
# CSE 157

**Lab 4: Connecting to the Cloud**
**Professor Katia Obraczka**

Adam Hidas (ID: 1832587)
ahidas@ucsc.edu
with Joey Vigil
Jovigil@ucsc.edu

July 2, 2025

# 1. Overview and Setup

This lab is a continuation of the previous lab where three 'weather stations' were connected in two different topologies. Things such as the set up for each individual sensor, starting the pis in an ad-hoc network, and communicating over sockets were covered in previous lab reports and therefore will not be gone over again. To understand these parts more in depth, refer to lab reports one, two, and three.

In this iteration, the pis will not only communicate with each other but also with the broader internet. A SQL database will be hosted on a personal computer along with a python flask server in order to gather readings from the pis and then have that data displayed. In order to facilitate understanding of the web application, a version of the web app is hosted on PythonAnywhere at ahidas2.pythonanywhere.com/. Note that this site has all the same functionality of the app when hosted on the personal device, however the data being displayed was generated rather than gathered by devices.

## 1.a. Wiring

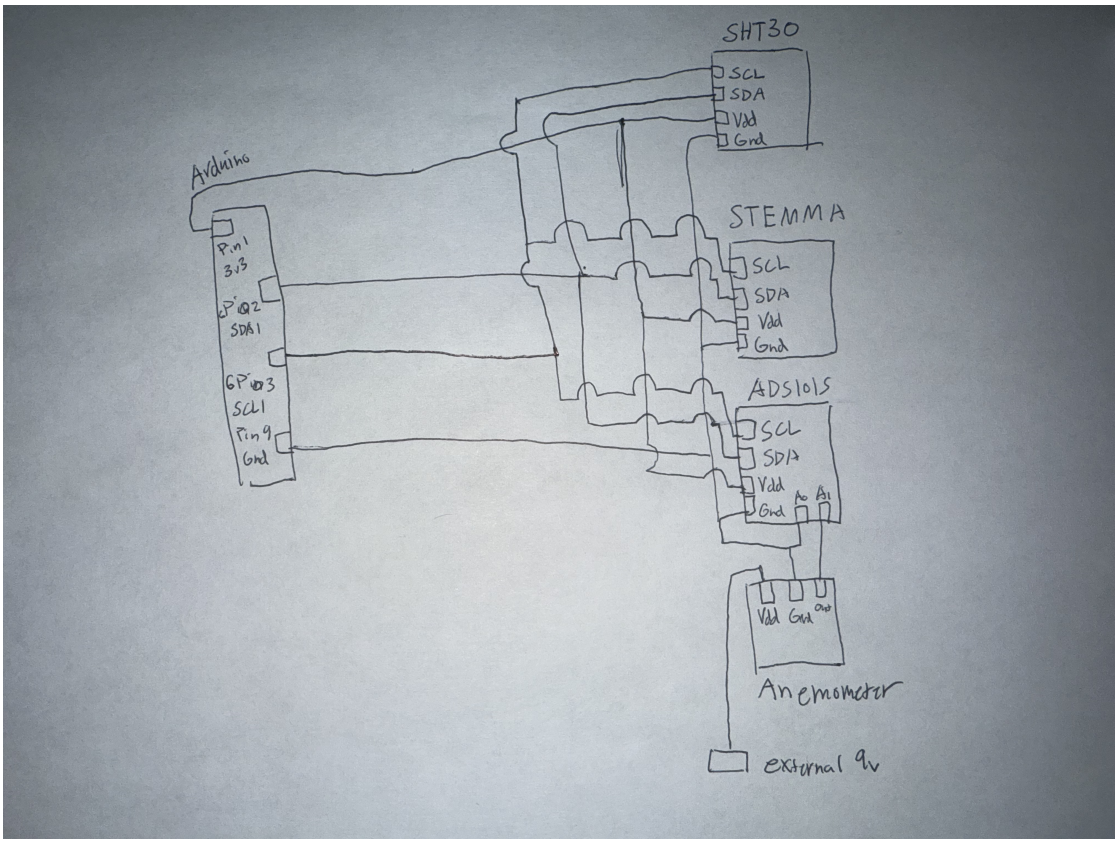


Figure 1: Wiring Diagram of an Individual Pi

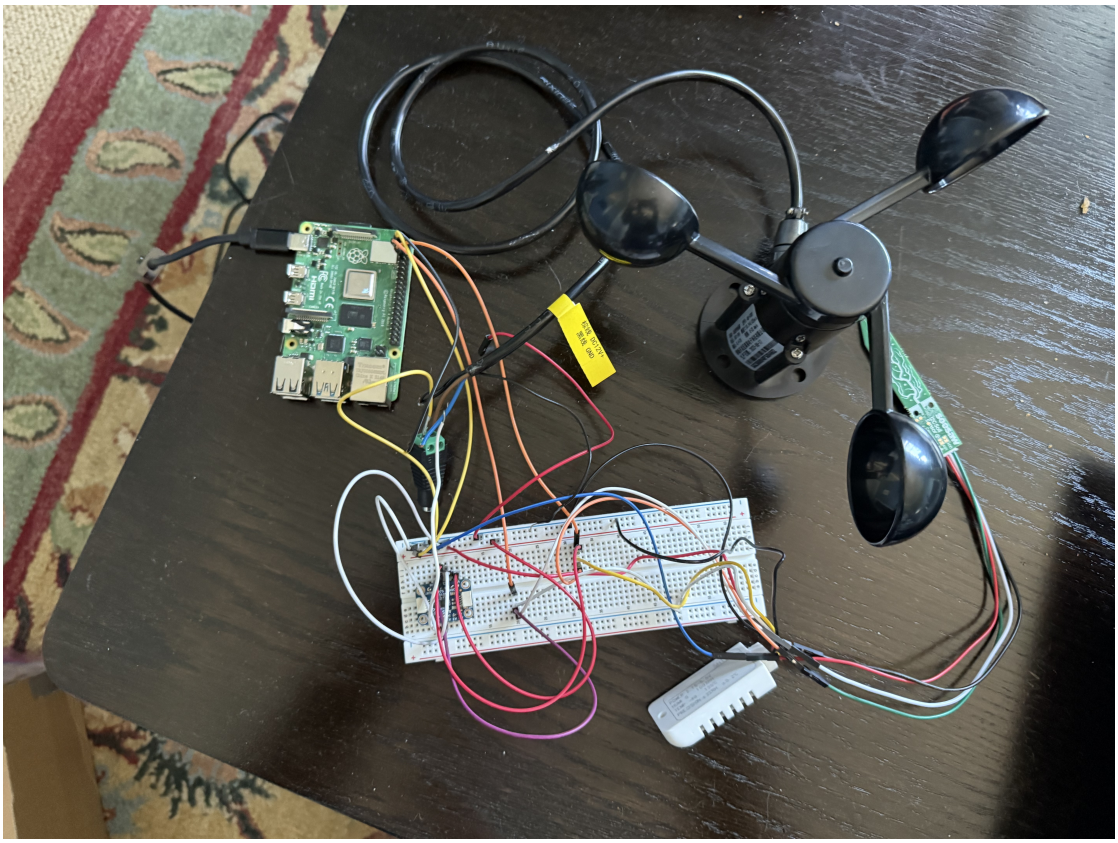

