ENSEIRB-MATMECA

CONCEPTION D'OBJETS CONNECTES PAR PROTOTYPAGE RAPIDE

Patrice KADIONIK
kadionik.enseirb-matmeca.fr

TABLE DES MATIERES

1. BUT DES TRAVAUX PRATIQUES

Ces Travaux Pratiques ont pour but de présenter la conception d'objets connectés par prototypage rapide.

Comme nous avons pu le voir en cours, le logiciel libre et le matériel libre sont les composants de base essentiels dans cette conception.

Nous n'allons pas réaliser la conception matérielle d'un objet connecté, nous partirons donc de matériels conçus pour le prototypage rapide et nous nous attacherons à la conception logicielle.

Les objets connectés contiennent généralement un système d'exploitation. On retrouve dans l'immense majorité des cas un socle Linux embarqué. Il est possible d'utiliser ce socle avec un langage de programmation comme C ou Python mais on peut aussi trouver au-dessus une machine virtuelle comme Java ou même Android.

Linux est donc incontournable dans ce domaine. Il est aussi possible d'avoir un système d'exploitation Temps Réel comme FreeRTOS (logiciel libre) très largement employé dans la conception d'objets connectés voire même du pur langage C embarqué (*bare metal*).

Les TP se feront autour d'une carte Raspberry Pi complétée d'une carte fille ou « *HAT* » incluant différents capteurs : le *Sense HAT*. Celui-ci contient des capteurs de pression, température, humidité relative et de mesure d'accélération ainsi qu'un *joystick* et un gyroscope.
Nous nous limiterons aux 2 premiers capteurs de la liste.

Nous allons balayer la conception d'objets connectés en utilisant une distribution pour Raspberry Pi (distribution *Raspbian*) et développer les applications logicielles en langage Python principalement.

Un premier TP est consacré à la mise en œuvre de Python et du *Sense HAT*.

Un deuxième TP est consacré à la mise en œuvre du protocole MQTT devenu standard de fait dans l'Internet des objets (IoT).

Enfin, un troisième et dernier TP est consacré à LoRaWAN utilisé dans une infrastructure bâtie autour du réseau communautaire TTN afin de concevoir un objet connecté LoRa. Pour cela, la carte Raspberry Pi sera complétée d'un deuxième *HAT* possédant une interface LoRa.

Mots clés : objet connecté, Raspberry Pi, Sense HAT, Draguino, ARM, Linux embarqué, Raspbian, langage C, langage Python, LoRa, LoRaWAN, TTN, MQTT, Cayenne

2. PRESENTATION DE L'ENVIRONNEMENT

2.1. Carte cible Raspberry Pi

La carte Raspberry Pi ou RPi est une carte bon marché largement utilisée pour le DIY (*Do It Yourself*) afin de développer de petits systèmes embarqués ou des objets connectés. Le modèle mis en œuvre dans ces TP est la carte Raspberry Pi 3B+.

Notre carte RPi possède ainsi les éléments suivants :

- Un SoC (*System on Chip*) Broadcom BCM2837 avec un processeur quadricœur ARM Cortex-A53 à 1,2 GHz.
- 1 Go de RAM.
- 4 ports USB.
- Sortie vidéo HDMI.
- Sortie audio HDMI et Jack 3,5 mm.
- Support microSD.
- Ethernet 10/100 Mb/s, Wifi 802.11n et Bluetooth 4.1.
- 7 E/S GPIO, 1 UART, 1 bus I2C et 1 bus SPI.

L'image suivante présente la carte cible RPi 3B+ :

