IoT-BAsed Indoor Air Quality Monitoring System

Simone Fabbri

February 1, 2023

1 Introduction

The scope of this project is to make an air quality monitoring system, based on two sensors:

- the `DHT22` sensor, for the temperature and humidity;
- the `MQ-2` sensor, for various gas monitoring (like CO and methane).

The sensors are connected to an `ESP-32` board; the *raw data* is gathered and stored on a InfluxDb instance, and is possible to view it through 2 Grafana dashboard, together with humidity, temperature and gas *predictions*.

Furthermore, **2 additional components** are developed:

- an **external temperature** gathering system, which collects the external temperature hourly thanks to the Open API Weather service and the **meteostat** python library;
- a **telegram bot**, which alerts when the air quality is below a certain threshold and sends a rendered image of the various Grafana panels at regular intervals.

The next sessions explain the technical details of how these goals were achieved.

2 Project's Architecture

2.1 Hardware architecture

The hardware components used in this project are:

- `ESP-WROOM-32`: single board microcontroller, with wifi and bluetooth capabilities; in this project, both available cores and the wifi are used;
- `DHT22`, humidity and temperature sensor;

- `MQ-2`, gas sensor; it can output an analog and a digital signal; the analog signal is used.

While the DHT22 is connected to the 3.3V board output, which is the operating sensor voltage, the MQ-2 is connected to the `vin` pin, which could also be used as an output voltage pin. A scheme is showed in figure 1.

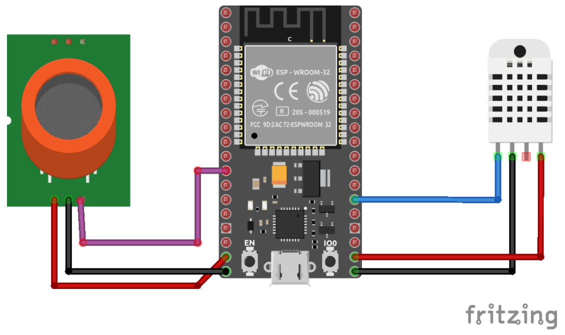


Figure 1: Electric scheme for the project's hardware architecture.

2.2 Software architecture

The software architecture is composed of the software (and libraries) for the development board, and the software for the various servers and services which run on a computer.

All the components running in a computer are developed in a docker image: this means that there is an image for the data proxy, an image for the data analytics... All the images where built from scratch, except for Grafana, a plugin called grafana-image-renderer and InfluxDB, which images are already provided from *hub.docker.com*.

All the containers are managed through a *docker compose* file.

2.2.1 IoT smart device

The code for the Esp 32 and the sensors is stored in the `esp_32` folder. There are two versions of the code:

- `main`: is the main version of the logic for the board.
- `perf_eval`: is a simplified version of the main source code, with the only purpose of evaluating the performance (time and packet loss) of the protocols used to exchange sensor data with the data management components.

The (main version of) the IoT smart device logic is in charge of connecting to the network through wifi, acquiring the data and sending it; there are 2 available protocols to do this:

- coap;
- http.

Through the coap protocol is also possible to change some settings, like the interval between the sensor acquisitions, or the `MIN/MAX_GAS_VALUE`.

To change these settings, an external software is used: `libcoap2-bin`, available in the linux apt repository. This program provides a binary, `coap-client`, through which is possible to send messages to a coap server.

The code is splitted into several files:

- `main.ino`, contains the two basic functions `setup()` and `loop()`;
- `settings.h, settings.cpp`: stores some settings like the coap/http server ip addresses and handles the changing of the settings;
- `data.h, data.cpp`: handles getting and sending all the data (sensors, AQI, gps location...).

For the performance evaluation, there is also an extra python script in the `performance_eval` folder which is used to analyze the data collected through the `perf_eval` version of the logic to control the board.

2.2.2 Data proxy

The data proxy consists in a python module which runs the coap and the http servers.

This server is started from a docker container, which image is defined through a **Dockerfile**. The ports of these services are exposed to the system.

Some common settings, like the InfluxDb token, are saved in a common file, `settings.json`, accessible to all the software components (except of course the ones for the Esp board). The use of this file is omitted in the next paragraphs, to avoid repetition.

The coap and http server are based on two libraries, *aiocoap* and *aiohttp*.

