CSE 157 Lab 4: Cloud Integration and Fault-Tolerant Data Logging with Raspberry Pis

Mylo Lynch

Overview: Finalizing a Real IoT System

This lab represented the final stage of our IoT sensor network, where the focus shifted from peer-to-peer data exchange to end-to-end cloud integration, robustness, and data visualization. We weren't building from scratch. Our sensors were already configured in Lab 2, and our networking logic (polling and token-ring)was developed and tested in Lab 3 (but now way more robust because of our new ACK + Timeout based approach). What this lab added was a MySQL database layer hosted on a separate machine, a Flask-based web app for visualization, and a set of retry-based logic flows to maintain functionality even when nodes failed or disconnected. We made our system work despite failure, and sent that data in a meaningful, visual format that a user or stakeholder could interact with. Although Lab 4 focused on cloud integration, the foundation of our IoT system was established in Lab 1, where we first flashed Raspberry Pi OS (Lite) onto each Pi and configured both infrastructure mode and ad-hoc mode networking. Infrastructure mode used standard WiFi via a router to allow broader Internet access and interaction with external machines like the database host. Ad-hoc mode created a direct mesh between Pis using static IPs, which was essential in Lab 3 for peer-to-peer communication without relying on a router. During initial setup, we used Raspberry Pi Imager to preconfigure

SSH access, WiFi credentials, and default users. Additional network configuration—such as setting a static IP or editing WiFi settings—was done using the wpa_supplicant.conf file and raspi-config. Passwords were managed through passwd, and SSH keys were set up for secure remote login. These flexible configuration methods allowed us to recover from networking issues and manage headless Pis effectively throughout the project.

To remotely access each Pi, I typically used ssh pi@<pi-ip>, which let us run scripts and monitor logs from my development machine. This was especially useful once the Pis were deployed away from a monitor. In practice, I found myself using SSH constantly it streamlined testing, debugging, and file transfers without needing to physically access the device.

Our system diagram shows the interactions between the three Pis, the MySQL database hosted on a separate XAMPP-enabled computer, and the Flask web server that reads data and serves time-series graphs. The network infrastructure supported both internal communication between Pis and outward facing access for the web dashboard (which is why we used infrastructure mode and not adhoc mode.).

Network interfaces played a key role in how each Pi communicated. wlan0 was used for all socket connections to other Pis, SQL insertions, and HTTP API calls to forecast services. The lo interface handled local routing, while eth0 remained unused. Comparing adhoc and infrastructure modes, adhoc assigned static IPs and manual routing, perfect for direct Pi-to-Pi socket programming without a DHCP server. Infrastructure mode offered more flexibility and external access, which is essential for Flask and database hosting. Both setups had pros and cons, but our system used a hybrid approach depending on the lab phase: adhoc for Lab 3’s isolated communication tests, and infrastructure mode for Lab 4’s full stack connectivity.

Components in the System

The system had five core components:

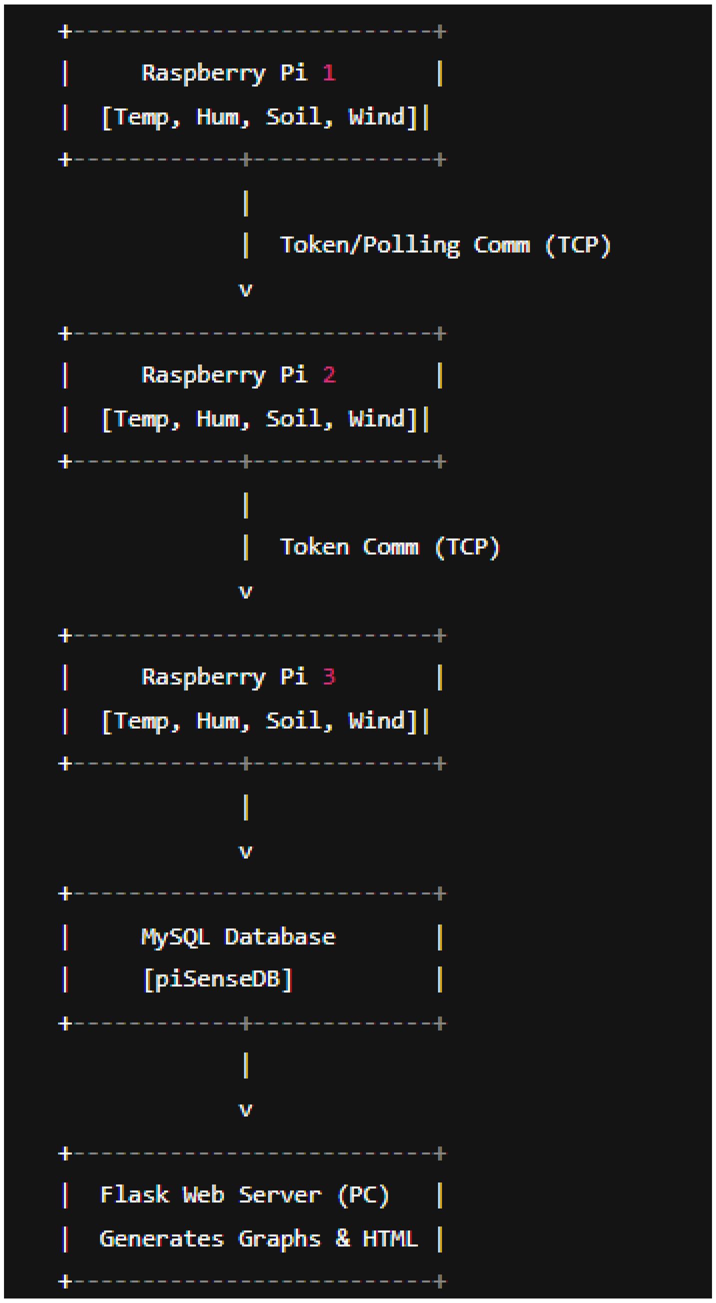
1. **Three Raspberry Pis** — Each Pi collected data from a temperature/humidity sensor (SHT31-D), a soil moisture sensor (Seesaw), and a wind sensor (ADS1015 + Anemometer).

