Déploiement d'une architecture LoRa pour l'IoT

Amar ABANE & Samia BOUZEFRANE
Emails: a_abane@hotmail.fr & samia.bouzefrane@lecnam.net

Conservatoire National des Arts et Métiers
Paris

Sommaire

1. Objectif du TP
2. L’architecture globale LoRa-IoT
3. Exemple de passerelle LoRa (LG01)
4. Première phase : déploiement local
    1. Configuration de la passerelle
    2. Installation de l’environnement et des bibliothèques
    3. Premier exemple
5. Deuxième Phase : Connexion à un serveur IoT
    1. Configuration du serveur
    2. Intégration au serveur IoT
6. Aller plus loin
7. Annexe

1. Objectif du TP

Ce TP explique les étapes nécessaires à la mise en place d'une application basique de l'IoT avec la technologie LoRa. La démonstration présentée consiste à envoyer des valeurs prélevées par des capteurs vers une plateforme Web/IoT publique.
L'objectif principal est la prise en main des composants LoRa et leur utilisation concrète dans l'IoT.
Les manipulations présentées reposent sur le kit LoRa de Dragino. Cependant, l'architecture adoptée et les notions abordées restent valables pour tout autre équipement de ce type.

2. L'architecture globale LoRa-IoT

Les principaux éléments d'un déploiement IoT avec LoRa sont donnés en Fig. 1. Trois étapes de communications peuvent être distinguées globalement :

- Une communication avec la technologie LoRa permet de connecter les capteurs aux passerelles. Avec LoRa, cette communication s'effectue en un seul saut capteur-passerelle.
- Différents types de communications peuvent connecter la passerelle LoRa au serveur-IoT/Cloud. On retrouve généralement une connexion filaire Ethernet, ou sans fil avec WiFi ou 3G/4G; ces liens hétérogènes représentent la connexion Internet. Le serveur IoT stocke les données collectées par les capteurs et relayées par la passerelle. Ici, la passerelle LoRa doit disposer d'au moins 2 interfaces de communication; une radio LoRa et une interface Ethernet, WiFi ou 3G/4G.
- Enfin, les données du serveur-IoT/Cloud sont accessibles aux utilisateurs via Internet.

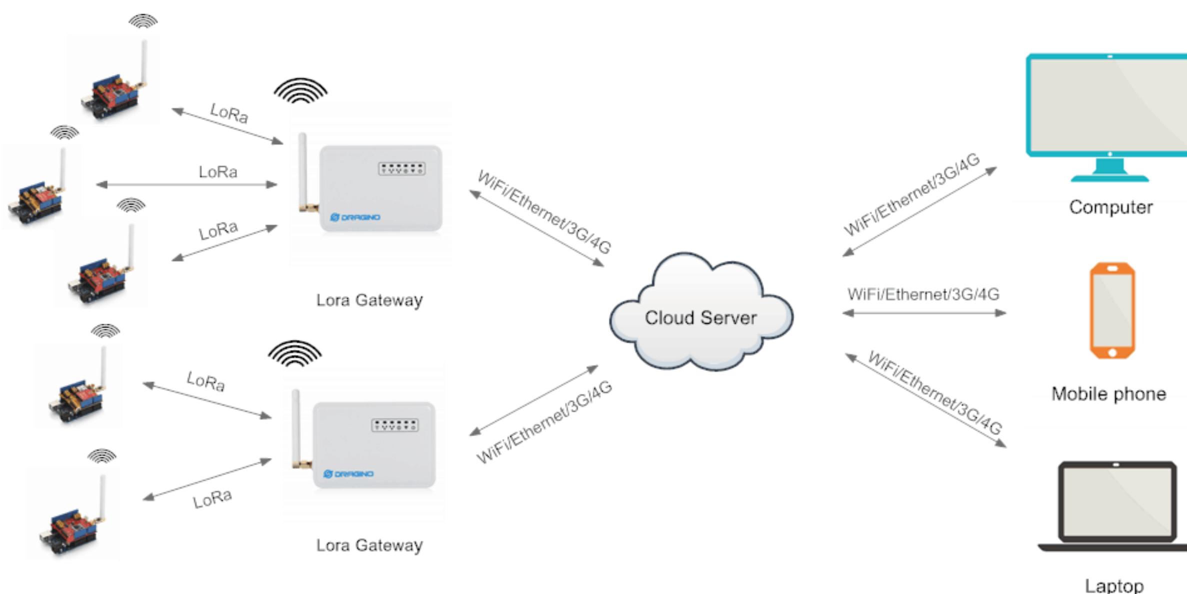


Figure 1 : Architecture LoRa-IoT type.

3. Exemple de passerelle LoRa (LG01)

Le kit utilisé dispose d'une passerelle simple et générique, contenant les composants nécessaires à la mise en place de notre architecture. La Fig. 2 représente une photo de l'intérieur de la passerelle. Elle dispose des éléments suivants :

1. Hardware
    a. Partie Linux
        i. Processeur 400 Mhz
        ii. 64 MB RAM
        iii. 16 MB Flash

b. Partie MCU
   . Microcontrôleur ATMega328P
   i. 32 KB Flash
   ii. 2 KB SRAM
2. Interfaces
   . Alimentation : 9v ~ 24v DC
   a. 2 ports RJ45
   b. 1 port USB 2.0
   c. 1 interface USB 2.0 interne
   d. 1 module WiFi (IEEE 802.11 b/g/n)
   e. 1 module LoRa
3. Software
   . Open source Linux (OpenWrt)
   a. Configuration via interface Web, SSH via LAN or WiFi.
   b. Serveur Web intégré
   c. MCU Compatible Arduino

La communication entre la partie MCU et la partie Linux se fait avec une transmission UART.