2.2.3 Data management

The data management is based on an InfluDb docker instance and a Grafana docker instance; furthermore, a *Grafana Image Renderer* server is used to render the Grafana panels as images.

All these components are instantiated through the **docker compose** file, and the ports are exposed to the system.

2.2.4 Data analytics

The data analytics consist of predicting the values of temperature, humidity and gas raw values.

This prediction is done by a python module, `data_prediction`, which accomplishes this goal thanks to the **Facebook Prophet library**.

2.2.5 Extra components

There are two extra components, one responsible for getting outdoor meteo information, and the other is the telegram bot. Both components are python modules.

Like the modules above, each run in its separate docker container, defined though its proper Dockerfile.

3 Project's Implementation

3.1 IoT smart device

The code for the IoT smart device is divided in several files, briefly introduced in the architecture section 2.2.

The most imporant libraries used to implement the different functions are:

- `coap-simple.h`, for coap communications;
- `WiFi.h`, to connect to a WiFi connection;
- `DHT.h`, to read values from the temperature/humidity sensor;
- `HTTPClient.h`, to send http requests.

In the next paragraphs, the relevant content of the various files is described.

The details for the `perf_eval` version of this code will be described in the section about the performance evaluation, since the small changes affect only parts specific to the performance evaluation task.

3.1.1 main

In this file, the SSID and password for the network to connect is hardcoded in two variables (`SSID` and `PASS`).

In the `setup()` function, the serial communication, the WiFi connection and the DHT sensor readings are initialized; furthermore, the coap server is also started, to be able to receive commands from a client; this last part is handled by the following instruction:

Listing 1: Coap server

```
// CoAP server endpoint url callback
void coap_change_settings(CoapPacket &packet, IPAddress ip, int port) {
  Serial.println("[Setting something]");
  set(packet);
  coap.sendResponse(ip, port, packet.messageid, "ok");
}

void setup(){
  //...
  coap.server(coap_change_settings, "set");
}
```

The `set()` function is defined in `settings.cpp`.

In the `main` file is also defined the coap callback function (which simply prints the response).

The `coap.loop()` function is not handled in the main thread, but in another thread, since the ESP 32 has two cores; this means that when we want to change settings, the board can handle the request almost immediately. This is achieved through the system function `xTaskCreatePinnedToCore()`.

Three things happen in the `loop()` function: the data is acquired, the data is sent and the thread sleeps for some time, as showed in the listing below:

Listing 2: Loop function

```
void loop() {
  Serial.println("Getting data...");
  get_data();
  if (TIME_WINDOW[4] >= 0){
    Serial.println("Sending data...");
    send_data();
  }
  delay(SAMPLE_FREQUENCY);
}
```

The functions are defined in `data.cpp`. Note that the data is not sent immediately, but after 5 times that we collected temperature/humidity/gas; this is because to compute the AQI, we need the mean of 5 measures. In the next section will be described how the array `TIME_WINDOW` works.

3.1.2 data.cpp

This file has two functions: `get_data()` and `send_data()`.

The first function reads the temperature and humidity through the `DHT` library. For the gas, the analog value of the pin where the MQ-2 is connected is read; this value varies between 0 and 4095. [4]

The value of the gas is then inserted in the `TIME_WINDOW` array; this array is initially initialized with negative values, so when in `TIME_WINDOW[4]` there is a ≥ 0 value, we know we have read at least 5 values, and we can compute the AQI.

When the 6th, 7th, ... and so on read is done, the value is inserted in `TIME_WINDOW` in a circular way, starting again from position 0.

All the values are then inserted into a data struct, as the listing below shows:

Listing 3: Loop function

```
struct {
  char temperature[64];
  char humidity[64];
  char AQI[32];
  char gas_conc[32];
  char RSSI[32];
  const char gps[16] = "44.4948,11.3426"; //bologna's coordinates
  const char device_id[18] = "30:C6:F7:22:83:F4"; //mac address
} DATA2SEND;
```

The function `send_data()` takes the values from the data structures; in case we want to use coap to transmit them, we first ”tag” the data: we append the gps coordinate and the device id to the value we want to send. Then, the data is sent to the appropriate handler. The example shows the tagged temperature:

