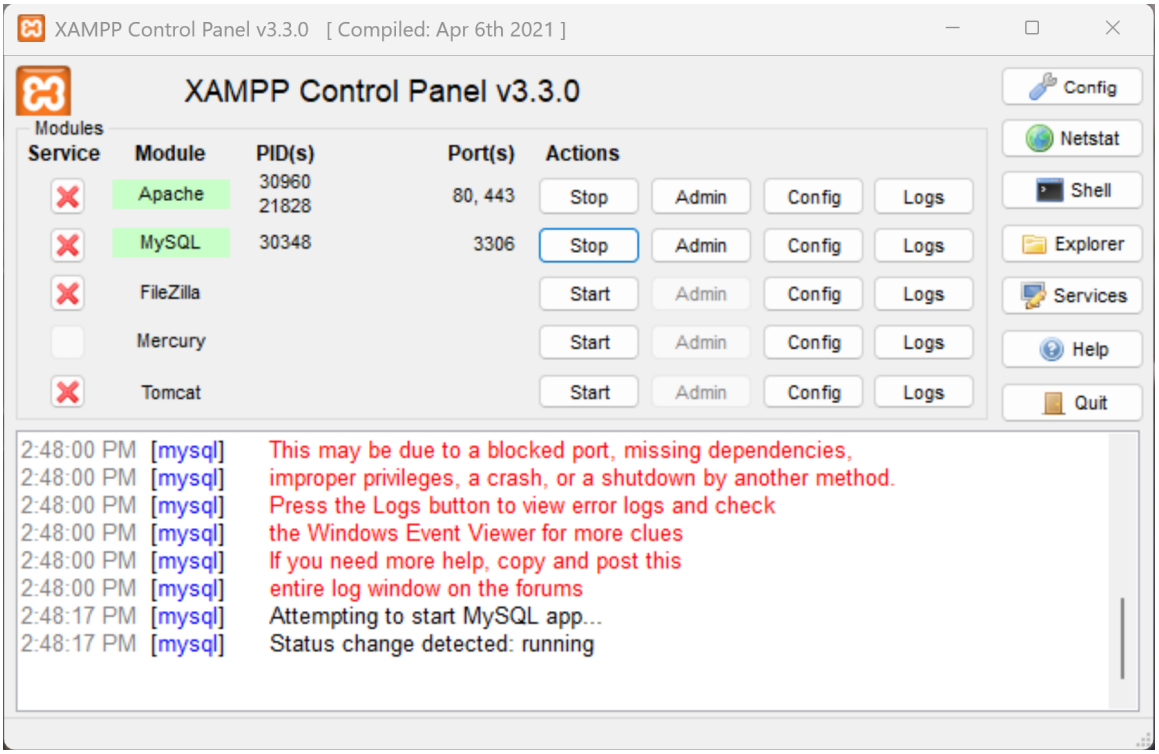
CSE 157
Lab4: Connecting to the Cloud

Isaac Garibay

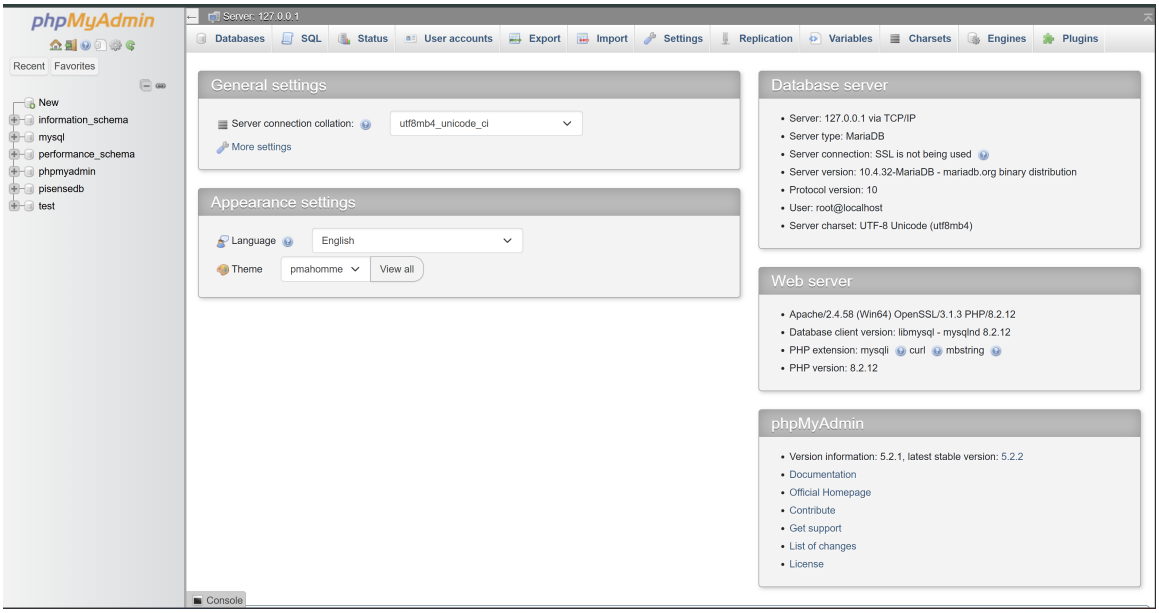
June 11, 2025

Part 1 Setting Up a Web Server & Getting Started

To begin the lab, I referred to the XAMPP tutorial: installation and first steps documentation. All that was required was downloading the appropriate installer for my machine (in my case Windows 11) and running it to install the XAMPP bundle. Once I completed this step, I went ahead and tested out how it runs by running the control panel as shown below.



