UNIVERSITÀ DI PARMA

DIPARTIMENTO DI INGEGNERIA E ARCHITETTURA

CORSO DI LAUREA MAGISTRALE IN INGEGNERIA

Communication Engineering

Wireless Sensors Network for Smart Home System

IoT Project Report made by:
Emanuele Pagliari and Jodi Oxoli

ACADEMIC YEAR 2017/2018

Table of Contents

Chapter 1

Overview of the Project

1.1 Main Goal

The aim of the project is to build a **low-cost wireless sensors network (WSN)**, whose data can also be seen and used outside the network and for future development of a feedback system (A/C options, Light Switchers, Smart LED and others). In particular, the main goal is to **create a decentralized network structure which maintains an acceptable fault tolerance**. In this implementation, we decided to use four different kind of nodes with different functions and role. Regardless of the node, **all the devices communicate their data to a central server, which takes data and puts them into a database**. Data relating to the last 24 hours can also be seen on a simple webpage hosted on the server, which is not a local server. There are many kinds of data retrieved, which involve the use of different kind of sensors and hardware boards. The main limit is the total **cost of the equipment**, which should be as low as possible in order to expand the network to more rooms and floors in the future. Also, another limit is the **energy autonomy** of the system, in particular of outdoor nodes, which can't be powered from powerline. So we have to choose the best approach and communication interface to maximize the efficiency.

1.2 Architecture

The network is made by a series of nodes and has a stellar topology. They all communicate with two different access points using the **IEEE 802.11 b/g/n Wi-Fi protocol**. The two access points are also the internet gateways, and they are located inside the same LAN. That's important, because **two nodes which are connected to different WLAN need to communicate among them**. At the beginning, we evaluated to introduce many constrained nodes to use in order to collect different kind of data (smoke, Carbon Monoxide, light intensity) inside the indoor environment and let them communicate with a central listener through 802.15.1 BLE protocol advertising packets. But, cause of some complications which were arisen, we decided to temporarily skip this part. Later we will see those complications. The communication of the data collected by the sensors of the various nodes to the web server is done using the **REST approach with the classic GET and POST methods of the HTTP protocol**. There aren't sessions or cookies. The web server consists on a PHP simple web interface and a Mysql database where the data are stored. The server also uses the REST approach to obtain data from the various nodes. **The web server is located on a hosting service and not locally**, so data can be accessed anywhere. Also, all the node communicate with a local CoAP server using POST methods of the CoAP protocol with a REST approach and CON (confermable) packet trasmission. It can be seen a schematic comparison of the system in Fig. 1.1.

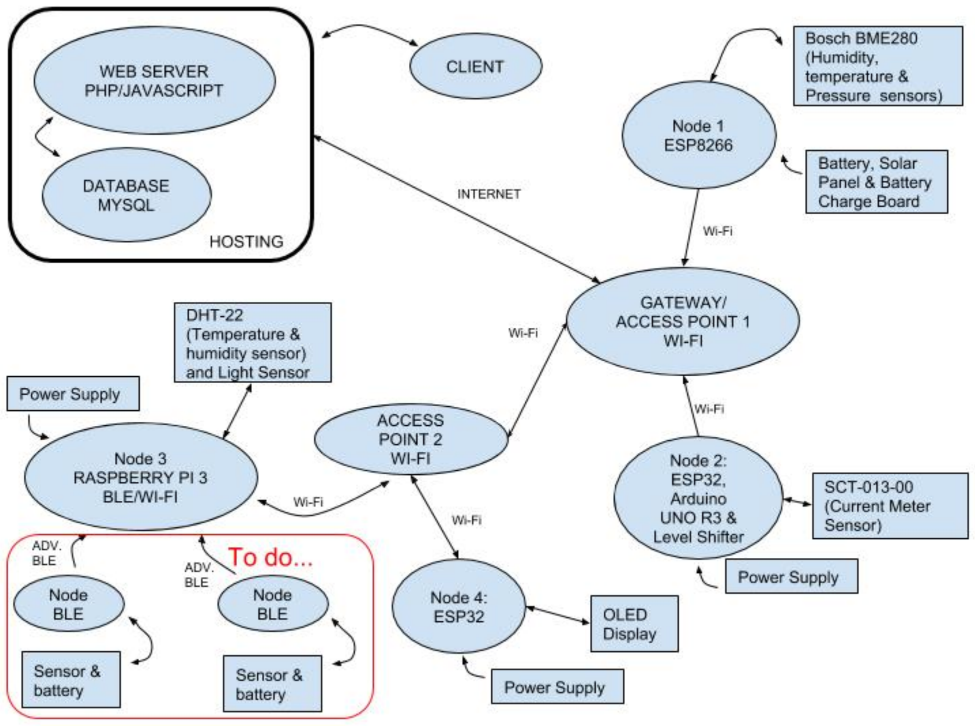
WEB SERVER PHP/JAVASCRIPT
DATABASE MYSQL
HOSTING
CLIENT
INTERNET
Bosch BME280 (Humidity, temperature & Pressure sensors)
Node 1 ESP8266
Battery, Solar Panel & Battery Charge Board
Wi-Fi
GATEWAY/ ACCESS POINT 1 WI-FI
Wi-Fi
DHT-22 (Temperature & humidity sensor) and Light Sensor
Power Supply
Node 3 RASPBERRY PI 3 BLE/WI-FI
ACCESS POINT 2 WI-FI
Wi-Fi
Wi-Fi
Node 2: ESP32, Arduino UNO R3 & Level Shifter
SCT-013-00 (Current Meter Sensor)
Power Supply
Wi-Fi
ADV. BLE
ADV. BLE
To do...
Node BLE
Node BLE
Sensor & battery
Sensor & battery
Node 4: ESP32
OLED Display
Power Supply

