CSE 157 - Lab 4

Dallas Meyer

Dec 10, 2023

1 Overall design

1.1 Brief Introduction to the Lab

In this lab, we continue the polling and token-ring Raspberry Pi (RPi) systems that we have created and discussed in lab 3. This is done via adding a computer that acts like a cloud server, which runs a SQL database to store data.

An additional front-end element was added via a website hosted on the same "cloud" computer, showing current sensor information.

As for the last part of the lab, which was part 4, we had to implement a system where if the sensors reach a certain values, then we can send out the wild-fire risk. This was indicated via the front-end/website, where information on whether the risk was low/moderate/high, along with whether an active wild-fire had occurred.

1.2 IoT System Components

The overall schematic of the two different IoT systems can be seen below: where Fig.1 is the polling system and Fig.2 is the token system.

Looking at the polling system (Fig.1), the two Secondary Pi's are in charge of collecting sensor data when requested from the Primary Pi. The Primary Pi collects its own data, and then sends all three sensor data to the "cloud"/laptop. More detail on these steps below.

To connect these Raspberry Pi's together, we re-used the Ad-hoc from lab3. Most importantly, the secondary RPi's were not connected to the internet. This allows communication in areas that don't have Internet.

However using Ad-hoc brings the question on sending data to the laptop. To handle this, my lab partner Keith connected the Primary RPi to the Internet via an Ethernet connection, which was a tricky and lengthy setup until we realized that the Raspberry Pi OS must be an older version (bullseye). Using the Ethernet was useful, since it and the WiFi adapter are separate, meaning there are technically two network interfaces available to use on the Raspberry Pi. Thus allowing simultaneous connections.

After establishing connection to the Internet, we had to figure out how to send the sensor data into the laptop's SQL server. To do this, we gathered the laptop's IP address and SQL port. Then using this information to insert the data into the SQL database tables via the MySQL python library.

For the laptop, we hosted a website using Flask, which reads off the data from the MySQL server. More details in the next section.

Now comparing this to the token server (see Fig.2), the only difference in regards to the cloud and Raspberry Pi system, is that the third Raspberry Pi in the token system is in charge of sending the data. Ignoring any error/special-cases, the token system would appear to function the exact same as the polling system.