Don't mind the scary x's, they were a harmless side effect of running XAMPP as admin

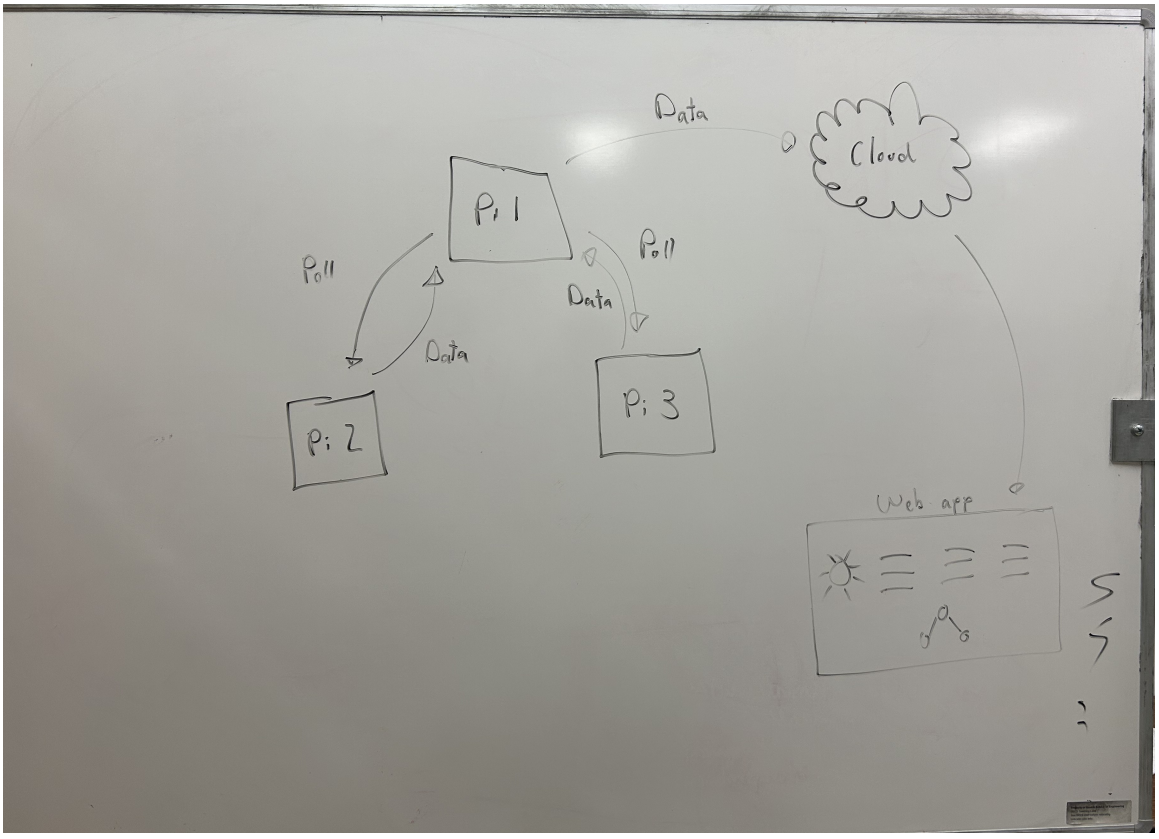


Then, using the Admin button, I navigated the *phpadmin* dashboard. One should see as shown above. There was no need for any custom configuration other than ensuring that XAMPP ran as an adminis-

trator process (**For Windows users**).

Once I understood how this software worked, the team and I consulted the whiteboard to think of the system architecture with the introduction of the cloud for both topologies we had previously implemented in Lab 3.

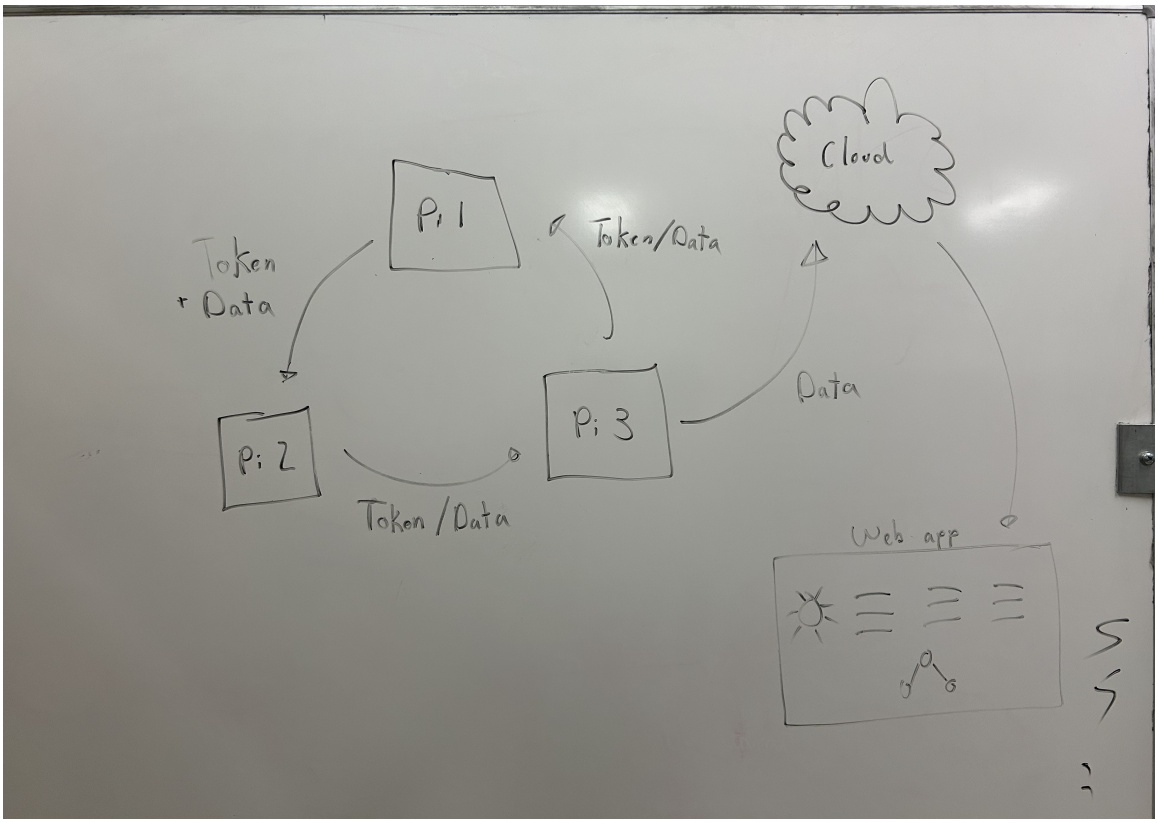
The Centralized Primary/Secondary polling topology



Given that the ultimate goal was to have a web app provide a presentation layer for data based on what we have the pi's push to the database, we concluded that it was best to have the primary make an insert query of the data into the tables for each pi's tables and have the web-app pull from the database. (*The logic of how this is done is discussed in part 3*). The alternative was adding more complexity and responsibility to the pi's scripts which we found to be the worst of the two directions our design would go.

The Token Ring topology

For our token ring topology, we did have an iteration of the token-ring.py script that had all pi's push to the database when the token was held. However, we found such a design defeats the purpose of what the token ring provides, which is an efficient and coordinated means to perform in-network data aggregation. Therefore, we decided to have the last node in the ring be the designated edge node that interacts with the cloud. The challenges in ensuring robustness to this approach proved difficult. (*The process is discussed at length in part 2*).