Figure 1.1: Overview of the system.

1.3 Protocol and Language used

Taking a look at the software part, at the application level, it can be seen that we used **two protocols** to communicate the data directly from the node to the web server and put them into the database. **We choose HTTP instead of CoAP** for this operation due to some limits of our hosting service that allows only the use of PHP scrypt. Given that the only CoAP library we found didn't work properly (at least with our use case), we decided to make this part using the classical HTTP protocol with a REST approach. In particular, the three sensor nodes use the HTTP protocol to send the data through a POST method to the right PHP scrypt, where we implemented a GET method to catch the data and add them into a MySQL Database. Instead, we used the **CoAP protocol** for intra-devices communication, so locally. Indeed, all three sensors nodes communicate their last lecture both to the web server and to a local CoAP server, the fourth node, as seen in Fig. 1.1.

The CoAP server collects all data, which are submitted to him through a CoAP POST method with a CON (confirmable) approach from the other devices. Then, it shows the data on a small OLED display. The implementation of the HTTP POST method and of the data sensing functionalities from the sensors has been implemented in language C, both on the ESP8266 and ESP32 nodes. Also, the CoAP functions on these type of nodes have been implemented in C. Instead, on the RaspBerry Pi 3 node, we decided to use **PYTHON**, both for the HTTP, data sensing and CoAP part. The web server is made in **PHP**, with some JAVASCRIPT parts for the graphs. All nodes collect data and communicate them to the web server in polling mode with a certain interval of time that varies according to the type of node, in order to maximize sleep period and reduce consumption. For example, the outside Weather Station Node collects and sends data every 5 minutes, while Current and Power Node every second. The other node, does these operations every 5 minutes. Every time they collect the data, they make two POSTs: one in HTTP to the web server, and another one in CoAP to the local CoAP server.

Chapter 2

Development

2.1 Hardware

Choosing the hardware was the **most difficult part**, which gave us many hitches. Let's see every single node, case by case.

2.1.1 Node 1 (Outdoor Weather Station)

Starting from the outside node, we decided to use an **ESP8266 board**. It is connected to a particular sensor (**Bosch BME280**) which can detect **temperature**, **humidity**, **atmospheric pressure** and then computes the **Dew Point** using the retrieved data. This node is the more constrained, since it's powered by two 3.5V 0.5A solar panels and a 1200mAh backup battery, properly connected using an IC which protects battery charge and discharge. It is possible to see the hardware of the system in Fig. 2.1.

All the hardware is **located outside** in a protective box with an appropriate shielding in order to prevent falsification of measurements and guarantee water resistant. The sensor has been located at about 2 meters of height from the soil, in order to prevent the falsification of the temperature due to the ground radiation and minimize the soil inversion effect during the winter season. Solar panels have been oriented towards the South with a tilting of