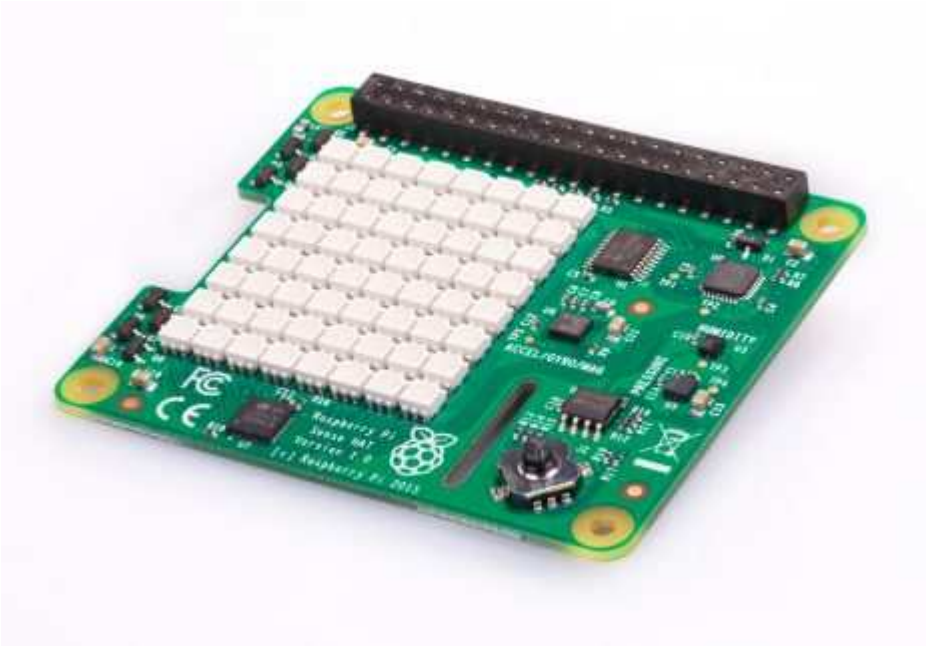
Carte cible RPi 3B+

La carte RPi 3B+ est complétée d'un *HAT*, le *Sense HAT* pour rajouter différents capteurs et un affichage.

La carte *Sense HAT* possède ainsi les éléments suivants :

- 1 gyroscope.
- 1 accéléromètre.
- 1 capteur d'accélération (2/4/8/16 g).
- 1 magnétomètre (4/8/12/16 Gauss).
- 1 baromètre (260 - 1260 hPa).
- 1 capteur de température (0-65°C).
- 1 capteur d'humidité relative (20-80% rH).
- 1 affichage par leds avec une matrice de 8x8.
- 1 *joystick* 5 boutons.

L'image suivante présente la carte *Sense HAT* :



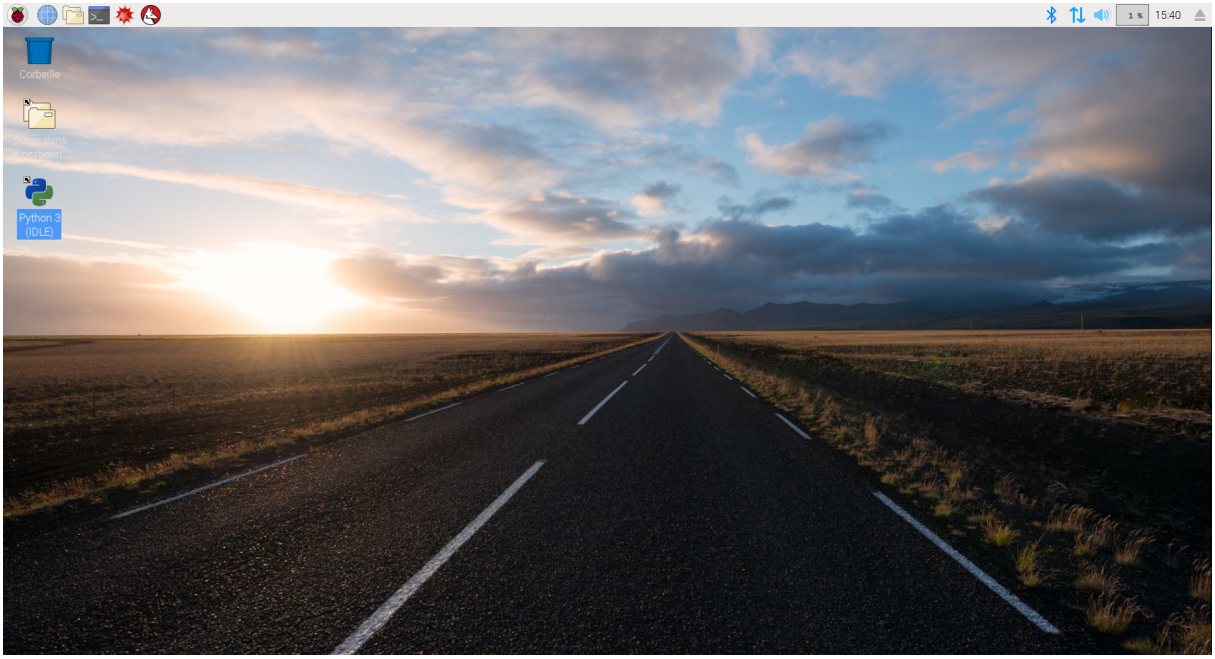
Carte *Sense HAT*

2.2. Distribution Linux embarqué Raspbian

Nous allons utiliser une distribution Linux officielle de la carte cible Raspberry Pi comprenant tous les outils nécessaires pour un développement natif directement sur la carte RPi. Pour cela, la distribution *Raspbian* est utilisée et a été installée et configurée sur la carte microSD de la carte RPi.