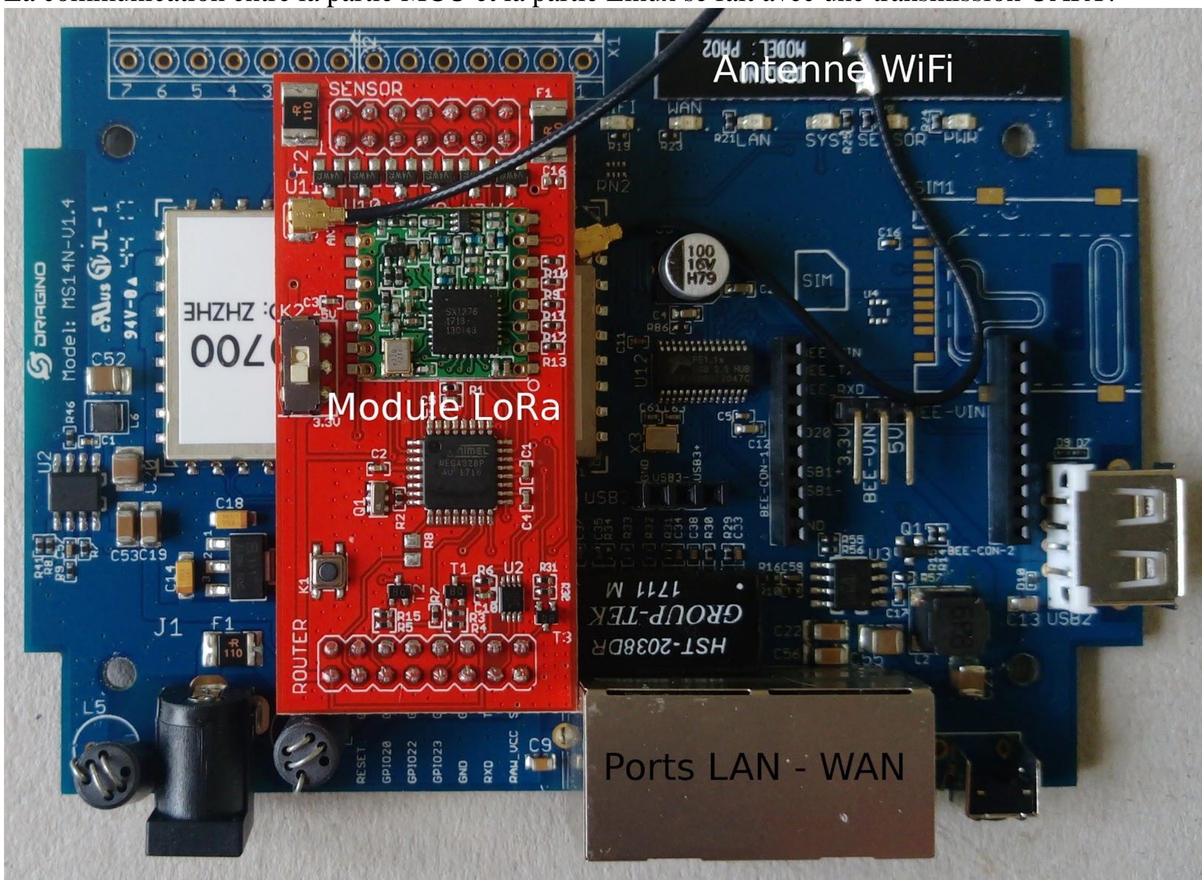


Figure 2 : Intérieur d'une passerelle LoRa.

4. Première phase : déploiement local

4.1 Configuration de la passerelle

Le type de connexion locale utilisé définit la configuration que doit avoir la passerelle. Dans notre cas, la passerelle doit pouvoir se connecter à Internet via le réseau local. Elle doit donc être configurée comme client LAN ou WiFi (Fig. 3). Nous optons pour la deuxième option.
Par défaut, la passerelle est configurée en point d'accès WiFi. Pour l'utiliser en client WiFi, allumer la passerelle et suivre les étapes suivantes :

1. Rejoindre le réseau **dragino2-xxxxxx** (sans mot de passe)
2. Accéder son interface Web via l'adresse **10.130.1.1** (username: **root**, password: **dragino**)
3. Aller dans **Network --> Internet Access**, et choisir **Access Internet via WiFi Client**, puis mettre **Way to Get IP** à **DHCP**. Entrer ensuite le SSID du réseau local, le mot de passe et le type de sécurité.
4. Dans **Network --> Access Point**, Désactiver le point d'accès WiFi.

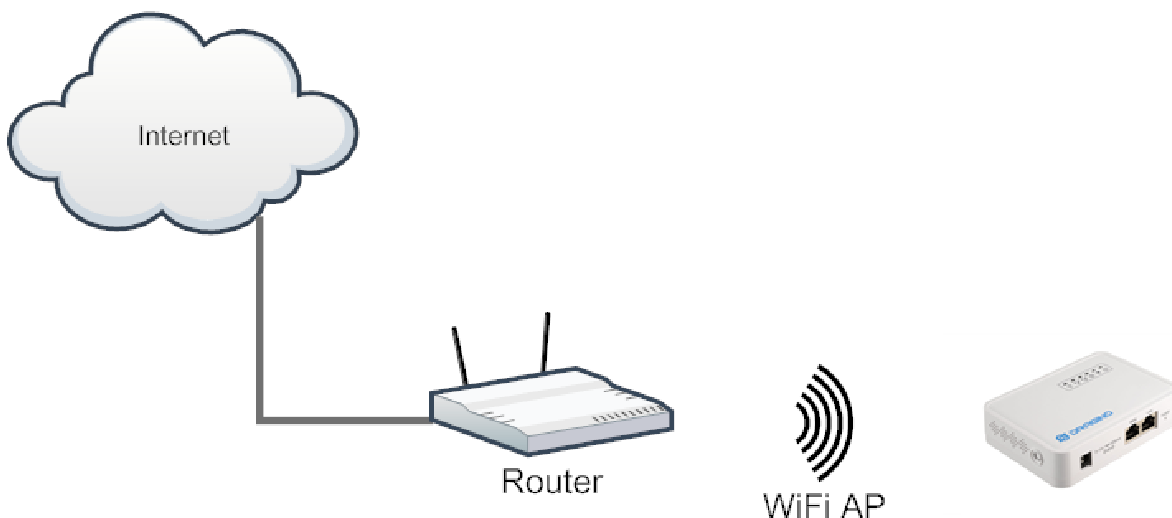


Figure 3 : Passerelle en mode client WiFi.