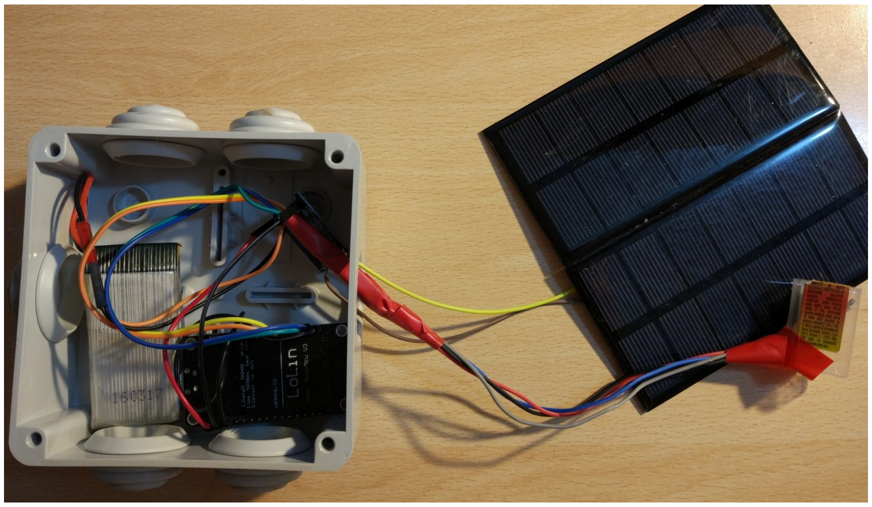

Figure 2.1: Hardware of the first node.

about 30 degrees, in order to catch the maximum radiation during the winter days. The final installation can be seen in the Fig. 2.1. Since **during the test we detect a power consumpition of about 50 mA on a period of 24 hours**, with a battery of 1200 mAh and the solar panels, the station should work for months (maybe one year) without charge manually the battery. We will see it.

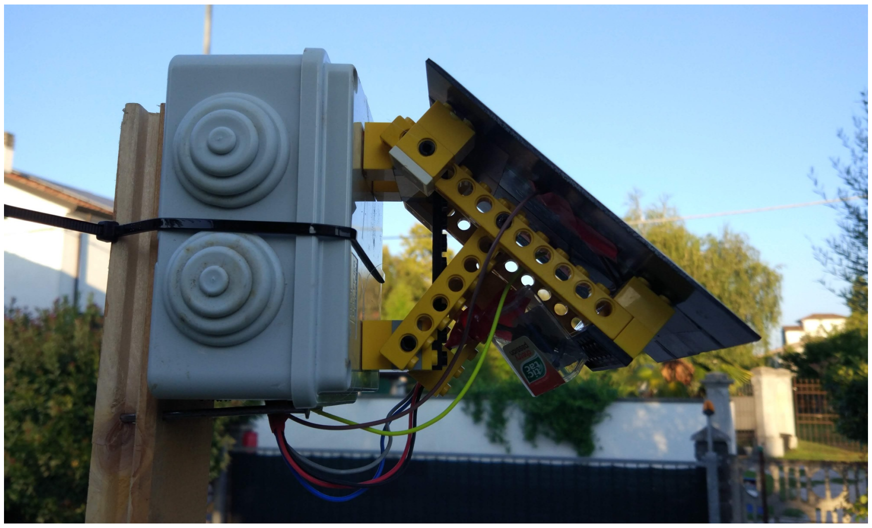

Figure 2.2: Final installation of the outdoor nodes.

2.1.2 Node 2 (Current and Power Meter)

The second network node does not consist of one board, but in **three boards**, because during the testing phase we discovered that the ESP32 can't provide enough power to the current sensor SCT-013-000, so we used an **Arduino One R3** to catch the data from the sensor and then to send it to the ESP board through a serial port. That was not so easy, because the Arduino Uno R3 works at 5 Volts, while the ESP32 board at 3.3 Volt. So, we needed to build a **voltage divider** using three resistor of 1 kOhm in order to decrease the voltage of the signal from 5 to 3.3 Volts. After the initial testing, we decided to buy a level shifter board (the CLW1070 in Fig. 2.3), in order to power both the Arduino and ESP with one power supply instead of two and provide the serial signal to the ESP32.

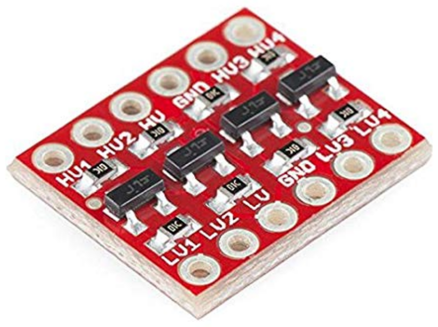

Figure 2.3: The level shifter board used.

