ENSEIRB-MATMECA
Filière électronique

Projet S9

Rapport

Infrastructure des objets connectés Wipy

Asmae Aazouzou
Youssef Khouribchy
Abdelaziz Arroyo Ben El Gada
Dina Benamar

Encadrant : Patrice Kadionik

Infrastructure des objets connectés Wipy

Asmae Aazouzou
Youssef Khouribchy
Abdelaziz Arroyo Ben El Gada
Dina Benamar

28 janvier 2022

Table des matières

1 Introduction

1.1 Contexte et objectifs du projet

Dans le cadre du projet avancé du semestre 9, nous étions amenés à travailler sur l'infrastructure des objets connectés en nous intéressant à l'application des systèmes embarqués pour l'IoT.

L'Internet des Objets (IoT) a connu une croissance importante durant ces dernières années, et a d'ores et déjà commencé à changer notre interaction avec le monde en facilitant notre vie quotidienne. Il est sans doute que les systèmes embarqués jouent un rôle primordial dans la fondation de l'écosystème des objets connectés. C'est la raison pour laquelle comprendre le fonctionnement de ces dispositifs et les mettre en oeuvre est un véritable atout à notre formation.

L'objectif principal de notre projet était de concevoir et de programmer des objets connectés à l'aide des cartes électroniques WiPy, permettant, d'une part, de collecter des données et de les transmettre sur un cloud, et d'autre part de les recevoir et de les afficher sur l'interface Grafana.

1.2 Cahier des charges

Durant le cours de ce projet, nous utiliserons deux cartes électroniques *WiPy* afin de récupérer les données nous intéressants, à l'aide des capteurs intégrés dans ces cartes : la température, l'humidité et la pression. Une fois collectées, ces données seront ensuite envoyées au service cloud *Adafruit IO* en utilisant le protocole MQTT.

D'autre part, la connexion établie entre ces objets et Internet sera fournie par un point d'accès Wi-Fi que nous concevrons en utilisant la carte électronique *Raspberry Pi* et le logiciel routeur sans fil *RaspAP*.

Afin de récupérer les données transmises depuis Adafuit IO nous utiliserons *Telegraf*, puis nous les stockerons à l'aide d'*InfluxDB* dans une base de données. Finalement, celles-ci seront affichées sur l'interface Grafana en temps réel.

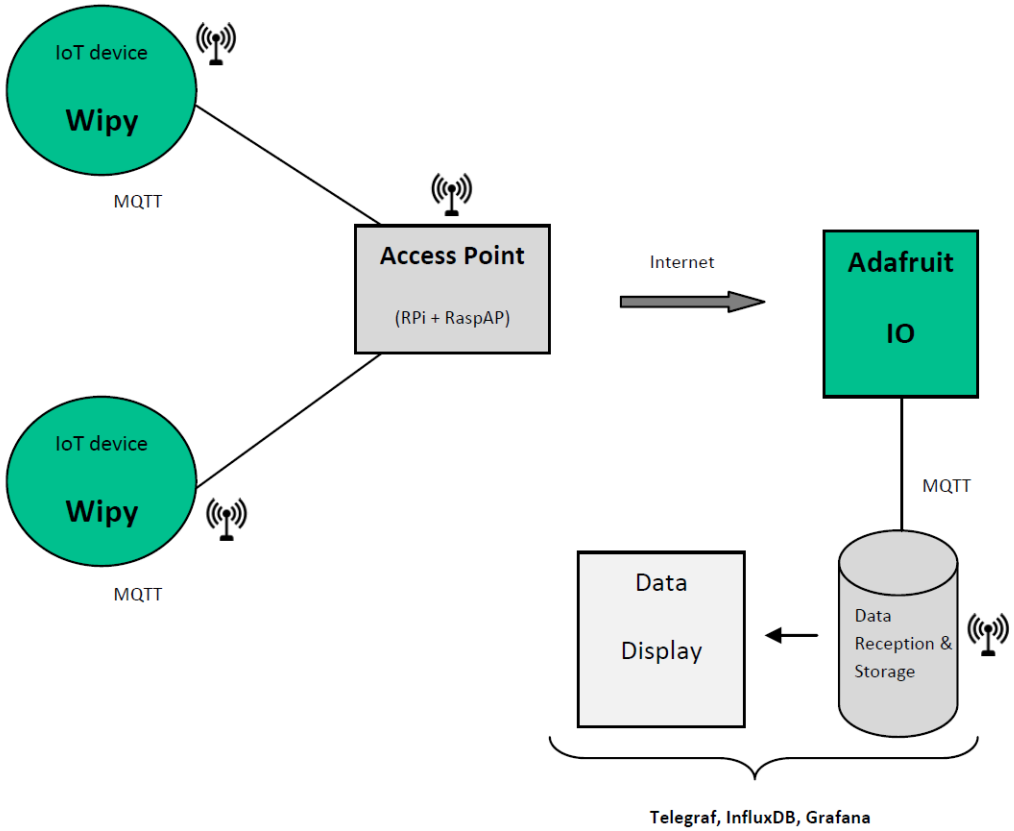

Figure 1 – Schéma illustrant l'infrastructure de la communication des objets connectés Wipy

2 Récupération et transmission des données Wipy

2.1 Carte Wipy

La carte électronique utilisée dans notre projet est **WiPy 3.0**, c'est une carte de développement open source WiFi et Bluetooth (compatible BLE) miniature basée sur le langage Python 3. Elle fonctionne en autonomie en exécutant un programme préalablement stocké dans sa mémoire interne ou par WiFi en envoyant directement les lignes de commande en Python.

La carte WiPy 3.0 dispose de 24 broches d'entrées/sorties (I2C, UART, PWM, etc) permettant la réalisation de projets connectés avec différents capteurs et modules. (Figure 2)

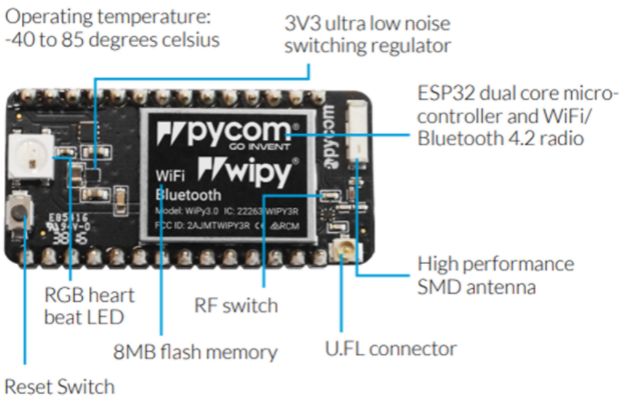

Figure 2 – Carte WiPy 3.0

Cette carte utilise le dernier chipset à double processeur Espressif ESP32, aidant à se connecter rapidement au WiFi, au Bluetooth Low Energy (BLE) ainsi que les technologies de communication IoT à longue portée LoRa et Sigfox. Dans notre cas, nous allons utiliser le RGB LED de la carte pour vérifier la connexion WiFi ; lorsqu'on est connecté au WiFi, la LED s'allume en vert.

Puisque notre projet se base sur l'envoi des données des capteurs, nous utilisons la carte d'extension **Pysense** développée par Pycom. Elle est composée de cinq capteurs : accéléromètre 3 axes (LIS2HH12), luminosité (LTR329ALS01), pression barométrique (MPL3115A2), température et humidité (SI7006A20) comme le montre la figure suivante :

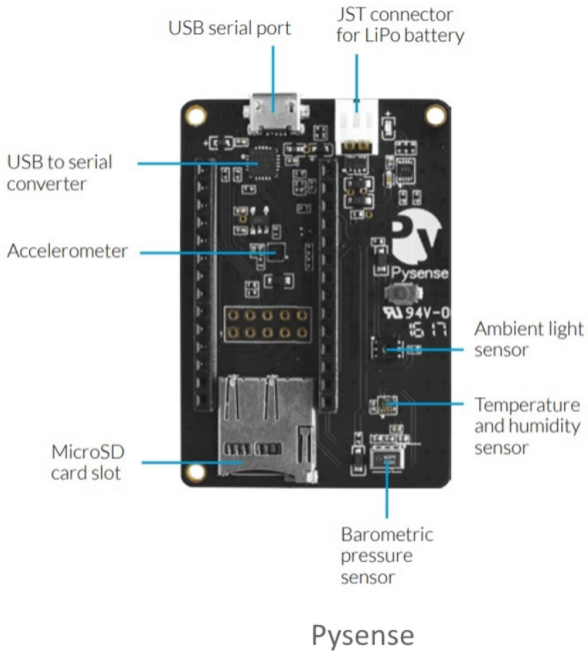

Figure 3 – Carte d'extension Pysense

a) Connexion au Wipy :

Nous utilisons un éditeur de texte qui prend en charge la communication avec le Wipy.

1. Il faut commencer par installer l'outil Atom (https ://atom.io/).
2. Dans Atom, il faut installer le module (package) permettant d'interagir avec le Wipy : **pymakr**. Ce dernier permet d'intégrer une console REPL dans l'éditeur Atom en synchronisant ainsi les fichiers entre l'ordinateur et le wipy.
3. Il faut indiquer au module pymakr l'adresse du port de communication. Il est possible d'obtenir cette information en cliquant sur le menu More (en bas à droite de la fenêtre d'Atom), puis en choisissant la commande *Get serial ports*.

Une fois ces étapes réalisées, l'éditeur Atom est opérationnel : cela signifie que l'on peut écrire des programmes en micropython et les transférer dans le Wipy.

b) Séquence de boot :

Lorsque Micropython démarre, 2 fichiers sont par défaut lus successivement :
— **Boot.py** est prévu pour contenir les fichiers qui sont censés être utilisés systématiquement.
— **Main.py** contient le code principal qui doit être exécuté au lancement.

2.2 Récupération des données

Nous souhaitons, dans un premier temps, collecter les données issues des mesures effectuées par les capteurs de la carte WiPy. Il s'agit principalement des mesures de température, d'humidité et de pression.

Pour ce, nous utilisons la bibliothèque *Pysense*, nous copions ensuite les fichiers correspondant aux capteurs nous intéressants depuis le dossier *Pysense*. Ces fichiers portent le nom des capteurs et accèdent directement à la partie de la carte qui leur correspond :
- SI7006A20 : Capteur d'humidité et de température.
- MPL3115A2 : Capteur de pression et d'altitude.

