. After receiving raw data from several sensor nodes, it summarizes the data and sends it to the Central Server after performing edge processing (such as anomaly detection and rolling averages). It serves as both a TCP client for the Central Server and a TCP server for sensor nodes. The drone mimics actions unique to drones, like running out of battery. It either queues incoming data or momentarily disconnects from the sensors when the battery level falls below a predetermined threshold. Anomalies, connections, disconnections, and battery warnings are all recorded. Real-time sensor data, processed results, anomaly indicators, battery life, and logs are all displayed in its graphical user interface (GUI), which also has interactive features like buttons and sliders to mimic drone behavior.

Central Server (TCP Server)

The Drone sends processed and aggregated data to the Central Server. It serves as a TCP server and uses a graphical user interface (GUI) to display the incoming data in real time. Together with logs of every message received, the GUI shows tables or graphs of temperature and humidity trends. The drone's detected anomalies are prominently displayed. For simulation purposes, data is stored in memory, and the user does not need to enter anything into the GUI.

TCP Communication Flow and Data Direction

Sensor_1 --> |

Sensor_2 --> | [Drone – TCP Server + Client] --> [Central Server]

Sensor_3 --> | - Aggregation & Anomaly Detection - Final GUI

TCP - Battery Simulation & GUI - Logging

Connections

Bidirectional TCP sockets are used to establish connections, and all data flows are unidirectional from sensors to drone to central server. Scalability, modular development, and the capacity to replicate real-time edge-based environmental monitoring are all guaranteed by the design.

1. `sensor.py` – Sensor Node Module

Through a TCP connection, this module replicates a headless environmental sensor that periodically transmits temperature and humidity data to the drone. A distinct sensor_id and connection parameters, including the drone's IP address, port, and transmission interval, are initialized for each sensor. Every few seconds, the sensor produces readings, which are then formatted as JSON objects with the sensor ID, temperature, humidity, and a timestamp in ISO 8601 format. In the event of a failure, the module's persistent TCP client will automatically try to reconnect. Additionally, it facilitates graceful shutdown by displaying a "disconnect" message upon manual termination.

2. `drone.py` – Drone Edge Node Module

The drone serves as a TCP client to send aggregated data to the Central Server and a TCP server to receive data from sensors. It uses threads to manage several sensor connections at once. The drone computes the average temperature and humidity after receiving data and keeps a rolling window of recent readings (such as the last five) per sensor. Configurable thresholds are used for anomaly detection, and out-of-bound values are noted and recorded. The drone mimics battery behavior by pausing data forwarding to the Central Server and going into a "returning to base" state when the battery falls below 20%. Data from sensor nodes is still being received in this state, but it is not queued or forwarded; instead, forwarding merely pauses until the battery level is restored. The GUI shows logs, battery level, computed averages, anomalies, and real-time sensor data. Additionally, it has interactive features like sliders to change thresholds and mimic battery changes.

3. `central_server.py` – Central Server Module

The Drone's processed and aggregated data is sent to this module, which displays it graphically. It functions as a TCP server, listening to incoming drone connections on a port that can be configured. After being parsed, the received messages are shown in two structured tables: one for anomalies and one for average sensor readings. Additionally, it updates each sensor node's connection status based on the messages forwarded by the drone and keeps a real-time log of all messages received, complete with timestamps. A comprehensive and well-structured overview of the system's functioning, including continuous monitoring and feedback on anomaly detection, is given by the Central Server.

Design Rationale

Our environmental monitoring system was designed with modularity, scalability, and real-time responsiveness in mind. Each sensor node, drone node, and central server component was created using the structures and technologies most appropriate for its intended use in a distributed edge computing system. The main design decisions and their justifications are described in detail below.

1. Data Format: JSON for Inter-Component Communication

For all data sent between components, we decided to use JSON as the standard format. This choice was driven by the popularity of JSON, its readability by humans, and Python's built-in support for it via the Json module. JSON offers a natural, lightweight, and readily serializable structure that makes encoding and decoding on both ends of a TCP connection easier because our sensor data consists of key-value pairs (sensor_id, temperature, humidity, timestamp, etc.). Additionally, JSON enables future extensibility, allowing us to add new fields (like sensor location or battery level) without interfering with the parsing logic that is already in place.