Figure 2: One of the Stations

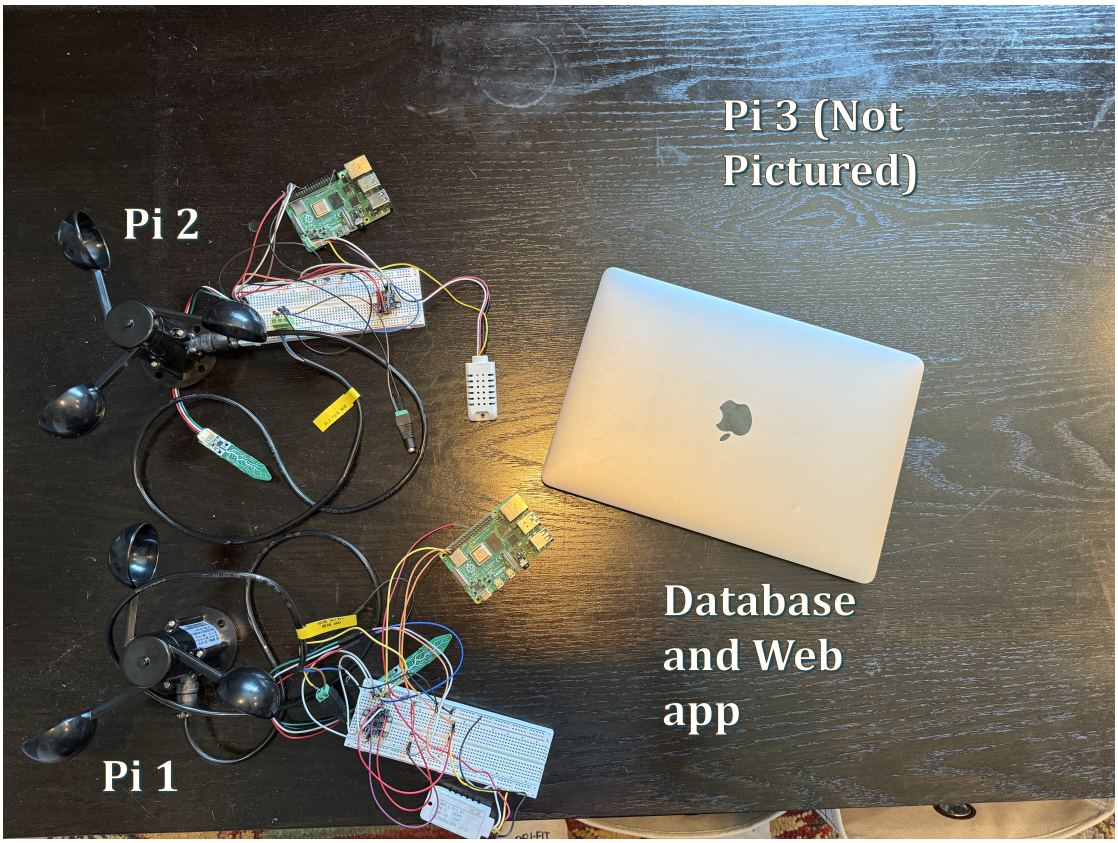


Figure 3: Full setup

## 1.b. System Overview

This setup consists of three Pi's and a personal computer to host both the data base and the web server. Then in order for the wider internet to access the web page we used an Ngrok tunnel. This service allows users to request the Ngrok server rather than the personal computer directly. The Ngrok server then requests the personal computer and the response is sent in the same chain. This allows a personal device to host a webpage while not having to expose itself directly to the global internet. A diagram of this setup is show below.