4.2 Installation de l'environnement et des bibliothèques

Les capteurs LoRa utilisés ainsi que la partie MCU de la passerelle sont basés sur des cartes Arduino. Il faut donc télécharger et installer la dernière version de l'IDE Arduino: https://www.arduino.cc/en/Main/Software.
De plus, le MCU de la passerelle utilise une carte qui n'est pas supportée par défaut dans l'IDE Arduino. Il faut donc ajouter et installer le gestionnaire de carte correspondant :

1. Démarrer Arduino et aller dans **File --> Preference**
2. Dans la partie **Additional Boards Manager URLs** (Fig. 4), ajouter le lien http://www.dragino.com/downloads/downloads/YunShield/package_dragino_yun_test_index.json
3. Aller dans **Tools --> Board --> Boards Manager**, chercher la carte Dragino et l'installer (Fig. 5).
4. Aller dans **Tools --> Board** et vérifier la présence de la carte **Dragino Yun-UNO or LG01/OLG01**

Remarque: Contrairement à la passerelle, les capteurs LoRa utilisés sont basés sur des cartes Arduino UNO qui sont supportées par l'IDE par défaut.

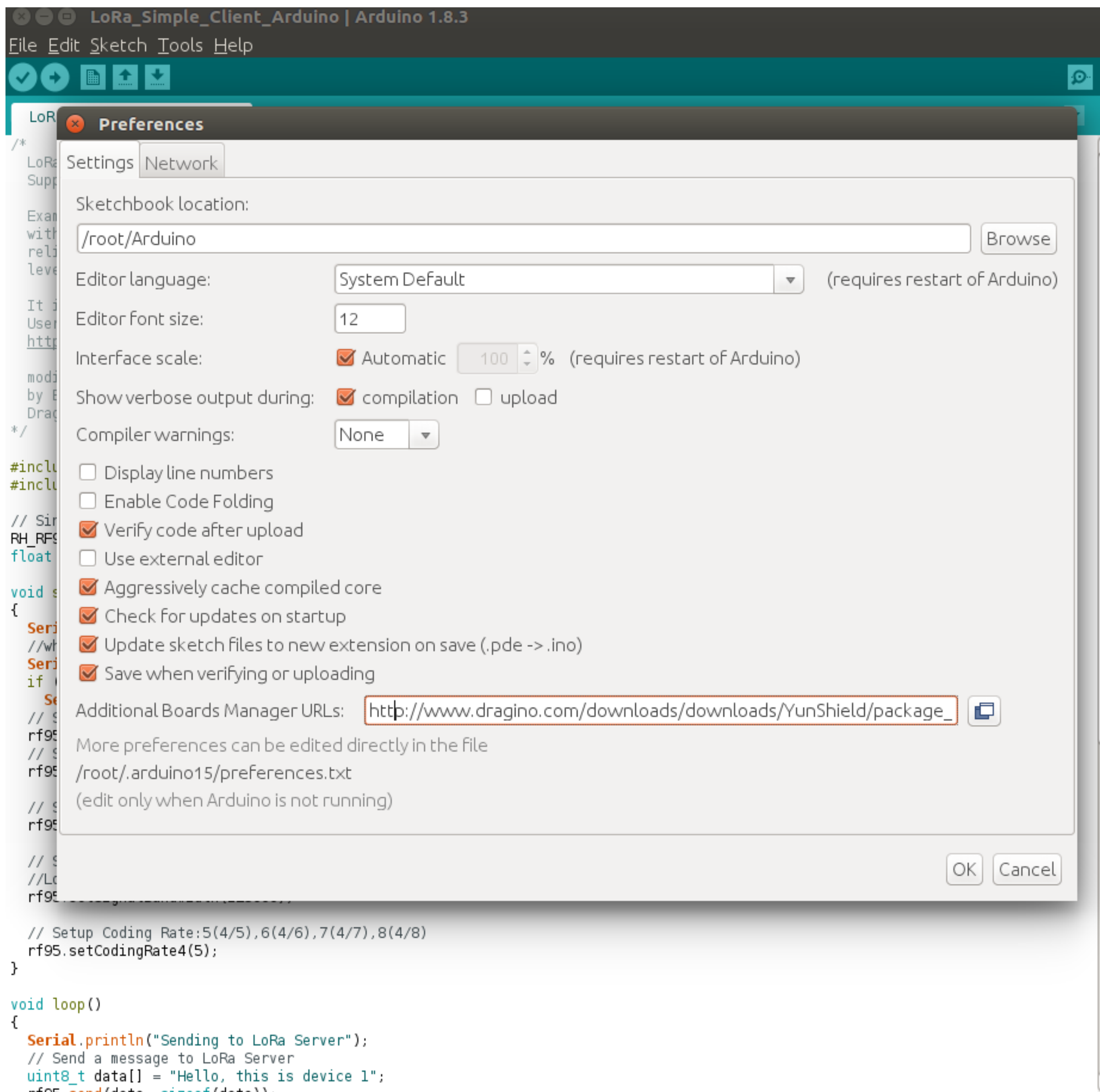


Figure 4 : Ajout d’un gestionnaire de carte.

