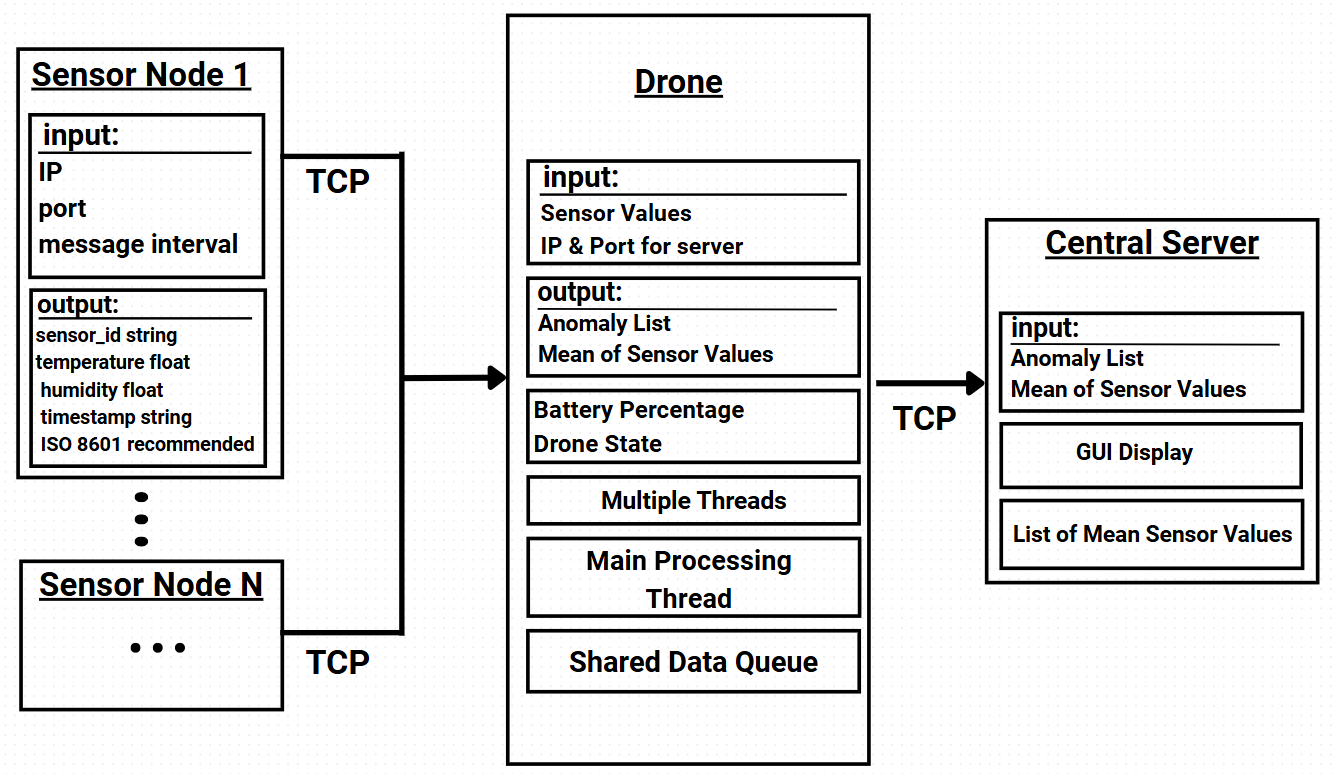
**CS 408 PHASE 1**

**Ertem Ata Kavaz - 32790**

**Ege Çetinkaya - 32661**

**Karya Üstünçelik - 30903**

**1) ARCHITECTURE DIAGRAM:**

**2) MODULE DESCRIPTION:**

**1) Sensor Module: python sensor.py --ip --port --interval**

* The sensor module requires **three** inline command parameters: ip, port, and message interval.
* It will generate random data for each specified data point using Python’s random class.
* The sensor will attempt to connect to the specified ip and port. If the connection fails, it will retry every **5** seconds.
* Once connected via TCP, it will send generated data as **JSON** at the specified **message interval**.

**{**

**"sensor\_id": 1,**

**"temperature":34, 22.5,**

**"humidity": 55,**

**"timestamp": "2025-02-10T10:00:00Z"**

**}**

**2) Drone Module: python drone.py --server\_ip --server\_port**

* The drone module requires **two** inline command arguments: the **port** and the **IP address** of the central server. A thread will connect to the central server using these parameters via TCP.
* There is a **“many-to-one"** relationship between sensor modules and the drone module. The drone listens for incoming sensor connections in an infinite loop.
* When a sensor connects, the main process creates a new thread for that sensor, resulting in a **“one-to-one"** relationship between the sensor and its dedicated thread.
* **Sensor Handling Thread:** Each sensor-handling thread checks the drone state:
  1. If the drone is **running**, it receives incoming data and places it into a shared queue.
  2. If the drone is **charging**, the thread disconnects from the socket and terminates.

Multiple threads share both a **lock** and the **queue** to manage concurrency, preventing race conditions while reading and writing.

* **Processing Thread:** Every **5** seconds, the main processing thread acquires the lock and processes all items in the queue. During this process:
  1. Values outside defined thresholds are added to an **anomaly list**.
  2. A mean is computed for each type of value, which is then stored.
  3. The lock is then released.

After updating means and the anomaly list, the processing thread sends both to the central server in JSON format.

* **Battery Thread:** A separate thread manages the **battery percentage**:
  1. Starts at 100% and decreases by 5% every 5 seconds.
  2. When it drops below a predetermined threshold, it triggers the drone’s **charging state** and sleeps for 5 seconds to simulate recharging.
  3. After 5 seconds, it sets the battery back to 100% and the drone resumes **running**.

While in **charging** state, all sensor sockets are disconnected.

* A **Tkinter**-based GUI displays connected sensors, their latest received values, any anomalies, the battery percentage, and the drone’s state.

**{**

**"average\_values": {**

**"temperature": 20.3,**

**"humidity": 45.2**

**},**

**"anomalies": [**

**{**

**"sensor\_id": 1,**

**"temperature": 1000,**

**"timestamp": "2025-02-10T10:05:00Z"**

**}**

**]**

**}**

**3) Central Server: python central\_server.py --port**

* It listens indefinitely for incoming connections from the drone.
* Upon receiving **JSON** containing mean values and anomaly data, it parses and processes the information.
* The central server has its own GUI that displays the mean values, anomalies, and relevant statistics in real time.

**3) DESIGN RATIONALE:**

**Random Data Generation:** We simulate sensor inputs using Python’s random module, eliminating the need for actual hardware while still testing the system’s functionality end to end.

**Dedicated Thread per Sensor:** Assigning each sensor its own thread enables concurrent data gathering, ensuring one slow sensor does not block others.

**Lock & Queue:** We employ a lock-and-queue mechanism to prevent race conditions when multiple threads simultaneously read and write shared data.

**Tkinter:** Chosen for its simplicity and direct integration with Python, allowing easy creation of user-friendly GUIs.