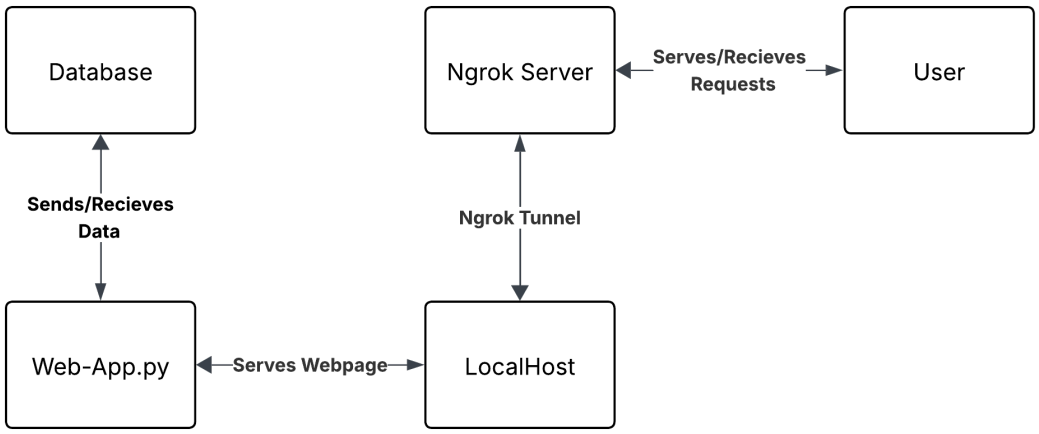


Figure 4: Database Setup

The pi's will then communicate with the database in two different configurations. The first configuration is where one pi is the primary and the other two secondaries.

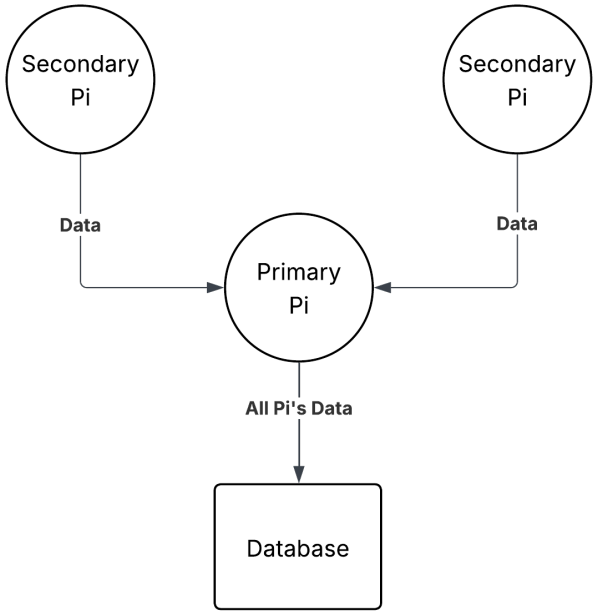


Figure 5: Polling Configuration of Pi's

The second configuration is a token ring configuration where data is passed circularly until it is sent to the database.

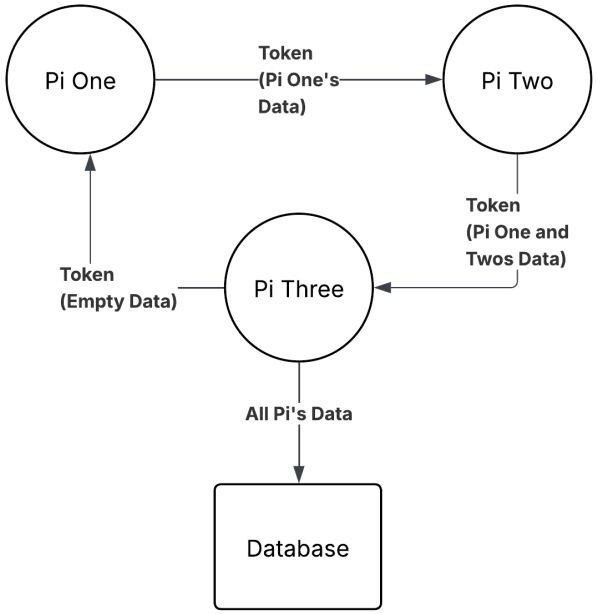


Figure 6: Token Ring Configuration of Pi's

# 2. Configuring the Database

## 2.a. Creating the Database

In this lab XAMPP was used to setup and run the database. XAMPP is an environment that contains tools to host and run a MySQL database. By simply downloading XAMPP it downloads all the relevant tools it needs, making installation simple. Once installed XAMPP provides a GUI for managing and creating databases. It does this through incorporating phpMyAdmin into the environment. To create the database it is as simple as choosing a name and hitting the 'Create database' button. Throughout the process of creating this database I was able to use the GUI for almost every action. The only SQL commands I manually needed to execute was when making the auto incrementing ID columns for the configuration tables.