But there is more. In order to make the current sensor (SCT-013-000) properly working, we needed to create a **small circuit which makes the signal readable from the analog input pin of the Arduino One**. We used a 100μF 16 Volts condensator and two 10 kOhm resistors. In Fig. 2.4 it can be seen how these parts are connected to the current sensor and to the ESP32.

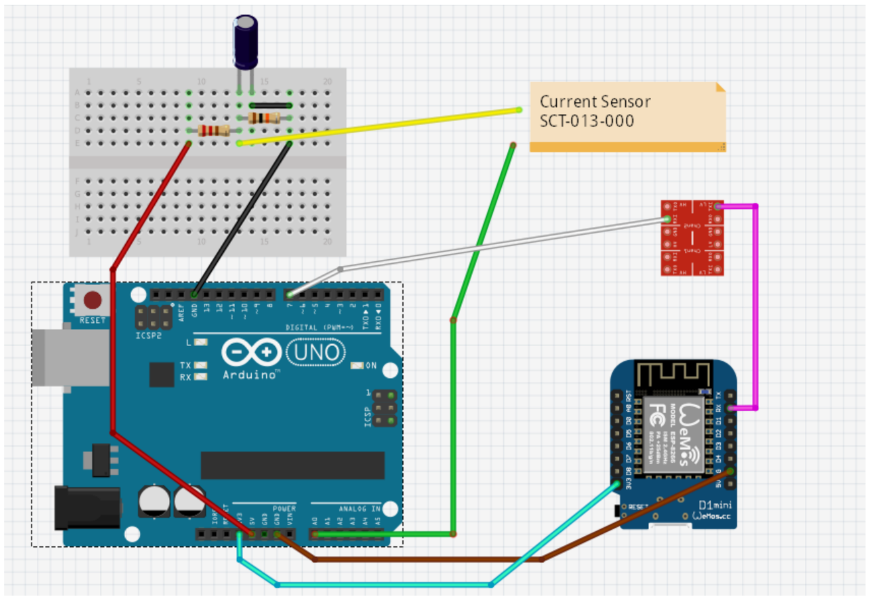

Figure 2.4: Scheme of the circuit.

After programming and wiring the boards using a solder, we put all the components inside a **case made of Lego bricks**. This is currently powered by a single 5V Power Supply Unit and monitors the consumption of all the devices connected to a power strip. In Fig. 2.5 the final construction of the node can be seen.

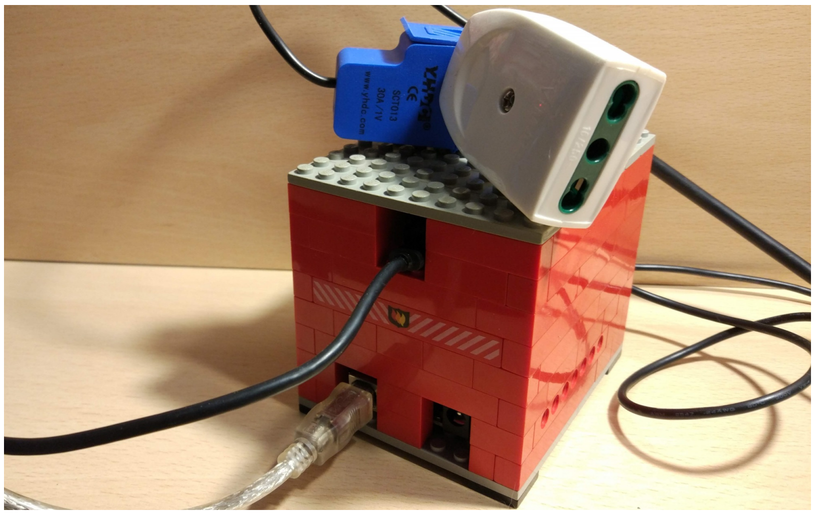

Figure 2.5: Final case of the second node.

2.1.3 Node 3 (Indoor RaspBerry Pi 3 Sensor)

Third node is represented by a **RaspBerry Pi 3** together with other two sensors: the **DHT22**, used for temperature and humidity sensing, and a **photoresistor** used as a light (ON/OFF) detector. No additional component are required this time, since we only connected the two sensors to the RaspBerry Pi 3 pin I/O shield. This node has both Wi-Fi b/g/n and BLE connectivity and is also powered by the power line, since it will also work as a local NAS. It's the less constrained node. In Fig. 2.6 a shot of the RaspBerry Pi 3 Casing and sensors wiring can be seen.