2. Communication Model: TCP with Threading for Concurrency

To guarantee dependable, ordered delivery of environmental data—which is essential for precise anomaly detection and time-sensitive monitoring—we chose TCP sockets over UDP. We used multithreading on the Drone server side because the Drone has to handle several Sensor Nodes' connections at once. Because each sensor is managed by a separate thread, the main server loop is not obstructed during reading, processing, or disconnection.
Because threading is easier to handle blocking socket operations and more compatible with GUI frameworks like tkinter, which are not asynchronous by nature, we chose it over asynchronous models (like asyncio). A single-threaded model was adequate on the Central Server side because managing a single stream from the Drone is less taxing on the connection.

3. Rolling Window Aggregation for Edge Processing

The drone uses a rolling window of the last N readings (with N set to 5) to process sensor data. This method lessens the effect of noise or temporary anomalies and enables more consistent average values. To facilitate quick, append and slice operations, we store recent readings for each sensor in Python lists. Even on edge devices with limited resources, this design guarantees that edge processing is computationally efficient and lightweight, making it appropriate for real-time operation.

4. Anomaly Detection Based on Configurable Thresholds

We used temperature thresholds that could be adjusted by the user to implement anomaly detection. The need for flexibility across various monitoring scenarios (such as cold environments versus industrial heat zones) is reflected in the decision to make thresholds adjustable via the Drone GUI. Additionally, this design makes the system interactive and flexible by enabling live tuning during tests or demonstrations.

5. Battery Simulation and “Return to Base” Behavior

We added a battery level input to the Drone GUI in order to mimic drone constraints. Data forwarding is stopped by the drone going into a "returning to base" state when the level falls below 20%. In scenarios involving mobile computing, this simulates realistic energy-aware edge behavior. Testers have control over when and how this behavior is triggered thanks to a slider in the GUI that allows for dynamic battery drain simulation.

6. GUI Design with Real-Time Feedback and Interactivity

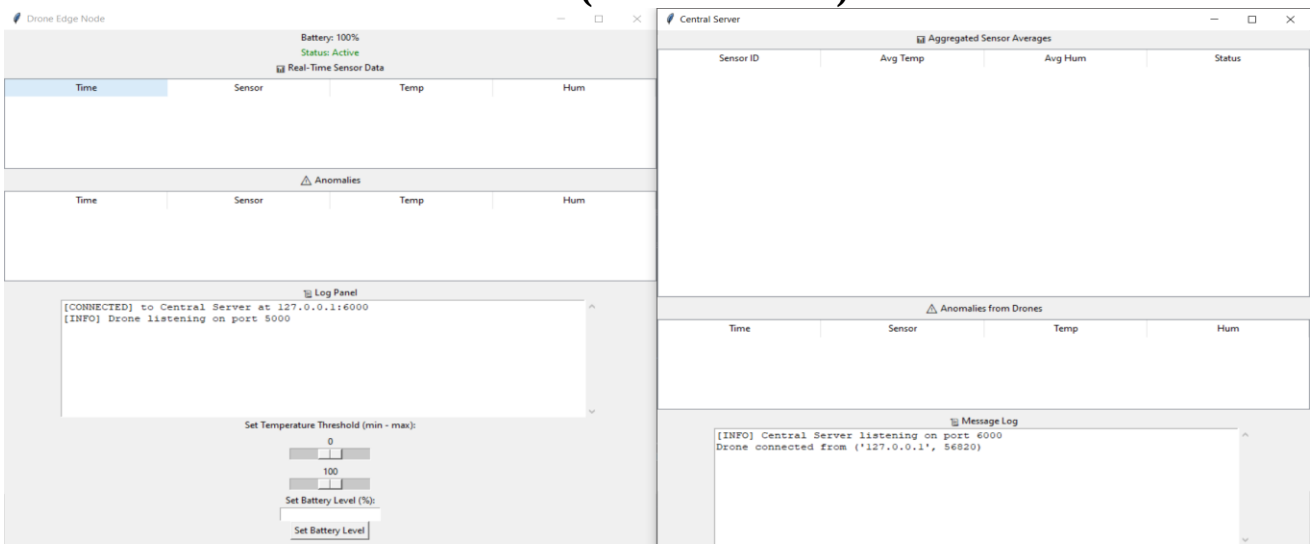
Tkinter was used to create the interactive graphical user interfaces (GUIs) for the Drone and Central Server because it integrates natively with Python and is simple to deploy without the need for external dependencies. The GUIs consist of interactive widgets (buttons, sliders) for simulating batteries and thresholds, scrollable tables that display real-time sensor readings and anomalies, and status banners and log panels for transparent event tracking. This design improves the user experience, makes live demonstrations possible, and gives graders a visual and intuitive way to see how the system behaves. To minimize cognitive load, GUI elements are logically arranged to separate incoming data, processed outputs, and system messages. All things considered, our design places a high value on testability, clarity, and robustness, guaranteeing that each part can be created, executed, and debugged separately while still working as part of a broader edge computing ecosystem.