## 2.b. Creating Tables

In the database there were three main tables: sensor_readings1, sensor_readings2, and sensor_readings3. These tables held the data collected by the pis. Along with these tables were three tables used for administrative duties: PrimConfig, TokenConfig, and UserAuth. Prim and Token Config were used in setting correct timeout and polling times for the pi's. UserAuth was used for user authentication, ensuring that only authorized users could make changes to these configs.

### 2.b.1. sensor_readings1,2,3

**Structure of the Sensor Readings SQL Tables**

| Column Name | Data Type | Default / Extra |
|---|---|---|
| timestamp | timestamp | 0000-00-00 00:00:00 |
| temperature | float | None |
| humidity | float | None |
| windspeed | float | None |
| soil moisture | float | None |

The above table shows the structure of the three sensor_reading tables. The timestamp column uses the MySQL timestamp datatype which was useful as it is the same as the python datetime.datetime data type.

### 2.b.2. PrimConfig and TokenConfig

**Polling Configuration Table Structure**

| Column Name | Data Type | Default / Extra |
|---|---|---|
| ID | int(11) | AUTO_INCREMENT |
| NUM_CONNECTIONS | int(11) | 0 |
| POLL_DELAY | int(11) | 1 |
| PRIMARY_TIMEOUT | int(11) | 1 |
| SECONDARY_TIMEOUT | int(11) | 5 |
| USER | text | None |

**Token Ring Configuration Table Structure**

| Column Name | Data Type | Default / Extra |
|---|---|---|
| ID | int(11) | AUTO_INCREMENT |
| NUM_CONNECTIONS | int(11) | None |
| TIMEOUT | int(11) | 5 |
| CONNATTEMPTS | int(11) | 5 |
| POLLDELAY | int(11) | 1 |
| USER | text | None |

The above two MySQL tables are very similar with the only differences being the change in timeouts and CONNATTEMPTS in the token ring table. Primary and Secondary timeouts need to be defined only in the polling configuration as pi's may be performing different functions in this configuration. In token ring only one timeout needs to be defined however another value, CONNATTEMPTS, is needed as well. CONNATTEMPTS refers to how many times a pi will try and connect to the next pi

in the ring before initiating a reconfigure.

The ID column is used to identify what the most recent configuration should be. This column is auto incremented whenever a row is added and ensures that only the freshest row is used. The NUM_CONNECTIONS column is changed by the Pi communicating with the database and is used to determine how long the timeouts should be in conjunction with the POLL_DELAY. If there are lots of pi's connected then the timeout needs to be longer than when there are less pi's connected. POLL_DELAY is the time that a pi waits before requesting data, in the polling case, or sending data, in the token ring case. Lastly, the USER column is used to log who made the config change. This could be any of the pi's in token ring, the primary pi in polling, or a user who is signed into the web service with adequate privileges. This allows for debugging as well as in a real scenario to see if there are unwanted changes being made by users.

#### 2.b.3. UserAuth Table

**User Authentication Table**

| Column Name | Data Type | Default / Extra |
|---|---|---|
| `User` | `text` | None |
| `Password` | `text` | None |
| `Auth` | `enum('Admin', 'Editor', 'Viewer')` | None |

This table is used to sign in users when they are accessing the web service. The USER and PASSWORD columns are used to check whether the user signing in has an account. Then the AUTH column is used to check the users privilege when attempting to perform some action. The admin privilege is the highest and allows the user to do everything. In the future this could extend to things such as adding more users or changing other users permissions. The editor privilege allows a user to change the polling delay of the pis, and also contains all access that the viewer privilege has. The viewer privilege allows users to query the database by date and see entries that match the specified date. Users that are not signed in are simply only able to see the graphs.