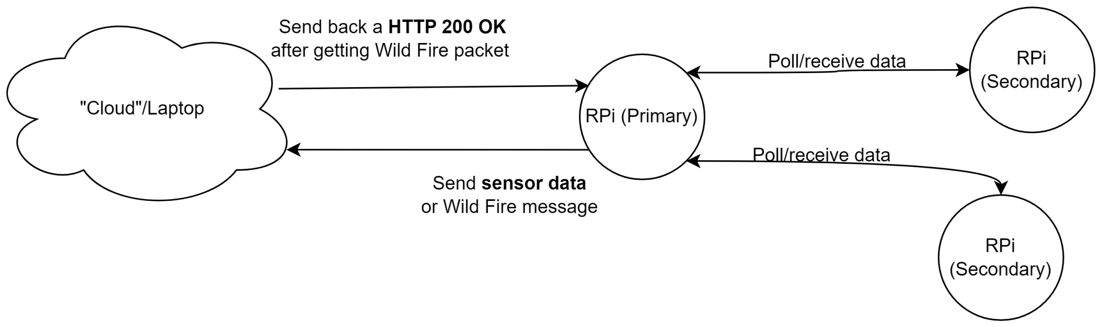


Figure 1: Cloud with polling system overview

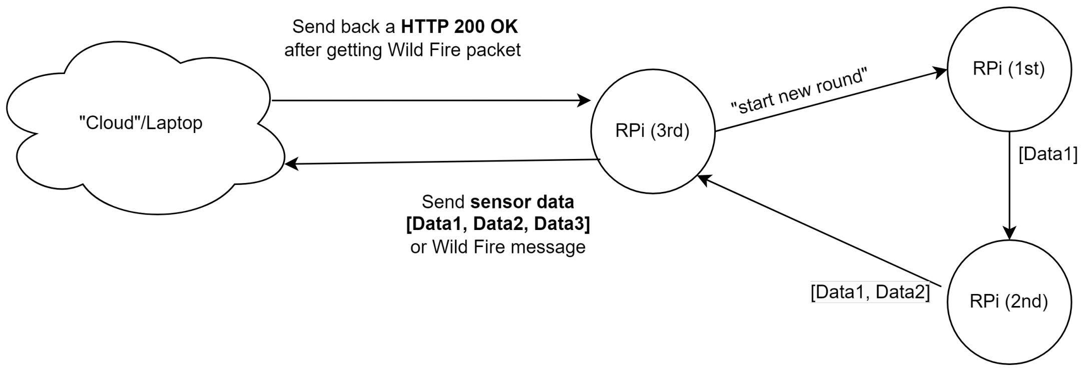
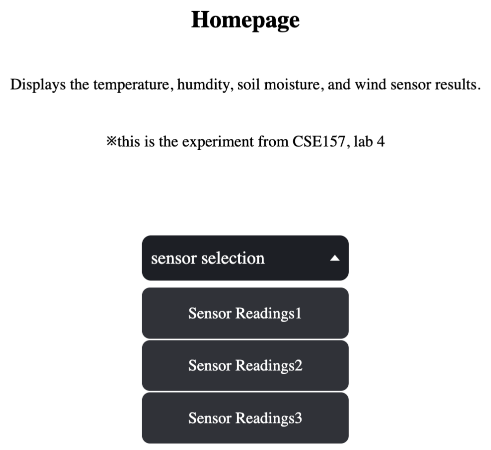


Figure 2: Cloud with token system overview

2 Web Server


Homepage

Displays the temperature, humdity, soil moisture, and wind sensor results.

※this is the experiment from CSE157, lab 4

Figure 3: Homepage for part 2-3

For the web server, it was implemented using the Flask Python library as this framework made it easier to render pages and provide dynamic HTML rendering.

My lab partner Airi did a huge chunk of the foundational HTML, CSS and Python programming to get just a basic website (see Fig.3 and Fig.5). I then added more dynamic functionality, such as dynamic plot (Fig.4) and data loading, color-coded risks (Fig. 14) that is evident in part 4 of the lab.

As for out-of-the-box functionality, Flask already handle lots of web-server handling. Luckily there were easy to use commands to set the IP address of the web server, so that the RPi and other devices on the LAN could see the website: **get.Flash.app.run(host="169.233.161.85"**).