2. **Polling Script (Primary Pi)** — The centralized controller that queried secondary Pis, assembled the round's data, and inserted it into the database.

3. **Token Ring Script (All Pis)** — A decentralized script where the token circulated among Pis, collecting and aggregating sensor data before being committed to the DB.

4. **MySQL Database Server** — Hosted on a separate machine running XAMPP; it stored all sensor values in three tables, one per Pi.

5. **Flask Web Server (web-app.py)** — Rendered graphs using sensor data + forecast data from Open-Meteo.



```
+--------------------------+
|     Raspberry Pi 1       |
|  [Temp, Hum, Soil, Wind]|
+------------+-------------+
             |
             |  Token/Polling Comm (TCP)
             v
+--------------------------+
|     Raspberry Pi 2       |
|  [Temp, Hum, Soil, Wind]|
+------------+-------------+
             |
             |  Token Comm (TCP)
             v
+--------------------------+
|     Raspberry Pi 3       |
|  [Temp, Hum, Soil, Wind]|
+------------+-------------+
             |
             v
+--------------------------+
|     MySQL Database       |
|     [piSenseDB]          |
+------------+-------------+
             |
             v
+--------------------------+
|  Flask Web Server (PC)   |
|  Generates Graphs & HTML |
+--------------------------+
```

Web Server Configuration

The Flask server was deployed on the same machine running the MySQL database using XAMPP. The base Flask app worked with minimal changes, what required customization was the rendering of matplotlib plots without GUI dependencies. We configured matplotlib.use("Agg") to allow plots to be rendered server side and saved as static images.

We created a static/ folder to store generated graphs for temperature, humidity, wind, and soil moisture. Flask routes were defined to dynamically:

- Query the SQL tables
- Calculate per timestamp averages
- Overlay forecast data (retrieved via Open-Meteo's HTTP API)
- Render all plots into a responsive HTML template

The server listens on port 8080 and can be accessed from any browser on the local network. While we didn't implement full RESTful endpoints or real time graphing with JavaScript, the approach struck a balance between simplicity, speed, and robustness. Forecast data is refreshed every session/refresh, ensuring accurate comparative overlays.