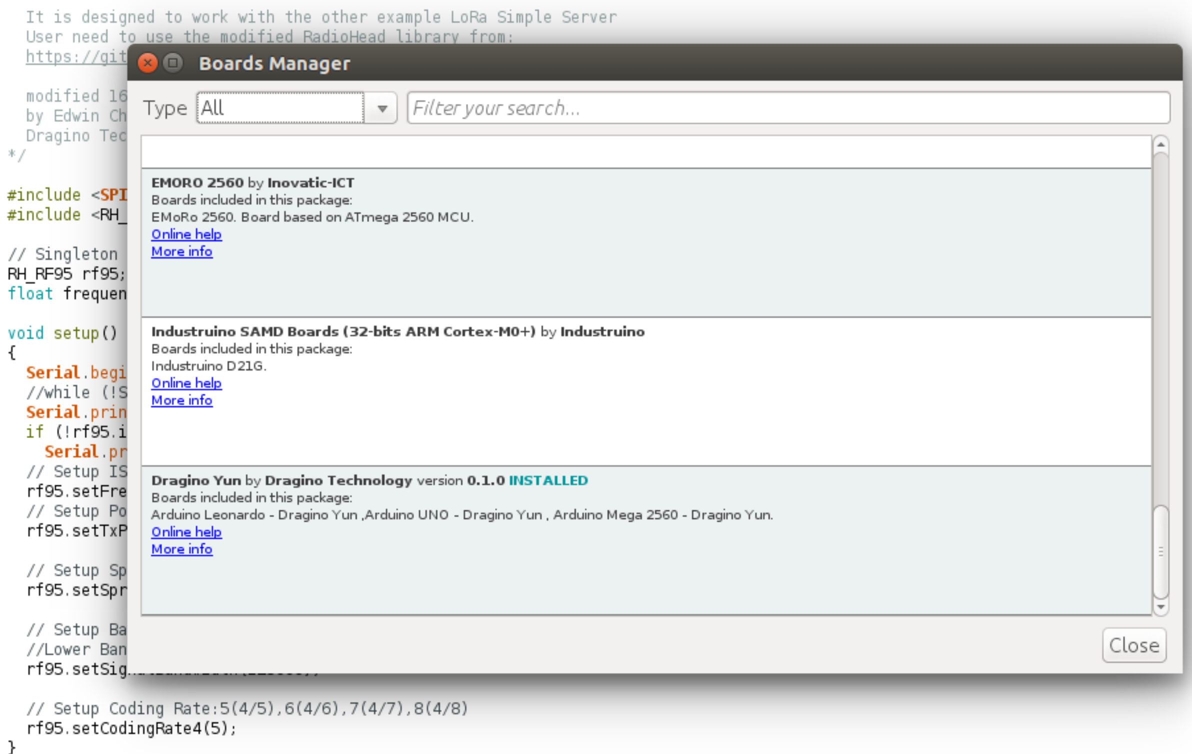


Figure 5 : Installation de la carte Dragino.

Enfin, l'utilisation du module LoRa dans les programmes Arduino nécessite de connaître un ensemble de fonctions définies dans la bibliothèque **Radiohead**. Cette bibliothèque permet au MCU de communiquer et contrôler le module LoRa pour envoyer et recevoir des paquets. Il faut donc l'installer:

1. Télécharger la bibliothèque sur https://github.com/dragino/RadioHead/archive/master.zip et la décompresser dans le répertoire **libraries** de Arduino.
2. Redémarrer l'IDE et vérifier que les exemples de la bibliothèque **Radiohead** sont bien présents dans **File --> Examples**.

Remarque : Ne pas confondre les exemples créés par la bibliothèque Radiohead et ceux créés par le gestionnaire de carte installé précédemment. Les exemples du gestionnaire de cartes (Dragino) ne sont visibles que lorsque la carte Dragino est sélectionnée.

4.3. Premier exemple

4.3.1. Le noeud LoRa

1. Brancher le capteur DHT11 à une Arduino UNO selon le schéma donné en Fig. 6.
2. Créer un nouveau sketch et copier le code **client_lora_dht11** donné en annexe.
3. Dans **Tools --> Board**, sélectionner la carte **Arduino/Genuino UNO**
4. Brancher la carte à un port USB, sélectionner le port correspondant (dans **Tools --> Port**) puis téléverser le programme.
5. Ouvrir le moniteur série (Ctrl+Shift+M)

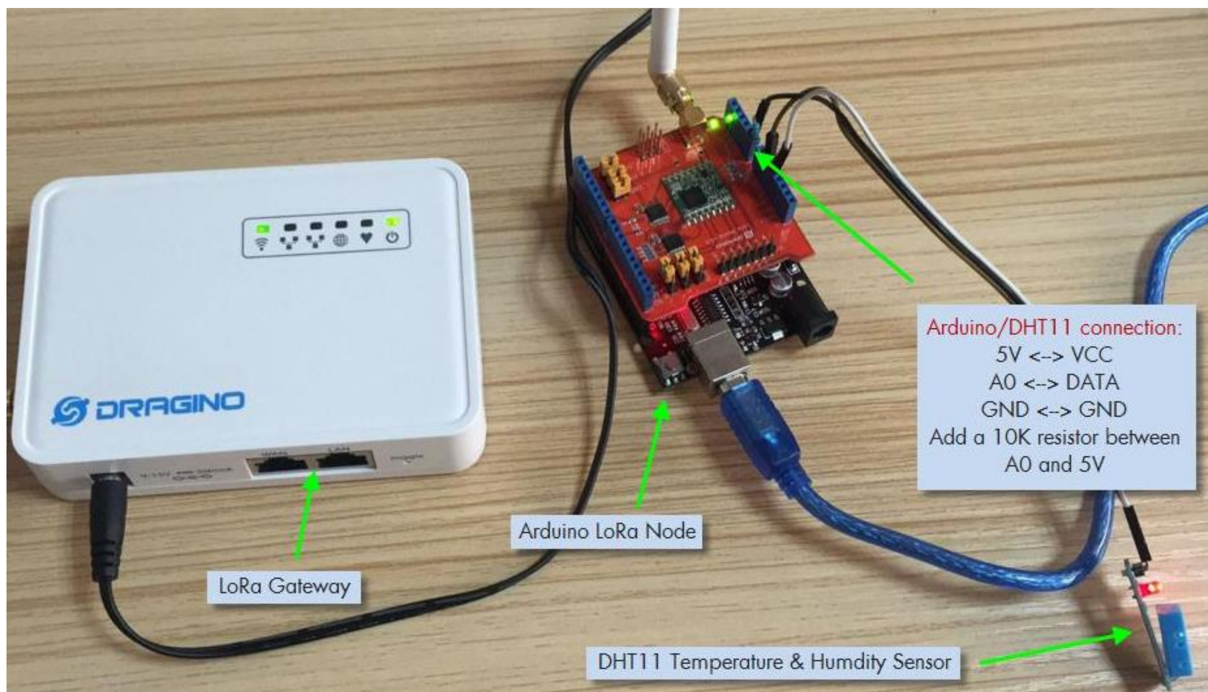


Figure 6 : Montage complet et branchement du capteur DHT11 avec la carte Arduino UNO.

4.3.2. La passerelle

Afin d'accéder à un autre moniteur série pour la passerelle, il faut lancer une nouvelle instance de l'IDE Arduino.

1. Créer un nouveau sketch et copier le code **gateway_lora_local** donné en annexe.
2. Dans **Tools --> Board**, sélectionner la carte **Dragino Yun-UNO or LG01/OLG01**.
3. Dans **Tools --> Port**, sélectionner le port de la passerelle et téléverser le programme.
4. Ouvrir le moniteur série.

Remarque 1 : Il est utile de noter la différence entre les ports utilisés pour le nœud et la passerelle. Le nœud étant connecté par USB, nous utilisons le port COM traditionnel. La passerelle étant accessible uniquement par le réseau local via WiFi, nous utilisons un port réseau détecté par l'IDE Arduino. Les deux méthodes font en réalité le même travail, seul le moyen de communication diffère.

Remarque 2 : Le nœud LoRa (client) commence à envoyer des paquets dès le démarrage, par contre la passerelle ne répondra qu'une fois le moniteur série est lancé. Ceci est dû à l'instruction **while (!Console)** qui force le programme à attendre le lancement du moniteur série.