One thing to note is that the retrieval of the data between the cloud and the web application is the same across both topologies. Moreover, we found it much more portable to have all pi's aswell as the web app itself utilize the same SQL user account. (*Yes, not the most secure choice. This decision was for the sake of simplicity and timely implementation of less important features.*)

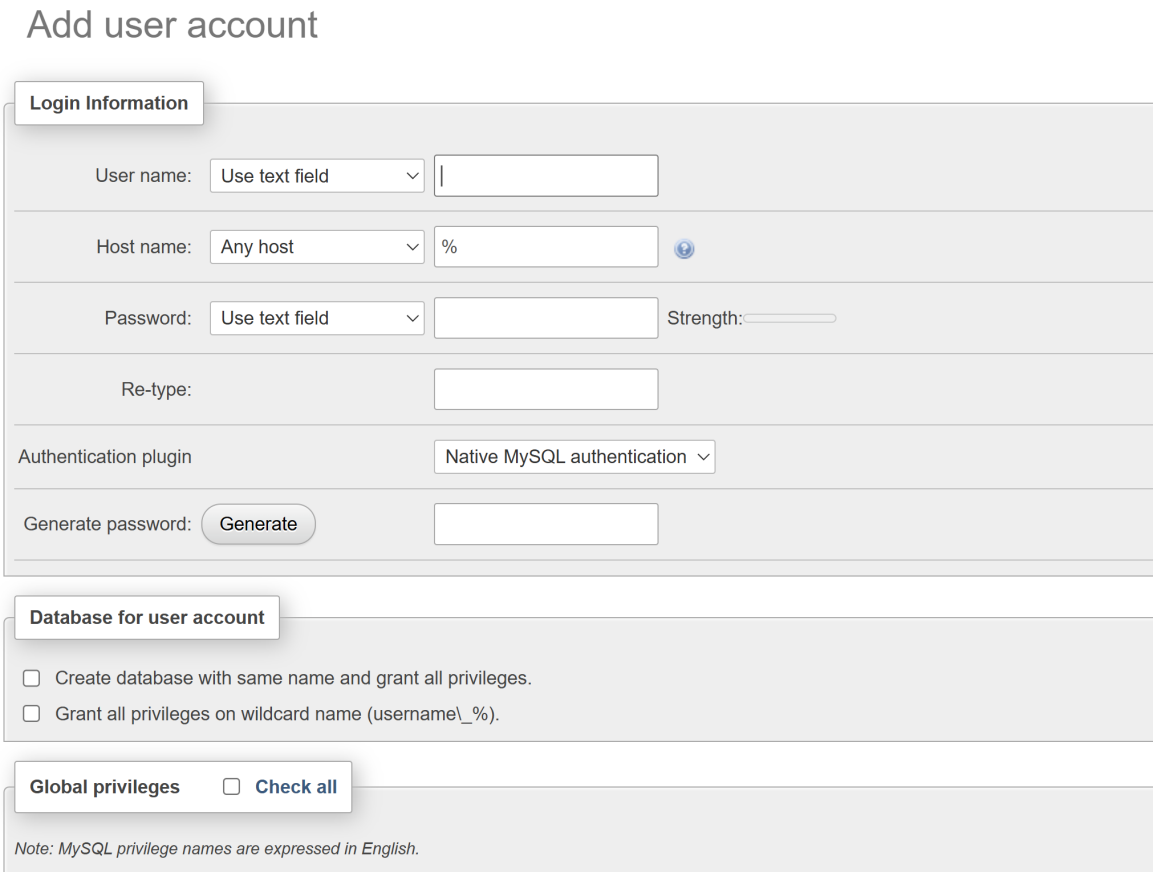
Part 2 Creating and Connecting to a Database

After the initial set up of the database software and testing interactions with the web server, I proceeded to move on to making the necessary additions to the existing networking scripts to allow them to push collected sensor data to the XAMPP hosted database. Referencing the Getting started with Python Connector documentation, I implemented the following logic as shown below.

The logic here first describes the credentials to authorize insert queries made to the database. Note that the insert query describes the schema necessary to insert into our table structures.

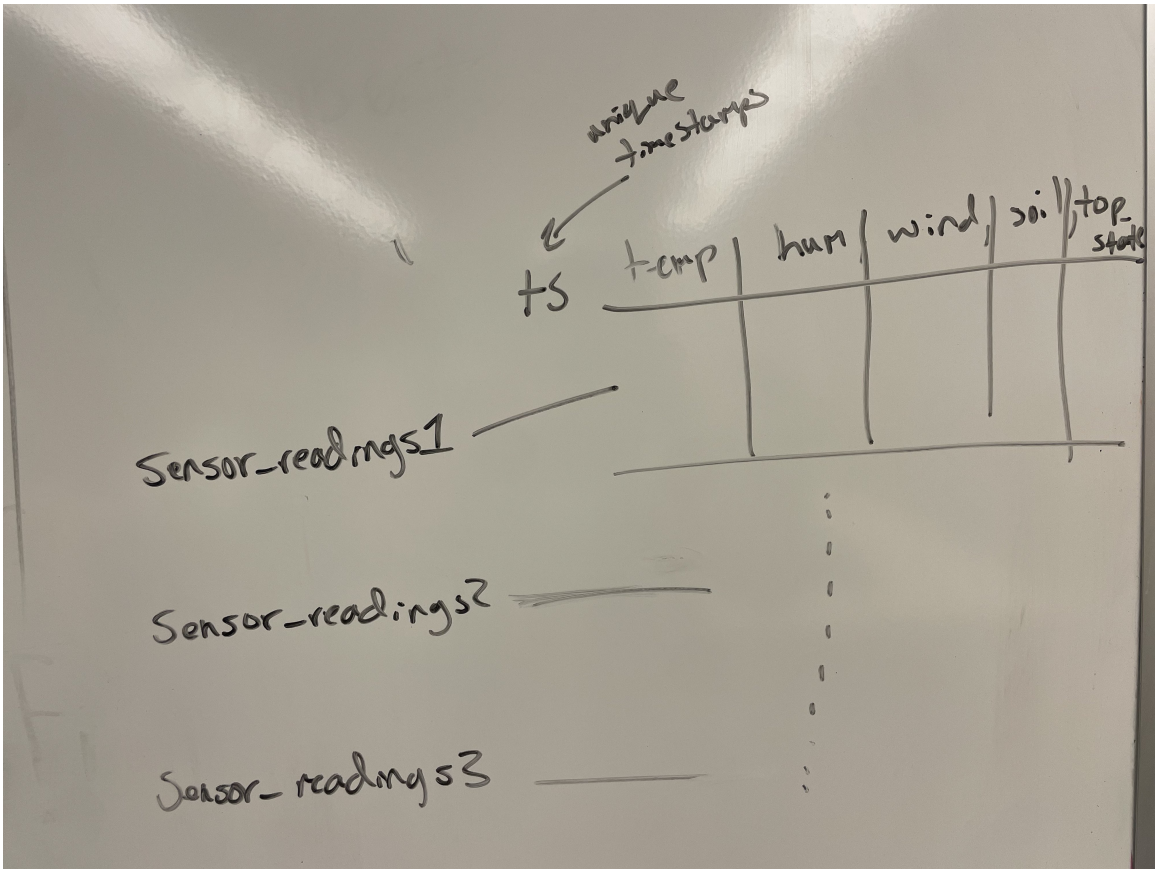
```
import mysql.connector

DB = dict(
    host     = "192.168.0.132",        # laptop IP
    port     = 3306,
    user     = "primaryPi",
    password = "theeIoTofGoats!",
    database = "piSenseDB"
)
conn = mysql.connector.connect(**DB)
cur  = conn.cursor(prepared=True)
INSERT = ("INSERT INTO {table} "
          "(temperature, humidity, wind_speed, soil_moisture, topology_state) "
          "VALUES (%s, %s, %s, %s, %s)
```