Test Scenarios

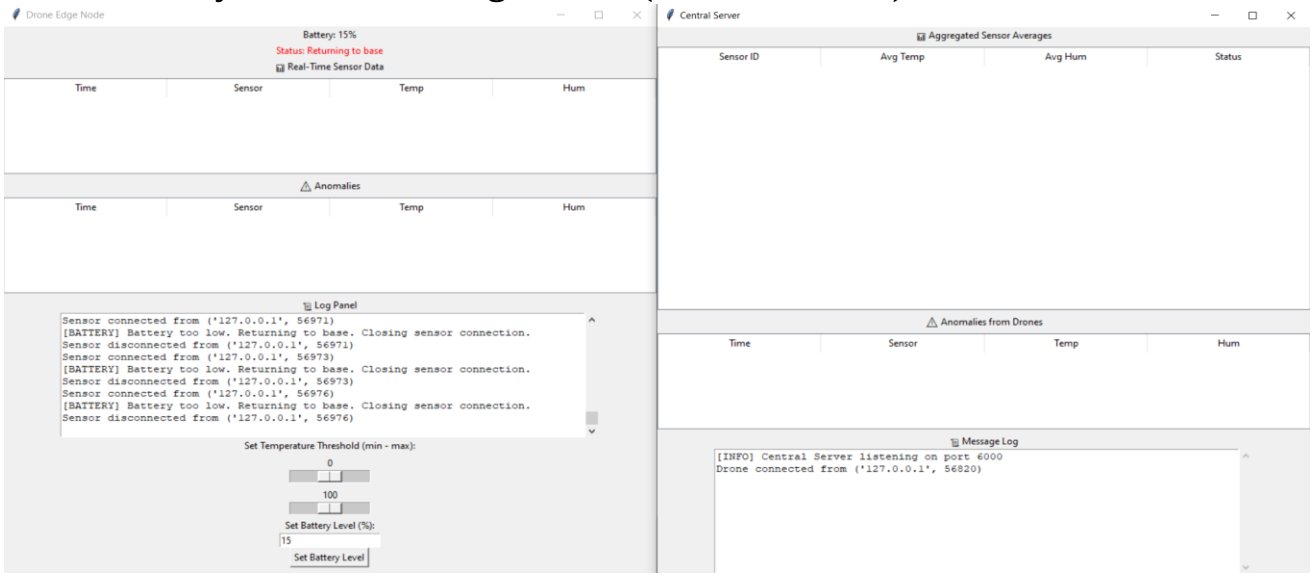
To confirm the system's fundamental features and resilience in practical use cases, it was put through four different testing scenarios. In order to confirm the system's end-to-end functionality, the first scenario concentrated on Normal Operation. Two sensor nodes, sensor1 and sensor2, were set up with 2-second transmission intervals after the Central Server was launched on port 6000 and the drone connected to the server. Both sensor nodes were able to establish a successful connection with the drone during this test, and they started sending data in JSON format on a regular basis. Using the last five readings, the Drone GUI calculated rolling averages and displayed incoming raw data accurately. The anomaly table was left empty because every value was within the typical range. After that, the Drone sent the combined data to the Central Server, which verified that every sensor was operational and updated its graphical user interface (GUI) in real time to reflect the current sensor averages. In order to evaluate the system's resistance to node failures, the second scenario tested Sensor Disconnection. The initial setup procedures were the same as in the case of regular operation. One sensor node was manually terminated using Ctrl+C following multiple successful transmissions. Before it shut down, the sensor gave the drone one last "disconnect" message. The disconnection event was immediately recorded by the drone, which then updated its graphical user interface to reflect the modification. The disconnected sensor stopped sending it more data. In order to confirm proper sensor state propagation across components, the Central Server simultaneously changed the sensor's status to "Disconnected" and reflected this state change in its user interface. Low Battery "Return to Base" Mode, which mimics battery-constrained behavior in edge computing environments, was the subject of the third scenario. The drone's battery level was manually lowered below 20 percent using the GUI slider after the system was initialized with at least one active sensor. In response, the drone entered the state known as “Returning to base.” The battery-critical mode was indicated visually on the drone's graphical user interface. In this state, data forwarding to the Central Server was momentarily stopped by the Drone. Incoming data was either queued or discarded, depending on how it was implemented. The accuracy of the system's internal state management was confirmed when the drone returned to normal operation and carried on sending data to the Central Server without needing to restart once the battery level was raised above the threshold. Anomaly Detection was the last scenario. One sensor was set up to transmit an out-of-range temperature value (such as 1000°C) during a transmission cycle when the system was first launched under typical circumstances. Using its preset threshold parameters, the drone detected the abnormal value right away. The anomaly was visually marked in the drone's GUI along with the sensor ID and timestamp, and it was noted in the log. The forwarded summary that was sent to the Central Server also contained the anomalous data. After receiving the anomaly data, the Central Server showed it in its own anomaly table, verifying that outlier had been appropriately identified, recorded, and spread. When taken as a whole, these four scenarios verify the system's primary functions: fault tolerance, edge-level decision making, real-time communication, and user-facing visualization. They show that the architecture can accommodate continuous operation and adapt to changes in the environment and system in a dynamic manner.