5. Deuxième Phase : Connexion à un serveur IoT

Après avoir connecté le capteur et la passerelle LoRa, la deuxième phase consiste à connecter la passerelle à Internet afin d'envoyer les données collectées à une plateforme Web, ou serveur IoT, ou Cloud.
Plusieurs services de serveur IoT sont proposés, allant du simple stockage des données jusqu'aux outils de visualisation, d'analyse et de fouille plus au moins sophistiqués.
Nous optons ici pour **ThingSpeak**, qui propose une interface intuitive et un fonctionnement simple.
Généralement, l'envoie et l'accès aux données se fait avec une API RESTFul. Cette API utilise des requêtes HTTP (GET, PUT, POST, DELETE) pour envoyer des données au -serveur et réaliser ensuite différentes opérations.

5.1. Configuration du serveur

1. Créer un compte sur https://thingspeak.com/users/sign_up
2. Créer un nouveau canal pour contenir les données collectées. (**New channel**)
3. Renseigner les champs nécessaires et cocher une nouvelle case **Field** pour chaque type de donnée contenu dans le projet (ici, température et humidité).
4. Dans l'onglet **API Keys**, copier la clé d'écriture (**Write API Key**) qui servira à sécuriser les requêtes d'envoie de données au serveur (Fig. 7).
5. Dans la section **API Requests**, le format des liens à utiliser sont donnés pour chaque opération. Nous nous intéressons à la première (Fig. 7).
6. Les données de chaque champ crée sont accessibles dans l'onglet **Private View** (Fig. 8). Pour tester le fonctionnement des liens, utiliser un navigateur et lancer par exemple le lien suivant : https://api.thingspeak.com/update?api_key=XXXXX&field1=31 en remplaçant **XXXXX** par la clé d'écriture copiée précédemment. Dans l'onglet **Private View** la valeur 31 s'affiche dans le graphe quelques secondes après avoir lancé le lien.

Figure 7 : Interface principale ThingSpeak.

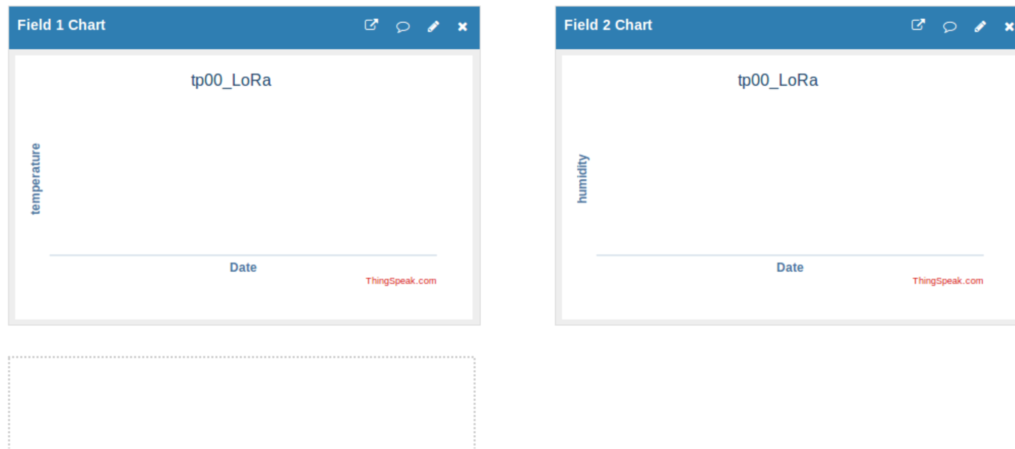


Figure 8 : Interface de visualisation des données reçues.

5.2. Intégration du montage local au serveur IoT

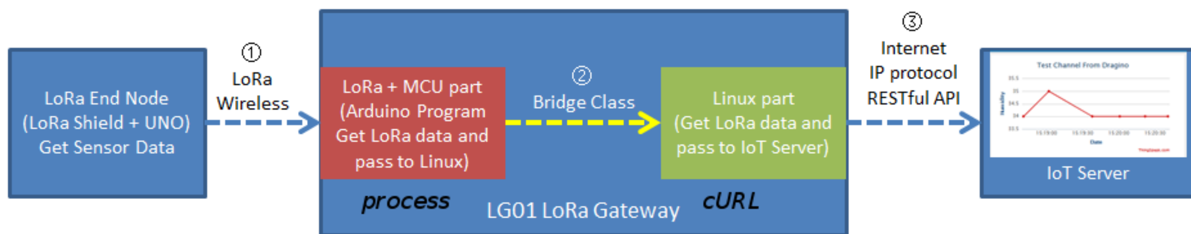
Pour envoyer des données, la passerelle effectue la même opération que la vérification de l'API réalisée sur le navigateur précédemment. La seule différence est que le lien est lancé par une commande Linux grâce au programme **curl**. Cependant, les données envoyées par le nœud LoRa sont reçues par la partie MCU de la passerelle. Il faut donc lancer la commande **curl** qui se trouve dans la partie Linux de la passerelle à partir de la partie MCU. Pour cela, la bibliothèque **Process** permet de faire des appels systèmes à partir d'un programme Arduino.

L'exemple précédent est exécuté avec la commande curl comme suit :

curl -k "https://api.thingspeak.com/update?api_key=XXXXX&field1=31"

En résumé, le processus complet est donné en Fig. 9. Le noeud LoRa envoie la valeur captée à la passerelle. La partie MCU de la passerelle reçoit cette donnée, la traite et lance la commande **curl** sur la partie Linux grâce à la classe Process. La partie Linux exécute la commande et le serveur reçoit la donnée.

Dans le montage réalisé dans la première phase, remplacer le programme de la passerelle par le code **gateway_lora_iot_server** donné en annexe, puis téléverser les sketchs.



Data Flow:
①: LoRa end node get data from sensor and send out via LoRa wireless protocol
②: LoRa/MCU part in LG01 get the sensor data from LoRa wireless. and pass the data to Linux side
③: Linux part in LG01 send the sensor data to IoT server in RESTful API format.

Figure 9 : Etapes de communication du nœud LoRa au serveur IoT.