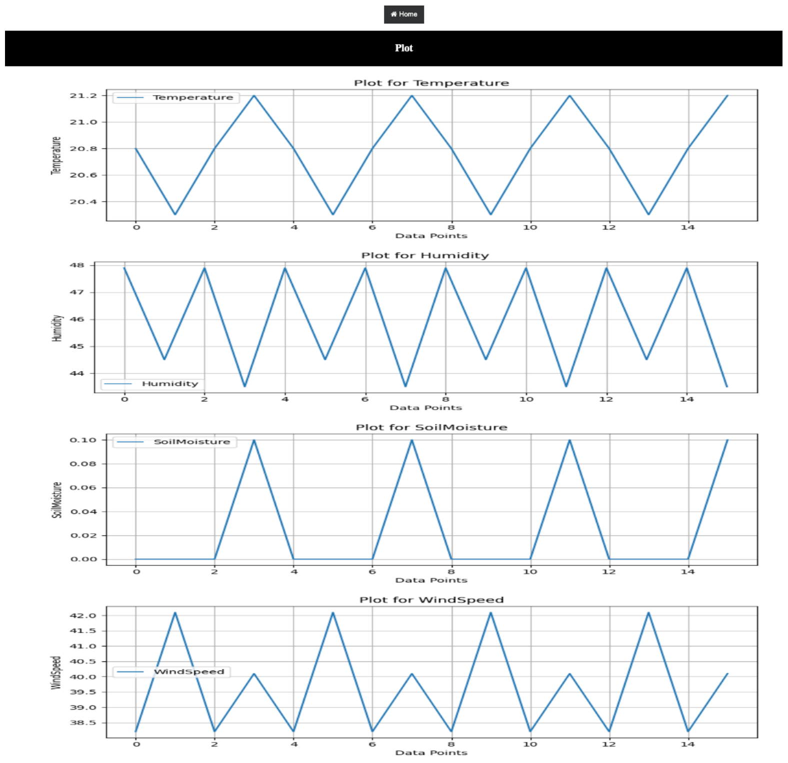


Figure 4: Example plots for part 2-3, showing the temperature, humidity, soil moisture, and wind speed.

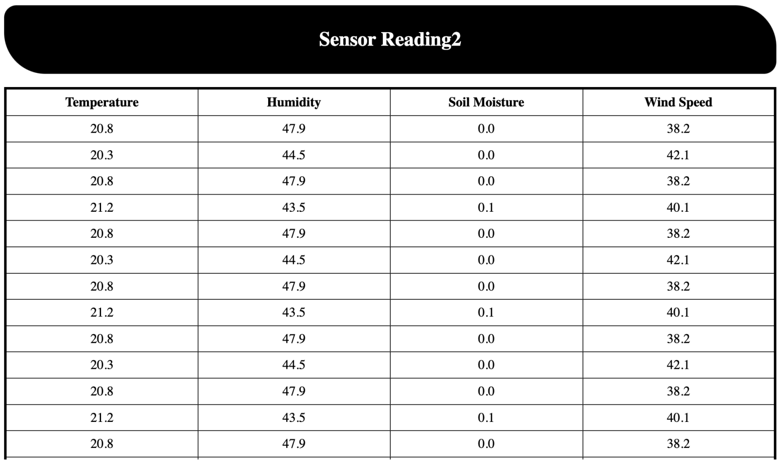
Since the plots used .png images, getting it to update automatically was a bit tricky. However Flask made it convenient since it runs specific Python code for each specific HTML file – or GET request. Thus we added a function to read SQL data and generate plots of it every time a GET request or page refresh occurred. See python code below:

Listing 1: web-app.py: Flask server class

```
class GetFlask:
    def __init__(self):
        self.risk_danger = False
        self.app = Flask(__name__)

        # Homepage
        @self.app.route("/")
        def check():
            return render_template("index.html")

        # Page for Sensor1
        @self.app.route("/sensor1")
        def sensor1():
            # receive data from mySQL and put it in the
                "array"
            mysql = mySQL()
            array = mysql.print_table("sensor_readings1")
            plot_data("sensor_readings1")
            # using the data "array", create the web page
                using the html file
            return render_template("sensor1.html",
                content=array, risk=self.risk_danger)
```



Sensor Reading2

Temperature	Humidity	Soil Moisture	Wind Speed
20.8	47.9	0.0	38.2
20.3	44.5	0.0	42.1
20.8	47.9	0.0	38.2
21.2	43.5	0.1	40.1
20.8	47.9	0.0	38.2
20.3	44.5	0.0	42.1
20.8	47.9	0.0	38.2
21.2	43.5	0.1	40.1
20.8	47.9	0.0	38.2
20.3	44.5	0.0	42.1
20.8	47.9	0.0	38.2
21.2	43.5	0.1	40.1
20.8	47.9	0.0	38.2

Figure 5: Example table showing the data from Sensor Reading2

As seen above in Fig.5, the temperature, humidity, soil moisture, and wind speed is printed out chronologically with decimal numbers on the website.

3 Database

For the SQL database, running the SQL server was handled by XAMPP, where just hitting ”run” on the SQL server was all that was needed to deploy it. XAMPP’s use of the phpMyAdmin GUI made it very convenient to quickly initialize any tables and manage the database as well.

3.1 Designing the SQL database

When designing the database, we had to figure out what columns/headers we wanted to use, along with how many different tables. Since we had three Raspberry Pi’s collecting data, it seemed intuitive to create three separate tables (Fig.6). Part 4 adds two extra columns. Since part 2-3 and part 4 used different table sizes, we used two different data modules: pisensedb, and pisensedb_part23.

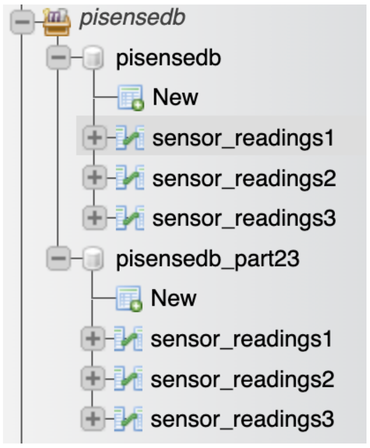
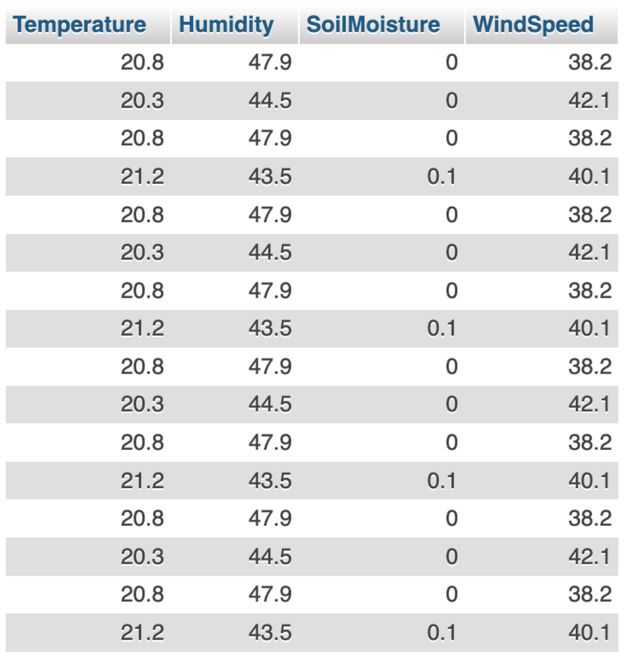


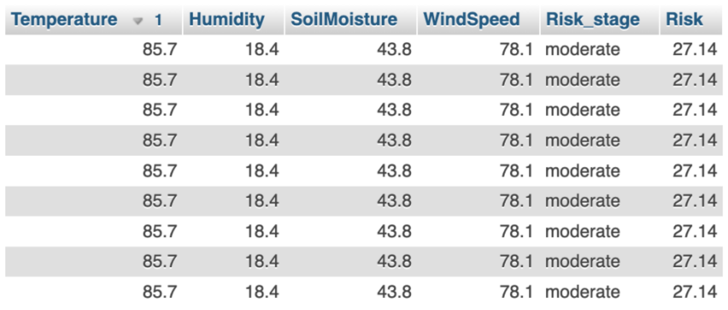
Figure 6: SQL database server table structure/diagram, showing pisensedb (part 4) and pisensedb_part23 (part 2-3) as separate data modules

As seen in Fig.7, there are four columns: ”Temperature”, ”Humidity”, ”SoilMoisture”, and ”WindSpeed”. Whereas in part 4 for the wild fire portion, an extra two columns ”Risk_stage” and ”Risk” (percent) was added. See Fig.8.



Temperature	Humidity	SoilMoisture	WindSpeed
20.8	47.9	0	38.2
20.3	44.5	0	42.1
20.8	47.9	0	38.2
21.2	43.5	0.1	40.1
20.8	47.9	0	38.2
20.3	44.5	0	42.1
20.8	47.9	0	38.2
21.2	43.5	0.1	40.1
20.8	47.9	0	38.2
20.3	44.5	0	42.1
20.8	47.9	0	38.2
21.2	43.5	0.1	40.1
20.8	47.9	0	38.2
20.3	44.5	0	42.1
20.8	47.9	0	38.2
21.2	43.5	0.1	40.1

Figure 7: Example table from part 2-3



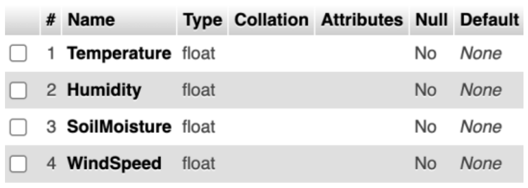
Temperature ▼ 1	Humidity	SoilMoisture	WindSpeed	Risk_stage	Risk
85.7	18.4	43.8	78.1	moderate	27.14
85.7	18.4	43.8	78.1	moderate	27.14
85.7	18.4	43.8	78.1	moderate	27.14
85.7	18.4	43.8	78.1	moderate	27.14
85.7	18.4	43.8	78.1	moderate	27.14
85.7	18.4	43.8	78.1	moderate	27.14
85.7	18.4	43.8	78.1	moderate	27.14
85.7	18.4	43.8	78.1	moderate	27.14
85.7	18.4	43.8	78.1	moderate	27.14

Figure 8: Example table from part 4, showing wild-fire risk

3.2 Data representation

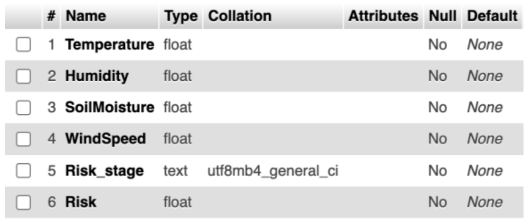
In regards to data representation, for part 2-3 we simply used float values to store the sensor data (Fig.9).

As for part 4, the ”Risk Percent” is a float. ”Risk stage” is a string/text (Fig.10).



	#	Name	Type	Collation	Attributes	Null	Default
☐	1	**Temperature**	float			No	*None*
☐	2	**Humidity**	float			No	*None*
☐	3	**SoilMoisture**	float			No	*None*
☐	4	**WindSpeed**	float			No	*None*

Figure 9: Data types for part 2-3



	#	Name	Type	Collation	Attributes	Null	Default
☐	1	**Temperature**	float			No	*None*
☐	2	**Humidity**	float			No	*None*
☐	3	**SoilMoisture**	float			No	*None*
☐	4	**WindSpeed**	float			No	*None*
☐	5	**Risk_stage**	text	utf8mb4_general_ci		No	*None*
☐	6	**Risk**	float			No	*None*

Figure 10: Data types for part 4

3.3 Operations used on the SQL database

When creating the SQL database and initializing the tables, we used the phpMyAdmin GUI as it made it easy to add tables/columns without needing to run SQL commands in a terminal.

However on the RPi, a little bit of setup was required. First we needed to determine how to connect to the SQL server across the Internet. This was easily done by simply specifying the IP address of the laptop running the SQL server, along with what database we wanted to edit. This was done in one python method when creating the connection object:

self.conn = mysql.connector.connect(host="cloud_IP_address", database="pisensedb", user="root", password="")

As for manipulating data using Python code, this part was a bit tricky and required a bit of trial and error. First we tested the commands (eg. INSERT) using the Python MySQL library locally on the laptop, so that we can have python code that works before putting it on the Raspberry Pi.

As to add more modularity and clean code, we also converted those commands into a Python class called mySQL. See the python code below, which shows the create_connection() and insert_update() functions. All that was needed was to then run these commands on the Primary/3rd Raspberry Pi to add new data.

Listing 2: web-app.py: mySQL class

```
class mySQL:
    """
    create connection with MySQL and handle tables
    """

    def __init__(self):
        self.conn = None
        self.data_array = []

    def create_connection(self):
        """
        Create connection with MySQL database
        """
        self.conn = mysql.connector.connect(
            host="localhost", database="pisensedb",
                user="root", password=""
        )

    def insert_update(self, sensor, data):
        """
        Insert data into the database table

        Args:
            sensor (string): name of the table
            data (float): list of data to insert into the table
        """
        try:
            self.create_connection()
            cursor = self.conn.cursor()
            insert_query = f"INSERT INTO {sensor} (Temperature, 
                Humidity, WindSpeed, SoilMoisture) VALUES 
                ({data[0]}, '{data[1]}', '{data[2]}', 
                '{data[3]}')"
            cursor.execute(insert_query)
            log.debug("data sent")
            self.conn.commit()
            cursor.close()
            self.close_connection()
        except mysql.connector.Error as e:
            log.debug(f"insert_update: Error inserting data: 
                {e}")
```

As seen in the code, we execute the SQL command that we created in the string insert_query through the cursor.execute() method.

Clearing the table was done similarly, where instead the query is "DELETE FROM sensor table".

4 Polling vs. Token-ring

As discussed previously, we have implemented a cloud server to work with the polling and token-ring IoT systems. Here will we compare the approaches in two scenarios: when some nodes have data to send, and when all have data to send.

4.1 Not all nodes have data to send

For the polling approach, if one of the secondary Pi's don't have data to send (eg. perhaps error with reading sensors), then following lab3's handing, the primary Pi will have a polling timeout for that secondary Pi and set its data values to zero. This won't trigger any wildfire warnings, as the conditions won't be triggered from zero values – this will be explained in the next section. If the primary Pi has no data to send, then it will also send values of zero.

For the token-ring approach, if one of the Pi's doesn't have any data to send, this means it could also default to having zero values sent. However if lets say if no data has ever been sent to the next listening Pi, then the timeout would be triggered and thus Zero values would be inputted. In the case of the 3rd Raspberry Pi having no data to send, then the SQL server would simply not receive any data.

When comparing the two approaches, the polling approach would be more useful in the idea that if one of the Pi's is completely in-operable or doesn't have data to send, then the Primary Pi could just skip it.

4.2 All nodes always have data to send

If all nodes always have data to send, then the polling approach could be more efficient than the token approach. This is because if needed the polling approach could use synchronous polling.

However in terms of having the Pi's out in the wilderness for actual readings, the token-ring may be more suitable since the Pi's could be more spread out and cover a larger distance, since the Pi's only have to rely on the other adjacent Pi's to network connectivity. On the contrary, the polling approach is more limited as each Pi must be within the distance of the Primary Pi.

4.3 Fairness to allocating access to the shared medium

Assuming the shared medium is the polling's data channel from the primary Pi, or the token in the token ring, there are different factors to think about.

When the Primary Pi polls, its fair when the Pi polls each secondary Pi one at a time and doesn't allocate any more time to another pi. For instance, if the secondary Pi sends out data after getting a polling request, right before it times out, this would be unfair since the network medium would be under utilized.

In the case of the token ring, one Raspberry Pi could keep the token and prevent any other Pi's from sending data.

In this situation, the token-ring would allow much more unfairness compared to the polling approach, as polling has a "controller" to determine who can send data and keep things fair.

5 Emergency message generation

5.1 Resources used on deciding Wild Fire Conditions

My lab partner Keith based our wild fire conditions from this resource: https://www.nwcg.gov/publications/pms437/fire-danger/nfdrs-system-inputs-outputs.

This page contains the threshold conditions for the wind speed, relative humidity, and temperature (Fig.11). This is because this website covered areas such as our Redwood Forests, as such it is relevant to Santa Cruz.

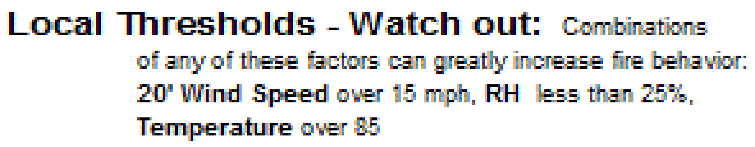

Local Thresholds - Watch out: Combinations of any of these factors can greatly increase fire behavior:
20' Wind Speed over 15 mph, **RH** less than 25%,
Temperature over 85

Figure 11: Enter Caption

Instead of just implementing the threshold values verbatim, we ensured that we also had some realistic risk thresholds: low, medium/moderate, and high. This is evident in Fig.13 and Fig.14, where we can see a green "low", yellow "moderate" and red "high risk" reading for the sensor's polled data. The way these risks are calculated is within our wildfire_calc.py file, which contains a class that calculates the percent risk, and then the subsequent risk stage/level.

To calculate this, we checked for two sensor readings at a time and see if they cross a certain thresholds. Depending on their current float value, each sensor would have a maximum risk of 25% (hence 25% + 25% = 50%). Thus the maximum risk percent is 50%, since anything higher than that wouldn't matter or seem realistic. In addition we have maximum values, in the case that the sensors error out, etc.

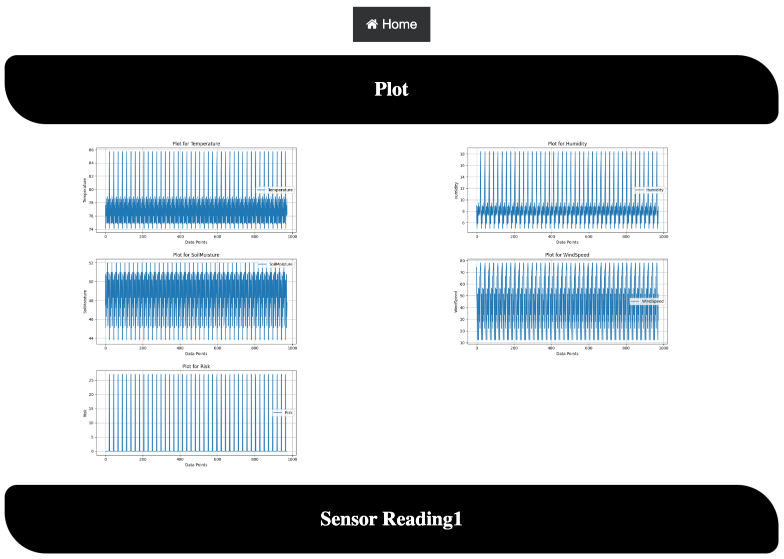
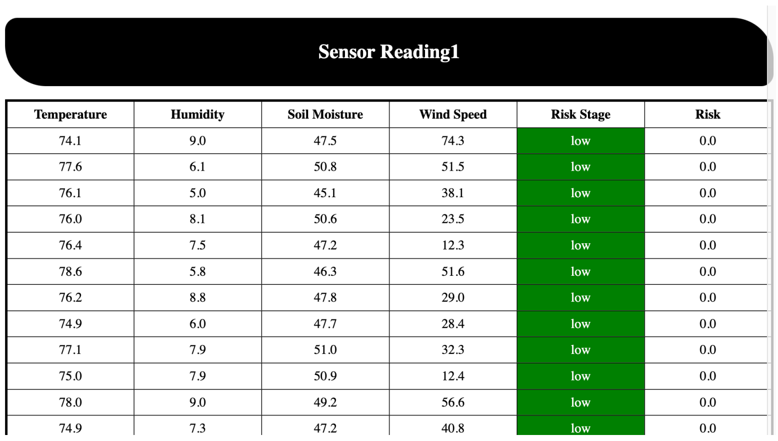


Figure 12: Continuation of part2-3 website, where plots for Wild-Fire risk percent and stages are added

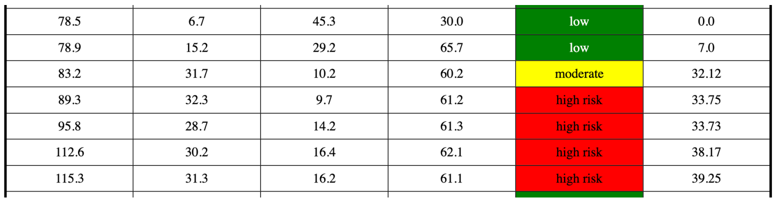


Sensor Reading1

Temperature	Humidity	Soil Moisture	Wind Speed	Risk Stage	Risk
74.1	9.0	47.5	74.3	low	0.0
77.6	6.1	50.8	51.5	low	0.0
76.1	5.0	45.1	38.1	low	0.0
76.0	8.1	50.6	23.5	low	0.0
76.4	7.5	47.2	12.3	low	0.0
78.6	5.8	46.3	51.6	low	0.0
76.2	8.8	47.8	29.0	low	0.0
74.9	6.0	47.7	28.4	low	0.0
77.1	7.9	51.0	32.3	low	0.0
75.0	7.9	50.9	12.4	low	0.0
78.0	9.0	49.2	56.6	low	0.0
74.9	7.3	47.2	40.8	low	0.0

Figure 13: Color-coded risks and readings for Sensor Reading1

[H]



78.5	6.7	45.3	30.0	low	0.0
78.9	15.2	29.2	65.7	low	7.0
83.2	31.7	10.2	60.2	moderate	32.12
89.3	32.3	9.7	61.2	high risk	33.75
95.8	28.7	14.2	61.3	high risk	33.73
112.6	30.2	16.4	62.1	high risk	38.17
115.3	31.3	16.2	61.1	high risk	39.25

Figure 14: Color-coded risks, showing a "high" risk example in red

5.2 Implementing the emergency Packet

In this step, we had to make it where the Primary/3rd Raspberry Pi sends a wildfire emergency packet to the "cloud"/laptop in the case there is a wildfire.

Our implementation used HTTP, where in the emergency_handler.py on the RPi, it sends a Post request (via requests.post(URL, payload)) of a packet to the URL

```
http://COMPUTER_IP:5000\emergency
```

with the contents of "EMERGENCY" in the payload. If successfully delivered, the website will show a banner at the top (Fig.15)

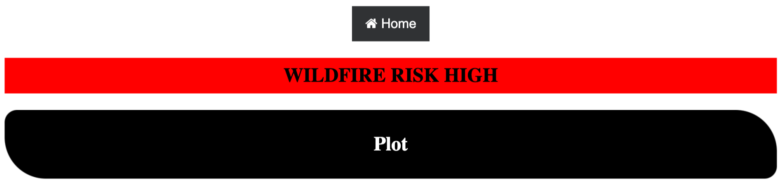


Figure 15: Wild-fire banner

5.3 Ensure Packet has been Sent to Server

Since we used HTTP, the response received from the Post request should be a **"200 OK"**. This is because HTTP automatically has responses to know if packets have been sent successfully.

5.4 Recover from an Unavailable Server

In our implementation, if the wild-fire high risk threshold has been achieved, the Raspberry Pi will keep attempting to send the message – in a blocking function – until it has been sent via an 200 OK response.

6 Conclusion

There are many takeaways in the lab. The most notable aspect of this lab was the front-end and back-end development nature of it.

Since I and Airi had no experience with SQL, this lab has been a learning experience of using a SQL server as a database to store data rather than a lackluster text file. In addition having a database allows easy multi-user access so that multiple machines can read or insert data.

In regards to front-end, learning to use Flask and creating a website was really nice and useful, as it allows us to make a user-interface that's very human-readable and adds more practicality to our lab.

Having this many different aspects to the lab, made it easier to divide up different tasks among our group. Regardless, having it has a group effort made it more fun and easier since we can combine different knowledge/practices when doing pair programming, troubleshooting, etc.

There were definitely many roadblocks, such as establishing the Ethernet connection of which somehow the newer Raspbian OS version prevented us from doing.

Overall, I'd say this lab was really fun since it felt like an actual IoT implementation that could be used in the real-world. Rather than just some sensors printing data out to a terminal.