Listing 4: Example of tagged data (temperature) - coap protocol

```
if (PROTOCOL==COAP){
  char tagged_data[50];
  //"tag" the data with gps and device id
  snprintf(tagged_data, 50, "%s;%s;%s", DATA2SEND.temperature,
    DATA2SEND.gps, DATA2SEND.device_id);
  int msgid = coap.put(IPAddress(COAP_SERVER_IP[0], COAP_SERVER_IP[1],
    COAP_SERVER_IP[2], COAP_SERVER_IP[3]),
    COAP_PORT, "env/temp", tagged_data);
  //...
}
```

If we want to send the data through http protocol, all the data in sent together in json format:

Listing 5: an http request contains all the data (omitted for simplicity)

```
int ret = snprintf(postData, 150, "{\"temp\":%s,\"hum\":%s\" /*...*/ }",
DATA2SEND.temperature, DATA2SEND.humidity /*...*/);
Serial.println("Sending post request...");
http.begin("http://192.168.1.17:8080/data");
http.POST(postData);
Serial.println("post done");
```

3.1.3 settings.cpp

This file contains the variables for the coap server ip, the sample frequency for the sensor values... the variables that can be changed are summarized in the table below, with the default values:

variable	default value	description
PROTOCOL	COAP	protocol in use to send the data
MIN_GAS_VALUE	500	value used for AQI
MAX_GAS_VALUE	1500	value used for AQI
SAMPLE_FREQUENCY	5000	sample frequency in ms

In this file is also defined a function, `set()`, which sets the variable requested from a client with the new value according to the request.

3.2 Data proxy

The data proxy consist of a python module where are defined the two servers, http and coap, thanks to two libraries: `HTTPServer` and `aiocoap`.

The servers are run asynchronously as coroutines thanks to the `asyncio` python library.

When a request arrives to one of the servers, every server is responsible to create a `Point` object; this object is defined in the `influxdb_client` library. For example, the http server creates this object:

```
d = [
  Point("data")
    .tag("device_id", post_data["id"])
        .tag("location", post_data["gps"])
        .field("rssi", post_data["rssi"]),
  Point("env")
        .tag("device_id", post_data["id"])
        .tag("location", post_data["gps"])
        .field("temperature", post_data["temp"])
        .field("humidity", post_data["hum"])
        .field("aqi", post_data["AQI"])
        .field("gas_concentration", post_data["gas"])]
```

then the function `save_data()` is called, to save the data to the Influx database.

Notice how the data is saved in two different *measures*, ”data” for the RSSI and ”env” for the sensor values and the AQI.

There is also a function called `compute_ppm()`, which takes the raw gas sensor value and computes the ppm of different gases (altough is impossible to know which gas stimulated the sensor in the first place). This computation is performed after some time, because it is necessary to first collect some raw values to compute the `R0` value (see the appendix A), which is initially set to 0. When the value becomes different than 0, this function computes the ppms and saves them with the `save_data()` function.

3.3 Data management

Data management is performed through an InfluxDB instance, which stores the data and does not require any special configuration, a Grafana instance to visualize the data, and a Grafana image renderer service to render the panels that will later be sent via a telegram bot.

As regards the Influx instance, some parameters like the token to access the buckets is defined in a yml file, which is used by `docker-compose` to start all the docker services, including Grafana and the image renderer.

In Grafana, there are two dashboards: one for the sensor data, in which is also shown the predictions, the RSSI, the AQI and the data collected from meteostat.

In the second dashboard is shown the AQI for the gases.

Finally, an alert was defined for the AQI; this is described in section 3.7.

3.4 Data analytics

The data analytics is composed by another python module, `data_predictions`, which is based on the library Prophet by Facebook.

This module predicts X seconds in the future, according to the `sample_frequency` value for the sensors:

Listing 6: Prophet prediction (X is `frequency` in the example)

```
future = m.make_future_dataframe(periods = periods,
        freq=str(frequency) + " S", include_history = False)
forecast = m.predict(future)
```

The value of $X = 100$ is used in the project. In the code listing, the parameter `periods` is set to $predict\_seconds/sample\_freq$, which corresponds to X/`sample_frequency`; `freq` is set to $frequency$, which is equal to `sample_frequency`. So, if our sample frequency is 5 seconds (i.e. we have a new value every 5 seconds), this corresponds to a prediction every 5 seconds, for $X/5$ times.