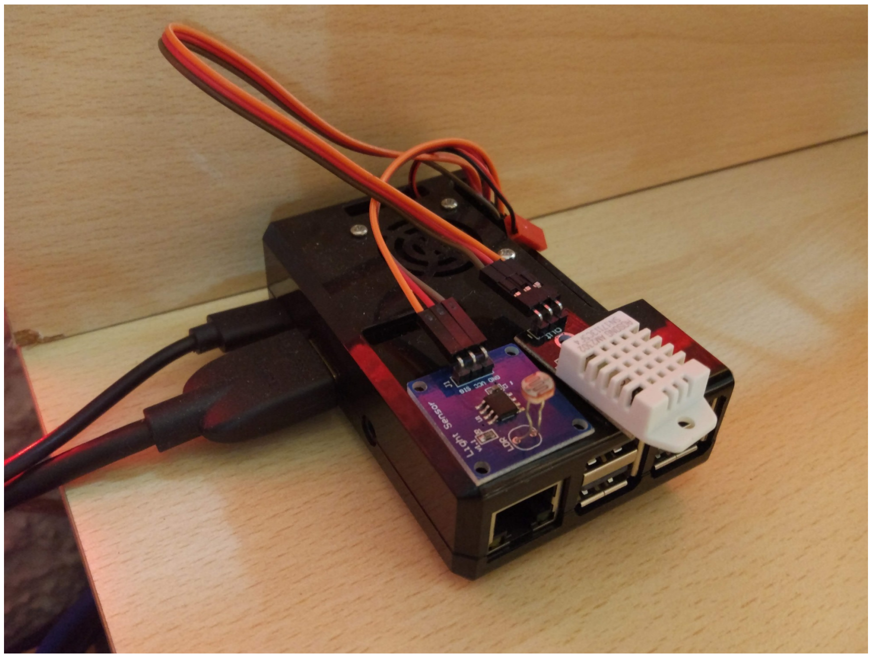

Figure 2.6: Final case of the third node.

2.1.4 Node 4 (Local CoAP Server)

Finally, fourth node is the **local CoAP Server**, which is a particular version of the ESP32: the **Heltec 32**, equipped with a small 0.96 inches **OLED Display**.

There are no other hardware component, except a single LED used for debug and future deployment (we will see it later). This node has both Wi-Fi b/g/n and BLE connectivity and is powered by a Power Supply Unit. The fourth node assembly can be seen in the Fig. 2.7.

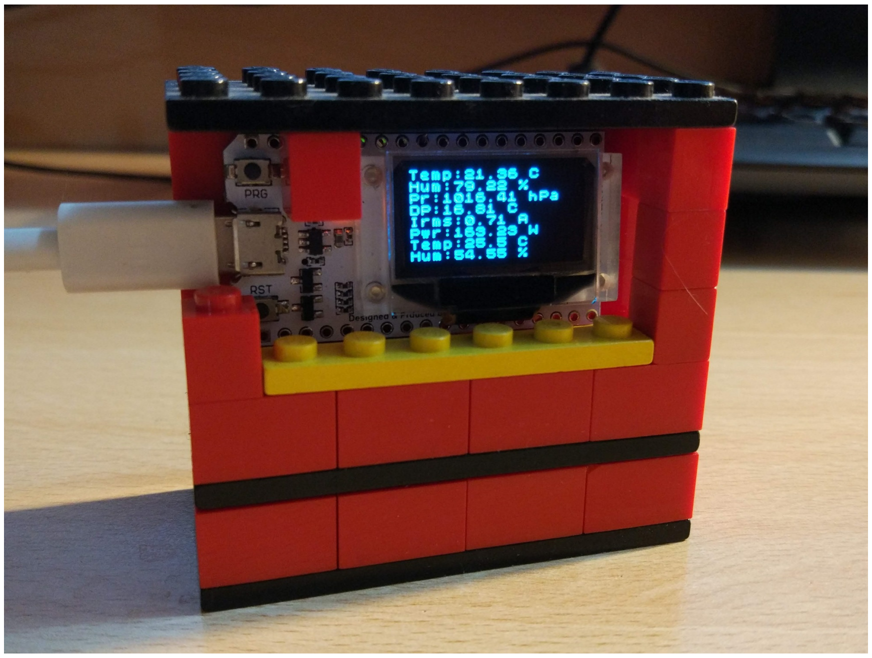
```
Temp:21.36 C
Hum:79.22 %
Pr:1016.41 hPa
DP:16.81 C
Irms:0.71 A
Pwr:163.23 W
Temp:25.5 C
Hum:54.55 %
```

Figure 2.7: Final case of the fourth node.