## 3. Configuring the Web Server

The python Flask library was used to create the web server. No additional configuration was needed other than importing the flask session library so that users could sign in. The web server has the following endpoints.

## 3.a. Endpoints

### 3.a.1. / "Index"

This is the homepage of the web site and is found by simply going to the url. This endpoint plots the graphs based on data in the database, pulls the weather forecast, and then renders this by using a template which it returns. This endpoint is only accessible via GET requests.

### 3.a.2. /Login

This endpoint is used to receive user and password information from the client and try and sign the client in to an existing account. On success it sets a session variable to the users name and returns a success message which is displayed on the webpage. On failure it returns a failure message which is also displayed. This endpoint is only accessible via a POST request.

### 3.a.3. /Update_Config

This endpoint is for authorized users to update the polling delay of the pi's. It checks for authorization and if successful inserts a new row in Prim and Token config with the specified poll delay and calculated timeouts. This endpoint is only accesible via POST requests.

### 3.a.4. /Search

This endpoint queries the database based on an entered date for readings taken on that date. It returns JSON containing the readings from all pi's that match with the specified date. This endpoint responds to both POST and GET requests.

## 3.b. Displaying the Data

### 3.b.1. Displaying Sensor Data

Data plots were created whenever a GET request was sent to the website. These plots used the current data in the three sensor reading tables. A challenge with plotting this data came when trying to plot the average. The challenge was that because the timestamps were not identical between pis for sensor readings, the sensor values were considered not at the same time and therefore not easily averaged. To solve this we used the binning technique to conflate similar timestamps. The function to do this is below:

```
def get_avg_data(all_data, sensor_type):
#first value is a timestamp, second value is a list of values from each table at that
all_data_of_sensor = []
for data in all_data.values():
    for entry in data:
```

```
            timestamp = entry['timestamp']
            if any(abs(timestamp - x[0][0]).total_seconds() < 2 for x in all_data_of_sens
                # If the timestamp is close to an existing one, average the values
                for x in all_data_of_sensor:
                    if abs(x[0][0] - timestamp).total_seconds() < 2:
                        x[1].append(entry[sensor_type])
            else:
                all_data_of_sensor.append(([timestamp], [entry[sensor_type]]))

    for data in all_data_of_sensor:
        # Calculate the average for each timestamp
        temp = [sum(data[1]) / len(data[1])]
        data[1].clear()
        data[1].append(temp)
        timestamps = [x.timestamp() for x in data[0]]
        # Compute average
        avg_timestamp = sum(timestamps) / len(timestamps)

        # Convert back to datetime
        avg_datetime = datetime.fromtimestamp(avg_timestamp)
        data[0].clear()
        data[0].append(avg_datetime)
    return sorted(all_data_of_sensor, key=lambda x: x[0])
```

This function stores data as a tuple of arrays. Index zero of the tuple is an array of timestamps within 2 seconds of the first timestamp in the array. Index one is an array of sensor values that are added when a timestamp is matched. These values are averaged and that becomes a timestamp and value pair to be plotted. Note that this was completed after the in class demo and was not included in the submitted code.

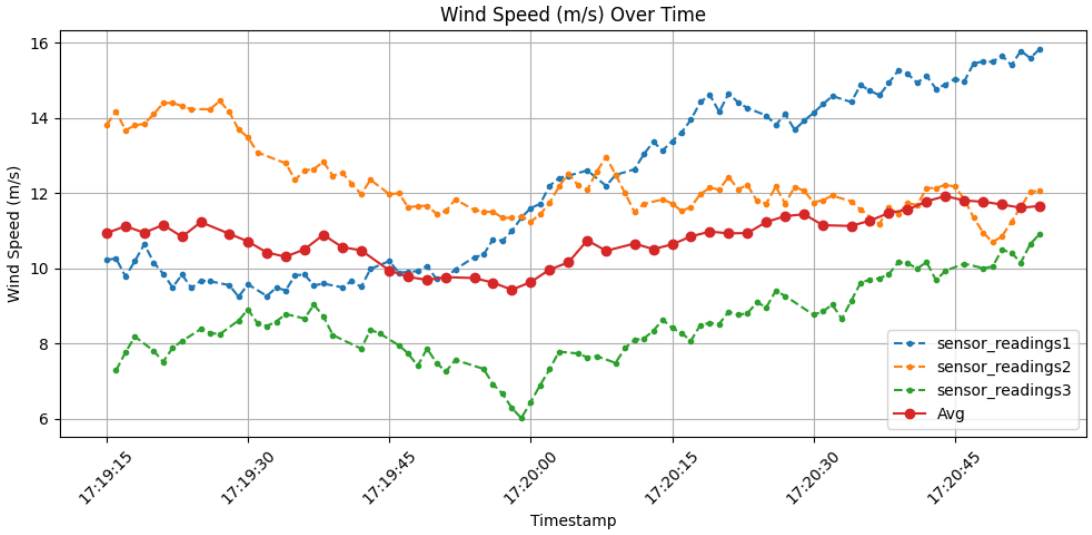


Figure 7: Example of Plot Made by Web Server

### 3.c. Displaying the Forecast

The forecast is displayed at the top of the webpage and is pulled whenever the homepage is requested. The api.weather.gov was used in order to get up to date forecast information. The response was then parsed so that today's forecast was shown. In the future more work could have been done to integrate the forecast more seamlessly into the graphs.

## 4. Adapting the Polling and Token Ring Configurations

The main challenges for both the polling and token ring configurations was allowing the pis to communicate directly with the database. The first issue that had to be solved was adding a rule to the database that allowed connections from any host on the network. This was done in the privileges tab in phpMyAdmin by creating a new rule giving any host on the network access to the database. This allowed devices to access the database however the firewall on the mac device was still blocking connections. To solve this the firewall was turned off for the duration of testing. If the project was meant to be run at all times then to improve security a rule could be added to the firewall which allows connections only when accessing the database.

The function wrote to access the database from the pis and insert the data was the same for both configurations and is shown below.

```
def write_to_db(sensor_data_):
sensor_data = sensor_data_
print(sensor_data)
```

```
timestamp = datetime.now()
cnx = mysql.connector.connect(user='root', password='',
                          host='172.20.10.2',
                          database='piSenseDB')
for i in range(len(sensor_data["wind_speed"])):
    pis_readings = []
    pis_readings.append(timestamp)
    for value in sensor_data.values():
        if isinstance(value, list):
            pis_readings.append(value[i])

    print(pis_readings)
    if(pis_readings[1] == 0 and
        pis_readings[2] == 0 and
        pis_readings[3] == 0 and
        pis_readings[4] == 0):
        print("Skipping empty readings")
        continue
    with cnx.cursor() as cursor:
        table = f"sensor_readings{i+1}"
        cursor.execute(f"INSERT INTO {table} (timestamp, temperature, humidity,`soil
cnx.commit()
cnx.close()
```

### 4.a. Polling

There were only minimal changes that needed to be made to the polling configuration. As only the primary pi needs to communicate with the database only primary.py needed to be changed. The change was just adding the above function to put the data in the database. Primary.py does this after polling all data from all connected pis.

### 4.b. Token Ring

The changes to token ring were more involved. This is because in the event of the the pi that is connected to the database disconnecting another pi needed to become the connector. To solve this a check was added that set the next pi in line to be the connector when the initial connector pi was disconnected.

## 5. Robustness

No changes were made to how either configuration handles robustness. A quick summary for each will be given here but a more in depth summary including diagrams is available in lab report three.

### 5.a. Polling

Both the primary and secondary pis use timeouts as their main technique for detecting lost connections. On startup of a secondary pi it attempts to connect to primary, on a timeout it tries again indefinitely. When a secondary pi connects to primary its ip address is added to a set which primary.py iterates through in order to ask for data. If a secondary pi takes too long to respond it is removed from the set and must reconnect through the initializing stage again. If a secondary pi times out while waiting to be polled it re-initializes by attempting to connect to primary again.

### 5.b. Token Ring

The failure handling for token ring is more in depth. When a pi fails to receive a response a set number of times it removes the pi in front of it from the ring and sends its data to the next pi in the chain. The benefit of this is that no other pis need to know change anything of how they operate. Then if the pi comes back online it must make itself known. To accomplish this the pi that used to receive data from the disconnected pi has a check to see if it its current sender is the original pi. If it detects that it is, it sends a reset flag forward telling the next pi to go back to the original configuration.

## 6. Comparing Polling and Token Ring

In comparing the two topologies two scenarios are important to consider. The first is if all pis have data to send. In this case, token ring has an advantage due to its lesser number of total messages sent. Each pi sends one message containing its and the previous pis data. In the polling configuration however, there are two messages exchanged for each exchange of data. The initial request from primary and the response from secondary. This is overhead that the token ring configuration does not encounter. Note that in the case of a very lossy network with lots of pis in the ring token ring could have worse performance. This is due to the messages being sent growing large in size and retransmissions becoming more costly. In the polling configuration this is not an issue.

In the second case only some nodes have data to send. In this case the polling configuration would perform better. In the token ring case even if a pi does not have data to send it still has to transmit the previous pis data. This could result in a large amount of transmissions containing only one pis data that could have been better served if just sent straight to the 'main' pi. Polling avoids this as if a pi does not have data to send it simply responds with no data and the primary pi can move to the next.

In terms of fairness both topologies are very similar in the way they were deployed in this lab. Token ring slightly provides more guaranteed fairness as in the polling configuration secondary nodes are reliant on the primary node.

A topology that could be interesting to implement would be a mesh topology where all nodes communicate with all nodes. This would lead to better fault tolerance which

is important in WSNs.

# 7. Extra features

## 7.a. Variable Polling Rates

In a real world scenario being able to change how often sensors poll data is useful as data needs may vary over time. To accomplish this the SQL database was used to store configuration data including things like timeouts, number of reconnects, and data rate. The tables used were PrimConfig and TokenConfig as mentioned above in the data base section of the report. Both users of the website as well as the pis themselves can update these configurations. On the website there is a form where an authorized user is able to set the polling delay. The web app then calculates based on the number of connections what to set the timeouts as and posts it to both tables. There is a user column in both tables which acts as a log for who made the changes.

At the beginning of each cycle, all pis query the most up to date configuration from their respective table, Prim or Token. This gives each pi its timeouts, polling delays, and connection attempts for each cycle. If the number of connections changes from whats in the table, the pi will insert a new configuration with the new number of connections and new calculated timeouts. This means that all pis have access to the most up to date configurations at all times.

## 7.b. User Sign in

In order to facilitate the variable polling rates a user sign in and authorization system was implemented. Users sign in and each account has a set level of authorization which allows them to do different things. Admin has full access, editor can edit the configuration and use the search function, and viewer can just use the search function.

To do this flask has a built in session library which is used to keep session variables for the duration of a users visit to the site. By setting a users session variable to their username when they sign in it is able to be checked for authorization when attempting to access the aforementioned functions. A future addition would be allowing the admin to add users as that currently needs to be done manually in phpMyAdmin.

## 7.c. Search by Date

The final extra function added was a search bar that takes a formatted date string and returns all readings from that date. To do this we used the DATE function given in SQL. Currently the search just responds with JSON which is displayed as is. To extend this adding plotting to this data or highlighting the selected dates on the graphs would make it a more beneficial feature.

## 8. Conclusion

In this lab we integrated a full stack app including the data collection. We learned how to use technologies flask, ngrok, and sql connector in order to accomplish this. We also gained familiarity with phpMyAdmin. We also learned the useful technique of binning when dealing with irregular data which proved to be very useful. To improve upon this lab we could better integrate the days forecast into the gathered data.