Nous utiliserons le langage C mais aussi le langage Python plus adapté à un contexte de prototypage rapide.

La distribution *Raspbian* propose un environnement graphique de bureau comme le montre la figure suivante :



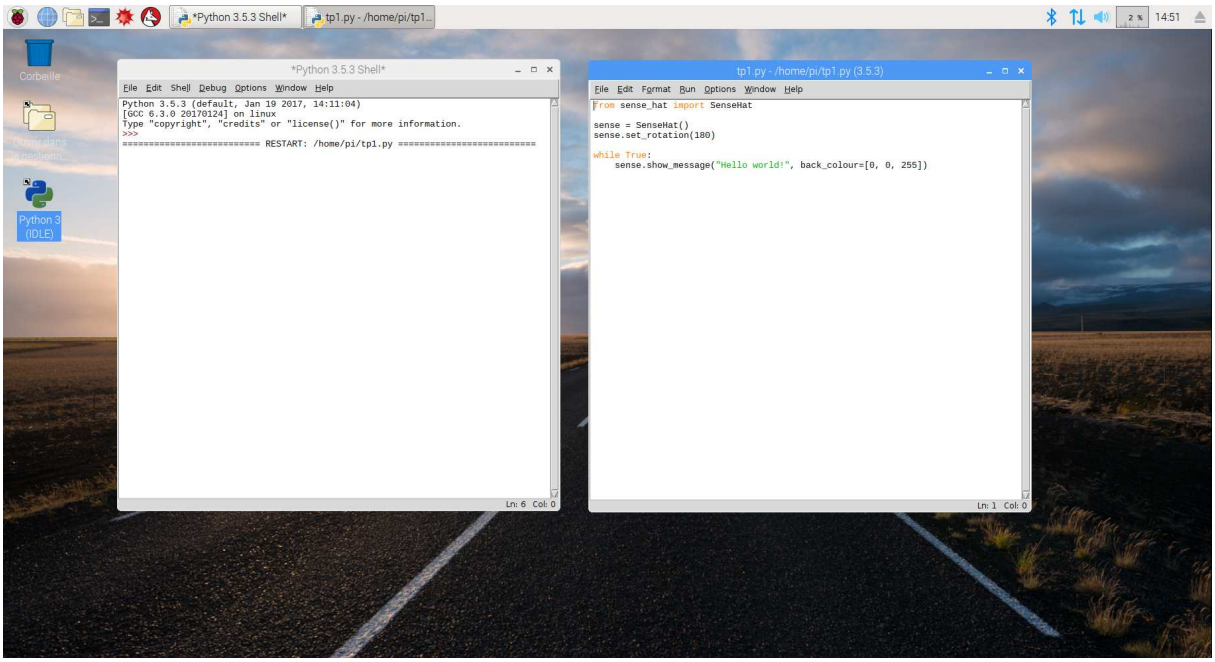
Bureau de *Raspbian*

3. TP 1 : RASPBERRY PI ET LANGAGE PYTHON

3.1. Introduction

De nombreuses bibliothèques Python permettent de développer simplement sur la carte RPi. Il existe par exemple la bibliothèque `SenseHat` pour piloter très simplement la carte *Sense HAT*.

Si l'on clique sur l'icône IDLE Python 3 (icône sur le bureau), on a alors accès à un IDE (*Integrated Design Entry*) de développement Python. On peut éditer un fichier Python (menu *File*), exécuter un fichier Python (menu *Run* ou touche F5), relancer l'interpréteur Python (CTRL F6)…



IDE IDLE 3 (pour Python 3)

L'IDE IDLE 3 est simple et intuitif. On pourra se référer aux annexes 1 et 2 pour avoir un résumé du langage Python par ailleurs déjà étudié en classes préparatoires aux grandes écoles…

Dès lors, nous allons développer des programmes Python avec l'IDE IDLE 3.

Pour avoir accès à la bibliothèque `SenseHat`, le fichier source Python aura la structure suivante :

```
from sense_hat import SenseHat

sense = SenseHat()
. . .
sense.show_message("Hello world!", back_colour=[0, 0, 255])
print(("Hello world!")
. . .
print("TEMP=%02.1f" % temperature)
. . .
```

L'API de la bibliothèque `SenseHat` est décrite dans les annexes 3 et 4.