2.2 Software

Now we will analyze the software part and the code we deployed on all the nodes and on the web server.

2.2.1 Web Server and Database

First, we started with the **web server**. We wrote **seven PHP scrypts**: three in order to create the web pages with the collected data and the graphs, three to catch the data through the GET method of the HTTP protocol and to insert them into the database; another one with the parameter to connect to the MySQL database. The three catch scrypt are quite similar, since they all do the same function but with different variables.

The structure is the following: at the begin we collected the parameters to connect to the database, located in the connect.php file, then we prepared the MySQL statement, where we declared the query used to insert data inside the database. Also, inside the query, we used the GET method to collect data incoming from the various nodes. We created three scrypts like this, one for the Weather Station (add_data.php), one for the indoor data retrieving (add_indoor.php) and another one for the power meter node (add_current.php).

Then we obtained the three web pages which show the data. The structure is the same: at the beginning we connected to the database, then we created a small table where we put the last data retrieved from the database using the right query.

At this point, we created the **graphs** used to display the data. We decided to use the **Chart.js javascript library** to create them. The graphs are quite simpler, they show the last 24 hours lectures of all the data, while in the current and power page, we decided to show only the last 10 minutes data, since data are collected every second.

The data to show inside the graphs are retrieved using the right queries and parsed into arrays. Then, using **PHP**, we printed them using the echo function inside the dataset for the graphs. As regards security, we decided to not apply any login system right now, because we had some issues during the test, but since those are not sensitive data, there are no problems.

At the end, we decided to add an **header menu** in order to switch between the different pages. The pages also automatically refresh every 15 seconds and show the last data.

The webpage can be reached at this URL: **www.emanuelepagliari.it/iot/**.

2.2.2 Node 1 (Outdoor Weather Station)

The second network component we made is the node 1 (the outdoor weather station). We programmed the ESP8266 board using arduino IDE, in order to find all the libraries more easily. The sketch we developed uses many libraries:

- *ESP8266WiFi.h*: for the board Wi-Fi functionalities, HTTP Client and also the deep-sleep mode;
- *Adafruit_BME280.h*: in order to use the Bosch sensor BME280 to read temperature, humidity and pressure;
- *coap_client.h*: for creating the CoAP client and execute the POST methods to the different resource URI.;

First, we declared the **main variables**, such as the IP Address of the CoAP server (Node 4), the port, the server name of the web server, the Wi-Fi name and password, and all the other support variables. Then, we declared the **void function** in order to manage the server response (ACK), since we used the **CON** (confirmable) approach.

The second function is the **Wi-Fi connectivity funcion**, where we managed the Wi-Fi connection and check operation. Then, we initialized the BME280 sensor and made another function for the fast calculation of the Dew Point.In the setup method we recalled the previous function that we need (Wi-Fi, CoAP response, CoAP client, sensor init, serial and input pin declaration) and, as all things were right, we started the real part of the program.

Infact, in the loop() class, we read the values from the sensor and put them into the right variable. We recall the Dew Point fast calculation function and printed all the data to the serial port for debugging. After that, we created the **URL** to POST to the web server by adding the values to the string we declared. Then, we started the Wi-Fi client in order to execute the HTTP POST method to the server. After checking the connection and the server, we created the URI and then we sent them to the server.

At this point, we had to do the **POST method over CoAP** for the local server. But, before that, we had to convert all the variables into char, since the CoAP method we used only accepts char variables. After that, we made four POSTs over CoAP, each one to the right resource URI, specified into the method coap.post().

At the end, we sent the ESP8266 to **deep sleep** for five minutes, in order to minimize the power consumption.

2.2.3 Node 2 (Current and Power Meter)

The second node is the more complex, since we had to develop two sketches for two boards: the **Arduino One** and the **ESP32**, which communicate through a serial port. Let's see them separately.

2.2.3.1 Arduino One R3

The Arduino One R3, as we saw in the chapter 2.1.2, **manages the current sensor** (SCT-013-000) and **transmits the data through the serial port to the ESP32**. The sketch we developed uses many libraries:

- *SoftwareSerial.h*: for the board serial communication;
- *Wire.h*: used to set the input pin as an Analog input;
- *EmonLib.h*: in order to elaborate the input and determinate the current value readed from the SCT sensor.