To get this working, you have to create a user account and specify privileges allowing the scripts to make an insert query. This can be done with the phpadmin dashboard UI as shown below.



The UI was also the preferred method to create our SQL tables rather than explicitly using SQL queries to create them. Each sensor value was represented as *float* data types aside from the timestamps of each entry which was simply specified as a *timestamp* type. Later on, we made the addition of a ***topology_state*** column, with *json* type values. This was so that a json dump could be performed to directly store the raw value of the list containing topology information.

On the next page is a simplified diagram for what our tables looked like.



Integrating the connector required installing the imported library shown in the code block by running pip install mysql−connector−python to retrieve the python specific SQL library to run queries in the script. (*You must activate the virtual environment to successfully run this command*).

Although using the XAMPP software was quite straighforward, one of my teammates faced some difficulties with running XAMPP on MacOS. It seems that the support for the current UI functionality is highly limited and struggles to start and manage the processes that run the Apache server and SQL database. In light of these difficulties, we relied on my Windows machine's ability to connect to the CSE157_5G router infrastructure. We leveraged the fact that the router assigned static IP's as opposed to eduroam's volatile assignment from it's VPN.

In my experience, I faced a mysterious issue of XAMPP being unable to run the SQL server at unpredictable times whenever starting the application back up. My only solution was simply uninstalling and installing XAMPP a few times and redefined the tables. Although a tedious task, it did not incur much technical debt since our scripts described the schemas necessary to make insert queries into the tables.

Discussing the Token Ring and Primary/Secondary Topologies

As far as getting the pi's to push to the database, that was a trivial matter compared to enforcing the in-network data aggregation with robustness in the token ring topology. As for the primary/secondary topology, there was not much to add to it for robustness. Of course, if one secondary goes down, the primary and the other live secondary pi operate as usual. If all secondary nodes are down. The primary

pi simply reports its own sensor readings to the database. When a node has nothing to send, it simply does not have it's data appended to the list that collects them at the primary node. This behavior is similar to what happens when a node has nothing to send within the token ring.

Regarding the case where all nodes continuously have something to send, the primary script simply takes the latest reading that it polled for from the secondary pis. The token-ring does something similar where upon receiving the token, a pi appends it's latest set of sensor readings to the token and forwards it to the next node in the ring. If the node is the last in the ring topology, then it acts as the edge and pushes all the readings to the database.

Considering the fairness in sharing the medium, I find that the centralized polling approach the primary/secondary topology gives similar opportunity to use the medium as the token ring topology since both configurations give each node an equal opportunity to poll their own sensors evenly. The greatest distinction is that the primary/secondary topology has a single point of failure–the primary pi. If the primary goes down, the secondaries sit idle. This could have been mitigated by having the secondaries push to the database by themselves with a timeout on it's **recv()** call that expects a poll from the primary. Although, this seems to circumvent the role of the primary pi but presents as the simplest and best method to ensure robustness for that particular topology.

One topology that I had considered in passing, though not so interesting, was a flat topology where the pi's all push to the cloud to their respective tables independently. This would get around the issues of overhead and latency in the two approaches we have built upon. The overhead at the primary pi is much greater and must handle all requests and responses from ***n*** secondaries. The token ring faces the challenge of latency due to it's functional inclination to perform in-network data aggregation via the token passed around the ring. Moreover, data freshness is of concern since the first node's data ages ***n-1*** hops around the ring before it's data is updated in the database.

In our case, the most interesting topology implementing robustness for was the token ring topology. Therefore, we discuss some of it's key features at length in the following sections.

Avoiding duplicate tokens

In our original ***token-ring.py*** script, we essentially had every pi perceive the same timeout. This meant that duplicate tokens were bound to occur which poses a problem for accurate data retrieval and maintenance. The approach that was taken to solve the issue of duplicate tokens, was staggering the timeout exceptions based upon the position of a given node in the ring.

```
def recv_token():
    """
    Wait for a token to arrive (or timeout). Inspect token["source"] to detect re-joins.
    """
    server.settimeout(TIMEOUT * (my_index + 1))
    try:
        conn_sock, addr = server.accept()
    except socket.timeout:
        return None

    with conn_sock:
        raw = conn_sock.recv(4096)

    try:
        token = json.loads(raw.decode())
    except Exception as e:
        print(f"[!] invalid token from {addr}: {e!r}")
        return []

    #Look at token["source"]
    source_addr = token.get("source")
    if source_addr and source_addr not in ring:
        print(f"[{role}] Detected rejoining node {source_addr} from token")
```

```
        update_topology_and_indices(source_addr)

    return token
```

This allowed for each pi to perceive relative timeouts before they attempt to re-initialize communication with a token of their own. In retrospect, we were attempting to solve a concurrent problem without using tried and true IPC practices. To be fair, we were stubborn about avoiding the use of the concurrent features provided by the python sockets api to avoid the technical debt of refactoring the features we had previously built in Lab 3. Ideally, we would have used a different language that supports true concurrency for what we were trying to build here.