On peut alors avoir accès à l'afficheur matriciel à leds et aux capteurs de la carte *Sense HAT*…

3.2. EX 1 : application Hello World

Nous allons écrire en langage Python la célèbre application « *Hello World!* ».

- Se créer un répertoire de travail à son nom et s'y placer :
```
host% cd
host% mkdir mon_nom
host% cd mon_nom
```

- Ecrire en langage Python le programme `tp1.py` d'affichage à l'écran et sur l'afficheur matriciel à leds de *Hello World!* ». Tester.

3.3. EX 2 : capteurs de température et de pression

- Ecrire en langage Python le programme `tp2.py` d'affichage à l'écran de la pression et de la température de la carte *Sense HAT*. Afficher sur l'afficheur matriciel à leds la température. On affichera les valeurs avec un « chiffre après la virgule » (à 10^{-1} près). Tester.

3.4. EX 3 : *threads* et classes Python

Python implémente l'API *thread* de façon native et on a accès à l'API via la directive :
`import threading`

- Ecrire en langage Python le programme `tp3.py` de création de 2 *threads*. On définira une classe de *thread* `task_affiche` qui dans sa méthode `run` affiche toutes les 2 secondes le nom de l'objet créé à partir de cette classe. Au bout de 10 secondes, les 2 *threads* seront arrêtés. Tester.

La structure générale de la classe `task_affiche` et donc du programme `tp3.py` est :

```
from sense_hat import SenseHat
from time import *
import threading

sense = SenseHat()

class task_affiche(threading.Thread):
    def __init__(self, nom = ''):
        threading.Thread.__init__(self)
        self.nom = nom
        self.Terminated = False

    def run(self):
        while not self.Terminated:
            . . .

    def stop(self):
        self.Terminated = True

t1 = task_affiche("Task t1")
t1.start()
```

3.5. EX 4 : *threads* et capteurs

- Ecrire en langage Python le programme `tp4.py` de création de 2 *threads*. La classe de *thread* `task_pression` pour le capteur de pression et la classe de *thread* `task_temp` pour le capteur de température seront créées. La pression sera affichée à l'écran toutes les 2 secondes tandis que la température le sera toutes les secondes.

3.6. EX 5 : protocole HTTP. Miniserveur Web

Python implémente l'API *sockets* de façon native et on a accès à l'API via la directive :
`import socket`

La structure générale d'un miniserveur Web correspond à celle d'un serveur TCP qui accepte une connexion TCP entrante et qui renvoie alors une réponse HTTP forgée manuellement. La réponse se compose d'une entête HTTP classique puis d'un saut de ligne et enfin des données codées en HTML correspondant à la page d'accueil du miniserveur Web.

Le canevas en langage Python du miniserveur Web est donc le suivant (les **???** sont à remplacer par les bonnes valeurs…) :

```
from sense_hat import SenseHat
from time import *
import threading
import socket

sense = SenseHat()

socket = socket.socket(socket.AF_???, socket.SOCK_???)

socket.bind(('', ???))

socket.listen(1)

while True:
    client, address = socket.accept()

    request = client.recv(1024)

    client.send(str.encode("HTTP/1.1 ???\n"))
    client.send(str.encode("Content-Type: ???\n"))

    client.send(str.encode("\n"))

    client.send(str.encode("<CENTER><H1>Welcome to the rpi-python Web
Server</H1></CENTER>"))
    client.send(str.encode("Hello World!"))

    client.close()
```

- Ecrire en langage Python le programme `tp5.py` de création d'un miniserveur Web en écoute sur le port 8080. Tester avec le navigateur Web.

3.7. EX 6 : protocole HTTP. Miniserveur Web et capteurs

- Ecrire en langage Python le programme `tp6.py` de création d'un miniserveur Web en écoute sur le port 8080 qui renvoie la température et la pression. Tester avec le navigateur Web. On pourra rajouter la balise HTML suivante pour forcer le rafraîchissement de la page Web toutes les secondes :
  `<meta http-equiv="Refresh" content="1">`

3.8. EX 7 : miniprojet

- Ecrire en langage Python le programme `miniprojet.py` de synthèse de TP précédents.
  - On créera une classe de *thread* `task_sensor` d'acquisition des capteurs de température et de pression.
  - On créera une classe de *thread* `task_led` d'affichage de la température courante sur l'afficheur matriciel à leds.
  - On créera une classe de *thread* `task_www` d'implémentation d'un miniserveur Web qui renvoie la température courante et la pression courante. On pourra rajouter la balise HTML suivante pour forcer le rafraîchissement de la page Web toutes les secondes :
    `<meta http-equiv="Refresh" content="1">`