As usual, we declared the used **variables**, the **serial port pins** (RX e TX) and other support variables. Then, in the setup() function, we **initialized the serial port**, the serial monitor and the sensor library initialization, with a properly tested calibration value (33).

At this point, we simply **read the data** (irms) from the sensor using the method of the library and printed it to the serial monitor. Notice that in order to determine the power value, we just multiply the Irms value for the voltage, which is around 230 Volt. After that, we **sent the Irms value to the EPS32** through the serial port.

We just sent the Irms value instead of both Irms and power value for a simple reason. Indeed, at the beginning we sent the two values separately, on two different println() methods, but we noticed a phase shift in the reading of the values which leds to an exchange of values, falsifying the data. So **we decided to transmit only Irms and then calculate the Power** (Irms*Voltage) on the ESP32 before the POST method. At the end, we inserted a delay of one second before repeating the operation.

2.2.3.2 ESP32

Then we have the second board, the ESP32. Let's see the used libraries:

- *WiFi.h*: for the board Wi-Fi functionalities;
- *coap_client.h*: used to create the CoAP client and execute the POST methods to the different resource URI;
- *PubSubClient.h*: used to declare the HTTP Client.

We declared the **used variables**, the IP Address of the CoAP server, the port, the server name of the web server and also the Wi-Fi name and password, as seen in the first node. Then, as seen before, we declared the **void function** to manage the server response (ACK), since we used the CON (confirmable) approach. After that, we created the **Wi-Fi**

connectivity function, and then the **setup() function**, where we initialized the serial monitor, the CoAP client and the Wi-Fi connectivity.

In the loop() function, we checked if the serial port retrieves some data, if true we read them and put it into a variable through the parseFloat() method. Then, we **calculated the power** and **printed both the Irms and Power to the serial Monitor**. Now, as seen in the first node, we started the Wi-Fi client in order to execute the HTTP **POST method** to the server. After checking the connection and the server, we created the URIs and then we sent them to the server. After that, we **converted the variables into char and made the POST method over CoAP** for the local server. We introduced a small **delay** of 800 ms in order to let the transmission finish.

2.2.4 Node 3 (RaspBerry Pi 3)

The Raspberry Pi 3 is the only node where we use Python for making the script. We used many libraries, but the main ones are:

- *Adafruit_DHT*: in order to use the DHT22 sensor and collect the data;
- *urllib2*: for the HTTP POST and other function;
- *Coapthon*: for the CoAP POST method to the local server.

In the Python script, we first declared the main variables, such as the delay between a data reading and another one, and the GPIO setup. Then we made two functions: the first one for **collecting data from the DHT22 sensor** and the second one for **collecting data from the light resistor**. After that, we created the **main()**, where we first created an HTTP opener in order to make the POST method. This was unnecessary, but since we tested it with an experimental authentication method, we needd to do this in order to correctly POST the data to the server.

After that, we made a while cycle where we continuously read the data from through the recall of the functions and then we used a **POST method for sending them to the web server**. We also printed the values in the terminal. At this point, we made to POST method over CoAP to the two URIs in order to send the values to the CoAP server. At the end, we introduced a **delay** of 300 seconds - five minutes - before repeating the operations.

2.2.4.1 BLE Part (Delayed)

At the beginning, our main idea for this node was to **collect data incoming from small Bluetooth Low Energy nodes trough Advertising packets** and send them to the servers. We created a pair of nodes using the ESP32 which transmits the data readed from the sensor as Advertising payload in the BLE ADV packets. We could verify that they work right using the BLE Sniffer.

The main problem here was to retrieve the BLE ADV payload on the Raspberry Pi 3. We used the **hcitool lescan method** to run a BLE scan and also we used the **hcidump –raw** to read the packet payload and data, but we could not find a solution using PIPEs to catch those data. After many and many days of tries, we decided to temporarily skip this part.

2.2.5 Node 4 (Heltec ESP32)

Finally we have the **CoAP server node**, built using the **Heltec ESP32 board**, which has a tiny OLED display. As on the other ESP boards, we used many libraries, such as:

- *WiFi.h*: in order to enable the Wi-Fi connectivity;
- *WiFiUdp.h*: for using UDP instead of TCP;
- *coap.h*: for the CoAP functionalities;
- *U8x8lib.h*: for using the onboard display.

Since this is the CoAP server, we needed to create all the **resource URI callback methods**. We created many resources: light ON/OFF (for future development, we will see it later), temperature, dew point, humidity, etc. In each callback method, first we copied the payload of the packet into some support variables, then we used the CoAP method **sendResponse** to send the ACK packet to the clients, since we are using a **CON** (Confermable) approach. Then we printed the values both on the serial monitor and on the display.