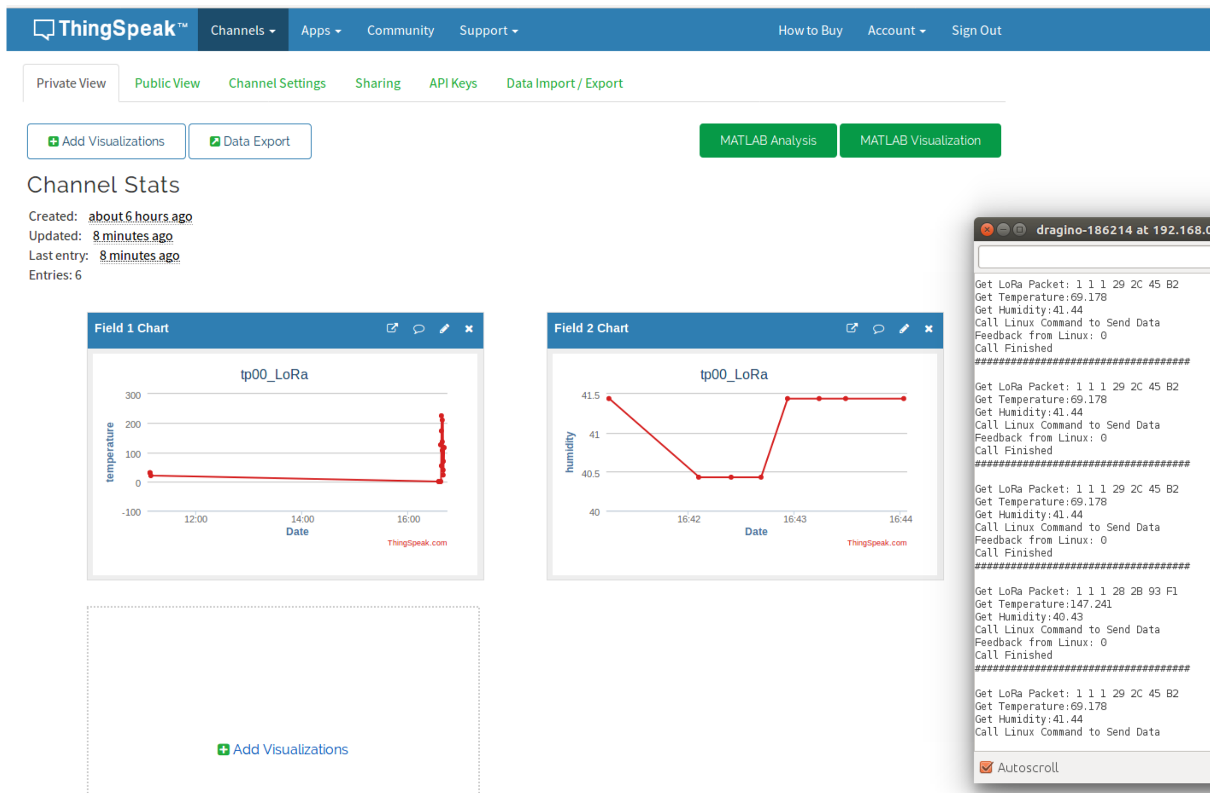
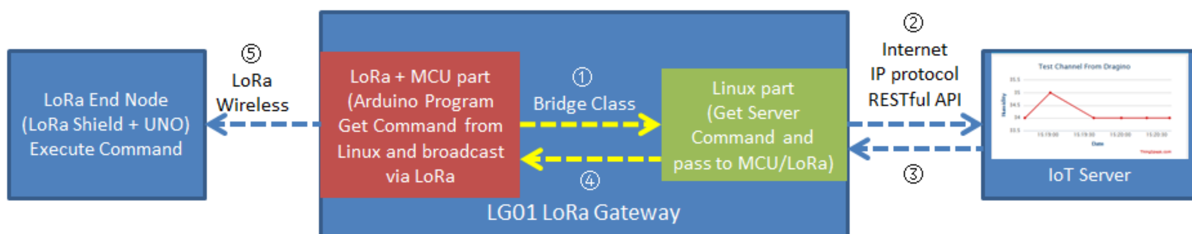


Figure 10 : Interface de visualisation des données après quelques minutes d’exécution.

6. Aller plus loin

À partir de la configuration réalisée, deux pistes intéressantes peuvent être envisagées pour rendre le scénario plus complexe :

- Exécuter une commande sur le nœud LoRa à partir du serveur IoT : En plus de visualiser et traiter les données, ThingSpeak propose un autre service permettant de créer une file de commandes côté serveur qui seront exécutées par le nœud LoRa. Le nœud jouera dans ce cas le rôle d'un actionneur, pour allumer/éteindre une lampe par exemple. Le principe de fonctionnement est similaire à celui de l'envoie de données vue précédemment. La Fig. 11 résume les étapes de communication qui réalisent cette tâche.
- Réaliser une topologie maillée avec plusieurs passerelles LoRa : Pour couvrir des surfaces plus larges, les données collectées par le nœud LoRa peuvent être envoyées sur plusieurs sauts. Pour cela, un ensemble de passerelles (clients maillés) sont placées entre la passerelle principale (passerelle maillée connectée à Internet) et les nœuds LoRa. Les clients maillés communiquent entre eux et avec la passerelle principale. La Fig. 12 donne un aperçu d'une telle topologie, et les détails de configuration sont disponibles dans la documentation du kit (http://www.dragino.com/)



Data Flow:
①: LoRa MCU part send a request to Linux side, ask the Linux side to check if there is command from IoT Server
②: Linux send this request to server via RESTful call
③: If there is new command, server send a new command to Linux
④: Linux pass this command to MCU/LoRa.
⑤: LG01 MCU part broadcast this command to its LoRa network. The LoRa end node will get this message and check if they should execute it.

Figure 11 : Exécution de commandes sur le nœud LoRa.