Screenshots

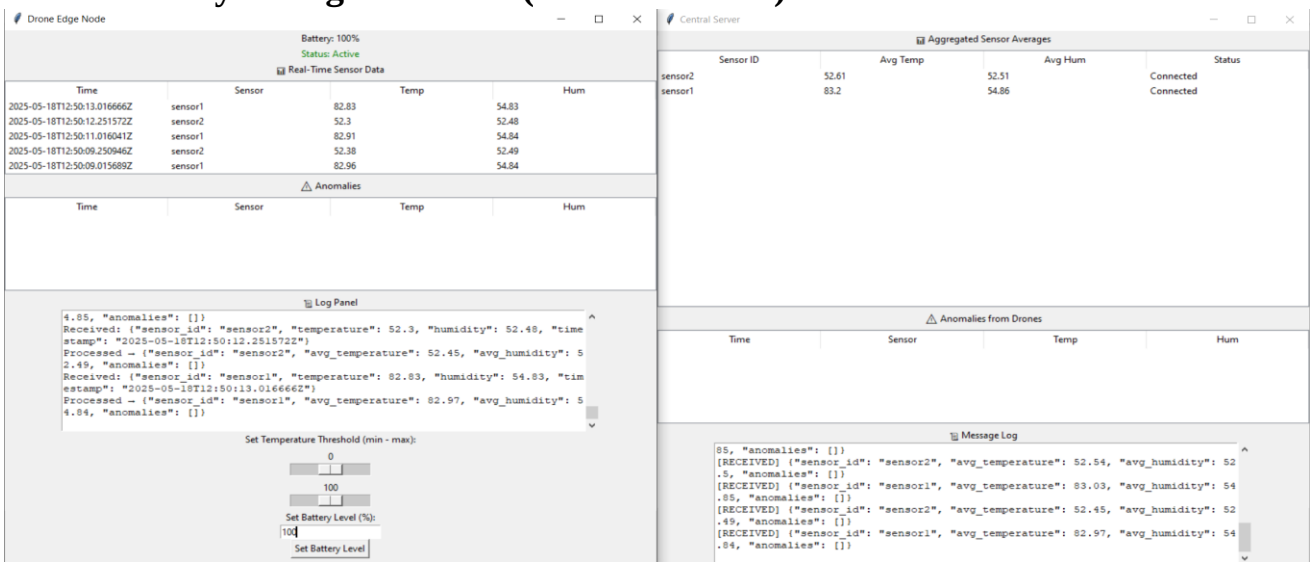
Drone Connected to Central Server (Sensors Offline):