Afin de récupérer ces informations, nous définissons la fonction *collect_ sensors()* qui retourne les valeurs ainsi mesurées.

```
def collect_sensors():
    py = Pysense()
    press = MPL3115A2(py,mode=PRESSURE) # Returns pressure in Pa
    dht = SI7006A20(py) # Returns humidity and temperature
    return press.pressure(), dht.humidity(), dht.temperature();
```

2.3 Transmission des données

Pour transmettre les données et les afficher sur l'environnement WEB Adafruit, il nous faut une connexion WiFi et un protocole de communication.

— **Connexion WiFi** : la connexion Wifi est établie grâce au module ESP32 du Wipy. Dans le code micropython il faut définir le nom, mot de passe du réseau et le mode de fonctionnement **station (STA)**, où chaque module Wipy se connecte à un point d'accès via une liaison sans fil.

```
# Wireless network
WIFI_SSID = "raspi-webgui"
WIFI_PASS = "1234567890" # Password of the PA module

wlan = WLAN(mode=WLAN.STA)
wlan.connect(WIFI_SSID, auth=(WLAN.WPA2, WIFI_PASS), timeout=5000)

while not wlan.isconnected():    # Code waits here until WiFi connects
    machine.idle()

print("Connected to Wifi")
```

— **Le protocole MQTT** : MQTT est l'abréviation de "Message Queing Telemetry Transport". Le protocole MQTT fonctionne comme un service de messagerie push avec un modèle éditeur/abonné (pub-sub). Dans ce type de communication, les clients se connectent à un serveur central appelé broker (courrier). Afin de filtrer les messages qui sont envoyés à chaque client, les messages sont classés dans des sujets organisés de manière hiérarchique. Un client peut publier un message dans un certain sujet (topic). D'autres clients peuvent s'abonner à ce sujet, et le broker leur enverra les messages souscrits. (Figure 4)

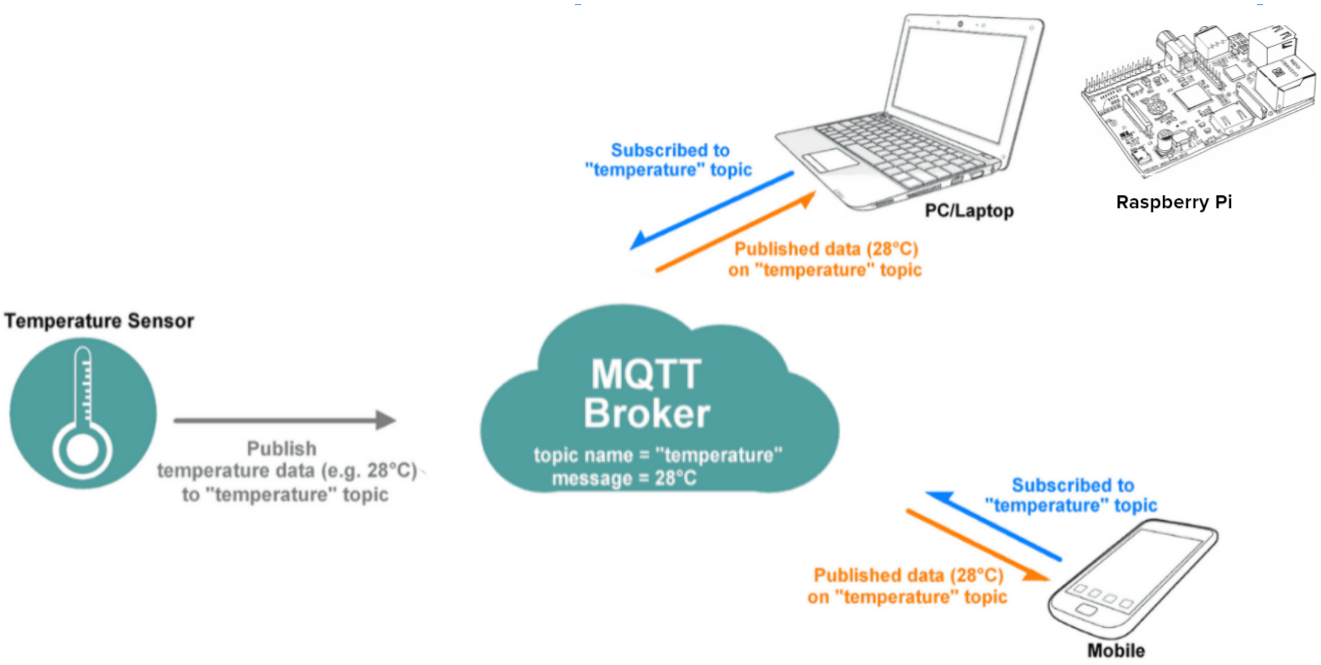

Figure 4 – Protocle MQTT

Dans notre projet, les "publishers" sont les deux modules Wipy utilisés, le "subscriber" est représenté par la carte Raspberry Pi, et le broker c'est le broker MQTT d'adafruit.

Nous nous connectons ensuite à Adafruit en utilisant le protocole MQTT. (La configuration d'Adafruit IO est détaillée dans la partie suivante) :

```
# Use the MQTT protocol to connect to Adafruit IO
client = MQTTClient(AIO_CLIENT_ID, AIO_SERVER, AIO_PORT, AIO_USER, AIO_KEY)
#client.settimeout = settimeout
client.connect() #connects to Adafruit IO
```

Par la suite, nous définissons la fonction *send_ data* qui va publier les données dans les *topics* créés sur Adafruit. Cette fonction appelle la fonction précédemment définie *collect_ senosrs*, pour récupérer les trois valeurs de température, pression et humidité, puis elle les envoie au broker d'adafruit via la fonction *client.publish* :

```
def send_data():
    global last_data_sent_ticks
    global DATA_INTERVAL

    if ((time.ticks_ms() - last_data_sent_ticks) < DATA_INTERVAL):
        return; # Too soon since last one sent.

    pressure, humidity, temperature = collect_sensors()

    try:
        client.publish(topic=AIO_TEMPERATURE_FEED, msg=str(temperature))
        print("Temperature sent")
        client.publish(topic=AIO_PRESSURE_FEED, msg=str(pressure))
        print("Pressure sent")
        client.publish(topic=AIO_HUMIDITY_FEED, msg=str(humidity))
        print("Humidity sent")
        print("DONE")
    except Exception as e:
        print("FAILED")
    finally:
        last_data_sent_ticks = time.ticks_ms()
```

2.4 Tests et résultats

2.4.1 Adafruit IO

Adafruit IO est un environnement WEB, *cloud service*, qui permet de collecter, stocker et visualiser des données. L'utilisation d'Adafruit IO comme *broker* MQTT est très répandue dans l'Internet des Objets. En plus de sa facilité d'utilisation, ce *cloud service* présente un moyen sécurisé d'accéder aux données provenant des objets connectés directement sur le Web et de les récupérer.

Après avoir créé un compte Adafuit (username : Project_Wipy), nous avons créé trois différents *feed* pour chaque carte WiPy correspondant aux trois types de mesure : température, humidité, pression. Ces feeds, *fils de données*, sont par défaut privés et accessibles à travers une clé.

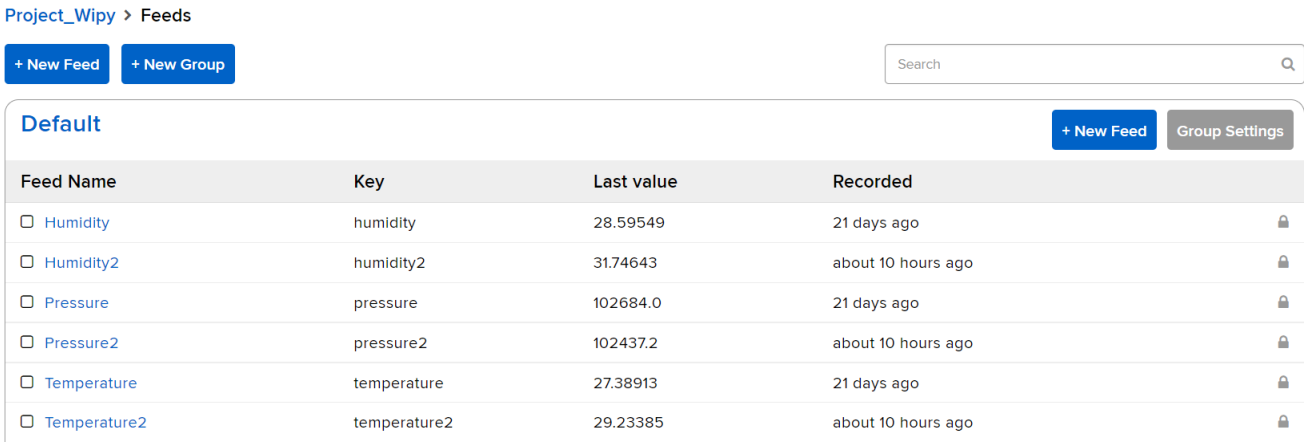

Figure 5 – Adafruit IO : création de *feeds*

Comme nous avons vu précédemment, la transmission des données s'effectue en utilisant le protocole MQTT. Celui-ci repose sur la publication et l'abonnement à un sujet spécifique (*publish-subscribe*). Dans notre cas, les *feeds* créés sont considérés comme des sujets (*topics*). Les données mesurées grâce aux capteurs sont les messages que nous "publierons" sur les différents *topics*.

Par la suite, nous précisions la clé des différents *feeds* dans notre code afin d'y publier nos messages. Chaque sujet est précisé comme suit : *Project_ Wipy/feeds/[type_ of_ measure]*.

```
# Adafruit IO (AIO) configuration
AIO_SERVER = "io.adafruit.com"
AIO_PORT = 1883
AIO_USER = "Project_Wipy"
AIO_KEY = "aio_hMWt73y8zhLsdawexevOpe7OSqu3"
AIO_CLIENT_ID = ubinascii.hexlify(machine.unique_id())  # Can be anything
AIO_TEMPERATURE_FEED = "Project_Wipy/feeds/temperature"
AIO_HUMIDITY_FEED = "Project_Wipy/feeds/humidity"
AIO_PRESSURE_FEED = "Project_Wipy/feeds/pressure"
```

2.4.2 Résultats d'affichage

Après avoir configuré l'accès à Adafruit IO, nous avons rassemblé les parties du code et testé notre programme.

Dans un premier temps, lorsque la carte WiPy est alimentée, la LED de celle-ci émet de la lumière rouge, une fois la connexion WiFi est établie, la LED devient verte pour l'indiquer. Afin d'obtenir ce comportement, nous faisons appel à la fonction *pycom.rgbled(@color)* (avec @color valant $0xff0000$ pour rouge et $0x00ff00$ pour vert).

Ainsi, le programme commence par tester la disponibilité de la connexion WiFi. Tant que celle-ci est établie, le programme entre dans une boucle infinie et fait appel à la fonction *send_ data()*. Chaque $15s$ pour la carte WiPy1 (et $20s$ pour Wipy2), les mesures de température, d'humidité et de pression sont récupérées puis envoyées en MQTT sur le sujet y correspondant.

Nous avons, par la suite, créé trois *dashboards* pour chaque carte sur Adafruit afin de visualiser les données reçues sur le cloud.

Les tests effectués nous ont permis de valider le bon fonctionnement de notre programme et le succès de cette première partie. En nous connectant sur Adafruit IO, nous pouvons voir les données se mettre à jour en temps réel et s'afficher sur les différents dashboards.

Les figures suivantes représentent le résultat d'affichage des données reçues sur Adafruit IO.

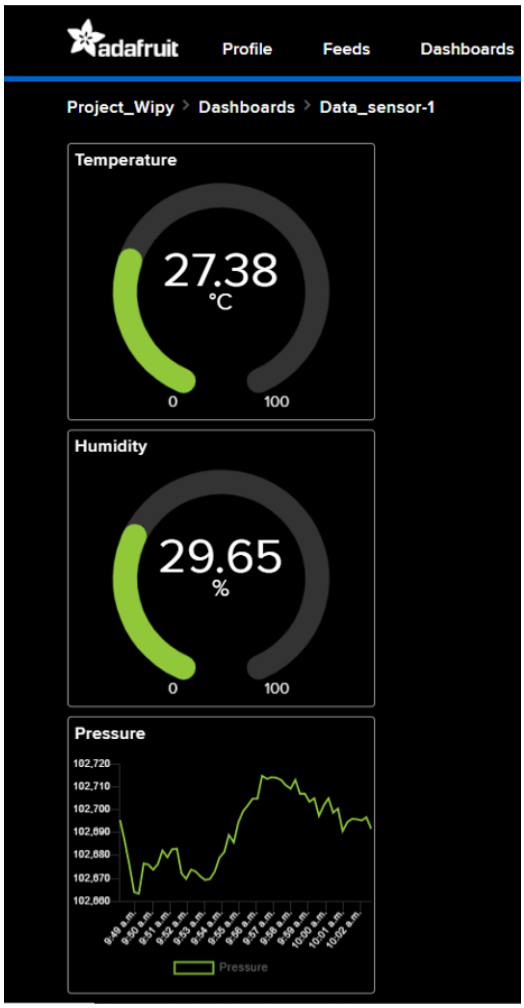

Figure 6 – Résultat d'affichage des données sur Adafruit IO pour la carte Wipy 1

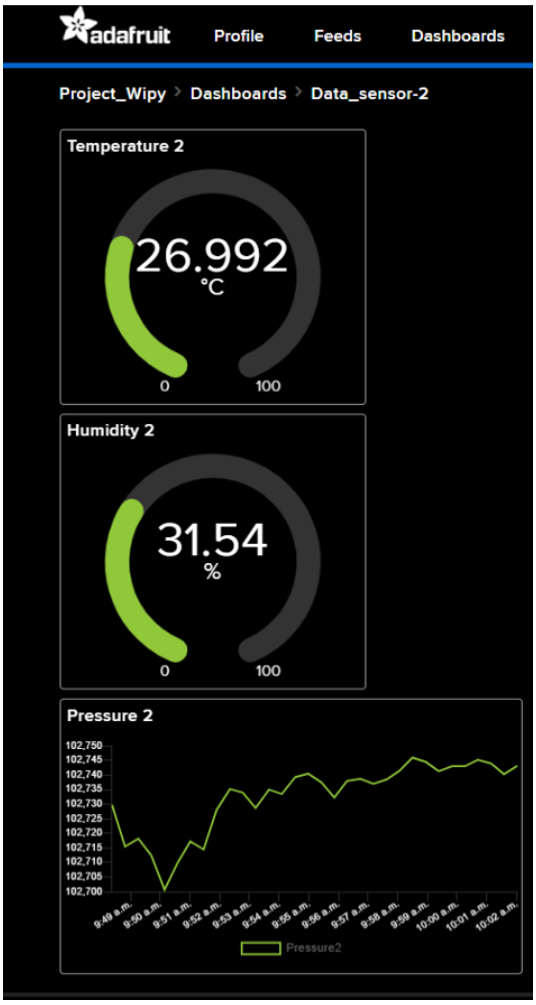

Figure 7 – Résultat d'affichage des données sur Adafruit IO pour la carte Wipy 2

3 Point d'accès RaspAP

Une fois les 3 mesures réalisées, les modules Wipy doivent se charger de les envoyer au service Cloud *Adafruit IO*. L'envoi des données se fera par le biais d'une connexion Internet. C'est ainsi, que nous allons définir notre propre point d'accés Wi-Fi. Pour ce faire, nous allons utiliser une carte Raspberry Pi 3b+ et RaspAp qui signifie tout simplement Raspberry Pi Acces Point (Point d'accès Raspberry Pi). Il est un logiciel de routeur sans fil riche en fonctionnalités qui fonctionne sur de nombreux périphériques populaires basés sur Debian, notamment la Raspberry Pi.

3.1 Configuration sur la carte

Nous avons choisi de définir les deux réseaux, dont nous avons besoin, d'une manière statique au lieu d'utiliser le protocole *DHCP* (Dynamic Host Configuration Protocol). Pour ce faire, il faut éditer les fichiers de configuration de la carte. Avec Raspberry Pi Os (anciennement Raspbian Jessie), le système n'obéit plus au classique/**etc/network/interfaces**. C'est à dire qu'il est inutile de s'escrimer à modifier les paramétrages de ce fichier pour basculer vers une adresse statique au lieu du DHCP par défaut. En effet sur Raspbian, le réseau est paramétré par le service DHCPCD. C'est donc à son niveau que nous allons agir, plus précisément sur le fichier **/etc/dhcpcd.conf**.

```
# RaspAP wlan0 configuration
interface wlan0
static ip_address=10.3.141.1/24
static routers=10.3.141.1
static domain_name_server=10.210.18.138
gateway


# eth0
interface eth0
static ip_address=192.168.4.31/24
static routers=192.168.4.253
static domain_name_server=10.210.18.138
```

Figure 8 – Les deux interfaces réseaux eth0 et wlan0

D'après la figure ci-dessus, nous possédons deux interfaces réseaux :
- L'interface **eth0**, c'est le réseau Ethernet qui permet à notre carte d'accéder à Internet.
- L'interface **wlan0**, c'est notre réseau Wifi qui sera utilisé par nos clients, les modules Wipy. Son adresse Ip est la suivante 10.3.141.1.

Un simple **ifconfig** nous permet de valider et de vérifier la configuration que nous avons réalisée.

3.2 Configuration Raspap

Il reste maintenant à configurer *RaspAp*. D'abord, il faut l'installer sur la carte Rpi en utilisant la commande : **wget -q https ://git.io/voEUQ -O /tmp/raspap bash /tmp/raspap** puis redémarrer la carte après.
Pour paramétrer RaspAP, nous devons accéder à l'interface d'administration via une page web, en se connectant sur l'adresse de notre point d'accès, c'est à dire 10.3.141.1. Cela peut être fait soit en local, c'est à dire sur la Rpi elle-même, soit depuis une autre machine connectée à notre point d'accès. Avant de commencer, il va falloir s'authentifier en indiquant le nom d'utilisateur et le mot

de passe. Par défaut, ils sont **admin** et **secret**. Ces paramètres peuvent bien sûr être modifiés si nous le souhaitions.

Une fois authentifié, sur l'onglet **Serveur DHCP** nous allons réaliser les modifications suivantes :

— Nous allons commencer par choisir l'interface réseau. Dans notre cas, c'est **wlan0**. Nous devrons fixer son adresse IP comme étant une adresse statique, pour ne pas créer des problèmes avec la configuration que nous avons déjà réalisée précédemment. Ensuite, il faut mentionner l'adresse Ip, le masque et le default gateway (L'adresse Ip de la passerelle par défaut). Ces informations doivent correspondre aux paramètres que nous avons déjà définis dans le fichier **/etc/dhcpcd.conf** (cf paragraphe précédent).

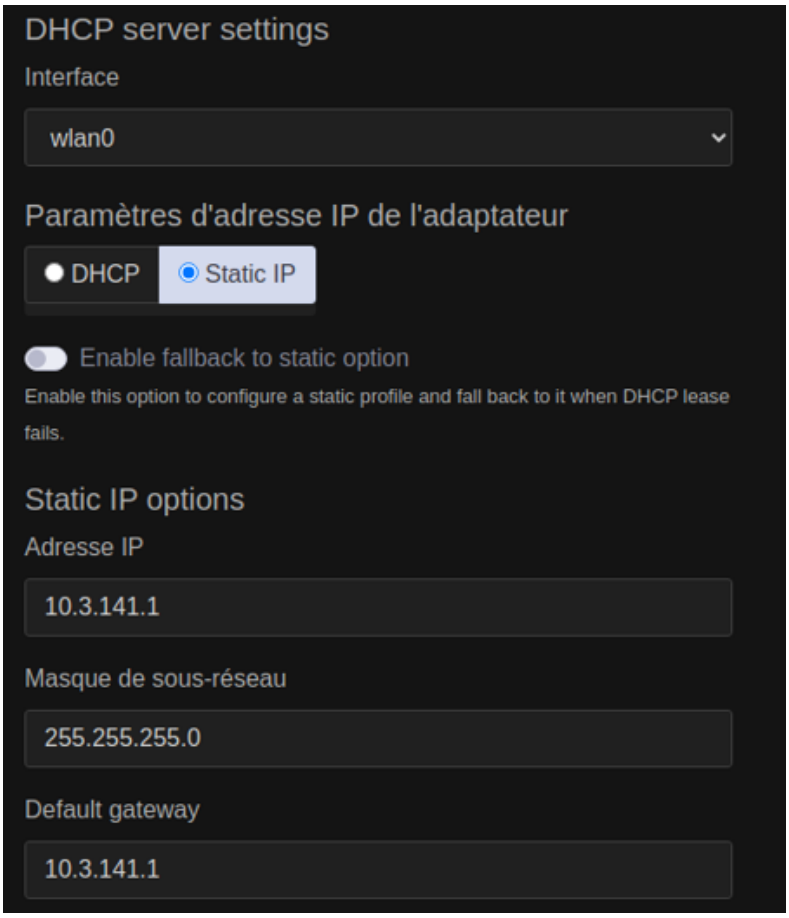

Figure 9 – Configuration de l'interface du réseau Wifi wlan0

— La deuxième et dernière étape consiste à définir la plage des adresses Ip consacrée aux clients/utilisateurs de notre réseau Wifi. Pour les adresses IP du départ et de la fin, nous avons choisi 10.3.141.50 et 10.3.141.255 respectivement. Donc, un maximum de 205 appareils peuvent se connecter simultanément à notre Wifi. A chaque nouvelle connexion, chaque client est attribué une adresse Ip qui appartient à la plage mentionnée précédemment. Cette attribution se fait suivant le protocole DHCP.

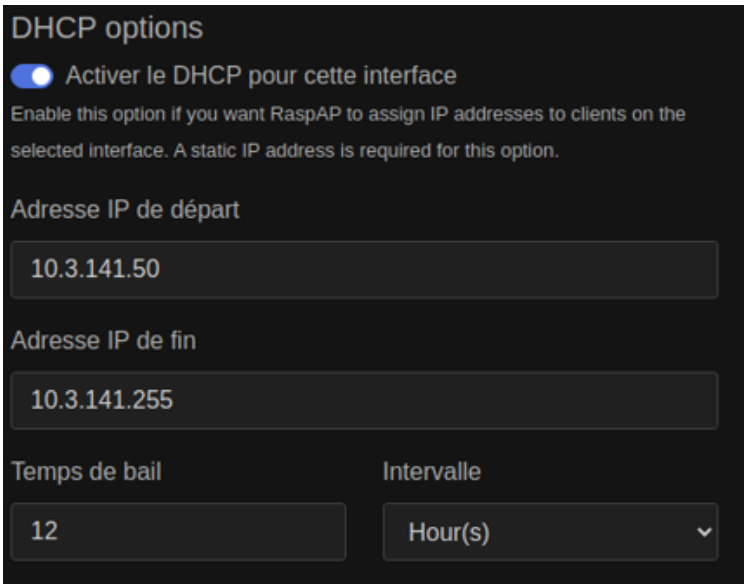

Figure 10 – Configuration DHCP des clients

Il nous reste juste à modifier le nom et le mot de passe de notre réseau. Au final sur le tableau de bord de l'interface d'administration, nous pouvons visualiser les informations relatives aux appareils connectés (leurs noms, adresses Ip et Mac). Nous pouvons aussi observer le trafic horaire illustré par les deux courbes (voir figure ci-dessous). La bleue et la blanche représentent respectivement la quantité en Mb de bits envoyée et reçue.

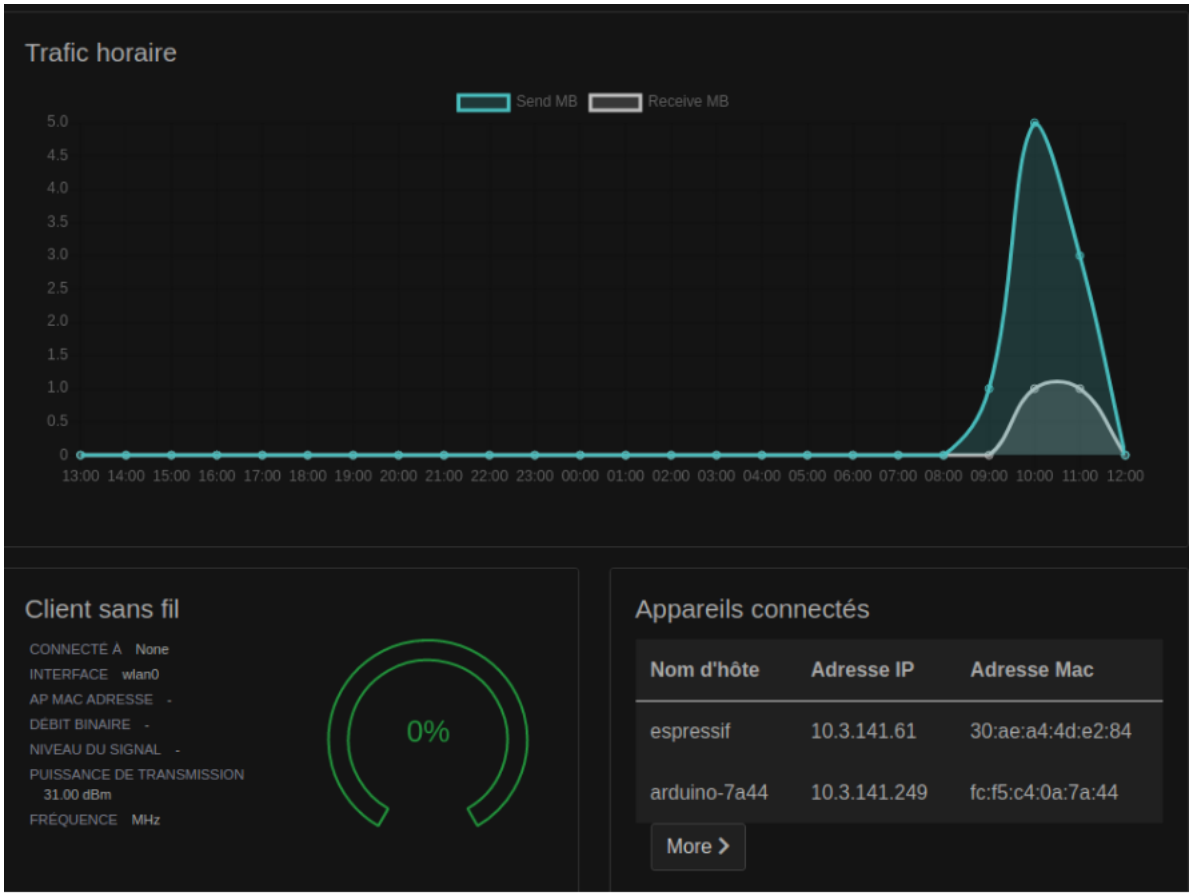

Figure 11 – Tableau de bord de RaspAp

4 Réception et affichage des données

Une fois notre point d'accès fonctionnel, nos modules Wipy sont maintenant capables d'interagir avec le broker MQTT *Adafruit* et de publier les données qu'ils ont collectées (température, pression et humidité). Dans cette partie, nous allons s'intéresser à la réception et l'affichage de ces données. Pour ce faire, nous allons utiliser la solution TIG, c'est à dire en utilisant le trio **Telegraf** pour la collecte des métriques/données, **Influxdb** pour le stockage et **Grafana** pour l'affichage.

Ainsi, l'architecture que nous allons utiliser est représentée ci-dessous.

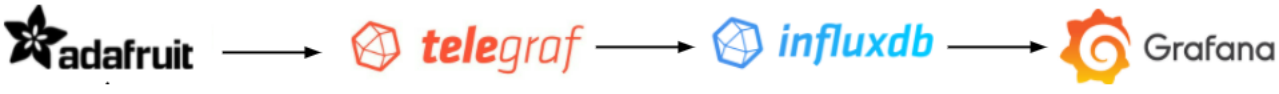

Figure 12 – Architecture TIG (Telegraf, InfluxDb et Grafana)

4.1 Telegraf

Telegraf est un agent de récupération de métriques, il permet de collecter des mesures à partir d'un large éventail d'entrées et les envoyer à un large éventail de sorties. Dans notre cas, il sera utilisé afin de récupérer les données depuis le MQTT broker *Adafruit* et il se chargera aussi de délivrer ce qu'il a récupéré à Influxdb.

Pour ce faire, nous avons besoin de l'installer et de modifier ses fichiers de configurations afin de réaliser ce qui est souhaité.

Une fois l'installation terminée, il faut modifier son fichier .conf qui se trouve à */etc/telegraf/-telegraf.conf*. Ce fichier contient tous les plugins possibles, ils permettent de réaliser des interfaces d'entrée ou de sortie avec un ou plusieurs service spécifiques. Dans notre cas, on n'utilisera que les deux plugins suivants : [[inputs.mqtt_consumer]] et [[outputs.influxdb]].

— Le premier plugin **[[inputs.mqtt_consumer]]** inclut toutes les informations nécessaires (le nom du broker, les topics, le type et le format des données ...) qui vont permettre à Telegraf de se connecter au broker, de s'abonner/souscrire à un topic afin de récupérer les mesures.

```
# # Read metrics from MQTT topic(s)
[[inputs.mqtt_consumer]]

servers = ["tcp://io.adafruit.com:1883"]
qos = 0
connection_timeout = "30s"

topics = ['Project_Wipy/feeds/temperature',
          'Project_Wipy/feeds/humidity',
          'Project_Wipy/feeds/pressure']

username = "Project_Wipy"
password = "aio_hMWt73y8zhLsdawexevOpe7OSqu3"

client_id = ""

data_format = "value"
data_type = "float"
```

Figure 13 – Description du plugin [[inputs.mqtt_consumer]]

— Le deuxième plugin **[[outputs.influxdb]]** regroupe toutes les informations nécessaires (l'url de la base de données, son utilisateur et son mot de passe) qui vont permettre à Telegraf de bien transmettre les métriques à la base de données Influxdb.

```
# Configuration for sending metrics to InfluxDB
[[outputs.influxdb]]
urls = ["http://127.0.0.1:8086"]

database = "project"

username = "utilisateur"
password = "motdepasse"
```

Figure 14 – Description du plugin [[outputs.influxdb]]

Pour finir, il faut redémarrer le serveur Telegraf pour prendre en compte la nouvelle configuration : **sudo systemctl restart telegraf**.

4.2 Influxdb

InfluxDB est une Time Series Database (TSDB), c'est à dire que chaque donnée stockée est sous forme d'un couple contenant la mesure collectée et l'instant durant lequel elle a été prise. Il doit être installé sur la même machine que Telegraf, car le plugin [[outputs.influxdb]] décrit précédemment travaille en localhost. Après son installation, on passe à la création de la base de données, de son utilisateur et son mot de passe. Il faut faire attention et s'assurer que ses informations soit identiques à celles indiquées dans le fichier *telegraf.conf*.

Les étapes à suivre sont les suivantes :

```
CREATE DATABASE project
CREATE USER utilisateur WITH PASSWORD 'motdepasse'
GRANT ALL ON project TO utilisateur
```

Figure 15 – Configuration de Influxdb

Au final, il reste juste à lancer le service Influxdb.

4.3 Grafana

Grafana est un outil de supervision simple qui permet de s'intégrer à une TSDB, ici InfluxDB. Grafana expose dans des dashboards les métriques provenant d'Influxdb.

Nous commençons par l'installer puis le lancer. Toute la configuration se fait sur l'interface d'administration via une page web (l'url par défaut est http ://localhost :3000/). Pour la première authentification, il faut mentionner le nom et le mot de passe qui sont par défaut admin et admin. Une fois connecté, il faut d'abord créer la source de data. Pour ce faire, il faut cliquer sur "Add data source" et spécifier le type de TSDB (time series database) avec laquelle nous allons travailler.

Dans notre cas, c'est la base de données Influxdb. Il suffit maintenant de spécifier l'url, le nom, l'utilisateur et le mot de passe de la base. Ces paramètres doivent être conformes avec ce qui a été configuré précédemment.

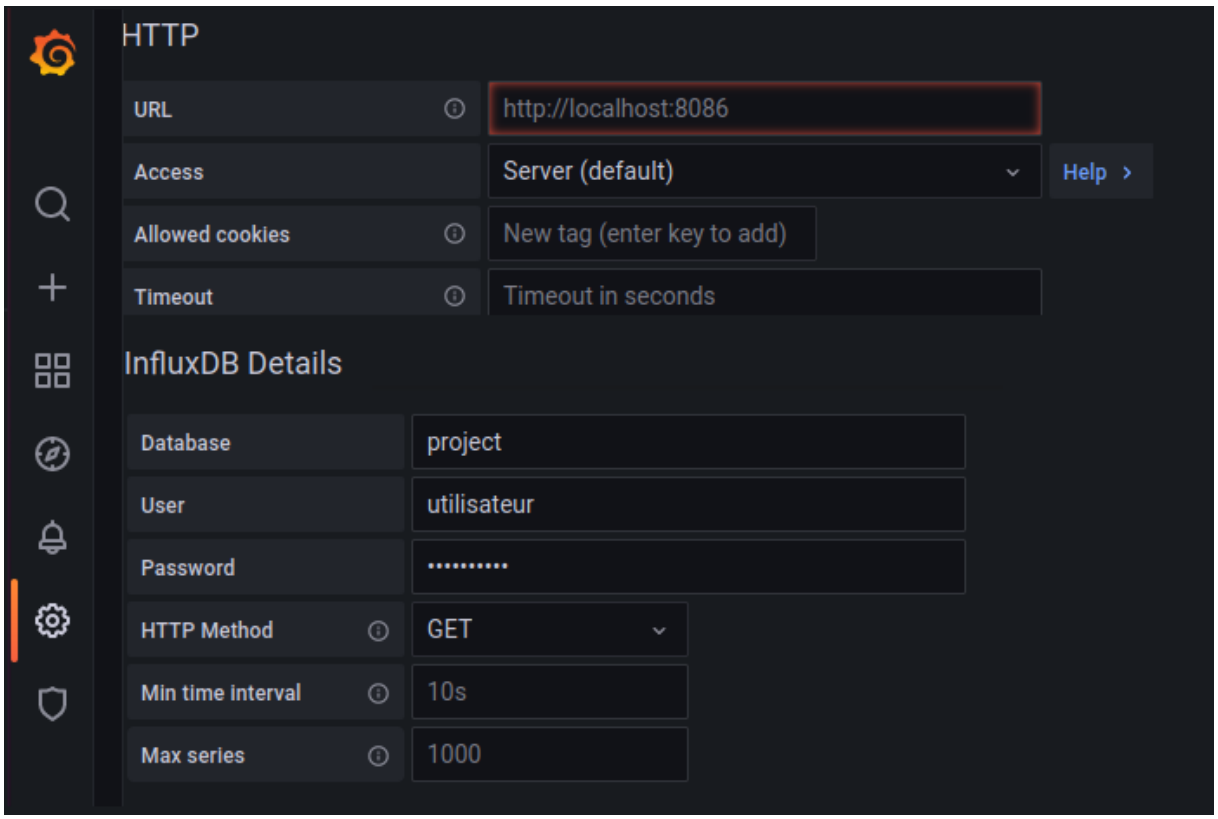

Figure 16 – Configuration de la "data source"

Une fois cette étape réalisée, il suffit de créer un dashboard en mentionnant le topic. Il existe plusieurs options qui permettent de créer des limites, choisir l'ordre croissant ou décroissant...

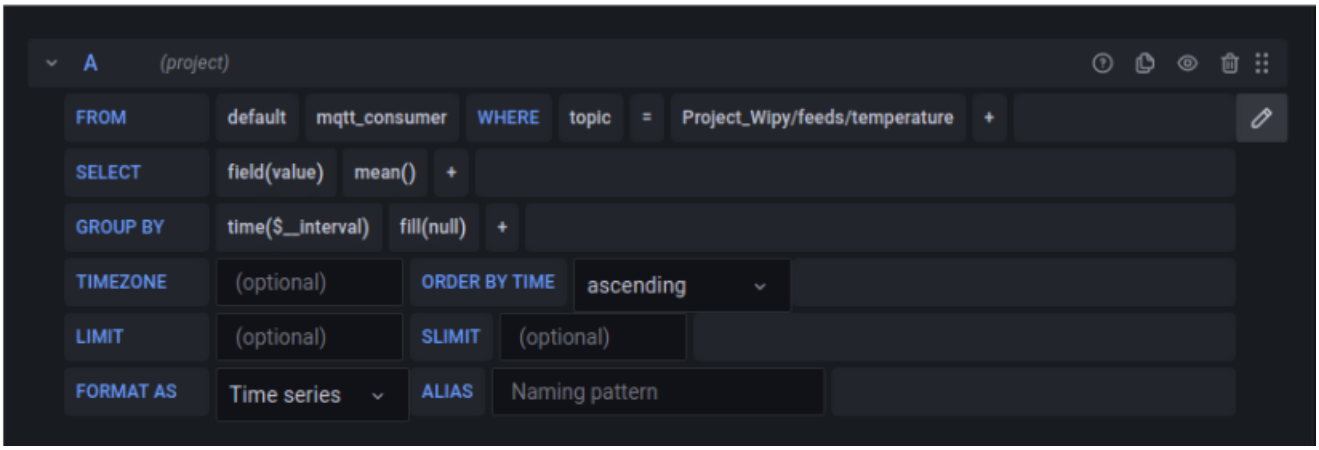

Figure 17 – Création du dashboard

Une fois toutes les étapes précédentes terminées, il est possible maintenant de visualiser l'évolution de nos graphes en temps réel. Par exemple, la figure suivante représente le tracé de la température.

Figure 18 – Graphe de la température

Grafana offre aussi la possibilité de définir des seuils d'alertes et les actions associées.

5 Partie supplémentaire

5.1 MaixDuino

5.1.1 Présentation de la carte Maixduino

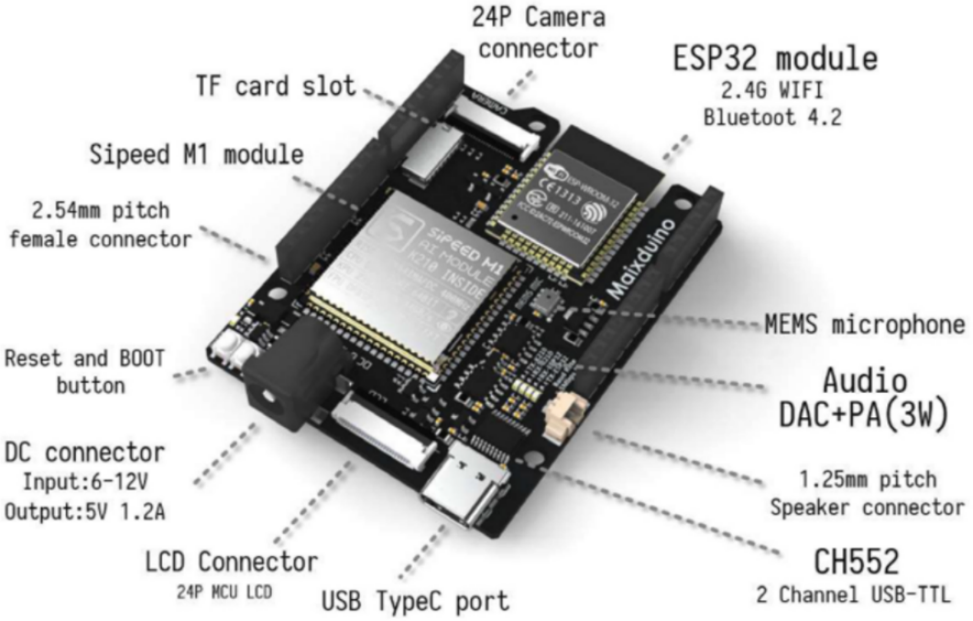

Figure 19 – Graphe de la température

La carte MaixDuino de Sipeed est une carte de test et de développement rapide d'applications IoT, au format compatible Arduino, composé de deux SoC principaux, dont un, le Kendryte K210 comportant notamment un double cœur RISC-V 64 bits RV64GC, et un accélérateur d'IA2. Le seconde SoC est basé sur un ESP323, utilisé principalement pour les fonctions réseau sans fil, mais comportant également un DSP.

La plateforme pré-installée est MaixPy, basé sur microPython, et se programmant à l'aide de l'interface et éditeur MaixPy IDE.

5.1.2 Récupération des données

Pour la partie d'émission, nous nous contentons d'envoyer seulement la valeur de température en utilisant un PMODTMP3 qui communique avec la carte via une liaison I2C.

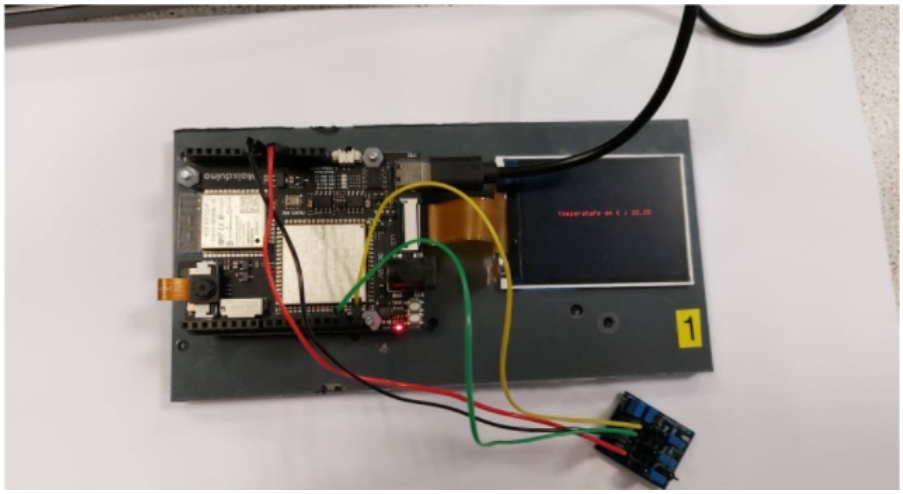

Figure 20 – MaixDuino + capteur de température

D'abord, nous récupérons la mesure de température sur 12 bits en communiquant avec le module PMOD TMP3 sur l'adresse 0x48. La valeur reçue du registre est en format complément à deux, pour cela nous avons défini la fonction *twos_comp*. Au final, afin d'avoir une valeur de température en C°, nous avons divisé la valeur de sortie par 16 (*0,0625).

```
tmp102_addr = 0x48

reg_temp=0x00
reg_config = 0x01

def twos_comp(val, bits):
    if(val & (1 << (bits - 1)))!=0 :
        val = val-(1<<bits)
    return val

def read_temp():
    val = i2c.readfrom_mem(tmp102_addr, reg_temp, 2)
    temp_c = (val[0] << 4)| (val[1] >> 5)

    temp_c = twos_comp(temp_c,12)
    temp_c = temp_c*0.0625
    return temp_c
```

Figure 21 – Mesure de la température avec un PMOD TMP3

5.1.3 Émission des données

Après avoir récupéré les mesures de température avec le capteur de la carte Maixduino, nous souhaitons maintenant les transmettre sur le *cloud* Adafruit IO. De façon similaire à l'envoi des données de la carte WiPy, la transmission des données est faite avec le protocole MQTT. Pour ce,

nous commençons par établir la connexion WiFi pour la carte Maixduino.

La connectivité de la carte est assurée par le module intégré ESP32 grâce à son interface WiFi. La communication entre ce microcontrôleur et le CPU de la carte Sipeed est réalisée via la liaison SPI (*Serial Peripheral Interface*), un bus de données série synchrone communiquant selon le schéma "maître-esclave".

Dans un premier temps, nous veillons à importer les bibliothèques nécessaires pour établir la connexion.

```
import network
from machine import SPI
from board import board_info
from Maix import GPIO
from fpioa_manager import *
```

Nous utilisons le module *network* pour gérer la mise en réseau et réaliser la communication WiFi. Comme, dans notre cas, le module ESP32, responsable de la connectivité, est en liaison SPI, il nous faut créer un objet *ESP32_SPI network*. Nous utilisons donc la fonction *network.ESP32_SPI()* qui prend en paramètre les fonctions des pin dont nous avons besoin (*fpioa_func*).

```
fm.register(25, fm.fpioa.GPIOHS10)
fm.register(8, fm.fpioa.GPIOHS11)
fm.register(9, fm.fpioa.GPIOHS12)
fm.register(28, fm.fpioa.GPIOHS13)
fm.register(26, fm.fpioa.GPIOHS14)
fm.register(27, fm.fpioa.GPIOHS15)

nic = network.ESP32_SPI(cs=fm.fpioa.GPIOHS10, rst=fm.fpioa.GPIOHS11, rdy=
    fm.fpioa.GPIOHS12, mosi=fm.fpioa.GPIOHS13, miso=fm.fpioa.GPIOHS14, sclk
    =fm.fpioa.GPIOHS15)
```

Par la suite, nous définissons les paramètres WIFI_SSID (nom du point d'accès WiFi) et WIFI_PASS (mot de passe du point d'accès WiFi), puis nous établissons la connexion :

```
while not nic.isconnected() :
    print("Connecting ...")
    nic.connect(WIFI_SSID, WIFI_PASS)
```

La ligne *nic.connect(WIFI_SSID, WIFI_PASS)* est ce qui effectue la connexion. Nous utilisons la boucle *while* afin que le programme cherche constamment à établir la connexion WiFi si tel n'est pas le cas.

Dans un deuxième temps, nous avons ajouté à notre code le module permettant d'envoyer des informations par MQTT. Comme expliqué précédemment, le protocole MQTT repose sur la notion de *publish-subscribe*, la publication et l'abonnement à un sujet (*topic*). Une fois de plus, le *topic* que nous considérerons est le *feed* créé sur Adafruit IO pour les données de température provenant de la carte Maixduino.

Nous commençons par créer un client MQTT en précisant les paramètres de configuration du cloud Adafruit IO que nous utilisons comme MQTT broker.

```
# Use the MQTT protocol to connect to Adafruit IO
client = MQTTClient(AIO_CLIENT_ID, AIO_SERVER, AIO_PORT, AIO_USER, AIO_KEY
    )
client.connect() #connects to Adafruit IO
```

Ensuite, dans une boucle infinie, nous faisons appel à la fonction *send_ data()* qui permet d'envoyer les données mesurées chaque quinzaine de secondes à Adafruit en les publiant sur le feed dont nous précisons la clé. La fonction permettant l'envoi des données par MQTT est :
client.publish(topic=AIO_ TEMPERATURE_ FEED, msg=str(temperature)).
Le code ci-dessous représente la fonction *send_ data()* telle que nous l'avons définie.

```
def send_data():
    global last_data_sent_ticks
    global DATA_INTERVAL

    if ((time.ticks_ms() - last_data_sent_ticks) < DATA_INTERVAL):
        return; # Too soon since last one sent.

    temperature = read_temp()

    try:
        client.publish(topic=AIO_TEMPERATURE_FEED, msg=str(temperature))
        print("Temperature sent")
        print("DONE")
    except Exception as e:
        print("FAILED")
    finally:
        last_data_sent_ticks = time.ticks_ms()
```

5.1.4 Résultat

Afin de tester notre programme, nous créons un nouveau *dashboard* sur Adafruit IO afin de visualiser les données de température mesurées avec Maixduino.

Une fois de plus, nous parvenons à voir en temps réel la mise à jour des valeurs de température transmises sur Adafruit IO. Ces tests nous ont permis de valider le bon fonctionnement de notre programme.

Les figures ci-dessous représentent le résultat du travail effectué.

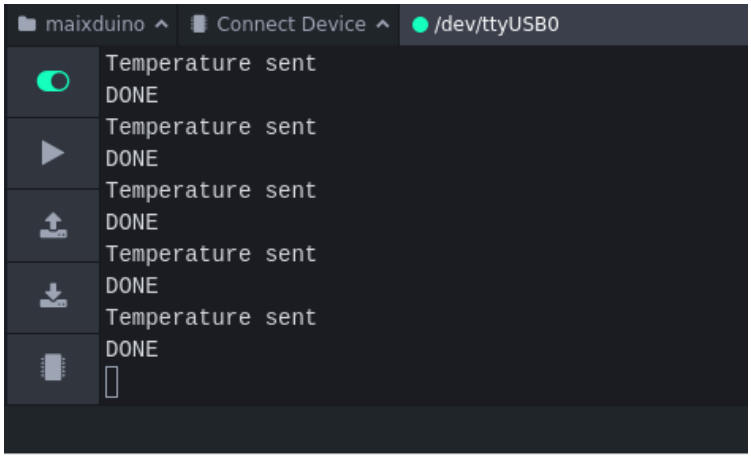

Figure 22 – Résultat de l'exécution du programme sur le terminal d'Atom Pymakr

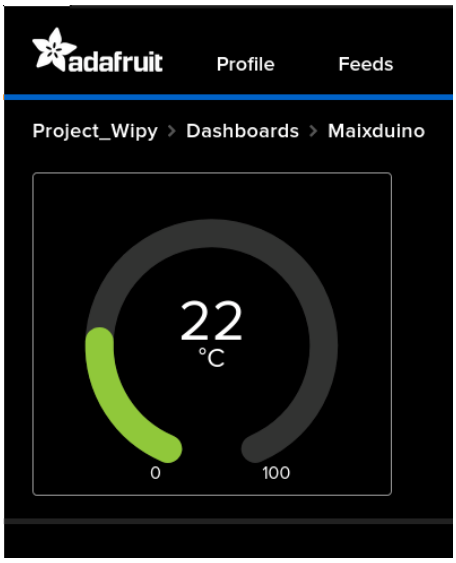

Figure 23 – Résultat d'affichage des données reçues sur Adafruit IO

5.2 Application graphique

Dans cette partie, nous avons décidé de créer une application dans laquelle l'utilisateur a la possibilité de choisir les paramètres (Broker à utiliser, données à afficher, numéro de la carte ...) manuellement sans avoir à modifier le code ou les fichiers de configuration à chaque fois. Dans cette application, nous avons mis à disposition de l'utilisateur une interface graphique pratique pour l'affichage des données. Le langage utilisé pour le développement de l'application est **Python** avec la bibliothèque **PyQt5**.

5.2.1 Pourquoi PyQt5 ?

Nous avons choisi PyQt5 parce qu'il est l'un des modules les plus utilisés dans la création des applications graphiques en Python. Cela est dû à sa simplicité, comme nous allons le voir après. Et aussi parce qu'il est facile de retrouver et d'obtenir une grande quantité de documentation sur le WEB.

5.2.2 Fonctionnement de l'application

Notre application possède trois parties principales.

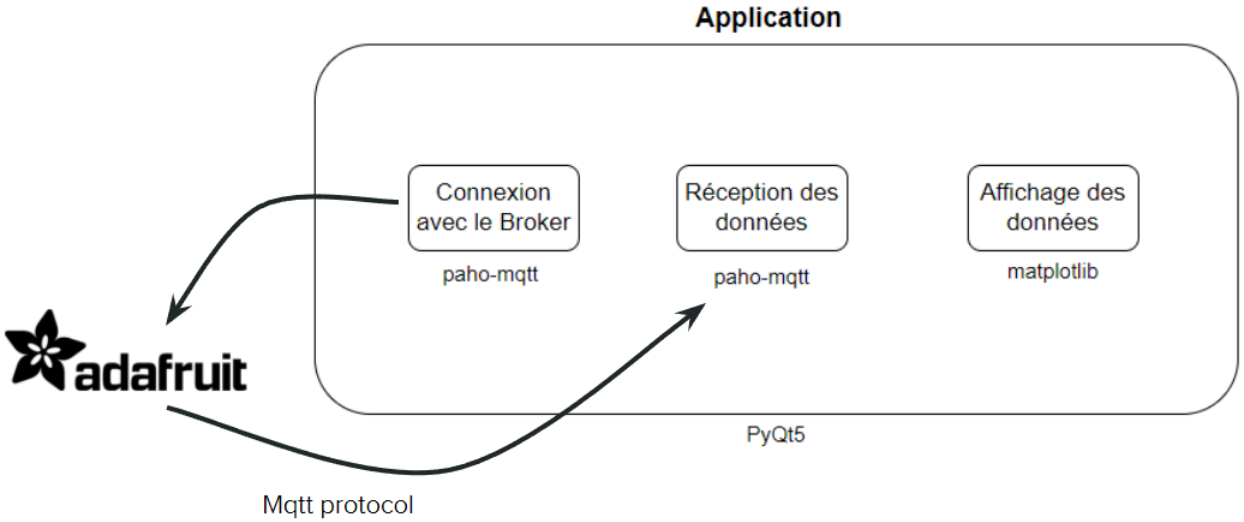

Figure 24 – Fonctionnement de l'application

Nous commençons par établir la connexion avec le Broker choisi par l'utilisateur (dans notre cas le broker est **Adafruit**). Ensuite, l'utilisateur choisit les données qu'il veut visualiser. Au final, la réception des données commence et l'affichage se fait au fur et à mesure.

5.2.3 Connexion avec le Broker

Pour cette partie, nous avons utilisé la bibliothèque Python **Paho MQTT**. Afin de nous connecter avec Adafruit, on définit la fonction **connect_mqtt** (voir image ci-dessous) pour laquelle il faut fournir le Broker à utiliser, le nom d'utilisateur et le mot de passe utilisés sur le site WEB de ce dernier et enfin le port à utiliser pour la connexion. On définit aussi la fonction **on_connect** qui se lance une fois la connexion établie avec succès.

```python
def connect_mqtt() -> mqtt_client:
    global username
    global password
    global broker
    global client
    global client_id
    def on_connect(client, userdata, flags, rc):
        if rc == 0:
            print("Connected to MQTT Broker!")
        else:
            print("Failed to connect, return code %d\n", rc)

    client = mqtt_client.Client(client_id)
    client.username_pw_set(username, password)
    client.on_connect = on_connect
    client.connect(broker, port)
    return client
```

Figure 25 – Fonction pour établir la connexion avec le Broker

5.2.4 Réception des données

Afin de commencer à recevoir les données, il faut se souscrire (subscribe) à un **topic**. Pour cela, on définit la fonction suivante :

```python
def subscribe(client: mqtt_client):
    def on_message(client, userdata, msg):
        global measures
        global time_axis
        measures.append(float(msg.payload.decode()))
        time = str(datetime.now())[11:19]
        time_axis.append(time)
        print("You have recieved and saved: ",len(measures), " measures")

    print("You have subscribed to: ",topic)
    client.subscribe(topic)
    client.on_message = on_message
```

Figure 26 – Fonction pour la réception des données

On définit aussi, la fonction **on_message** qui est lancée automatiquement dès qu'on reçoit une donnée. Dans notre cas, on a deux tableaux globaux **measures** et **time_axis** dans lesquels on ajoute respectivement la donnée reçue en **float** et l'instant (en Heure :Minute) en **string** durant lequel la mesure a été réalisée.

5.2.5 Affichage des données

Pour afficher les données en temps réel, on a utilisé la bibliothèque **matplotlib**. Cette dernière nous offre une interface d'affichage des graphes déjà prête avec plusieurs outils pratiques. Afin d'afficher les données au fur et à mesure qu'on les reçoit, on utilise des fonctions spécifiques de **matplotlib** qu'on peut voir dans la figure suivante.

```
def plot(self):
    global ploting
    global time_axis
    global measures
    ploting = True
    while (len(measures) < 2 ):
        continue
    ax = self.figure.add_subplot(111)
    ax.set_title(topic)
    ax.set_xlabel('time (s)')
    if (topic == "temperature"):
        topic = topic + " °C"
    if (topic == "pressure"):
        topic = topic + " Pa"
    if (topic == "humidity"):
        topic = topic + " %"
    ax.set_ylabel(topic)
    line1, = ax.plot(time_axis, measures, 'b-')
    while(1):
        line1.set_ydata(measures)
        line1.set_xdata(time_axis)
        self.figure.canvas.draw()
        self.figure.canvas.flush_events()
```

Figure 27 – Fonction pour l'affichage des données

Dans cette fonction on utilise les tableaux globaux **measures** et **time_axis** qui contiennent les données reçues et le temps de réception. On utilise aussi la variable ploting pour informer les autres parties de notre programme qu'on est entrain d'afficher les données. On remarque aussi que notre affichage ne commence que lorsqu'on reçoit au moins 2 mesures.

5.2.6 Fenêtres de l'application

Notre application se compose de 3 fenêtres :

a) Fenêtre principale :

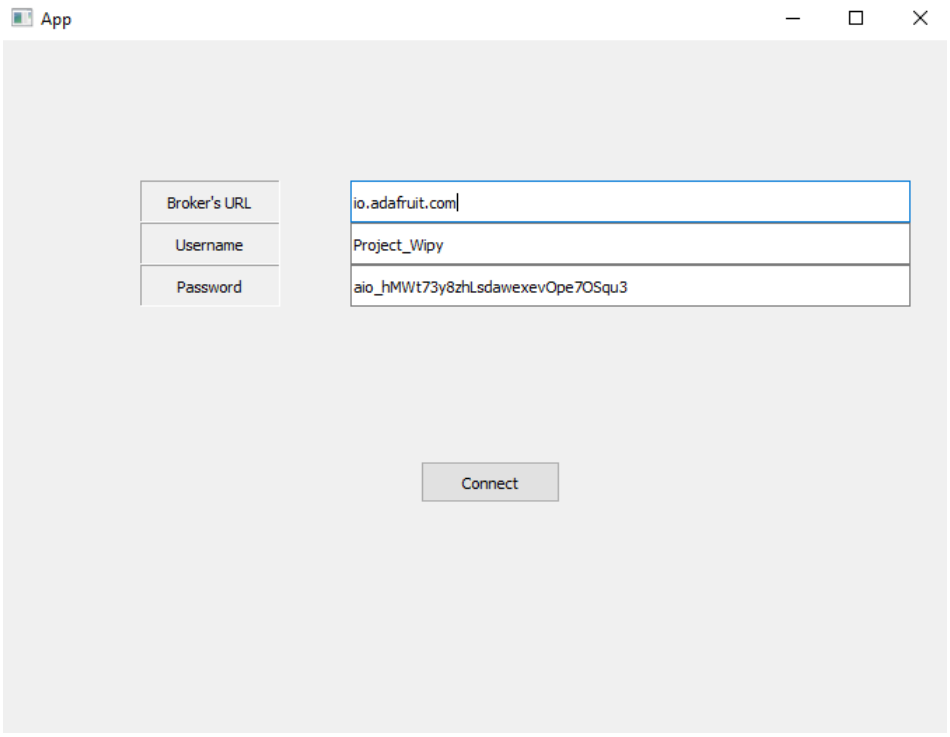

Figure 28 – Fenêtre principale de l'application

Cette première fenêtre nous donne l'option de choisir le Broker, le nom d'utilisateur et le mot de passe. On a aussi le bouton **Connect** pour initier la connexion. Pour définir cette fenêtre, on définit la classe suivante :

```
class Window(QMainWindow):
    def __init__(self):
1       super().__init__()
        self.title = "App"
        self.top = 100
        self.left = 100
        self.width = 680
        self.height = 500
        self.InitUI()

    def InitUI(self):
2       self.setWindowTitle(self.title)
        self.setGeometry(self.top, self.left, self.width, self.height)

        LABEL1 = QLabel(self)
        LABEL1.setText("URL")
        LABEL1.setAlignment(Qt.AlignCenter)
        LABEL1.move(100, 100)
        LABEL2 = QLabel(self)
3       LABEL2.setText("Username")
        LABEL2.setAlignment(Qt.AlignCenter)
        LABEL2.move(100, 130)
        LABEL3 = QLabel(self)
        LABEL3.setText("Password")
        LABEL3.setAlignment(Qt.AlignCenter)
        LABEL3.move(100, 160)

        self.lineEdit1 = QLineEdit("io.adafruit.com", self)
        self.lineEdit1.setGeometry(250, 100, 400, 30)
4       self.lineEdit2 = QLineEdit("Project_Wipy", self)
        self.lineEdit2.setGeometry(250, 130, 400, 30)
        self.lineEdit3 = QLineEdit("aio_hMWt73y8zhLsdawexevOpe7OSqu3", self)
        self.lineEdit3.setGeometry(250, 160, 400, 30)

        buttonWindow1 = QPushButton('Connect', self)
5       buttonWindow1.move(300, 300)
        buttonWindow1.clicked.connect(self.buttonWindow1_onClick)
        self.show()
```

Figure 29 – Classe à définir pour la fenêtre principale de l'application

Cette classe doit hériter de la classe **QMainWindow**. Dans l'initialisation de la classe (**partie 1**), on définit le titre de la fenêtre, sa position avec top et left et sa taille avec width et height et on finit par appeler la méthode **InitUI()**.

Cette dernière est constituée de 3 parties principales : **la partie 2** sert à appliquer les paramètres de la fenêtre définis durant l'initialisation de la classe.

Dans **la partie 3**, on définit les étiquettes de texte non-modifiable à afficher à l'aide de la classe **QLabel** et la méthode **setText()**. On peut aussi déplacer ces étiquettes de texte avec la méthode **move()** et on peut modifier l'alignement du texte avec la méthode **setAlignment()**.

La partie 4 sert à afficher des champs de saisie dans lesquels les utilisateurs peuvent écrire

et fournir des informations. Ces champs sont créés avec la classe **QLineEdit** et on modifie leurs position et leurs tailles avec la méthode **setGeometry(top, left, width, height)**.

Pour afficher des boutons (**partie 5**), on utilise la classe **QPushButton**. Pour les boutons, il faut définir la méthode qui va s'exécuter une fois le bouton appuyé, dans notre cas c'est la méthode **buttonWindow1_onClick** qu'on va détailler par la suite.

Enfin, pour afficher la fenêtre, on utilise la méthode **show()**.

La méthode buttonWindow1_onClick :

```
def buttonWindow1_onClick(self):
    global broker
    global username
    global password
1   broker = self.lineEdit1.text()
    username = self.lineEdit2.text()
    password = self.lineEdit3.text()

    client = connect_mqtt()
    self.statusBar().showMessage("Window1")
2   self.cams = Window1()
    self.cams.show()
    self.close()
```

Figure 30 – Méthode lancée après l'appui sur le bouton **Connect**

Dans cette méthode, il y a deux parties principales. On commence dans **la partie 1** par récupérer les données (Broker, nom d'utilisateur et le mot de passe) fournies par l'utilisateur à travers les champs de saisie.

Dans **la partie 2**, on essaye de se connecter avec le Broker. Si la connexion est établie, on ferme la fenêtre principale et on lance la fenêtre **Window1**.

De la même manière on définit les 2 fenêtres restantes (**Window1** et **Window2**).

b) Fenêtre Window1 :

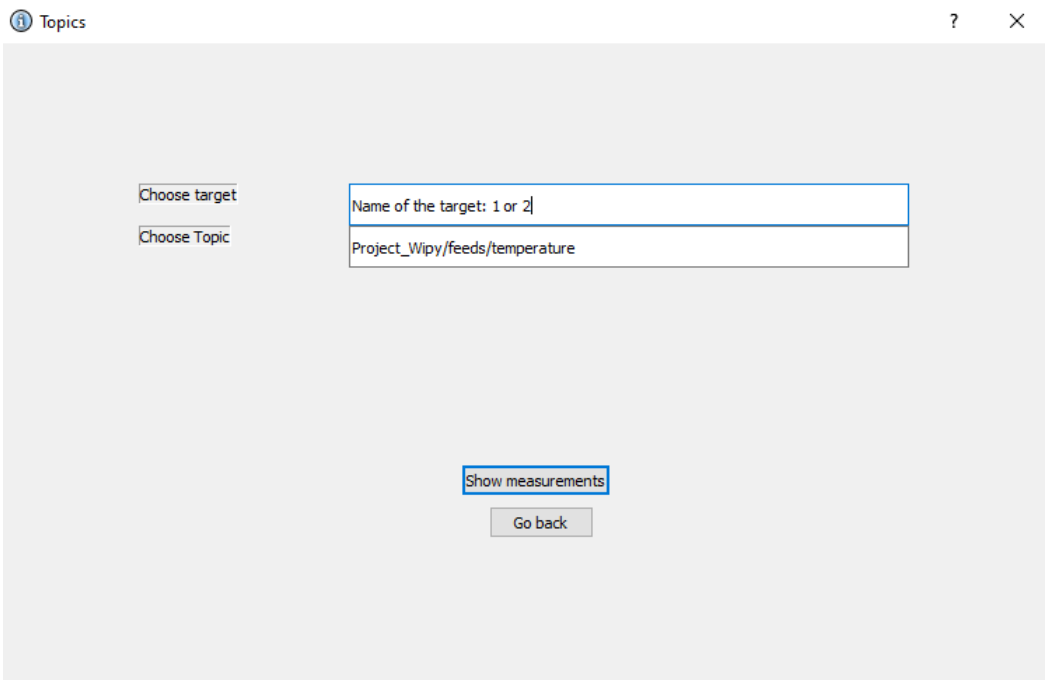

Figure 31 – La fenêtre **Window1**

Une fois la connexion avec le Broker établie avec succès, on a cette fenêtre qui s'affiche et qui nous donne la possibilité de choisir la carte et le topic pour lequel on doit s'abonner. On a en bas deux boutons : **Show measurements** pour commencer à afficher les données en temps réel et le bouton **Go back** pour arrêter la connexion avec le Broker et revenir vers la fenêtre principale.

c) Fenêtre Window2 :

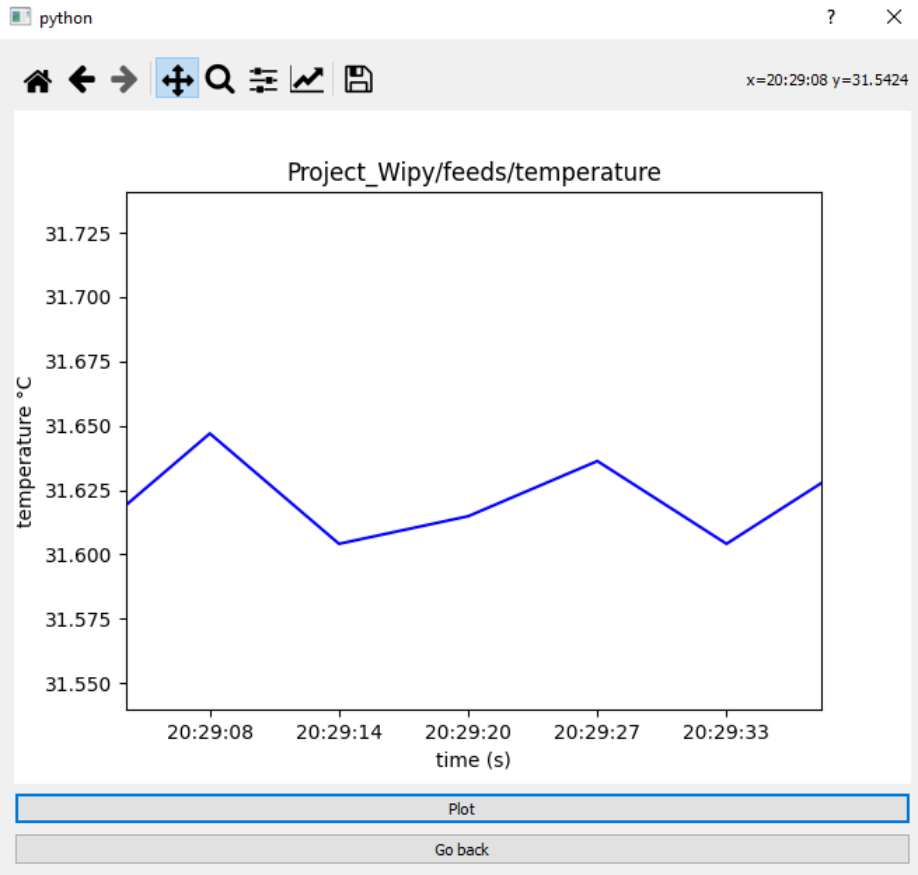

Figure 32 – La fenêtre **Window2**

Si on choisit un numéro de carte et un topic valides dans ce qui précède, cette fenêtre s'ouvre. Pour commencer à afficher les données, on appuie sur le bouton **Plot**. Dès qu'on reçoit 2 données au moins, le graphe s'affiche. Le graphe est affiché à l'aide de l'interface graphique de **matplotlib** et on voit qu'en haut de la fenêtre, on a plusieurs outils. Les plus importants sont ceux avec les deux flèches croisées et la disquette. Le premier nous permet de nous déplacer dans le graphe mais aussi d'ajuster les axes. Le deuxième sert à sauvegarder une capture du graphe.

En bas de la fenêtre, on a encore une fois le bouton **Go back** qui nous permet d'aller vers la fenêtre Window1 pour choisir une carte différente ou un type de données différent.

6 Conclusion

Dans ce projet, on a eu l'occasion de découvrir le monde d'IOT (Internet Of Things). On a étudié les principes et les outils utilisés dans ce domaine, notamment le fameux protocole de communication (MQTT) qui est devenu le protocole standard pour les communications IoT, et quelques outils (telegraf, influxDB et Grafana) qui sont utilisés pour l'envoi, la réception, la gestion

et l'affichage des données avec ce protocole. On s'est aussi familiarisé avec la notion de Broker en utilisant Adafruit. Durant ce projet, on a appris comment récupérer des mesures de différents types (température, pression et humidité) en utilisant deux cartes différentes (Wipy et Maixduino). On a également étudié comment créer et configurer notre propre point d'accès avec l'outil RaspAP. Enfin, nous avons créé notre propre application graphique pour offrir à l'utilisateur la possibilité de choisir les paramètres de connexion, le numéro de la carte et le type de données à recevoir et à afficher en temps réel.

7 Bibliographie

Références

[1] Quick reference for the WiPy : http://docs.micropython.org/en/latest/wipy/quickref.html

[2] Pycom Pysense Getting Started Guide : https://core-electronics.com.au/tutorials/pycom-pysense-getting-started.html

[3] Pycom Getting Started : https://docs.pycom.io/gettingstarted/

[4] Adafruit IO Guide : https://learn.adafruit.com/welcome-to-adafruit-io/what-is-adafruit-io

[5] Réaliser un routeur WiFi : https://eco-sensors.ch/router-wifi-4g-hotspot/

[6] Raspberry Pi Documentation - Configuration : https://www.raspberrypi.org/documentation/computers/configuration.html#setting-up-a-routed-wireless-access-point

[7] Transformer Raspberry Pi en routeur Internet : http://rpi.noirpatin.eu/rpi_box4g.html

[8] Monitoring avec Telegraf, chrongraph et influxdb https://connect.adfab.fr/devops/monitoring-avec-telegraph-chronograph-et-influxdb

[9] How to setup and secure Telegraf, InfluxDB and Grafana on Linux : https://sysadmin.info.pl/en/how-to-setup-and-secure-telegraf-influxdb-and-grafana-on-linux/