- Each Pi collects data from 4 sensors

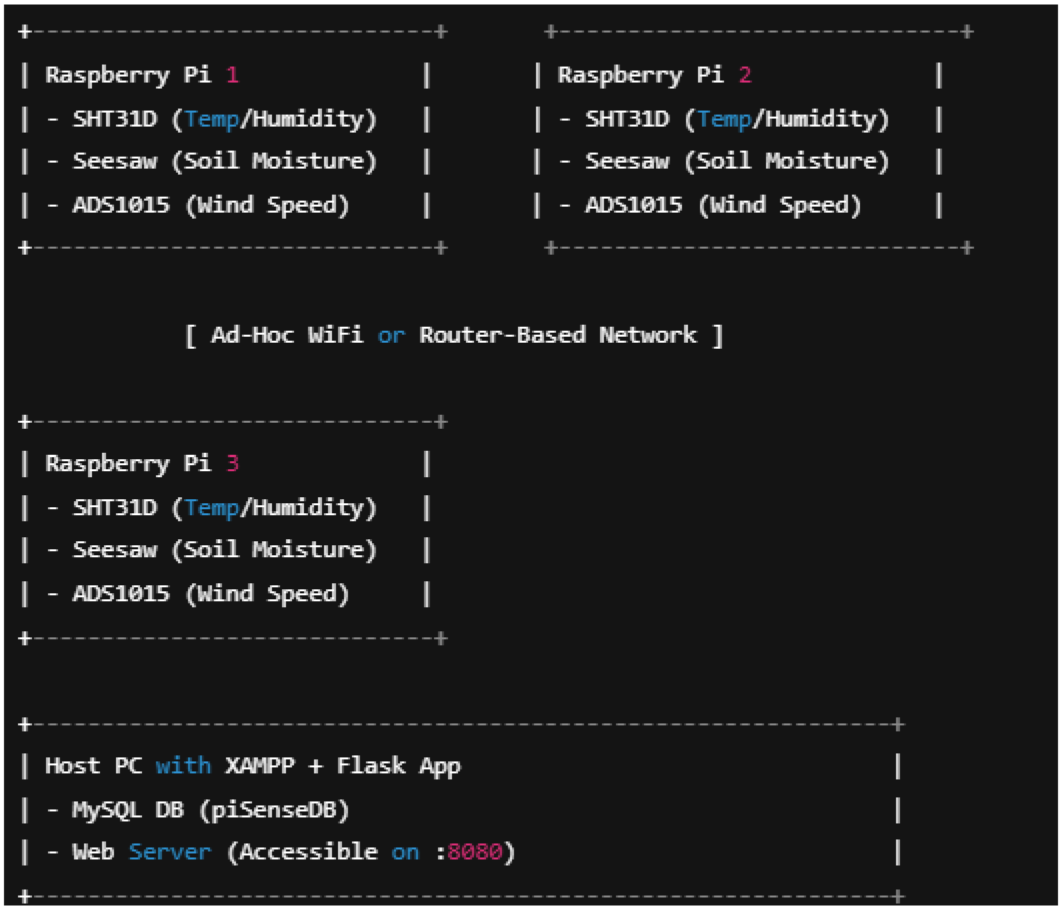
- Token ring flows Pi1 → Pi2 → Pi3 → DB

- Polling queries initiated by Pi1

- MySQL DB stores data from all Pis

- Flask app reads DB, generates plots

Physical set up:



```
+----------------------------+      +----------------------------+
| Raspberry Pi 1             |      | Raspberry Pi 2             |
| - SHT31D (Temp/Humidity)   |      | - SHT31D (Temp/Humidity)   |
| - Seesaw (Soil Moisture)   |      | - Seesaw (Soil Moisture)   |
| - ADS1015 (Wind Speed)     |      | - ADS1015 (Wind Speed)     |
+----------------------------+      +----------------------------+

             [ Ad-Hoc WiFi or Router-Based Network ]

+----------------------------+
| Raspberry Pi 3             |
| - SHT31D (Temp/Humidity)   |
| - Seesaw (Soil Moisture)   |
| - ADS1015 (Wind Speed)     |
+----------------------------+

+---------------------------------------------------------------+
| Host PC with XAMPP + Flask App                                |
| - MySQL DB (piSenseDB)                                        |
| - Web Server (Accessible on :8080)                            |
+---------------------------------------------------------------+
```

Development Environment

To develop and debug the Flask application and Pi scripts, I used vim as my primary text editor directly on the Raspberry Pi. I chose vim because I'm already familiar with its modal editing system and keyboard shortcuts, which let me work quickly without needing a mouse or GUI. I didn’t install a full IDE like VSCode or Thonny because headless SSH access made terminal-based editing more practical. All package installations (e.g., Flask, mysql-connector-python) were handled via pip3 inside a virtual environment I set up with python3 -m venv venv. This setup allowed me to keep dependencies isolated and reproducible across devices.

Database Design & Usage

The database was built using MySQL through phpMyAdmin. We created a schema called piSenseDB containing three tables:

- sensor_readings1
- sensor_readings2
- sensor_readings3

Each table contained:

- timestamp (UTC)
- temperature (°C)

- humidity (%)

- wind_speed (m/s)

- soil_moisture (%)

By separating data by device, we reduced complexity while allowing easy filtering, averaging, and fault isolation. This also aligned well with our fallback logic: polling could insert nulls for any downed Pi without risking primary table integrity, token ring also had nulls for missing sensor readings.

The Flask app accessed the DB via SQL SELECT queries, ordered by timestamp. These results were binned by time intervals and used to calculate per-round averages. Null values were excluded from the average calculations. On the insertion side, we used mysql.connector to execute parameterized INSERT queries from Python scripts.

Diagram:

```
+---------------------+
| sensor_readings1    |
+---------------------+
| timestamp      DATETIME (UTC) |
| temperature    FLOAT          |
| humidity       FLOAT          |
| wind_speed     FLOAT          |
| soil_moisture   FLOAT          |

+-------------------------------+


+---------------------+

| sensor_readings2    |

+---------------------+

| timestamp       DATETIME (UTC) |

| temperature     FLOAT          |

| humidity        FLOAT          |

| wind_speed      FLOAT          |

| soil_moisture   FLOAT          |

+-------------------------------+


+---------------------+

| sensor_readings3  |

+---------------------+

| timestamp       DATETIME (UTC) |

| temperature     FLOAT          |

| humidity        FLOAT          |

| wind_speed      FLOAT          |

| soil_moisture   FLOAT          |

+-------------------------------+
```

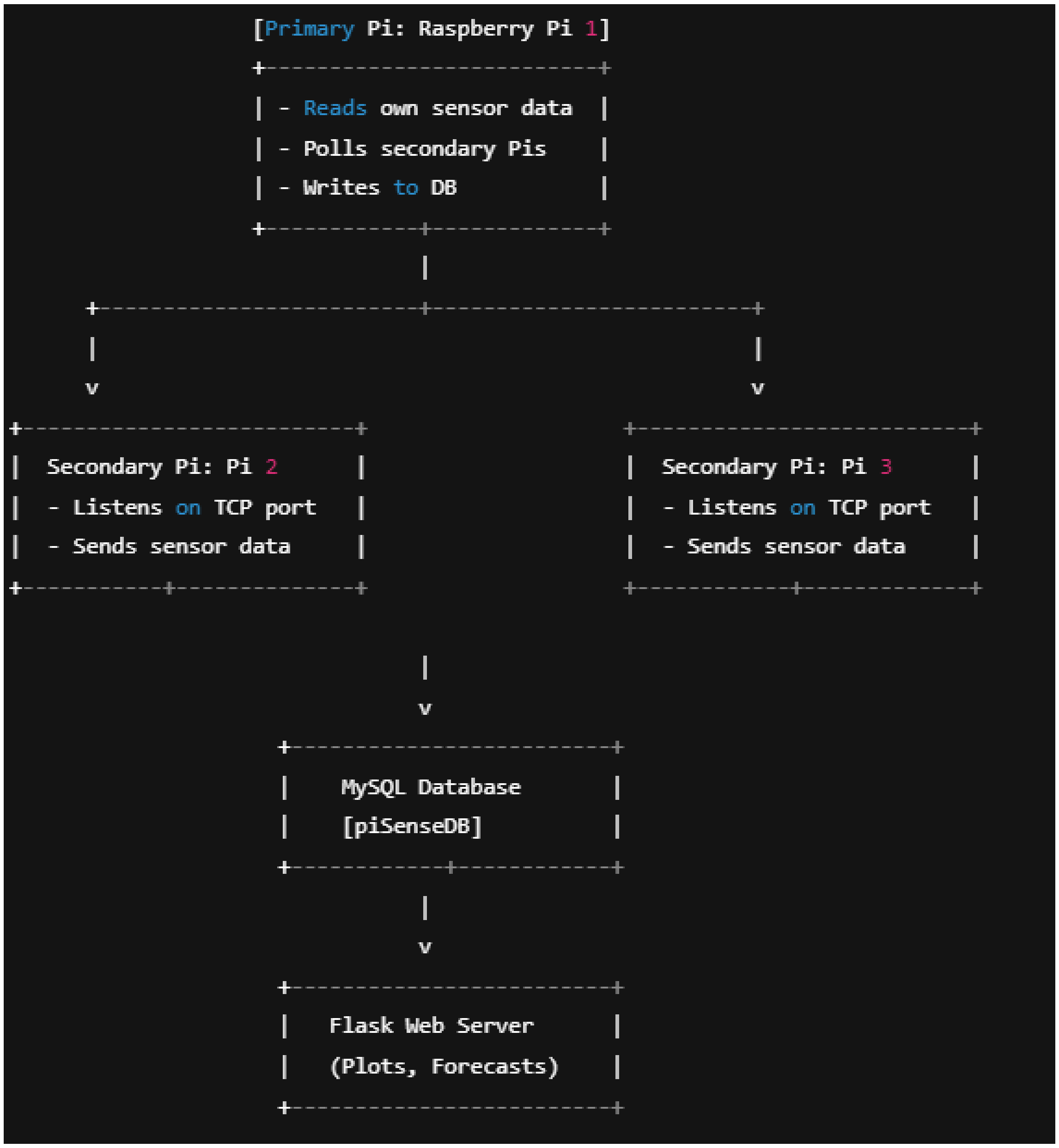
Polling vs. Token Ring: Architecture and Analysis

Polling Design

In the polling architecture, Pi 1 initiates each round every 10 seconds. It reads its own sensors and sends TCP requests to Pi 2 and Pi 3. If a Pi doesn't respond within 5 seconds, Pi 1 logs a timeout, inserts null values for that Pi's row, and continues.

This design offers high availability at the cost of completeness. Even if one or two Pis are down, data from the others is still inserted into the database and reflected on the dashboard. It tolerates failure well, but introduces slight desynchronization, Pi 1's readings are freshest, while Pi 3's are ~10 seconds stale by the time it responds.

This is ideal for systems that prioritize uptime, such as greenhouses or live dashboards. It also centralizes all responsibility into the primary Pi, which simplifies coordination but introduces a single point of failure.



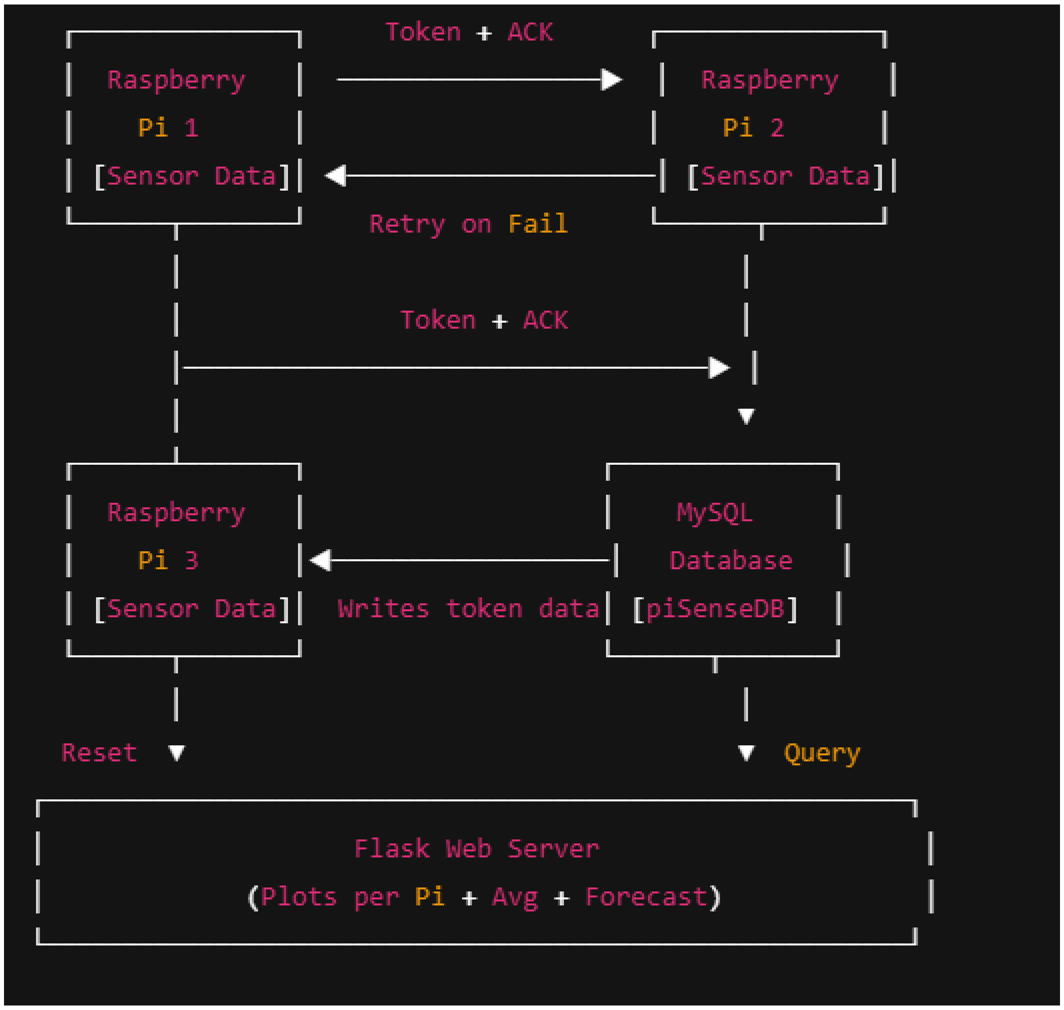
```
                   [Primary Pi: Raspberry Pi 1]
                   +--------------------------+
                   | - Reads own sensor data  |
                   | - Polls secondary Pis    |
                   | - Writes to DB           |
                   +------------+-------------+
                                |
      +-------------------------+-------------------------+
      |                                                   |
      v                                                   v
+--------------------------+                +--------------------------+
|  Secondary Pi: Pi 2      |                |  Secondary Pi: Pi 3      |
|  - Listens on TCP port   |                |  - Listens on TCP port   |
|  - Sends sensor data     |                |  - Sends sensor data     |
+-----------+--------------+                +------------+-------------+

                                |
                                v
                     +-------------------------+
                     |    MySQL Database       |
                     |    [piSenseDB]          |
                     +-----------+-------------+
                                 |
                                 v
                     +-------------------------+
                     |   Flask Web Server      |
                     |   (Plots, Forecasts)    |
                     +-------------------------+
```

Token Ring with Timeout + ACK

Our token ring model was a robust, decentralized system. Pi 1 starts the round by sending a JSON token with its own sensor data to Pi 2. If Pi 2 doesn't ACK within 5 seconds, Pi 1 sends the token to Pi 3. This retry logic continues until a responsive node is found. If no responsive Pi is found, Pi1 interacts with the database and retries at the start of the next round. Each Pi that receives the token appends its data, sends it forward, and repeats the ACK process.

Once the final Pi receives and appends its data, it writes the complete round to the SQL database and sends a reset token to Pi 1. If Pi 1 is down, it sends it to Pi 2. If both are unreachable, it starts the next round by itself and will retry reaching the others after.

This logic ensures data is never lost and fixes the infinite hanging issue our previous token ring approach had for Lab 3. You can change the timeout value to be as fast or as slow as you'd like, making this approach optimal in settings like assembly lines and factories, where sensor data must be acquired at a fixed time. Better hardware can allow for the timeout value to get smaller and smaller as how fast tokens can be passed and ACKs can be received affects the minimum timeout value the system can have. No longer an infinitely hanging system that needs complete data, our Token Ring system has become a predictable and fast system with minimal latency on failure.


Token + ACK
Raspberry Pi 1 [Sensor Data]
Raspberry Pi 2 [Sensor Data]
Retry on Fail
Token + ACK
Raspberry Pi 3 [Sensor Data]
MySQL Database [piSenseDB]
Writes token data
Reset
Query
Flask Web Server
(Plots per Pi + Avg + Forecast)

Working normal

START OF ROUND

[Pi 1] creates token with its sensor data

```
    |
    | Token + ACK
    v
```

[Pi 2] appends its data

```
    |
      | Token + ACK
      v
[Pi 3] appends its data
      |
      | Writes token to DB
      v
[MySQL Database] <- All 3 rows inserted
      |
      | Reset signal
      v
[Pi 1] receives reset, starts new round
```

Pi 2 down (works similarly if only Pi3 is down but Pi2 takes on Pi3 responsibilities)

```
START OF ROUND
[Pi 1] creates token with its data
      |
      | Token sent to Pi 2
      | (NO ACK received within timeout)
      |
      | Retry logic → fallback to Pi 3
      v
[Pi 3] appends data
      |
      | Writes token to DB (Pi 1 + Pi 3 data only)

      v

[MySQL Database]

      |

      | Reset signal

      v

[Pi 1] receives reset, starts new round
```

Pi1 down mid round

```
START OF ROUND

[Pi 1] created token, sent to Pi 2

    |

    v

[Pi 2] appends data

    |

    | Token sent to Pi 3

    v

[Pi 3] appends data

    |

    | Writes token to DB

    v

[MySQL Database]
```

```
    |
    | Sends reset token to Pi 1
    | (NO ACK — Pi 1 is down)
    |
    | Retry logic → send reset to Pi 2
    v
[Pi 2] receives reset, starts new round
only pi 1 online


START OF ROUND
[Pi 1] creates token with its data
    |
    | Attempts Pi 2 → No ACK
    |
    | Attempts Pi 3 → No ACK
    |
    | Writes token (Pi 1 data only) to DB
    v
[MySQL Database]
    |
    | Starts next round after timeout
    v
[Pi 1]
```

Robustness Implementation

Designing for robustness was central to our implementation, especially with the requirement that the system continue collecting and storing data even if one or more Pis went offline. In both polling and token ring configurations, we deliberately engineered mechanisms to ensure that the system did not stall, crash, or silently fail.

In the polling system, robustness was mainly achieved through the use of socket timeouts (sock.settimeout(5)) and exception handling. The primary Pi initiates TCP connections to each secondary, and if a connection fails or takes too long to respond, the system logs the timeout and proceeds by inserting null values for that Pi's sensor data into the MySQL database. This keeps the round moving forward and avoids indefinite blocking. Instead of crashing or skipping the entire round, we ensure that partial data is still recorded, which keeps the system useful even under failure conditions. We use explicit try/except blocks around every socket connection and sensor read, ensuring that the primary doesn’t fail due to a bad port, a broken sensor, or even a malformed response from a secondary.

In contrast, our token ring system emphasizes distributed robustness through timeout and ACK logic. Each Pi, upon receiving the token, attempts to forward it to the next Pi in the ring. After sending, it starts a timeout countdown (5 seconds) and waits for an ACK. If the ACK is not received, the sender Pi assumes that the next node is down and attempts to forward the token to the next available Pi in the fallback order. This retry based design means the system doesn’t immediately give up on unreachable nodes, it reroutes, giving each node a chance to contribute if and when it becomes reachable again.

To prevent data loss and ensure system continuity, we also implemented extensive error checking mechanisms in the token ring protocol:

- Malformed tokens are caught using JSON validation and safely discarded with a debug log.

- Dropped or incomplete connections trigger a retry sequence.

- Unacknowledged resets are rerouted to alternate nodes, preventing the system from freezing at the end of a round.

Together, these features make our system not just fault tolerant, but fault resilient. The system doesn't just "survive" failure, it adapts to it, maintains state, and resumes operation as soon as possible. The token ring won’t blindly proceed without trying every option; the polling loop won’t hang indefinitely waiting for a ghost. These systems were built to withstand real world conditions like power loss, sensor corruption, and inconsistent Wi-Fi, without ever dropping the ball entirely.

Fairness Comparison

Robustness isn’t the only important system level property fairness plays a huge role in distributed systems, particularly in how computational and communication burdens are shared. Our two architectures make fundamentally different choices here.

The polling approach is inherently centralized. The primary Pi initiates all activity, performs all database insertions, and is responsible for graph completeness. This makes it simple to reason about, but creates an obvious bottleneck. If the primary Pi goes down, the system stops completely. Even when all Pis are functional, the load is asymmetrically distributed, the secondaries simply wait to be asked and respond, while the primary does all the work. In systems with more Pis, this design becomes increasingly unsustainable, as the primary would scale linearly in its responsibilities.

The token ring system, on the other hand, is built around equal participation. Each Pi both receives and forwards the token, appends its data, and has the potential to write to the database if it becomes the final node. There is no single point of control, and any Pi can initiate or complete a round in the event of a failure. This design not only promotes robustness through decentralizatio it also boosts load balancing. Each node is responsible for a similar share of messaging, error handling, and computation. From a system design perspective, this is far closer to what you'd see in industrial deployments or sensor swarms, where each node operates semiautonomously and contributes to global coordination without a master controller.

Implementing fairness comes with challenges. Coordinating resets, ensuring token integrity, and maintaining ACK reliability are all more difficult than writing a centralized polling loop. But the benefits are there: increased resilience, better scalability, and reduced dependency on any single component. Our token ring system may be harder to implement, but it is fairer, more fault tolerant, and a better model for real world sensor networks.

Topology Alternatives and Design Justification

During the design process, we considered a number of other possible topologies that could have improved performance or robustness under different constraints.

A mesh topology would theoretically offer the highest level of redundancy and fault tolerance. In a mesh, every Pi could talk to every other Pi, which would allow dynamic rerouting of data, multiple token paths, and simultaneous data collection. However, this comes at a huge complexity cost. Synchronization becomes extremely difficult, message collisions have a high chance of happening without an additional control layer, and managing which Pi is responsible for writing to the DB requires coordination protocols (like leader election) that are way harder than what this lab needs. Mesh is great on paper, but extremely difficult to implement correctly at this scale without middleware.

We also considered a hybrid star ring model, where the token flows in a ring but database writes and resets are always routed back to a central node. This would combine some decentralization with a known reset point, potentially simplifying coordination. However, this too introduces problems: if that central reset node goes down, the whole ring is broken. It reintroduces the same single point of failure problem we were trying to avoid.

In the end, we chose to enhance our basic token ring model with timeout and ACK based fallback logic. This allowed us to:

- Maintain decentralization

- Handle mid round failures

- Promote fairness
- And reduce the complexity of message routing

This topology struck the best balance between simplicity, fault tolerance, and equitable participation. It matched the lab's emphasis on robustness while remaining implementable within our time constraints and toolset.

Web Application Logic

Graphs display sensor values per Pi, a bolded average line, and a dashed forecast overlay. Forecast data is retrieved at runtime using Open-Meteo’s REST API and parsed from JSON. Each forecast value is matched to the nearest timestamp in our data using pandas style logic.

We binned data using datetime rounding to ensure consistent time intervals across all three tables. The graphing backend excluded nulls from averages and used Matplotlib’s fill between method to show data spread. Since soil moisture forecasts don’t exist, we only plotted actual sensor values for that metric.

The main limitation was the lack of real time updates. We sidestepped this by rendering fresh images on each page load. While not fully dynamic, it maintained performance and stability. We briefly experimented with JavaScript charts (Chart.js), but this required client side SQL wrappers or extra REST endpoints, which we decided were overkill for the lab's goals. Also, running Matplotlib in headless mode fixed all other problems (crashing on reload).

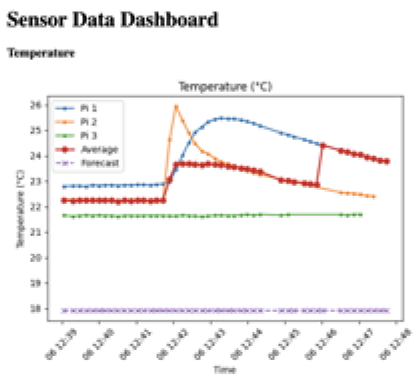


Figure 1: Temperature Plot Example

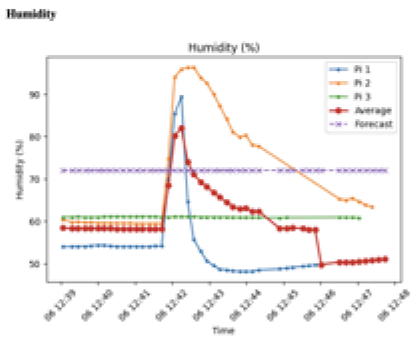


Figure 2: Humidity Plot Example

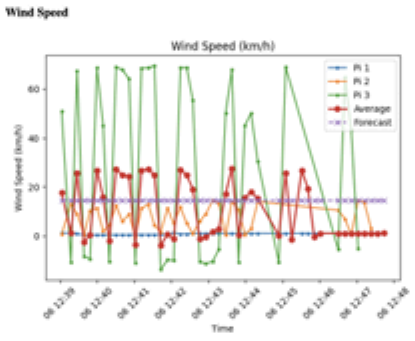


Figure 3: Wind Speed Plot Example

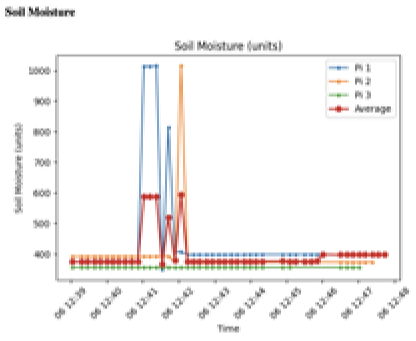


Figure 4: Soil Moisture Plot Example

Conclusion: What I Learned

This lab drove home the difference between a functioning system and a robust one. Through retry logic, distributed design, and SQL integration, we built a network that didn't just "work", it adapted when things failed. Polling gave us simplicity. Token ring gave us completeness. The MySQL+Flask stack gave us a tangible frontend to confirm the system's behavior.

I’ve worked with SQL to make web apps in my previous internships, so that part was nothing new. The previous labs already taught me about polling and token ring/ connecting actual sensors to my system. The real new part was how to effectively make the token ring setup robust (which I took some inspiration from my CSE 156 assignments for).

I walked away with a deeper understanding of failure recovery, decentralized communication, and the importance of data fidelity in real world IoT deployments. Most importantly, I saw how tradeoffs, between latency and integrity, complexity and resilience, shape every system architecture.

This wasn’t just about connecting Pis. It was about making a system that *keeps going*. Even when the network doesn’t.