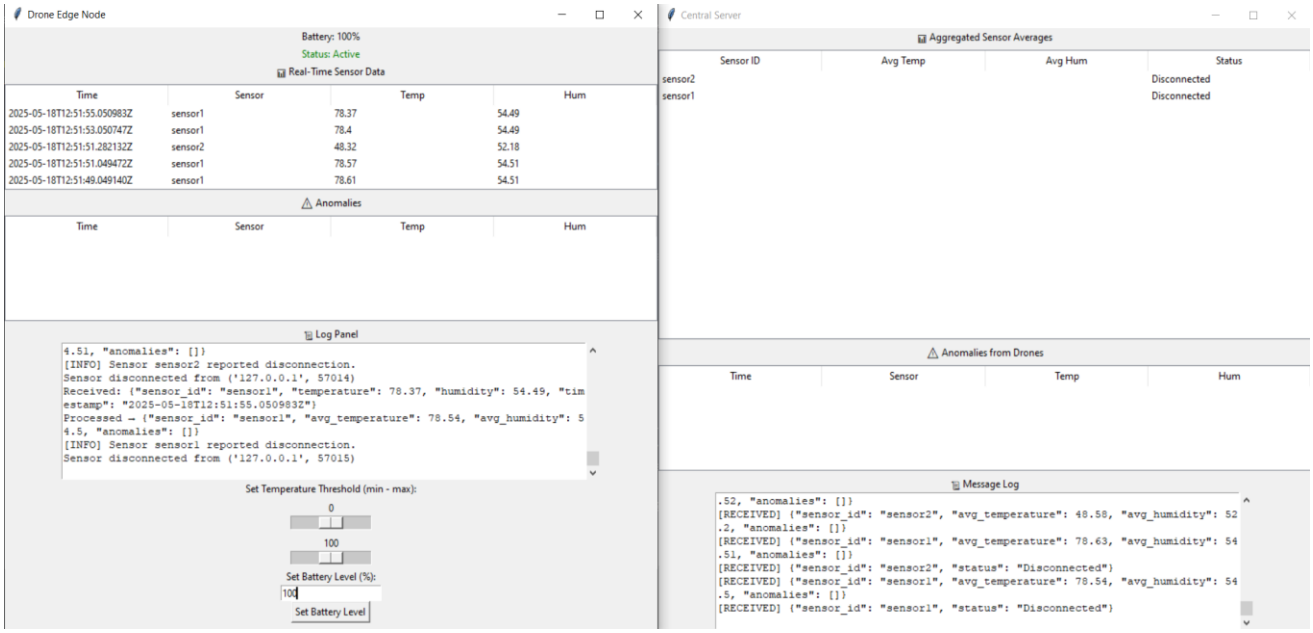
Drone Battery Low / Returning to Base (Sensors Online):