Minimizing Memory Usage

As for the memory usage problem, it was observed that the tokens grew indefinitely. In the long run, this would have an IoT crash in a fairly short amount of time. So, I had made the following edits to the main logic.

```
try:
    while True:
        if role == "start" and round_num == 1:
            reading = attach_topology(sensor_polling.get_local_measurements(my_index))
            token = {
                "source": my_addr,    #include source
                "data":   [reading],
                "round":  1,
                "closed": False
            }
            print(f"[start] initial token = {token}")
            forward_token(token)
            token = recv_token()
            continue

        tok = recv_token()

        if tok is None:
            print(f"[{role}] no token     re-initiating token ring")
            reading = attach_topology(sensor_polling.get_local_measurements(my_index))
            token = {
                "source": my_addr,   #include source
                "data":   [reading],
                "round":  round_num,
                "closed": False
            }
            forward_token(token)
            time.sleep(RETRY_PAUSE)
            continue

        token = tok or []
        print(f"[{role}] got token: {token}")

        reading = attach_topology(sensor_polling.get_local_measurements(my_index))
        token["data"].append(reading)

        if len(token["data"]) == N:
            # End of lap for current live ring
            for rec in token["data"]:
                db_insert(f"sensor_readings{rec['node']+1}", rec)
            plot_token(token["data"], token["round"])

            next_round = token["round"] + 1
            empty = {
                "source": my_addr,    #include source
                "data":   [],
                "round":  next_round,
                "closed": False
            }
```

```
            print(f"[{role}] completed lap (size={N}). Starting empty token for round={next_round}")
            time.sleep(PLOT_PAUSE)
            forward_token(empty)

            round_num = next_round
            time.sleep(PLOT_PAUSE)
            continue

        # Otherwise, not end of lap forward normally (with updated ring)
        token["source"] = my_addr  #reset source before forwarding
        if not forward_token(token):
            # Last-alive fallback
            for rec in token["data"]:
                db_insert(f"sensor_readings{rec['node']+1}", rec)
            plot_token(token["data"], token["round"])

            next_round = token["round"] + 1
            empty = {
                "source": my_addr,
                "data":   [],
                "round":  next_round,
                "closed": False
            }
            print(f"[{role}] last-alive fallback. Starting empty token for round={next_round}")
            time.sleep(PLOT_PAUSE)
            forward_token(empty)

            round_num = next_round
            time.sleep(PLOT_PAUSE)
            continue

        # If forwarded successfully, bump round counter
        round_num += 1
        time.sleep(PLOT_PAUSE)
```

Note that there is a condition checking for the position of the node to see whether or not the lap around the ring is finished by the time the token has reached it. If so, then it will iterate over each sensor's readings of every pi in the received token then indicate a new round with a fresh list for a new round of data to aggregate into.

Enhancing Fault-Tolerance

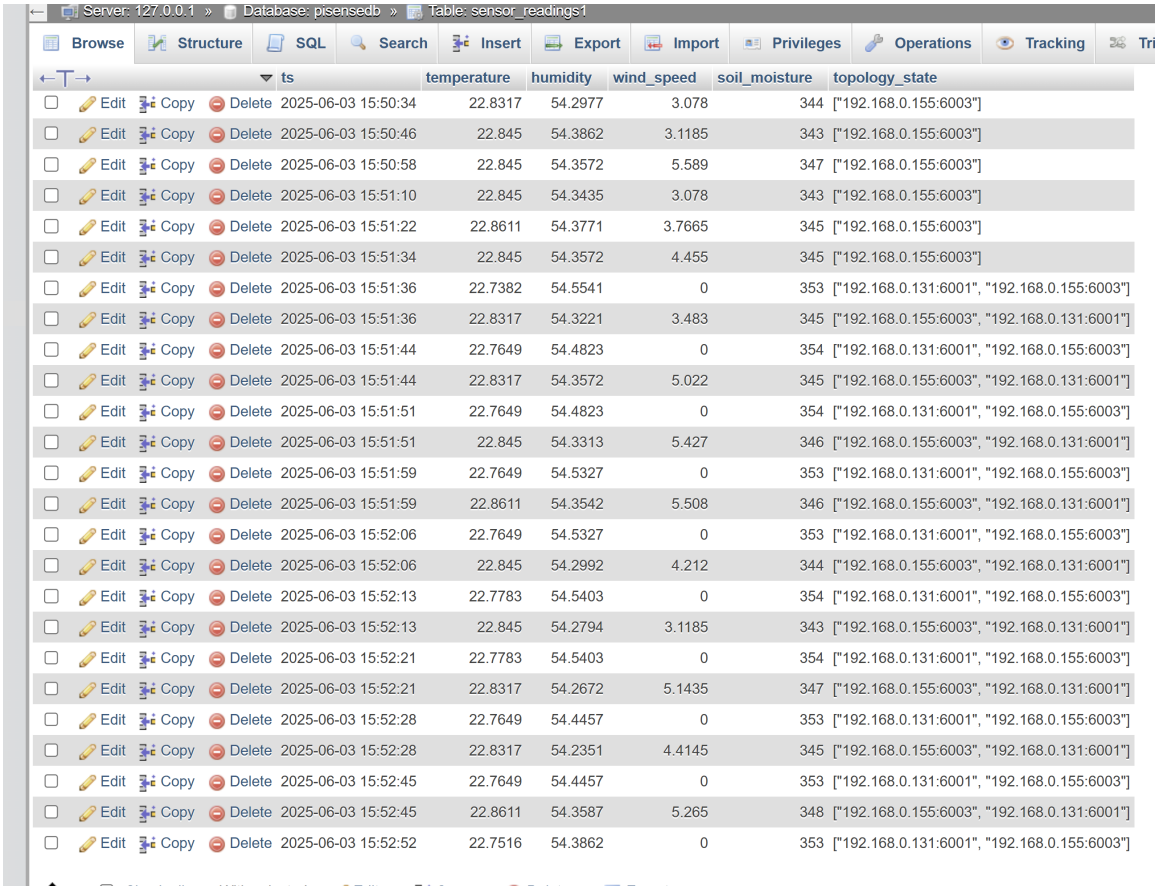
One may notice that there are some helpers involving the topology state. To enforce our design of having the last node in the ring act as the edge, we had to provide a means for the nodes and ultimately the web app to know about the state of the network. We modified the structure of our token to have additional fields such as the source of the token, the data list that collects each pi's sensor data, and the round number.