The predictions are then saved in Influx.

The function is executed by the python module periodically, every X seconds. The data used to make the predictions is all the raw sensor values from 24 hours before the prediction: so, if $X = 100$, if we want to predict the temperature from now, e.g. from 2023-01-01T10:00:00 to 2023-01-01T10:01:40, the module asks to the InfluxDB instance the temperature values from 2022-12-31T10:00:00 to 2023-01-01T10:00:00.

In case of an exception (likely because there is not enough data to make the first prediction), the module sleeps more before try again.

3.5 Extra components

3.6 Meteostat

A python module gets the hourly external temperature of Bologna, using the meteostat library to access the Open API Weather service.

The system runs every hour, and asks for the temperature of the last hour, and then saves the result in Influx.

3.7 Telegram bot

This is another python module; it is based on the python library `python-telegram-bot`.

The bot does not actually interact with the user, but only sends some images and notifies him when the AQI is too low.

To fulfill the first purpose, first it read a parameter from the json settings, `telegram_update_frequency`: this is the frequency of the updates. The bot simply downloads an image of every panel in the Grafana dashboards, with the data from `telegram_update_frequency` hours ago to now, thanks to the grafana image renderer plugin and the python library `requests`; after all the images have been collected, they are sent to the user. This is accomplished sending a media group to a specific chat id (also read from the json settings).

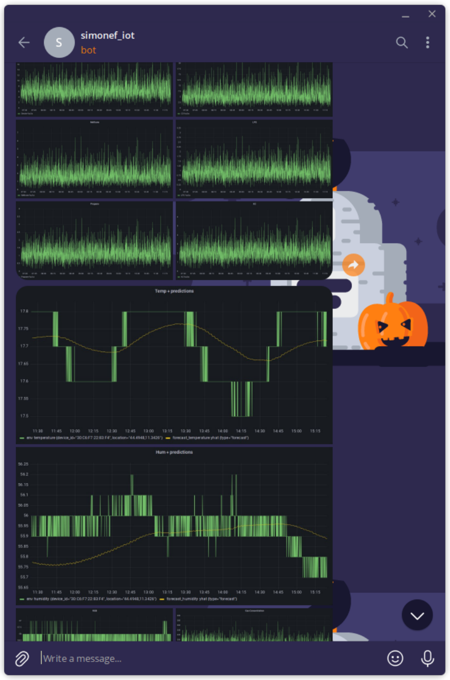


Figure 2: Telegram chat with the charts sent by the bot.

Regarding the **AQI alert**, this is a feature of Grafana: a simple contact point was added to use telegram, and configured with the telegram chat id. Then, a rule was added; the rule is composed of three expressions:

1. **Influx query:** a simple query retrieves the AQI values;
2. **reduce:** the last element of the query is taken;
3. **threshold**: if the element is below 2, the alert is triggered.

4 Results

In this section the results of various performance tests is discussed. The elements analyzed were:

- protocols (http and coap):

  - average packet delay
  - packet delivery ratio

- data analytics (Facebook Prophet):
  - MSE

The results are shown in different subsections.

4.1 Protocol evaluation

For this evaluation, 1000 packets were send with each protocol, at three different sensor collection frequency values: acquisition every 100ms, 500ms and 1000ms; so in total 6 tests were performed.

A different version of the main firmware was used: all the output messages were deleted, and some new were added, as described below.

For the coap protocol, for every packet sent the ESP32 board printed through the serial connection an 'S', the timestamp and the message id; for every packet received, an 'R', the timestamp and the message id.

The output was collected through the `screen` command and saved in a csv:

Listing 7: Command to capture the board serial output

```
screen -L \
   -Logfile perfomance_analysis_coap_500ms.csv \
   /dev/ttyUSB0 9600

# example of csv file
# S, 36984, 256285
# R, 36984, 256662
# S, 62982, 256787
# R, 62982, 257163
```

The board did non sent all the sensor data, but just the temperature.

The same approach was used to analyze the http protocol, except in this case was printed the http status code of the request and the total time required to perform it.

On the server side, the received data was not saved, to avoid the overhead involved in saving the data, since the goal was to analyze the raw protocol performance.

4.1.1 Coap

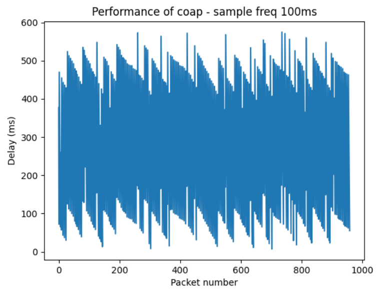
For the **coap** protocol, using `grep S`, `grep R` and `wc -l` to count the sent and received packets, we can know how many were lost. This data along some other statistics are shown in the following table:

So, in conclusion, increasing the data acquisition (and thus data sending) frequency did not change the packet loss ration much, but did positively affect the average and median delay, i.e. decreasing them.

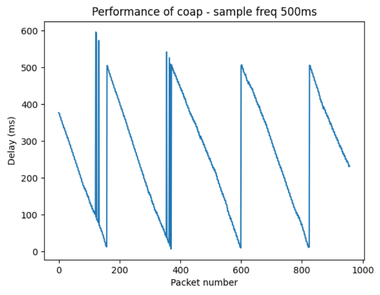
Measure	**100 ms**	**500 ms**	**1000 ms**
Packet ratio	96.1%	95.8%	96.1%
Average delay	278.9 ms	266.3 ms	264.6 ms
Min delay	7 ms	7 ms	6 ms
Max delay	575 ms	596 ms	788 ms
Median delay	278 ms	273 ms	261 ms

Table 1: Coap stats table.

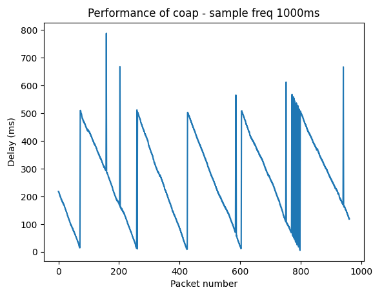
The delay for every packet is shown in the following images:



(a)



(b)



(c)

As we can see, there is a toothsaw profile, which means that after some fast packets, the delay increased very rapidly, to slowly decrease over time. This phenomenon is exacerbated in the 100 ms test.

4.1.2 http

The same techniques used in the previous section were applied here. In the http protocol, there is no packet loss, since it is based on tcp; in spite of this, some timeout due to the server not responding in time occurred, so the packet ratio is still reported in the table below:

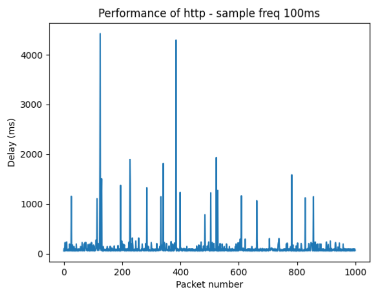
This shows that, in this setup, while http is more reliable than coap and can

Measure	**100 ms**	**500 ms**	**1000 ms**
Packet ratio	100%	99.7%	99.9%
Average delay	124.6 ms	251.1 ms	267.9 ms
Min delay	54 ms	30 ms	54 ms
Max delay	4424 ms	4515 ms	4575 ms
Median delay	65 ms	75 ms	95 ms

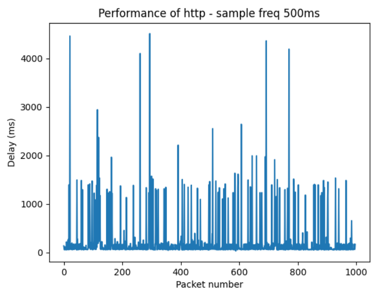
Table 2: Http stats table.

be faster on average, the minimum delay is still 9 to 13 times bigger, and also the max delay is 5 to 9 times bigger.

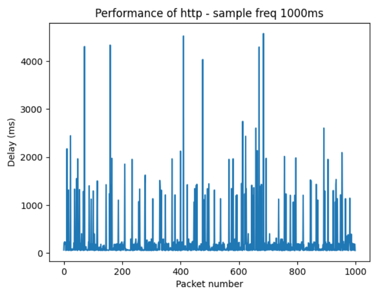
In the following images the delay every packet is shown:



(a)



(b)



(c)

It is possible to observe that the delay is more constant than with the coap protocol, but still peaks of delay are not uncommon.

4.2 Data analytics

In this section Facebook Prophet performance are analyzed.

Sensor's data was collected for about 2-3 days. Since the goal was to predict the values for X seconds in the future, 100 seconds was chosen as the value for X. Sensor data is collected every 5 seconds, so to match the real data,

Prior	**Rms temperature**	**Rms humidity**	**Rms gas concentration**
0.001	0.0153	1.9485	1683.3
0.1	**0.0121**	**0.3733**	**1304.5**
0.5	0.0156	1.9018	1709.4
10	0.0156	1.9015	1713.2

Table 3: Rms table for various priors and sensors.

the predictions were made every 5 seconds. This means every 100 seconds, 20 predictions were made. Before every prediction, the system was fitted with data from the previous 24 hours .

Is it possible to make sub daily predictions with Prophet, as [2] explains. To better achieve this, we can set a parameter, `changepoint_prior_scale` [3]: changepoints are points where the trajectory of the time series changes, and this parameter changes the prior which Prophet uses to estimate the magnitude of the rate change. The bigger this value is, the more flexible we allow the trend to be. The default value is 0.05.

While collecting sensor's data, a value of 0.1 was used. After collecting the data, the predictions were made again changing the prior to see if was possible to achieve a lower MSE. The results are shown in the table 3 and the charts 5.

The RMS for the 0.1 prior was computed using data between 2023/01/16, 23:59:59 and 2023/01/17, 23:59:59 (sensor's data and predictions). In order to remake the predictions for the period written above with other priors, the data used to fit Prophet was collected between 2023/01/15, 23:59:59 and 2023/01/17, 23:58:39 (just sensor's data).[1]

[1]This is because to compute the predictions from, say, 2023/01/17, 01:10:30, to the next 100 seconds (01:11:50) the model used data collected in the previously 24 hours: from 2023/01/16, 01:10:30 to 2023/01/17, 01:10:30.

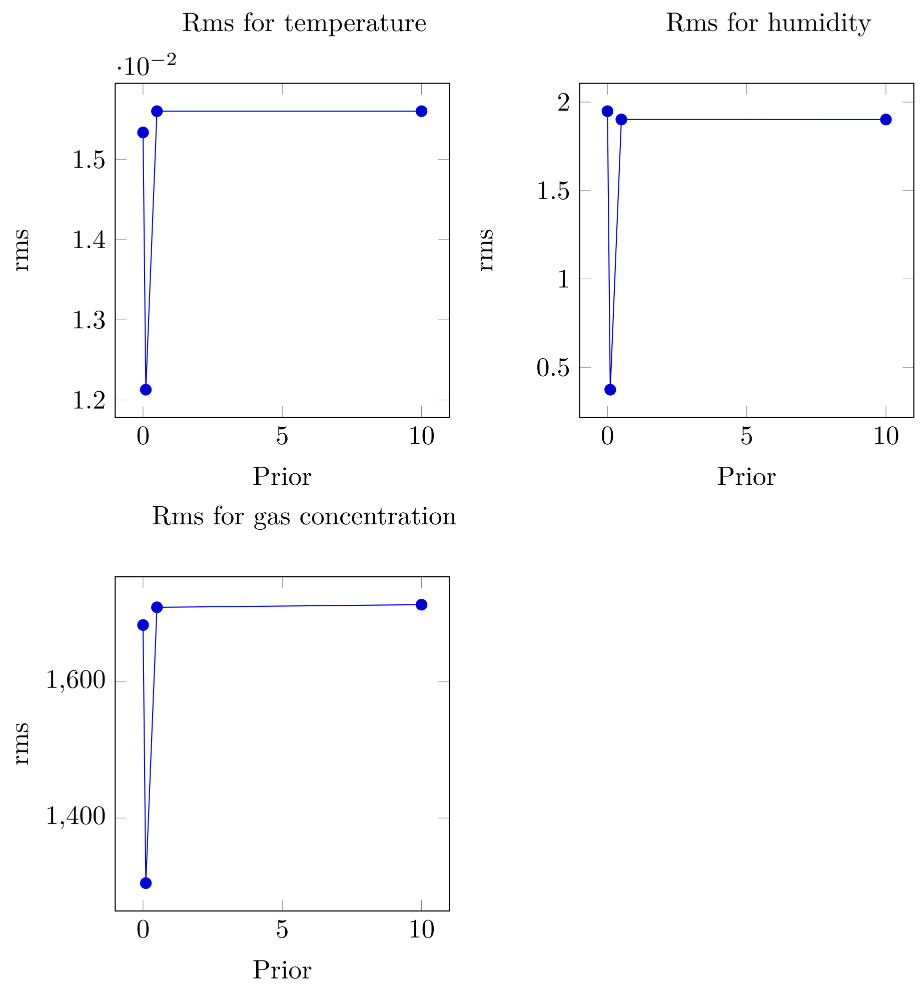


Figure 5: Rms charts

Appendix

A PPM computation

The system reads a value from the pin where the gas sensor is connected. Since the sensor is sensible to various gases, it is not possible to know which gas stimulated it. Nevertheless, it is still possible to compute the ppm of the various gas that the sensor can measure [5].

The input voltage we read from the sensor depends on the amount of gas in the air, and while it depends on the type of gas, can be approximated by the equation of a straight line:

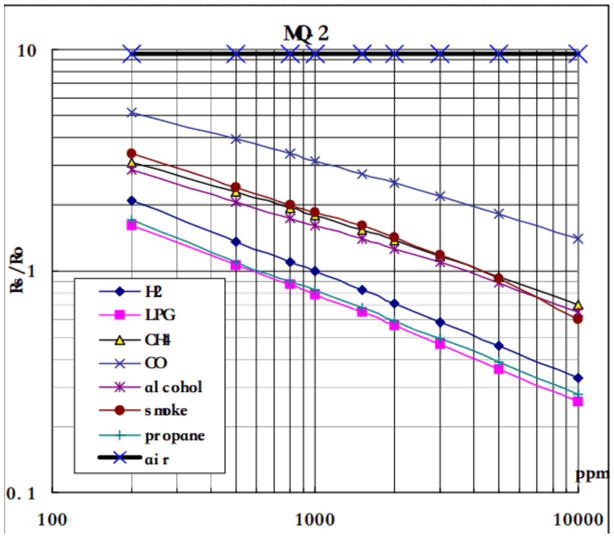


Figure 6: MQ2 gas sensitivity

$$y = mx + b$$

Since the producer of the sensor provides a log scale6, the formula becomes:

$$log(y) = m \cdot log(x) + b$$

What we want to know is on the ppm, which is on the x axis (and is x in the formula); the y axis, $RS/R0$, is the value of the resistance of the sensor in presence of the gas, over the resistance in the fresh air, without concentration of other gases.

RS is computed from the input voltage read from the sensor:

$$RS = (Vin - Vout)/Vout$$

In out case, $Vin = 3.3$, and $Vout = (3.3 * analogRead)/4096$, since we can read a value between 0 and 4095, shifted to the 1-4096 range to avoid division by 0.

Since we know that the resistance ratio in fresh air is a constant:

$$RS/R0 = 9.8$$

then $R0$ is simply:

$$R0 = RS/9.8$$

.

To compute m, the slope of the line, it suffices to know two points along the line, and then the formula is:

$$m = \frac{\log(y_1) - \log(y_0)}{\log(x_1) - \log(x_0)}$$

b is computed knowing m and another point in the line:

$$b = \log(y_3) - m \cdot \log(x_3)$$

Finally, the gas ppm concentration is obtained computing the (inverse log) of x:

$$log(x) = \frac{\log(y) - b}{m}$$
$$x = 10^{\frac{\log(y)-b}{m}}$$

References

[1] Grafana image renderer plugin, `https://grafana.com/grafana/plugins/grafana-image-renderer/`

[2] Sub-daily data, `https://facebook.github.io/prophet/docs/non-daily_data.html#sub-daily-data`

[3] Adjusting trend flexibility, `https://facebook.github.io/prophet/docs/trend_changepoints.html#adjusting-trend-flexibility+`

[4] ESP32 ADC – Read Analog Values with Arduino IDE, `https://randomnerdtutorials.com/esp32-adc-analog-read-arduino-ide/`

[5] Interfacing MQ-2 Gas Sensor with evive, `https://thestempedia.com/tutorials/interfacing-mq-2-gas-sensor-with-evive/`