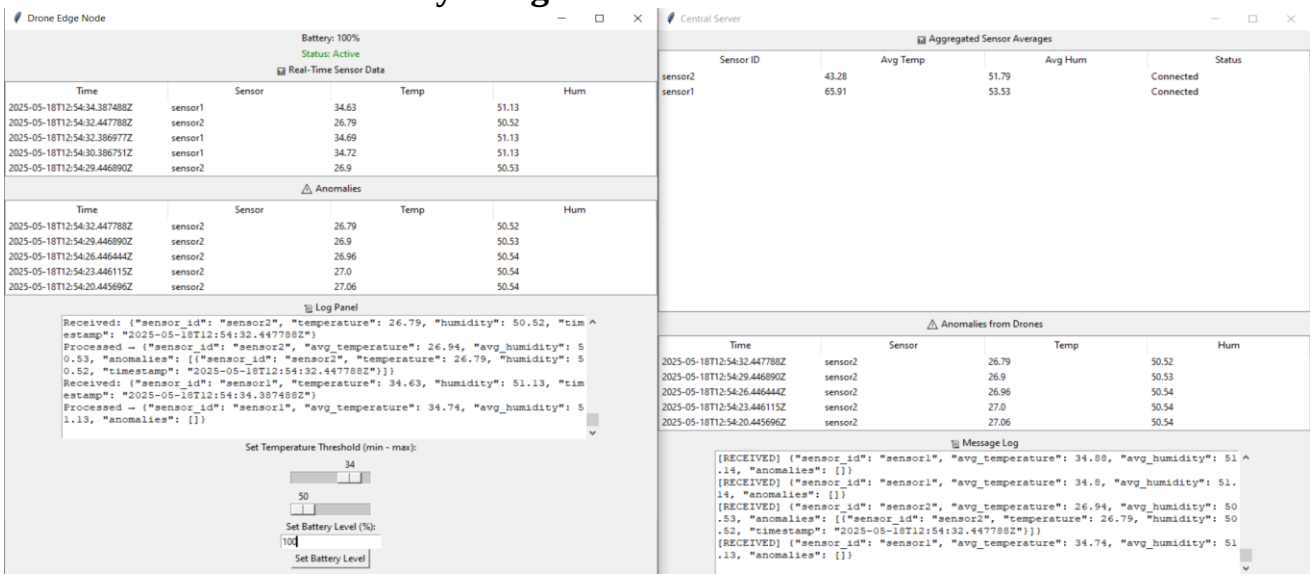
Drone Battery Charged / Active (Sensors Online):



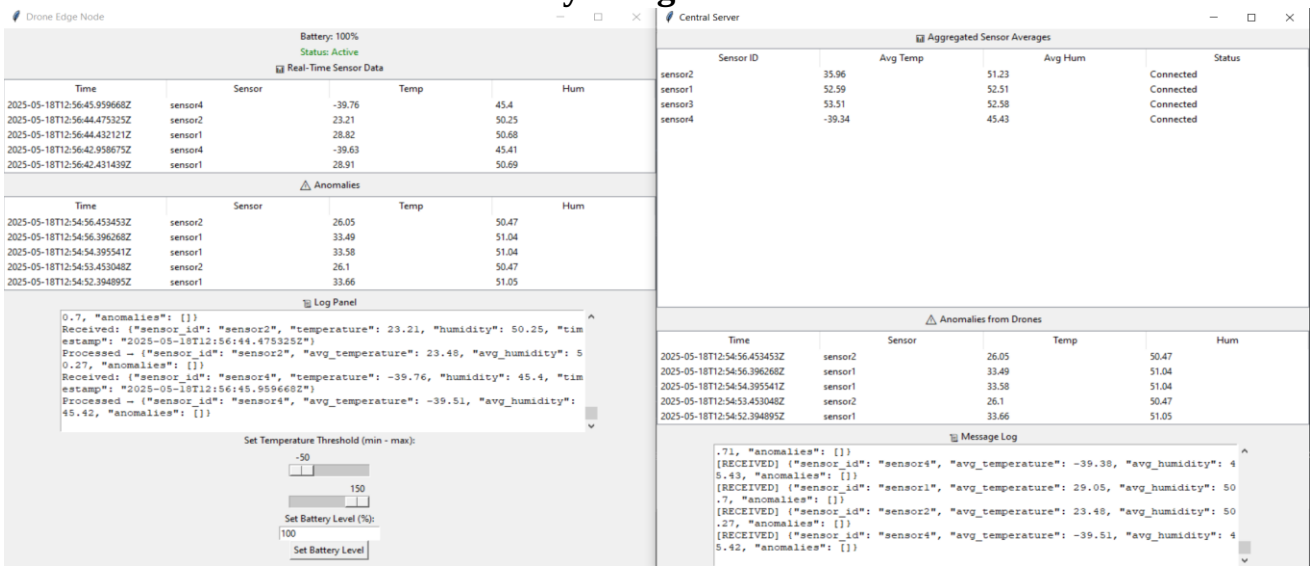
Sensors Turned Off:



Sensors On and In Anomaly Range:



More Sensors and Outside Anomaly Range:



Sensors and Drone Disconnected: