Ruche connectée

BUREAU D'ETUDE

HABERT Loïc / HOYO Pierre

L3 EEA REL | UT3 PAUL SABATIER

BE RUCHE CONNECTEE

TABLE DES MATIERES

INTRODUCTION

PRESENTATION DU SUJET

Notre bureau d'étude est porté sur les ruches connectées et leur instrumentation. Elle portera tout d'abord sur la compréhension des informations (signaux) renvoyées par les différents instruments, puis, la poursuite de l'étude sera dirigée vers la récupération de ses informations à distance sur une application pour mobile ou sur un site internet.

Le but de cette étude est de simuler une véritable ruche connectée pour comprendre les tenants et aboutissants d'un système électronique mais pas seulement. Notre choix s'est porté sur ce sujet car l'environnement et la biodiversité aujourd'hui est au cœur de l'attention, et il est intéressant de comprendre en quoi la technologie peut soutenir les écosystèmes à se reconstituer.

POURQUOI INSTRUMENTER UNE RUCHE ?

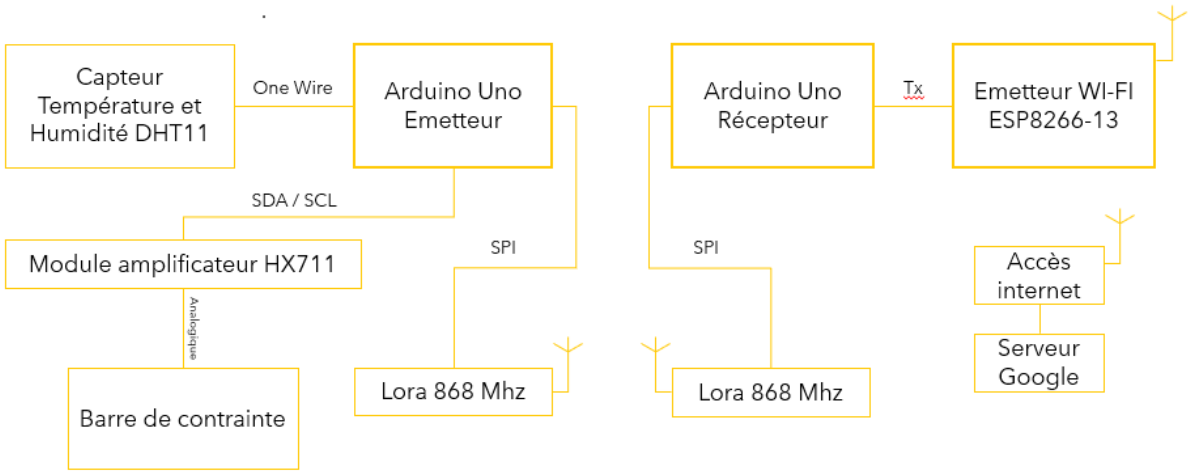
Les diminutions de population d'insecte depuis ces 30 dernières années n'ont pas épargné les abeilles. Importante à l'écosystème en pollinisant les plantes à fleurs, les butineuses garantissent la reproduction de nombreuses espèces végétales représentant pas moins d'un tiers de l'alimentation mondiale. Mises en danger par des pratiques agricoles néfastes (insecticides : néonicotinoïdes), l'enjeu est donc de protéger les abeilles en coopérant avec les apiculteurs afin d'instrumenter leurs ruches.

Une ruche connectée est constituée de différents capteurs permettant de mesurer la lumière, la température, la masse, l'humidité, la pression atmosphérique, l'orientation...etc. Ces différentes mesures vont aider les apiculteurs à surveiller l'état de leurs ruchers à distance, optimiser leur rendement et enfin à protéger leurs abeilles en suivant leur cycle de production.

DEVELOPPEMENT

ANALYSE FONCTIONNELLE

Notre projet de recherche s'est limité à l'étude d'un capteur de température et humidité, un capteur de masse, un module LORA et un module WIFI. Nous le décomposerons en deux parties, une partie émettrice et une partie réceptrice.



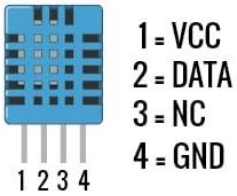
PRINCIPE DE FONCTIONNEMENT DE CHAQUE COMPOSANT

CAPTEUR DHT11

UTILISATION

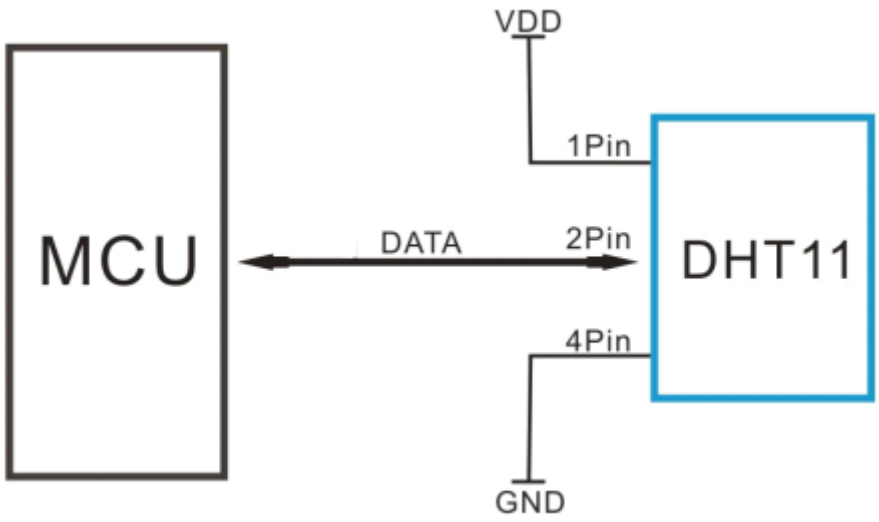
Le capteur DHT11 mesure la température et l'humidité. L'utilisation de ce type de capteur est intéressante car il permet de mesurer deux grandeurs physiques en même temps, à l'aide d'une thermistance pour la température et d'un capteur capacitif pour l'humidité. Le capteur DHT11 est composé de 4 broches. Il communique directement avec l'Arduino au travers d'une de ses 2 entrées numériques, les 2 autres broches sont pour l'alimentation 5 V et la masse (GND). Il est principalement utilisé pour des projets tels qu'une station météo, une serre connectée, une ruche connectée…

JUSTIFICATION



On utilise ce type de capteur car il est simple d'utilisation et propose une précision convenable à notre utilisation. Plus ou moins 2% de précision à 25°C, pour une plage de température de 0 à 50°C et plus ou moins 5% de précision pour une plage d'humidité comprise entre 20 et 90% RH. Alimenté en basse tension, de 3,5 à 5V, il consomme 2,5mA.

CONFIGURATION



3 Pin- Null

MCU= Micro-computer Unite (Arduino UNO)

COMPREHENSION

PROCESSUS DE COMMUNICATION

Le protocole de communication qui partage la « DATA » *(voir figure ci-dessus)* est le One Wire. C'est un système d'interface série (monofilaire bidirectionnelle) utilisé pour la communication et la synchronisation entre le microcontrôleur et le capteur. Le processus entier de communication des données dure environ 4ms.

COMPOSITION DES DONNEES :

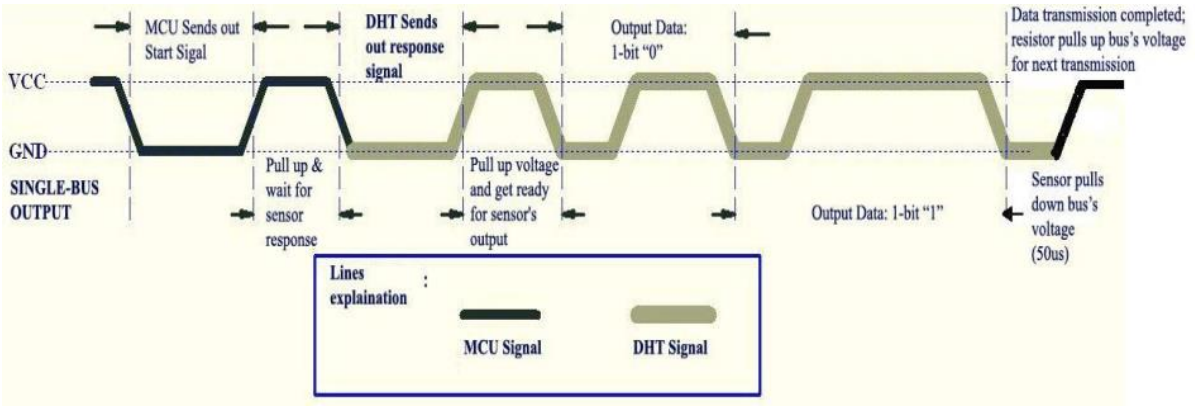
Les données sont composées de parties décimales et entières. Une transmission de données complète se fait en 40 bits avec comme premier bit envoyé par le capteur, le bit de poids fort.

FORMAT DE DONNEES :

8 bits Humidité (entier) + 8 bits Humidité (décimal) + 8 bits Température (entier) + 8 bits Température (décimal) + 8 Bits de contrôle

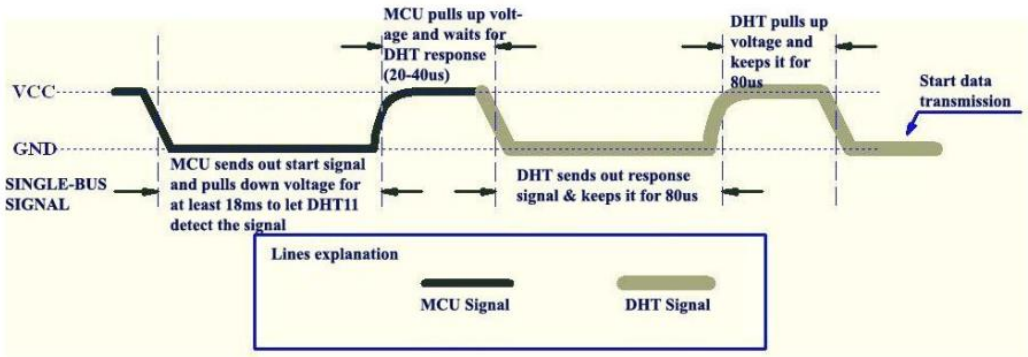
Si la transmission est correcte, les 8 bits de contrôle représentent la somme de tous les autres bits.

APPREHENSION DU SIGNAL :



- PROCESSUS DE DEMARRAGE

Le microcontrôleur en mode de fonctionnement (VCC) envoie un signal de démarrage et passe en mode basse consommation (GND). Après un temps minimum de 18ms, le microcontrôleur repasse en mode fonctionnement (VCC) et attend la réponse du capteur (entre 20 et 40 µs). Lorsque ce dernier envoie un signal retour, le signal devient celui du capteur et passe en mode basse consommation (GND). Après 80 µs, le capteur passe en mode fonctionnement 80 µs aussi et est prêt à envoyer l'information.



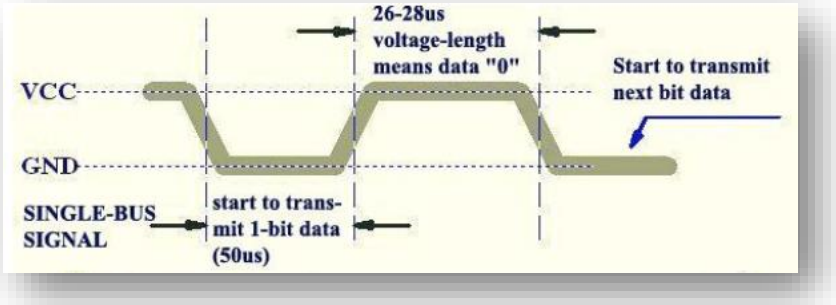
REMARQUES *:*

- *Sans le signal de départ le capteur ne pourrait pas renvoyer de réponse au microcontrôleur.*
- *Une fois les données collectées le capteur passe en mode basse consommation jusqu'à recevoir encore un signal de démarrage.*

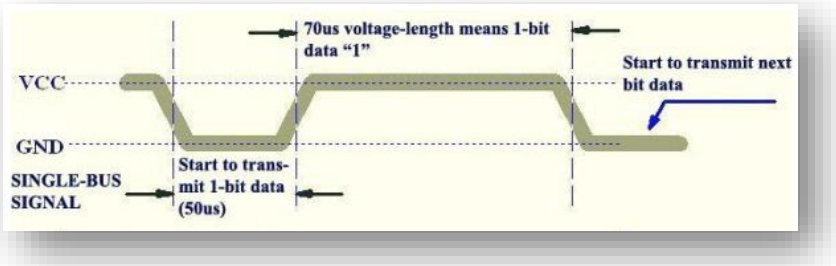
- LECTURE DES BITS

Une fois le processus de démarrage terminé, le capteur envoie le signal de données de 40 bits incluant les informations d'humidité et de température au microcontrôleur. Chaque bit de données commence par le niveau de basse tension pendant 50 µs et la longueur du signal de niveau de haute tension détermine si le bit de données est 0 (26-28 µs) ou 1 (70 µs).

Bit 0



Bit 1



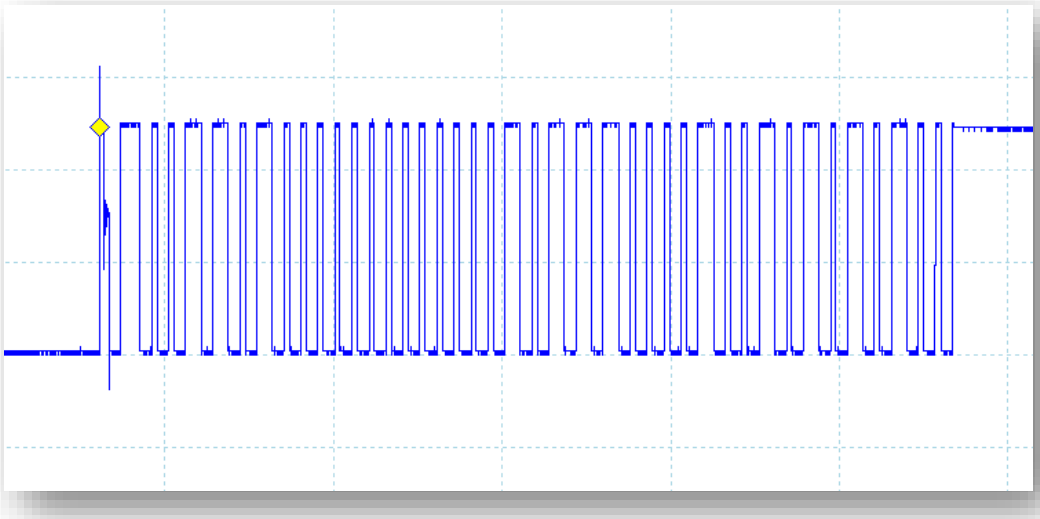
REMARQUES :

- Si la réponse du signal est toujours au niveau de haute tension, le capteur ne répond sûrement pas.
- Lorsque le dernier bit est transmis, le capteur abaisse le niveau de tension et le conserve pendant 50 µs.

ANALYSE DU SIGNAL OBSERVE :

Avec un appareil nommé picomètre, il est possible d'observer le signal « DATA » transmis *(voir figure ci-dessous)*.

En zoomant sur la partie « Bit de données » on reconnait le premier bit, celui de réponse du capteur et le deuxième bit, celui « d'initialisation ». Ensuite on peut lire les bits de données :



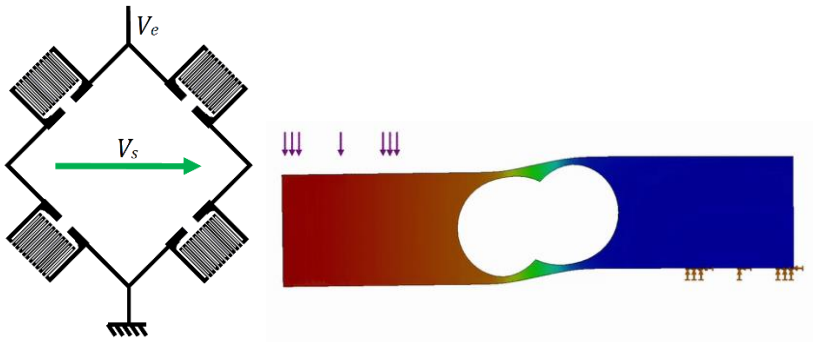
Bits Humidité		Bits Température		Bits de contrôle
Entier	Décimal	Entier	Décimal	
00110100	00000000	00010111	00001001	01010100
52	0	23	9	84

On retrouve une information sur l'humidité qui est de 52%, une information sur la température de 23,9 °C et les bits de contrôle.

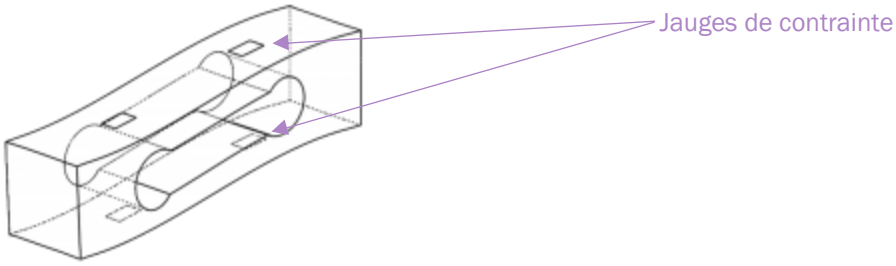
CAPTEUR DE MASSE

UTILISATION

La barre de contrainte permet la mesure de force associé à un convertisseur analogique-numérique (ADC) HX711. La mesure se fait à l'aide d'une jauge de déformation qui permet de mesurer l'allongement et d'un montage en pont complet de Wheatstone (voir figure ci-dessous) amplifié par le module HX711. Ce dernier sépare l'Arduino de la barre de contrainte à l'aide de 4 fils, 2 interfaces d'alimentation et 2 interfaces utilisées distinctement pour l'horloge et les données. Il est notamment utilisé pour le contrôle de processus industriels comme balance de haute précision.

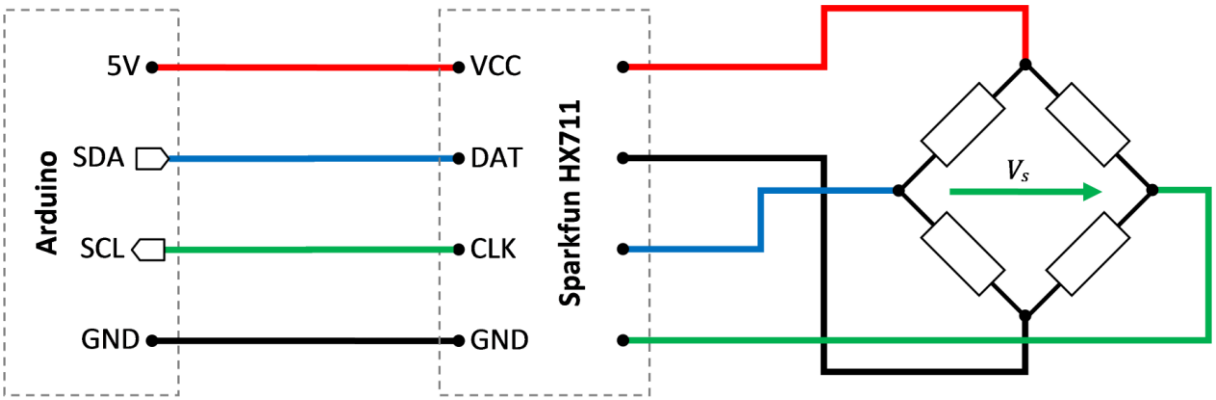


JUSTIFICATION

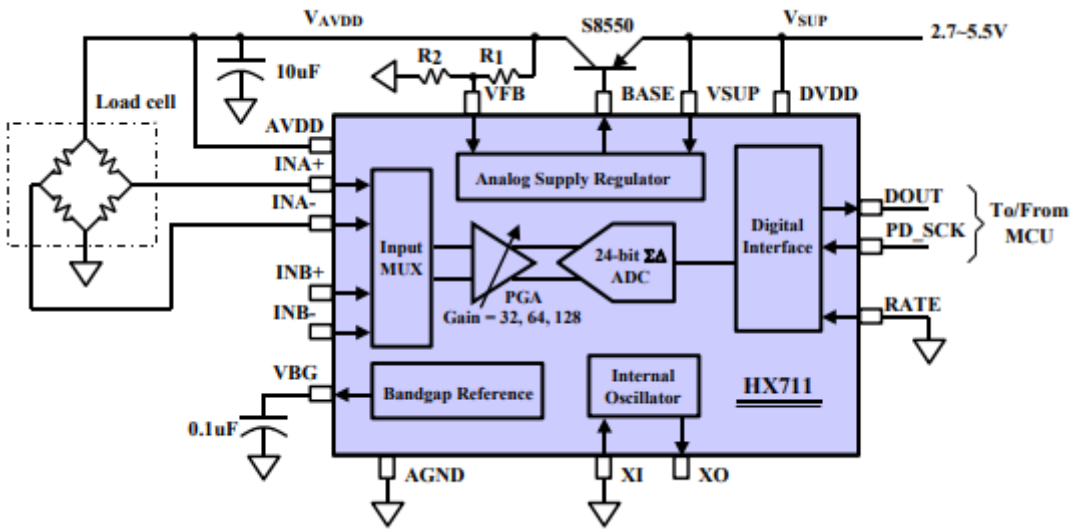


On utilise ce type de capteur car il a une très grande précision dû à son indifférence aux variations de température de -40° à 85°C, il est en revanche assez sensible à la casse. Alimenté en basse tension, de 2,6 à 5,5V, il consomme moins d'1,5 mA et 1 µA en veille.

CONFIGURATION



Les fils de couleurs verts et bleus de la partie gauche ne se suivent pas avec ceux de la partie droite.



COMPREHENSION

PROCESSUS DE COMMUNICATION

Une tension est délivrée par le capteur extensiométrique et reliée au module HX711 par des entrées analogiques sur les bornes IN+ et IN-. Un amplificateur programmable augmente le gain différemment selon le choix de l'entrée voie A ou B. La voie A propose un gain de 128 ou 64 selon le nombre d'impulsions positives qu'il y aura dans le signal d'horloge, déterminé dans le programme.

COMPOSITION DE DONNEES

L'information arrive sur un convertisseur analogique numérique 24 bits qui va convertir le signal analogique en un signal binaire numérique complémenté à deux.

APPREHENSION DU SIGNAL

Les broches PD_SCK et DOUT sont utilisées pour la sélection d'entrée de récupération de données, la sélection de gain et pour les commandes de mise hors tension dans une interface digitale joignant le microcontrôleur et le module HX711.

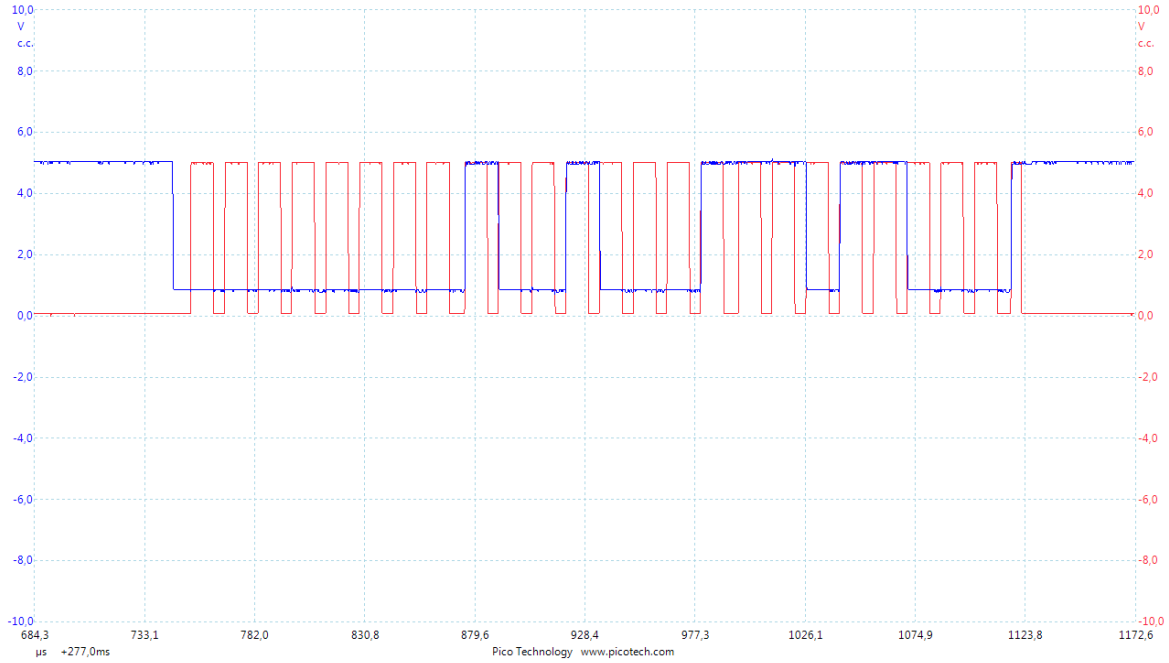
Lorsque les données de sortie ne sont pas prêtes à être récupérées, la broche de sortie numérique DOUT est élevée. L'entrée d'horloge série PD_SCK doit être basse. Lorsque DOUT passe au niveau bas, cela indique que les données sont prêtes à être récupérées.

Remarque :

- Les impulsions d'horloge PD_SCK ne doivent pas être inférieures à 25 ou supérieures à 27 dans une période de conversion, pour éviter de provoquer une erreur de communication série.

ANALYSE DU SIGNAL

A l'aide d'un picoscope, il est possible d'observer les informations dans le convertisseur *(voir ci-dessous)*.



Dans ce cas présent, nous retrouvons en bleu le signal de données et en rouge le signal d'horloge. Ce dernier est de 25 impulsions positives car nous avons choisi un gain de 128, dû à la tension très faible délivrée par le capteur. Ce signal donne l'expression en binaire complément à 2 (de droite à gauche) suivant :

2^3+2^4+2^6+2^7+ 2^8+2^12+2^15= 37 336

Ce résultat est ensuite converti dans le code de programmation afin d'obtenir les valeurs réelles voulus dans la bonne unité.

MODULE LORA

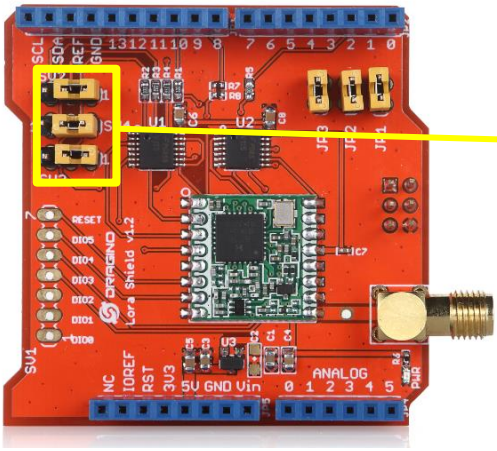
UTILISATION

Les modules LoRa (Long Range) assurent une communication sans fils interne entre un premier module émetteur et un deuxième module récepteur. Dans notre bureau d'étude, les modules LoRa chacun raccordés sur un Arduino différent, vont nous permettre d'échanger des informations à distance tels que la température, l'humidité, et la masse. Ils sont souvent utilisés pour des scénarios d'application longue distance et à faible consommation sur des objets connectés.

JUSTIFICATION

Le choix de ce matériel a pour but est de simuler une communication avec une ruche connectée placée dans une zone hors connexion Wi-Fi, en montagne par exemple. Les modules peuvent fonctionner à des fréquences soit de 434 Mhz, soit 868 Mhz, soit de 915 Mhz. L'avantage de sa faible consommation de 10,3 mA en fonctionnement permet d'allonger son autonomie.

CONFIGURATION



	Jumper On Right (default)	Jumper On Left
SV2	LoRa CLK <--> ICSP CLK	LoRa CLK <--> Arduino D13
SV3	LoRa DI <--> ICSP MOSI	LoRa CLK <--> Arduino D11
SV4	LoRa DO <--> ICSP MISO	LoRa CLK <--> Arduino D12

COMPREHENSION

PROCESSUS DE COMMUNICATION

Le protocole de communication du module LoRa est le SPI (Interface série périphérique). Cette communication est synchrone ce qui veut dire que le signal d'horloge est identique entre tous les appareils connectés sur un même système. Afin de profiter de ce mode de communication sur ce modèle LoRa, il faut placer les 3 shunts SV1, SV2 et SV3 à droite. En faisant cette manipulation on obtient :

MOSI -> D11; MISO -> D12; SCLK -> D13 ; SS -> D10

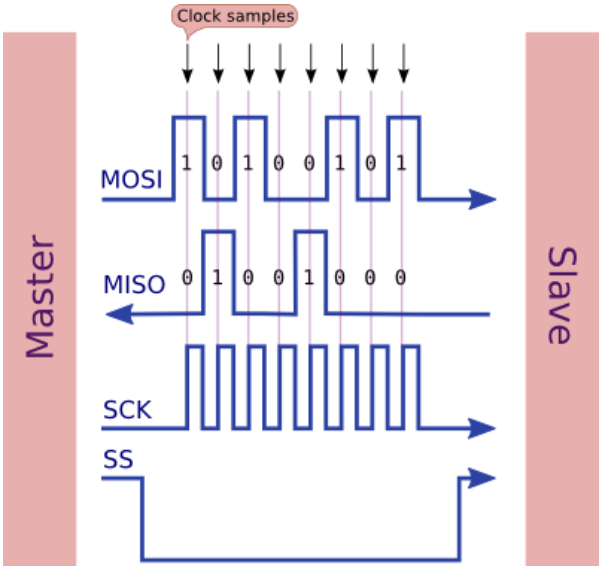
COMPOSITION DES DONNEES

Un bus SPI est composé de 4 fils entre l'émetteur et le récepteur :

- SCLK: Serial Clock (horloge série)
- MOSI : Master Output, Slave Input (sortie maitre, entrée esclave)
- MISO : Master Input, Slave Output (entrée maitre, sortie esclave)
- SS : Slave Select (choix de l'esclave)

FORMAT DE DONNEES

Le module maître génère une horloge et sélectionne le module esclave avec lequel il veut communiquer par l'utilisation du signal SS. Une fois ce dernier à bas, l'esclave est prêt à répondre aux requêtes du maître. À chaque coup d'horloge le maître et l'esclave s'échangent un bit. Dans notre cas, l'Arduino (maitre) peut choisir de recevoir ou envoyer une information, s'il doit émettre une information ce sera sur le fil MOSI et s'il veut en recevoir ce sera sur le fil MISO.



MODULE WIFI

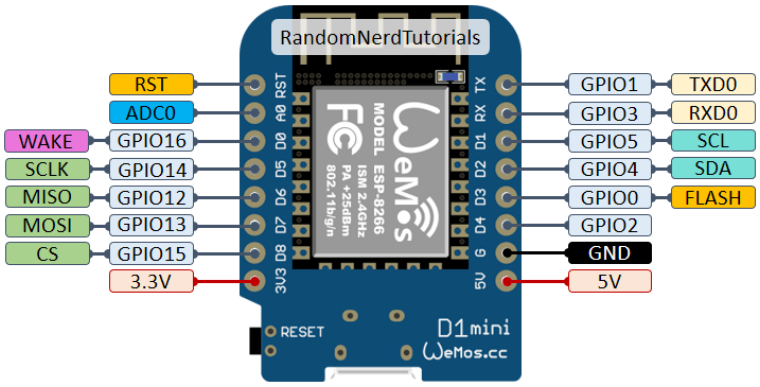
UTILISATION

Afin de consulter les différentes informations en ligne que le module Arduino (récepteur) récupère des différents capteurs, il est nécessaire d'utiliser un module Wi-Fi. Il possède son propre programme pour fonctionner et nécessite donc une installation depuis l'IDE Arduino, la nouvelle carte « Generic ESP8266 Module » afin de lui téléverser le code rédigé. Ce module peut être indépendant et faire fonctionner directement des objets à distance (exemple : avec des LED).

JUSTIFICATION

Dans notre installation il va permettre à la ruche connectée placé hors zone de connexion d'être relié au réseau internet.

CONFIGURATION

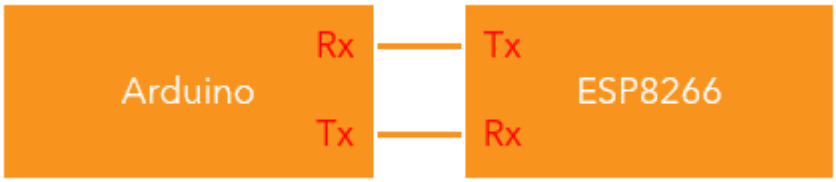


Ce module Wi-Fi peut être client ou serveur. Lorsqu'il est en mode serveur, il attend qu'un appareil se connecte à lui pour lui émettre des requêtes. Par exemple si un capteur est branché sur ce module, on pourra récupérer l'information de ce capteur. Au contraire, lorsqu'il est en mode client, il a besoin d'être connecté à un réseau pour émettre des requêtes au monde entier. Dans notre cas, on choisira le mode client pour pouvoir bénéficier des informations de la ruche, à condition d'être connecté à un réseau Wi-Fi.

COMPREHENSION

PROCESSUS DE COMMUNICATION

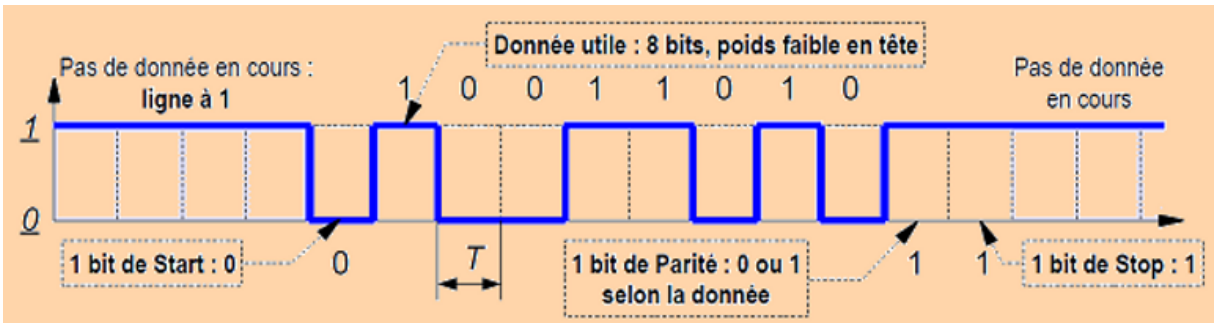
Le système de communication utilisé est la liaison série. C'est le protocole de communication le plus simple à mettre en place, ayant besoin seulement d'un fil de DATA ainsi qu'une masse commune pour échanger des données.



Pour effectuer cette communication, nous avons branché la borne Tx de l'Arduino à la borne Rx du module ESP8366. Dans notre cas l'émetteur Wi-Fi n'envoie pas de message à l'Arduino, le fil de la borne Rx de l'Arduino à la borne Tx du module Wi-Fi n'est donc pas obligatoire.

FORMAT DES DONNEES

Le protocole de cette liaison série est simple, la ligne est initialement à « 1 », puis passera à « 0 » pour signaler le début d'un message. Ensuite, seront envoyés les 8 bits du message en commençant par le bit poids faible, suivi du bit de parité et le bit de stop. Lorsque ce bit est à « 1 », cela signifie que le message est terminé et le signal restera à « 1 » jusqu'au prochain message.



PROCEDURE REELLE :

Afin que le module Wi-Fi ESP8266 puisse émettre le message à afficher sur internet, il faut tout d'abord le connecter à un réseau Wi-Fi. Pour cela, nous avons mis notre ordinateur portable en mode « point d'accès Wi-Fi ».

Pour se connecter au point d'accès wifi avec le programme, il faut utiliser la fonction « WiFi.begin(ssid, password) » accessible dans la bibliothèque « WiFiClient.h ». Remplacer le « ssid » par notre nom de connexion (PC_PIERRE) et le « password » par le mot de passe de connexion (123456789).

Maintenant connecté au réseau Wi-Fi, on veut émettre une requête au serveur créer spécialement (voir serveur chapitre suivant) pour le projet afin de lui communiquer la donnée qu'il devra stocker et afficher sur une page Web.

Pour émettre une requête, on utilise la fonction « http.begin(XXX) » accessible depuis la bibliothèque « ESP8266HTTPClient.h ». En remplaçant les « XXX » par l'URL ainsi que la route d'accès, ici serverName+carlu avec : http://ruche-connecte-348912.ew.r.appspot.com/sendinfo/ pour serverName et le contenu du message à afficher pour carlu.

SERVEUR

UTILISATION

L'utilisation d'un serveur est nécessaire pour stocker une valeur et l'afficher sur une page web. Nous avons donc créé un serveur en NodeJS que l'on a fait héberger sur des serveurs Google.

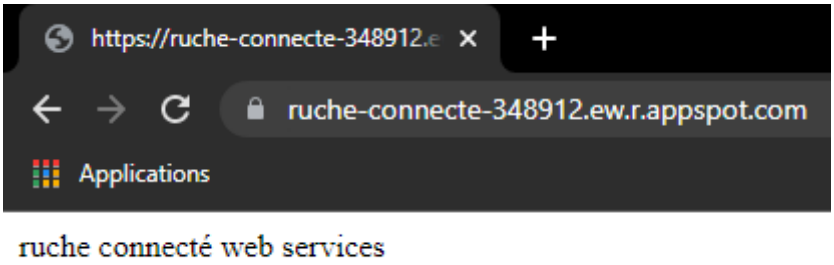
JUSTIFICATION

Nous avons choisi de faire stocker le serveur avec Google car il est simple d'utilisation et il à prix raisonnable pour un petit programme comme le nôtre.

FONCTIONNEMENT

En faisant héberger notre serveur par google une adresse, nous a était donnée comme URL : http://ruche-connecte-348912.ew.r.appspot.com

Lorsque l'on recherche ce site sur internet, nous arrivons simplement sur une page nous indiquant que nous sommes bien sur le bon site.

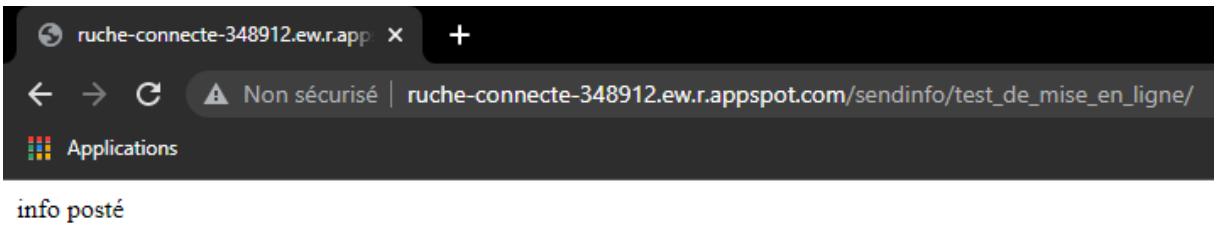


Le rôle du serveur et d'afficher différentes informations en fonction de la route d'accès écrite derrière cette adresse URL. Voici les deux routes d'accès utilisées :

- Première route : /sendinfo/ :data

Cette route permet de faire stocker à notre serveur une donnée avec « :data » étant la variable contenant notre température, humidité et notre masse.

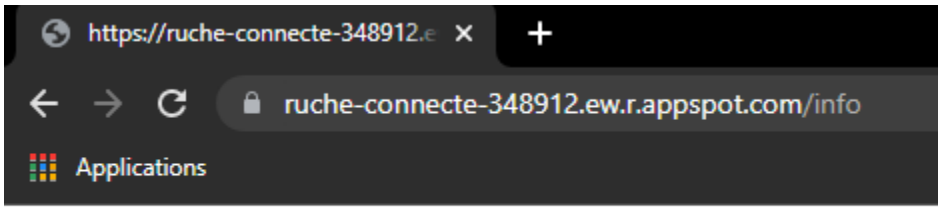
Sur la photo ci-dessous, on vérifie que lorsque l'on envoie « http://ruche-connecte-348912.ew.r.appspot.com/sendinfo/test_de_mise_en_ligne/ » le serveur répond « info posté »



- Deuxième route : /info

Cette route permet de faire afficher sur la page web la valeur de :data.

Sur la photo ci-dessous on voit que lorsque l'on envoie « https://ruche-connecte-348912.ew.r.appspot.com/info » le serveur nous répond par la variable qu'il avait stocké.



PROCEDURE REELLE :

Dans le programme de l'Arduino émetteur on a codé la ligne suivante :

String data = "humidite_"+String(h)+"_%_temperature_"+String(t)+"_degre_masse_"+String(m)+"kg}";

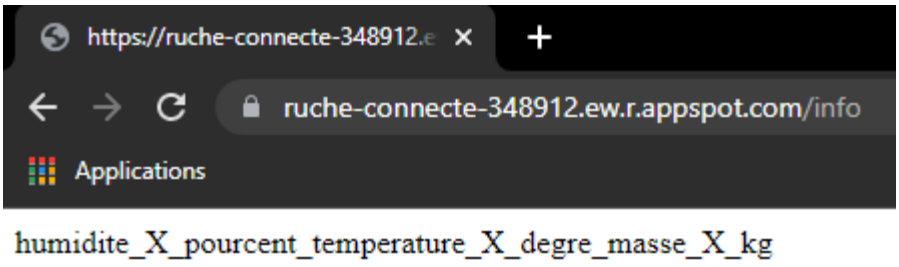
Cette ligne concatène les différentes variables « h » pour humidité, « t » pour température et « m » pour la masse. L'avantage de concaténer permet d'envoyer seulement un message au lieu de 3 messages différents.

Ce message est donc envoyé par le module LoRa de l'Arduino émetteur au module LoRa de l'Arduino récepteur pour enfin être émit par le module Wi-Fi.

L'émetteur Wi-Fi émet donc la requête :

http://ruche-connecte-348912.ew.r.appspot.com/sendinfo/humidite_pourcent_%_temperature_X_degre_masse_X_kg/

Sur la photo ci-dessous on voit l'affichage que l'on obtient lorsque le système est en route.



ANNEXE 1 : PROGRAMME ARDUINO EMETTEUR

```
#include "DHT.h"
#define DHTPIN 5
#define DHTTYPE DHT11

#include "HX711.h"
#define calibration_factor 100000.0 //Cette valeur est obtenue par calibration
#define DOUT  3
#define CLK  2

#include <SPI.h>
#include <RH_RF95.h>
RH_RF95 rf95;
HX711 scale; //Mise a 0,0kg
DHT dht(DHTPIN, DHTTYPE);

void setup() {
  Serial.begin(9600);
  dht.begin();
  if (!rf95.init())
     Serial.println("init failed");
     rf95.setFrequency(868.0);
     Serial.begin(9600);
     Serial.println("HX711 scale demo");

     scale.begin(DOUT, CLK);

     scale.set_scale(calibration_factor);
     scale.tare();

     Serial.println("Readings:");
}

void loop() {

     Serial.print("Reading: ");
     Serial.print(0.454*scale.get_units(), 1); //scale.get_units() returns a
float
     Serial.print(" kg"); //You can change this to kg but you'll need to
refactor the calibration_factor
     Serial.println();
     float h = dht.readHumidity();
     float t = dht.readTemperature();
     float f = dht.readTemperature(true);
     float m = 0.454*scale.get_units();
     if (isnan(h) || isnan(t)) {
        Serial.println("Failed to read from DHT sensor!");
        return;
}
    Serial.print("Humidity: ");
    Serial.print(h);
    Serial.print(" %\t");
    Serial.print("Temperature: ");
    Serial.print(t);
    Serial.println(" *C ");
```

```
    Serial.print("Masse: ");
    Serial.print(m);
    Serial.println(" kg ");
    //concaténation des variables afin d'avoir un seul message à envoyer
    String data =
"humidite_"+String(h)+"_%_temperature_"+String(t)+"_degre_masse_"+String(m) +
"kg}";
    int dataLength = data.length();
    dataLength++;
    uint8_t total[dataLength];
    data.toCharArray(total, dataLength);
    Serial.println(data);
    rf95.send(total, dataLength); //data envoyé
    rf95.waitPacketSent(1000);
}
```

ANNEXE 2: PROGRAMME ARDUINO RECEPTEUR

```
#include <Arduino.h>
#include <SoftwareSerial.h>

#include <SPI.h>
#include <RH_RF95.h>
#include <Wire.h>
#include <Adafruit_GFX.h>
#include <Adafruit_SSD1306.h>
#include <Fonts/FreeMonoBold12pt7b.h>
#include <String.h>

RH_RF95 rf95;
int turn;
int h, t, m;
String str;

void setup() {

Serial.begin(9600);

while (!Serial) ;
if (!rf95.init())
    Serial.println("init failed");
    rf95.setFrequency(868.0);
    pinMode(A0, INPUT);
}


void loop() {
delay(1000);
uint8_t buf[RH_RF95_MAX_MESSAGE_LEN];
uint8_t len = sizeof(buf);
if (rf95.waitAvailableTimeout(500))
   {
   if (rf95.recv(buf, &len))
      {
      String dataTotal = (char*)buf;
      str =String(dataTotal);
      Serial.println(str);
      explode(dataTotal);
      }
  else
  {
     Serial.println("recv failed");
  }
  }
}

//Fonction qui lit la donnée recu et découpe la donnée des qu'on croise un "-"
void explode(String req) {
char str[20];
req.toCharArray(str, 20);
char * pch;
pch = strtok (str, "-");
```

```
while (pch != NULL)
  {
  String sementara = pch;
  turn++;
  if (turn == 1) {
     h = sementara.toFloat();
  }
  if (turn == 2) {
     t = sementara.toFloat();
  }
  if (turn == 3) {
      m = sementara.toFloat();
  }
  pch = strtok (NULL, "-");
  delay(100);
  turn = 0;
  }
}
```

ANNEXE 3: PROGRAMME ESP8266

```
#include <SoftwareSerial.h>
#include <ESP8266WiFi.h>
#include <ESP8266HTTPClient.h>
#include <WiFiClient.h>

const char* ssid = "PC_PIERRE";
const char* password = "123456789";

String data;
int turn;
float h= 0;
float t = 0;
float m = 0;
String dataTotal;
String carlu = "";
String serverName = "http://ruche-connecte-348912.ew.r.appspot.com/sendinfo/"
;

unsigned long lastTime = 0;
unsigned long timerDelay = 5000;

void setup()
{
  Serial.begin(9600);

  WiFi.begin(ssid, password);//Connexion au reseau Wi-Fi
  Serial.println("Connecting");
  while(WiFi.status() != WL_CONNECTED)
  {
    delay(500);
    Serial.print(".");
  }
  Serial.println("");
}

void loop() {
  char monchar = '}';
  String cardispo = "";
  data = Serial.readStringUntil(monchar);//lecture port serie et stockage dans
la variable data
  carlu = data;
  Serial.println(carlu);
  Serial.println(data);
    if(WiFi.status()== WL_CONNECTED){
        WiFiClient client;
        HTTPClient http;
        String serverPath = serverName + carlu;
        Serial.println(serverPath);
        delay(5000);
        http.begin(serverPath);//Envoie de la requête
        Serial.println(carlu);
        Serial.println(data);
        delay(4000);
        // Envoie de la requête HTTP
```

```
      int httpResponseCode = http.GET();
      delay(1000);

      if (httpResponseCode>0) {
        Serial.print("HTTP Response code: ");
        Serial.println(httpResponseCode);
        String payload = http.getString();
        Serial.println(payload);
      }
      else {
        Serial.print("Error code: ");
        Serial.println(httpResponseCode);
      }
      http.end();
    }
    else {
      Serial.println("WiFi Disconnected");
    }
    lastTime = millis();
  delay(1000);

}
```

ANNEXE 4: PROGRAMME SERVEUR NODEJS

```
const express = require('express')
const app = express()
//middleware
const cors = require("cors")
let data = "0";

var corsOptions = {
    origin: '*',
    credentials: true,
    optionsSuccessStatus: 200
}

app.use(cors(corsOptions));

app.get('/', function (req, res) {
    res.send('ruche connecté web services')
    console.log("request");

})

app.get("/info", function (req, res) {

   res.send(data);
    console.log("request2");
})

app.get("/sendinfo/:data", function (req, res) {

    data = req.params.data;
    res.send('info posté');
     console.log("request3");

})

app.listen(8080, () => {
    console.log("app started");
})
```