Since our design intended to have the end node act as the edge, we had it ensure that the topology state is reported by the insert query of the database. To make this possible, we kept the feature of specifying the topology to every node in their command line arguments at boot but implemented a feature that has the ring dynamically update upon a change in the topology such as a drop or a re-join. Below are some key functions that helped in that endeavor.

```
# Compute my_index and predecessor index initially
my_index   = ring.index(my_addr)
pred_index = (my_index - 1) % N
pred_host, pred_port = ring[pred_index].split(":")

server = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
host, port = my_addr.split(":")
server.bind((host, int(port)))
server.listen(1)
```

```
print(f"[{role}] bound to {my_addr}, predecessor={ring[pred_index]}, ring={ring}")

def db_insert(table, reading):
    if not reading:
        return
    vals = (reading.get("temperature"),
            reading.get("humidity"),
            reading.get("wind_speed"),
            reading.get("soil_moisture"),
            reading.get("topology_state")) or json.dumps(ring)
    cur.execute(INSERT.format(table=table), vals)
    conn.commit()

def attach_topology(reading: dict) -> dict:
    """
    Adds a 'topology_state' field     JSON string with the CURRENT ring order.
    called right after sensor_polling.get_local_measurements().
    """
    reading["topology_state"] = json.dumps(ring)
    return reading


def update_topology_and_indices(node_addr):
    """
    If node_addr is already in ring, remove it (because unreachable).
    Otherwise, append it (because rejoining).
    Then recalc N, my_index, pred_index, etc.
    """
    global ring, N, my_index, pred_index, pred_host, pred_port

    if node_addr in ring:
        ring.remove(node_addr)
        print(f"[{role}] Removing unreachable node {node_addr} from ring.")
    else:
        ring.append(node_addr)
        print(f"[{role}] Adding node {node_addr} back into ring.")

    N = len(ring)
    if N == 0:
        return

    try:
        my_index = ring.index(my_addr)
    except ValueError:
        # If somehow this node was removed, re-add it
        ring.append(my_addr)
        my_index = ring.index(my_addr)

    pred_index = (my_index - 1) % N
    pred_host, pred_port = ring[pred_index].split(":")
    print(f"[{role}] Updated ring={ring}, N={N}, my_index={my_index}, predecessor={ring[
    pred_index]}")
```

In practice, this feature worked with some caveats. One emergent property was that re-joining nodes would sometimes perceive themselves as an isolated member of the ring and create an independent ring between itself and the next viable successor. However, this did not appear to affect the correctness of data pushed to the database. After a few rounds, the ring would eventually regulate and form the full expected ring topology. When pushing to the database, with either the centralized primary polling or the token ring, we had dedicated a column for the topology state represented as a list of ip's or role names depending on the configuration as shown below.



	ts	temperature	humidity	wind_speed	soil_moisture	topology_state
Edit Copy Delete	2025-06-03 15:50:34	22.8317	54.2977	3.078	344	["192.168.0.155:6003"]
Edit Copy Delete	2025-06-03 15:50:46	22.845	54.3862	3.1185	343	["192.168.0.155:6003"]
Edit Copy Delete	2025-06-03 15:50:58	22.845	54.3572	5.589	347	["192.168.0.155:6003"]
Edit Copy Delete	2025-06-03 15:51:10	22.845	54.3435	3.078	343	["192.168.0.155:6003"]
Edit Copy Delete	2025-06-03 15:51:22	22.8611	54.3771	3.7665	345	["192.168.0.155:6003"]
Edit Copy Delete	2025-06-03 15:51:34	22.845	54.3572	4.455	345	["192.168.0.155:6003"]
Edit Copy Delete	2025-06-03 15:51:36	22.7382	54.5541	0	353	["192.168.0.131:6001", "192.168.0.155:6003"]
Edit Copy Delete	2025-06-03 15:51:36	22.8317	54.3221	3.483	345	["192.168.0.155:6003", "192.168.0.131:6001"]
Edit Copy Delete	2025-06-03 15:51:44	22.7649	54.4823	0	354	["192.168.0.131:6001", "192.168.0.155:6003"]
Edit Copy Delete	2025-06-03 15:51:44	22.8317	54.3572	5.022	345	["192.168.0.155:6003", "192.168.0.131:6001"]
Edit Copy Delete	2025-06-03 15:51:51	22.7649	54.4823	0	354	["192.168.0.131:6001", "192.168.0.155:6003"]
Edit Copy Delete	2025-06-03 15:51:51	22.845	54.3313	5.427	346	["192.168.0.155:6003", "192.168.0.131:6001"]
Edit Copy Delete	2025-06-03 15:51:59	22.7649	54.5327	0	353	["192.168.0.131:6001", "192.168.0.155:6003"]
Edit Copy Delete	2025-06-03 15:51:59	22.8611	54.3542	5.508	346	["192.168.0.155:6003", "192.168.0.131:6001"]
Edit Copy Delete	2025-06-03 15:52:06	22.7649	54.5327	0	353	["192.168.0.131:6001", "192.168.0.155:6003"]
Edit Copy Delete	2025-06-03 15:52:06	22.845	54.2992	4.212	344	["192.168.0.155:6003", "192.168.0.131:6001"]
Edit Copy Delete	2025-06-03 15:52:13	22.7783	54.5403	0	354	["192.168.0.131:6001", "192.168.0.155:6003"]
Edit Copy Delete	2025-06-03 15:52:13	22.845	54.2794	3.1185	343	["192.168.0.155:6003", "192.168.0.131:6001"]
Edit Copy Delete	2025-06-03 15:52:21	22.7783	54.5403	0	354	["192.168.0.131:6001", "192.168.0.155:6003"]
Edit Copy Delete	2025-06-03 15:52:21	22.8317	54.2672	5.1435	347	["192.168.0.155:6003", "192.168.0.131:6001"]
Edit Copy Delete	2025-06-03 15:52:28	22.7649	54.4457	0	353	["192.168.0.131:6001", "192.168.0.155:6003"]
Edit Copy Delete	2025-06-03 15:52:28	22.8317	54.2351	4.4145	345	["192.168.0.155:6003", "192.168.0.131:6001"]
Edit Copy Delete	2025-06-03 15:52:45	22.7649	54.4457	0	353	["192.168.0.131:6001", "192.168.0.155:6003"]
Edit Copy Delete	2025-06-03 15:52:45	22.8611	54.3587	5.265	348	["192.168.0.155:6003", "192.168.0.131:6001"]
Edit Copy Delete	2025-06-03 15:52:52	22.7516	54.3862	0	353	["192.168.0.131:6001", "192.168.0.155:6003"]

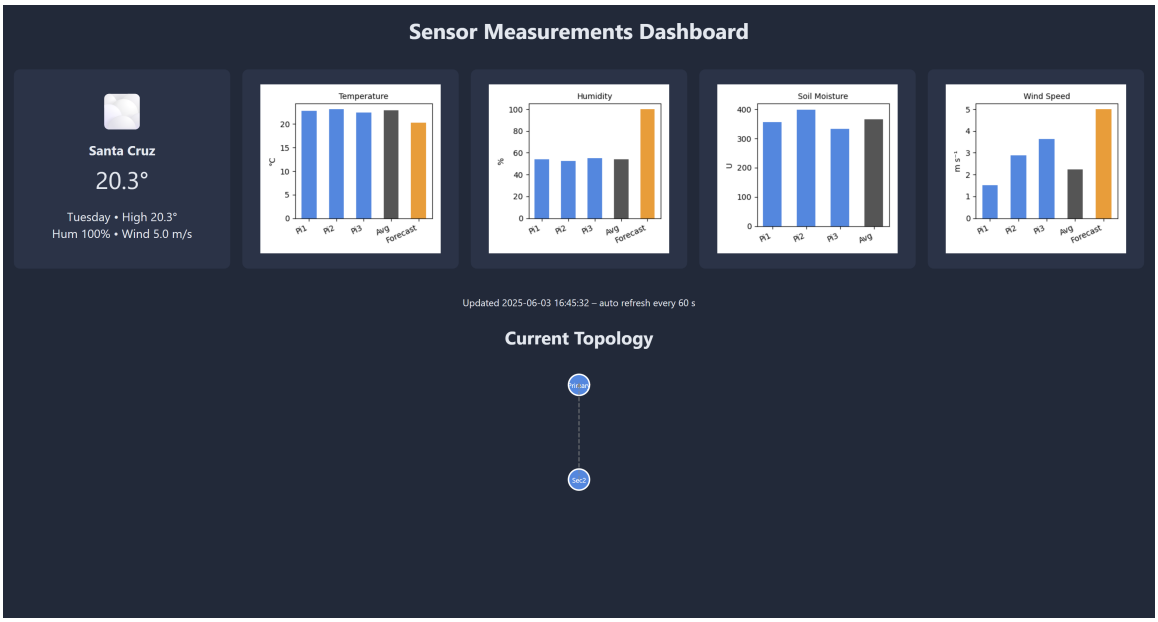
Testing between two pi's in the token ring topology

Part 3 Displaying Sensor Measurements on a Web Application

Given my previous experience with the Flask framework, it wasn't too difficult to implement the dashboard shown below. The rest of the features on the dashboard leveraged known libraries such as the **Matplotlib** library which was used in the topology scripts in Lab 3. The averages are computed traditionally by taking the sum of each pi's readings and dividing by the number of total set of readings for each sensor of each pi. The truly novel feature that was introduced was the use of the **svgwrite** library to have the web app draw the state of the topology based upon the values stored in the *topology_state* column in our tables.

Our development process to enforce robustness in the token ring while enforcing the last node to act as the edge node had us arrive at the need of the topology_state column. In fact, I believe it was the crux of our intended use case for our deployment dashboard application. (*This notion is discussed further in the use cases section*).

Below shows how our dashboard presents with varying network conditions. All information is nicely organized in one place posing as a tool that can be used in field deployments.



Our web app dashboard when a node drops out of the network

Our web app dashboard with a fully stable network

The logic of how we have the web app pull from our database is featured in the set of helper functions shown below.

```
from flask import Flask, render_template_string, Response, request
import pandas as pd, matplotlib.pyplot as plt, io, base64, requests, mysql.connector
from datetime import datetime, timedelta, date
import svgwrite, math, json
DB = dict(host="192.168.0.132", port=3306,
          user="primaryPi", password="theeIoTofGoats!", database="piSenseDB")

TABLES   = ["sensor_readings1", "sensor_readings2", "sensor_readings3"]
METRICS  = ["temperature", "humidity", "soil_moisture", "wind_speed"]
```

```
LABELS   = {"temperature": " C ", "humidity": "%", "wind_speed": "m   s   ", "
    soil_moisture": "U"}
LAT, LON = 37.0, -122.06
TIME_HRS = 24

app = Flask(__name__)
ring_state = ["Pi1", "Pi2", "Pi3"]


def db_to_frame():
    cnx = mysql.connector.connect(**DB)
    stop, start = datetime.utcnow(), datetime.utcnow() - timedelta(hours=TIME_HRS)
    frames = []
    for idx, tbl in enumerate(TABLES, 1):
        q = (f"SELECT ts, temperature, humidity, wind_speed, soil_moisture "
             f"FROM {tbl} WHERE ts BETWEEN %s AND %s")
        df = pd.read_sql(q, cnx, params=(start, stop), parse_dates=["ts"])
        if not df.empty:
            df["node"] = f"Pi{idx}"
            frames.append(df)
    cnx.close()
    return pd.concat(frames, ignore_index=True) if frames else pd.DataFrame()


def latest_topology() -> list[str] | None:
    cnx = mysql.connector.connect(**DB)
    cur = cnx.cursor()
    newest_ts, newest_json = None, None
    for tbl in TABLES:
        cur.execute(
            f"SELECT ts, topology_state "
            f"FROM {tbl} WHERE topology_state IS NOT NULL "
            f"ORDER BY ts DESC LIMIT 1"
        )
        row = cur.fetchone()
        if row and (newest_ts is None or row[0] > newest_ts):
            newest_ts, newest_json = row
    cnx.close()
    if newest_json:
        try:
            return json.loads(newest_json)
        except Exception:
            pass
    return None


def normalize_labels(nodes: list[str]) -> list[str]:
    """Convert raw IP:port     Pi#, keep nice human labels unchanged."""
    out = []
    for idx, n in enumerate(nodes, 1):
        out.append(f"Pi{idx}" if ":" in n else n)
    return out
```

The same user account used to provide authorization for querying the database in both the primary/secondary and token ring topologies is used here. Rather than making an insert, the web app makes read query with the **db_to_frame** helper function to retrieve the data from the tables using it to produce the bar graphs shown in the previous figure. To ensure that the latest topology is used to draw the network graphic, the **latest_topology()** to retrieve the latest value in the topology state column from the database tables. The **normalize_labels(nodes)** function simply helps to convert the labels that the topology graphic uses.

To retrieve forecasting data, we used the **openmeteopy** API to display weather forecast data. Below are the functions responsible for producing the bar graphs, the svg of the topology state respectively, using the data retrieved from the database.

```
def forecast_today():
    global _fc, _ts
    if (datetime.utcnow() - _ts).seconds < 3600 and _fc:
        return _fc
    try:
        url = (f"https://api.open-meteo.com/v1/forecast?latitude={LAT}&longitude={LON}"
               "&daily=temperature_2m_max,weathercode,windspeed_10m_max,
    relative_humidity_2m_max"
               "&temperature_unit=celsius&windspeed_unit=ms&timezone=auto"
               f"&start_date={date.today()}&end_date={date.today()}")
        d = requests.get(url, timeout=10).json()["daily"]
        _fc = dict(weathercode=d["weathercode"][0], temperature=d["temperature_2m_max"
    ][0],
                   humidity=d["relative_humidity_2m_max"][0], wind_speed=d["
    windspeed_10m_max"][0],
                   soil_moisture=None)
    except Exception as e:
        print(f"[wx] forecast fetch failed: {e}")
        _fc = {}
    _ts = datetime.utcnow()
    return _fc


WX_EMOJI = {0: "      ", 1: "   ", 2: "   ", 3: "      ", 45: "         ", 48: "
            ",
            51: "         ", 53: "         ", 55: "         ", 61: "         ", 63: "
             ", 65: "         ",
            71: "      ", 73: "      ", 75: "      ", 95: "      "}
def wx_icon(code): return WX_EMOJI.get(code, "         ")


def make_bar(df: pd.DataFrame, metric: str, fcst_val):
    if df.empty or df[metric].dropna().empty:
        per_pi = pd.Series(dtype=float)
    else:
        per_pi = df.groupby("node")[metric].mean().sort_index()

    bars, labels = per_pi.tolist(), per_pi.index.tolist()
    bars.append(df[metric].mean()); labels.append("Avg")
    if fcst_val is not None:
        bars.append(fcst_val); labels.append("Forecast")

    plt.figure(figsize=(3.2, 3))
    colors = (["#1e88e5"] * len(per_pi)) + ["#555555"] + (["#ff9900"] if fcst_val is not
     None else [])
    plt.bar(range(len(bars)), bars, color=colors, width=0.55)
    plt.xticks(range(len(labels)), labels, rotation=25, ha="right")
    plt.ylabel(LABELS[metric]); plt.title(metric.replace("_", " ").title(), fontsize=10)
    ; plt.tight_layout()
    buf = io.BytesIO(); plt.savefig(buf, format="png"); plt.close()
    return base64.b64encode(buf.getvalue()).decode()


def ring_svg(nodes: list[str]) -> str:
    """Return SVG of the ring (single-node handled)."""
    if not nodes:                                           # nothing? return blank
    SVG
        return "<svg width='1' height='1'/>"

    dwg = svgwrite.Drawing(size=("180px", "180px"))
    cx, cy, r = 90, 90, 70

    if len(nodes) == 1:                                     # lone survivor
        dwg.add(dwg.circle((cx, cy), 14, fill="#1e88e5", stroke="white", stroke_width=2)
    )
        dwg.add(dwg.text(nodes[0], insert=(cx, cy + 4), text_anchor="middle",
                         fill="white", font_size="9px"))
        return dwg.tostring()
```

```
60
61      arrow = dwg.marker(insert=(3, 3), size=(6, 6), orient="auto")
62      arrow.add(dwg.path(d="M0 0 L6 3 L0 6 Z", fill="#888")); dwg.defs.add(arrow)
63      n = len(nodes)
64      for i, name in enumerate(nodes):
65          ang = 2 * math.pi * i / n
66          x, y = cx + r * 0.9 * math.sin(ang), cy - r * 0.9 * math.cos(ang)
67          x2, y2 = cx + r * 0.9 * math.sin(ang + 2 * math.pi / n), cy - r * 0.9 * math.cos
    (ang + 2 * math.pi / n)
68          dwg.add(dwg.circle((x, y), 14, fill="#1e88e5", stroke="white", stroke_width=2))
69          dwg.add(dwg.text(name, insert=(x, y + 4), text_anchor="middle", fill="white",
    font_size="9px"))
70          dwg.add(dwg.line((x, y), (x2, y2), stroke="#888", stroke_dasharray="5,3",
71                           marker_end=arrow.get_funciri()))
72      return dwg.tostring()
```

(*Emoji's were used for extra pizzazz, however LaTeX refuses to display them properly in the code.*)

As the name suggests, the **make_bar()** produces the bar graphs for each pi's sensor readings based upon the data read from the database. To get the topology state svg created, we implemented the **ring_svg()** function to draw the graphical depiction of the topology state.

Extra Credit: The use cases of our web application and robustness features

The vision I saw for our dashboard application was to act as a useful presentation layer of the state of an IoT deployment. A clean UI with all information in one place can be used in practical applications in maintenance, analysis, and field testing.

The dynamic topology state updates in both the primary/secondary and token ring topologies were inspired from the behaviors of MANET protocols and proved interesting to attempt to implement. The graphical depiction of the topology state is something that I figured would be useful since it allows for monitoring changes in the network live without pain painstakingly browse through them. Moreover, I find it is needed to better verify data representation of sensor readings. Some of the limitations we observed from our design was that it relied solely on the database running live and had data stored in the tables to begin with. In addition, it's refresh rate was set to every minute which had it display stale data representation although this is not much of a concern and takes a simple change to the refresh rate in the web app script.

Conclusion

Some of the takeaways that I got from my experience in this lab was that writing sophisticated protocols that can handle lossy conditions is anything but trivial. It was interesting to attempt to write one that accounts for topology changes (e.g. drops, re-joins, or additions). In hindsight, implementing a broadcast via UDP with Python's socket programming API was the right design decision to effectively have all pi's have a consensus on what the topology state is. This would have likely solved the issues discussed in Part 2. Another takeaway that I gained from the lab was that creating an application for a deployment gives a glimpse into the real world applications of what an IoT deployment would require. In particular, a dashboard would prove useful for engineers or researchers testing or using IoT in remote regions to understand the state of the devices. Depending on what insights the data is expected to provide, it would prove beneficial to have the application display with a higher refresh rate to avoid making data driven decisions on stale data. Overall, I found it entertaining to think through and develop fault tolerant algorithms for IoT's. I learned a lot from trouble shooting through unwanted behavior and collaborating on the system architecture aswell. In the future, I will likely find myself working on some projects of my own developing niche algorithms for specific use cases.