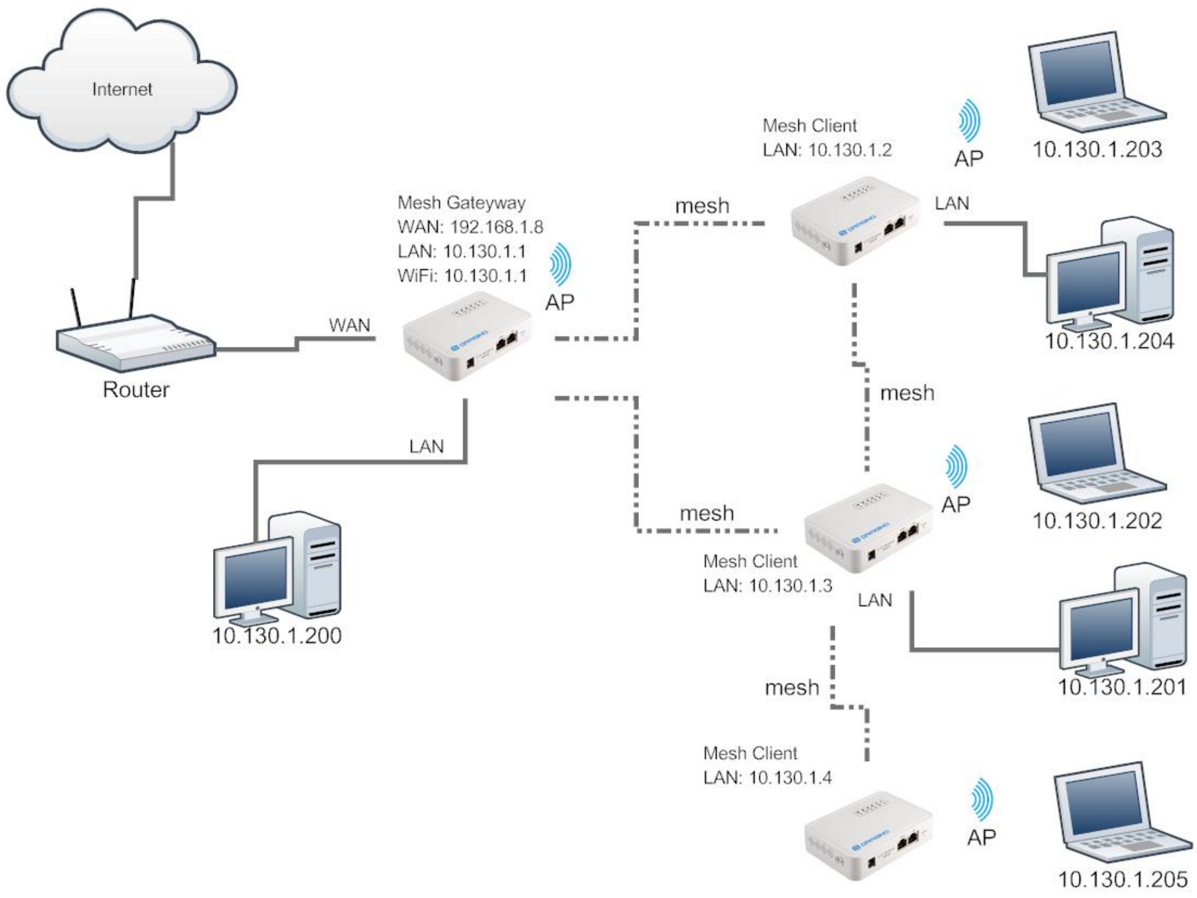


Figure 12 : Topologie LoRa maillée.

7. Annexe

7.1. Code client_lora_dht11 (à compléter)

```
#include <SPI.h>
#include <RH_RF95.h>

RH_RF95 rf95; // instancier l'objet pour communiquer

#define dht_dpin XXX // broche de lecture du capteur
byte bGlobalErr;
char dht_dat[5]; // stocker les données du capteur
float frequency = XXX;  // Fréquence du module

void setup()
{
  InitDHT();
  Serial.begin(9600);
  if (!rf95.init())
    Serial.println("init failed");
  // Paramétrer la fréquence ISM
  rf95.setFrequency(frequency);
  // Paramétrer la puissance de transmission
  rf95.setTxPower(13);
  Serial.println("Humidity and temperature\n\n");
}

void InitDHT()  // Initialise le capteur DHT11
{
  pinMode(dht_dpin,OUTPUT);
  digitalWrite(dht_dpin,HIGH);
}

void ReadDHT()  // Lit les valeur de température et d'humidité
{
  bGlobalErr = 0;
  byte dht_in;
  byte i;

  digitalWrite(dht_dpin,LOW);
  delay(30);
  digitalWrite(dht_dpin,HIGH);
  delayMicroseconds(40);
  pinMode(dht_dpin,INPUT);

  dht_in = digitalRead(dht_dpin);
  if( dht_in ){
    bGlobalErr = 1;
    return;
  }
  delayMicroseconds(80);
  dht_in = digitalRead(dht_dpin);

  if( !dht_in ){
    bGlobalErr = 2;
    return;
  }
  delayMicroseconds(80);
  for (i = 0; i < 5; i++)
    dht_dat[i] = read_dht_dat();
  pinMode(dht_dpin,OUTPUT);
  digitalWrite(dht_dpin,HIGH);
```

```
  byte dht_check_sum = dht_dat[0]+dht_dat[1]+dht_dat[2]+dht_dat[3];
  if(dht_dat[4]!= dht_check_sum)
    bGlobalErr = 3;
}

byte read_dht_dat(){
  byte i = 0;
  byte result = 0;
  for(i = 0; i < 8; i++)
  {
    while(digitalRead(dht_dpin) == LOW);
    delayMicroseconds(30);
    if (digitalRead(dht_dpin) == HIGH)
      result |=(1<<(7-i));
    while (digitalRead(dht_dpin) == HIGH);
  }
  return result;
}

uint16_t calcByte(uint16_t crc, uint8_t b)
{
  uint32_t i;
  crc = crc ^ (uint32_t)b << 8;
  for ( i = 0; i < 8; i++)
  {
    if ((crc & 0x8000) == 0x8000)
      crc = crc << 1 ^ 0x1021;
    else
      crc = crc << 1;
  }
  return crc & 0xffff;
}

uint16_t CRC16(uint8_t *pBuffer,uint32_t length)
{
  uint16_t wCRC16 = 0;
  uint32_t i;
  if (( pBuffer == 0 )||( length == 0 ))
  {
    return 0;
  }
  for ( i = 0; i < length; i++)
  {
    wCRC16 = calcByte(wCRC16, pBuffer[i]);
  }
  return wCRC16;
}

void loop()
{
  ReadDHT();
  char data[50] = {0} ;
  // Utiliser Data[0]. Data[1], Data[2] pour exprimer le deviceID.
  data[0] = X ;
  data[1] = X ;
  data[2] = X ;
  data[3] = dht_dat[0];  // ajouter la valeur de l'humidité
  data[4] = dht_dat[2];  // ajouter la valeur de température

  switch (bGlobalErr)  // traiter le retour de lecture
  {
    case 0:
      Serial.print("Humdity = ");
```

```
      Serial.print(data[3], DEC);
      Serial.print("%  ");
      Serial.print("Temperature = ");
      Serial.print(data[4], DEC);
      Serial.println("C  ");
      break;
    case 1:
      Serial.println("Error 1: DHT start condition 1 not met.");
      break;
    case 2:
      Serial.println("Error 2: DHT start condition 2 not met.");
        break;
    case 3:
      Serial.println("Error 3: DHT checksum error.");
      break;
    default:
      Serial.println("Error: Unrecognized code encountered.");
        break;
  }
  // calcul de la longueur des données à envoyer
  int dataLength = XXX;
  // calcul du CRC
  uint16_t crcData = XXX;
  unsigned char sendBuf[50] = {0};    // buffer d'envoie
  int i;
  for(i = 0; i < dataLength; i++)
    XXX;
  // Ajout de la première partie du CRC au paquet paquet LoRa
  sendBuf[dataLength] = (unsigned char)crcData;
  // Ajout de la deuxième partie du CRC au paquet paquet LoRa
  sendBuf[dataLength+1] = (unsigned char)(crcData>>8);

  rf95.send(XXX, XXX);  // envoyer le paquet LoRa
  uint8_t buf[RH_RF95_MAX_MESSAGE_LEN];  // buffer de la réponse
  uint8_t len = sizeof(buf);  // longueur de la réponse

  // attendre une reponse durant 3s
  if (rf95.waitAvailableTimeout(3000))
  {
    if (rf95.recv(XXX, &len))  // vérifier si le message est correcte
    {
    // vérifier si le message est pour le noeud
      if(buf[0] == X||buf[1] == X||buf[2] == X)
        {
        pinMode(4, OUTPUT);
        digitalWrite(4, HIGH);
        Serial.print("got reply: ");  // afficher la réponse
        Serial.println((char*)buf);
        delay(400);
        digitalWrite(4, LOW);
        Serial.print("RSSI: ");  // afficher le RSSI
        Serial.println(rf95.lastRssi(), DEC);
        }
    }else{
      Serial.println("recv failed");
        XXX;  // renvoyer le paquet
    }
  }else{
    Serial.println("No reply, is rf95_server running?");
    XXX;  // renvoyer le paquet
  }
  delay(30000);  // Lire le capteur toutes les 30 secondes
}
```

7.2. Code gateway_lora_local (à compléter)

```
#include <SPI.h>
#include <RH_RF95.h>
#include <Console.h>
#define BAUDRATE 115200
RH_RF95 rf95;

uint16_t crcdata = 0;
uint16_t recCRCData = 0;
float frequency = XXX;

void setup()
{
  Bridge.begin(BAUDRATE);
  Console.begin();
  if (!rf95.init())
    Console.println("init failed");
  rf95.setFrequency(frequency);
  rf95.setTxPower(13);
  Console.println("LoRa Local Gateway Example");
}

uint16_t calcByte(uint16_t crc, uint8_t b)
{
  uint32_t i;
  crc = crc ^ (uint32_t)b << 8;
  for ( i = 0; i < 8; i++)
  {
    if ((crc & 0x8000) == 0x8000)
      crc = crc << 1 ^ 0x1021;
    else
        crc = crc << 1;
  }
  return crc & 0xffff;
}

uint16_t CRC16(uint8_t *pBuffer, uint32_t length)
{
  uint16_t wCRC16 = 0;
  uint32_t i;
  if (( pBuffer == 0 ) || ( length == 0 ))
    return 0;
  for ( i = 0; i < length; i++)
  {
    wCRC16 = calcByte(wCRC16, pBuffer[i]);
  }
  return wCRC16;
}

uint16_t recdata(unsigned char* recbuf, int Length)
{
  crcdata = CRC16(recbuf, XXX);  // calculer le CRC du paquet
  recCRCData = recbuf[XXX];     // récupérer le CRC reçu
  recCRCData = recCRCData << 8;
  recCRCData |= recbuf[XXX];
}

void loop()
{
```

```
  if (rf95.waitAvailableTimeout(2000))  // écouter si un noeud transmet
  {
    uint8_t buf[RH_RF95_MAX_MESSAGE_LEN];  // buffer de reception
    uint8_t len = sizeof(buf);
    if (rf95.recv(XXX, XXX))  // Vérifier si un paquet est arrivé
    {
      recdata(XXX, len);
      Console.print("Get LoRa Packet: ");
      for (int i = 0; i < len; i++)
      {
        Console.print(buf[i],HEX);
        Console.print(" ");
      }
      Console.println();
      if(XXX == XXX)   // Vérifier le CRC
      {
        // Vérifier si c'est le noeud attendu
        if(buf[0] == X||buf[1] == X||buf[2] == X)
        {
          uint8_t data[] = "XXX";  // ACK de la passerelle
          data[0] = X;
          data[1] = X;
          data[2] = X;
          rf95.send(XXX, XXX);  // Send Reply to LoRa Node
          rf95.waitPacketSent();
        }
      }else
        Console.println(" CRC Fail");
    }
  }
}
```

7.3. Code gateway_lora_iot_server

Pour obtenir le programme correspondant, il faut compléter les blocs d'instructions suivants et les placer au bon endroit dans le code donné précédemment (7.2).

```
1: #include <Process.h>

2: String myWriteAPIString = "XXX";

3: String dataString = "";

4: Console.println("LoRa Gateway to ThinkSpeak");

5: int humidity = newData[0];
   int temperature = newData[1];

6: dataString ="XXX";
   dataString += XXX;
   uploadData();
   dataString = "";

7: void uploadData()
   {
     String upload_url = "XXX";
     upload_url += XXX; // ajouter la clé d'ecriture API
     upload_url += "XXX";
     upload_url += XXX; // ajouter les données

     Console.println("Call Linux Command to Send Data");
```

```
  Process p;  // utiliser la class Process
  p.begin("XXX"); // préparer la commande cURL
  p.addParameter("XXX");  // ajouter le paramètre
  p.addParameter(XXX);  // ajouter l'URL
  p.run();    // lancer la commande cURL
  Console.print("Feedback from Linux: ");
  // afficher la réponse de Linux
  while (p.available() > 0)
  {
    char c = p.read();
    Console.write(c);
  }
  Console.println("");
  Console.println("Call Finished");
  Console.println("####################################");
  Console.println("");
}
```