After those declaration, we built the setup() with the classical Wi-Fi connection function recall, display initialization, serial monitor, the coap server functionalities and so on. For all the resources, we had to declare the URI during the initialization, in order to make the CoAP server work.

Chapter 3

Conclusions and future developments

All the devices work properly, and the collected data can be seen at this url:
www.emanuelepagliari.it/iot/.
Right now the webpage is quite basic, but as future development, we would like to make a more complex web page, with the possibility to show a certain period of data (a particular day for example), and also to visualize the maximum and minimum value of the last 24 hours. We decided to skip all this stuff right now due to lack of time.

Two words about the accuracy of the measurements. The outdoor **temperature, pressure and humidity sensor is quite reliable** since there is a really short gap in the value of the humidity and pressure, which is also influenced by the height. The temperature accuracy is also good, but it's still influenced by the sun and soil radiation, **so it could be imprecise sometimes, especially during sunny days, when the value could became 5-6 degrees higher than the real value**. We're still working on this problem, which already got partially fixed, since at the beginning we reached 15-20 degree higher temperature. The solution could be a 3D-printed shield (something like this one swhoeed in Fig. 3.1) made of stacked perforated cones, in order to create a passive air flow from the bottom to the top. In the worst case, we would locate the sensor in the shadow and then move the solar panel away.

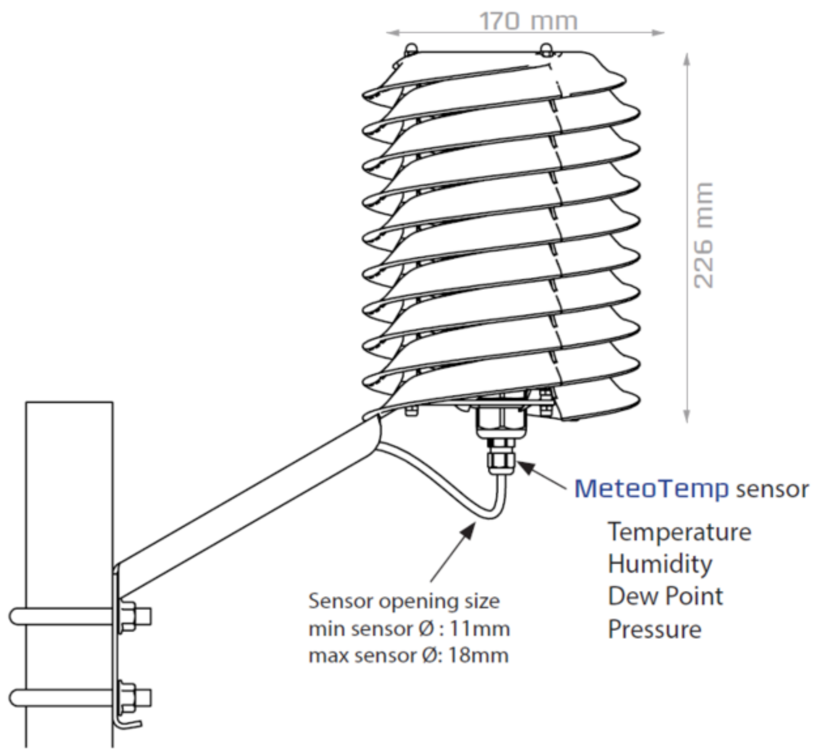

Figure 3.1: Sun radiation shielding system.

The less reliable sensor is the power meter sensor. At low load (less than 5-10 Watt), it can't detect the real power consumption, while applying a greater load (more than 15-20 Watt), the values are quite good, with a gap of just 2-3 Watt. This is not a great problem, since the environment where the power meter is located will rarely have to work with low load. So the collected data can be assumed as quite reliable.

As you may see in the CoAP server, there is an **URI dedicated to the light switch**. This part, which already works, allows the Raspberry Pi 3 to turn on and off a LED light when it detects through the light sensor a small environment brightness. It's just an experiment for a future development which can be used to control the security light or many others.

Another future development could be the expansion of the system with a feedback system that uses the data read by the sensor to make some **actions**, such as turn on or off a fan when a certain temperature is reached, or close/open blinds using an actuator, and many others. This is just the beginning. It is also possible to **add many and many other sensors** to the nodes, like a moisture sensor and rain sensor for the weather station and then use those data for deciding when irrigate a garden.