Une variable globale (à plusieurs *threads*) se déclare en langage Python comme suit :

```
temp = 0

class task_t1(threading.Thread):
      def __init__(self, nom = ''):
            threading.Thread.__init__(self)
            . . .
      def run(self):
            global temp
            while not self.Terminated:
                  . . .
                  temp = lecture_temp()
                  . . .
      . . .

class task_t2(threading.Thread):
      def __init__(self, nom = ''):
            threading.Thread.__init__(self)
            . . .
      def run(self):
            global temp
            while not self.Terminated:
                  while (True):
                        client, address = self.socket.accept()
                        . . .
                        client.send(str.encode("TEMP=%02.1f °C" % temp))
                        . . .
      . . .
. . .
```

- Tester avec le navigateur Web.

3.9. Conclusion

Nous avons pu voir la mise en œuvre de Python avec la distribution Linux *Raspbian* sur la carte RPi.

Nous avons contrôlé les capteurs et l'affichage de la carte fille *Sense HAT* de la carte RPi.

Nous avons aussi intégré un serveur Web écrit en langage Python pour un contrôle à distance de la carte RPi par Internet.

Nous avons enfin développé une application IoT (*Internet of Things*) en langage Python pour notre objet connecté à travers le miniprojet.

Quels sont les avantages et les inconvénients d'une telle méthodologie (complexité, temps de développement…) ?

4. TP 2 : MISE EN ŒUVRE DU PROTOCOLE MQTT

Ce TP est présenté dans un autre document…

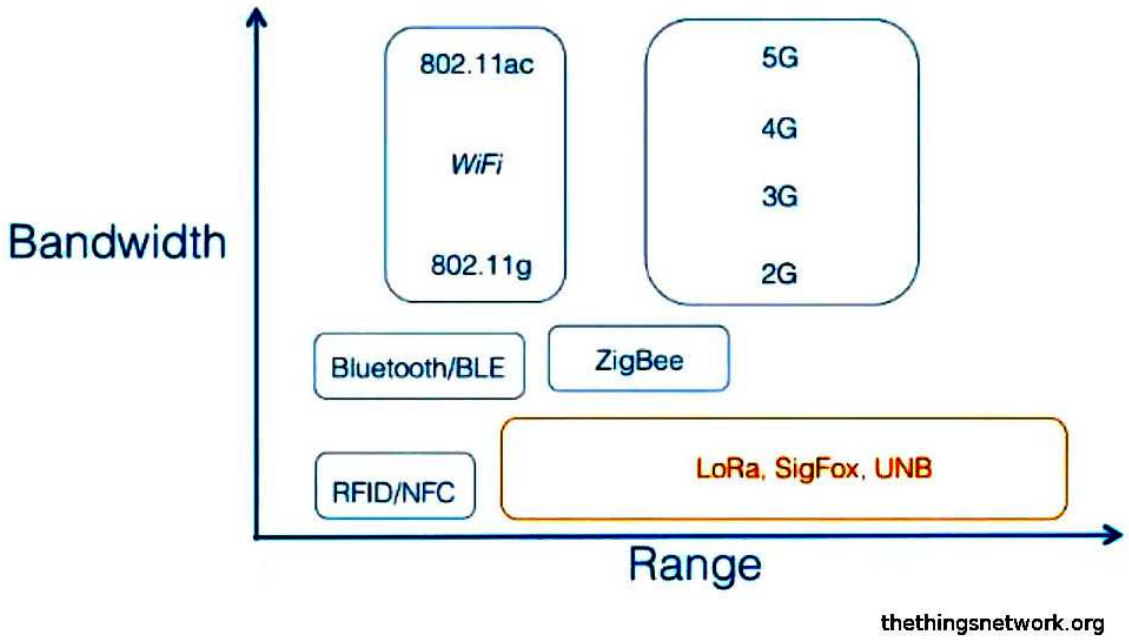
5. TP 3 : CONCEPTION D'UN OBJET CONNECTE A LORAWAN

5.1. Introduction

Créé par le consortium « LoRa Alliance », LoRaWAN (*Long Range Radio Wide Area Network*) est un protocole de communication dédié à l'Internet des objets (IoT) permettant des communications sans fil longue portée (plusieurs kilomètres), faible débit et avec une faible consommation électrique.

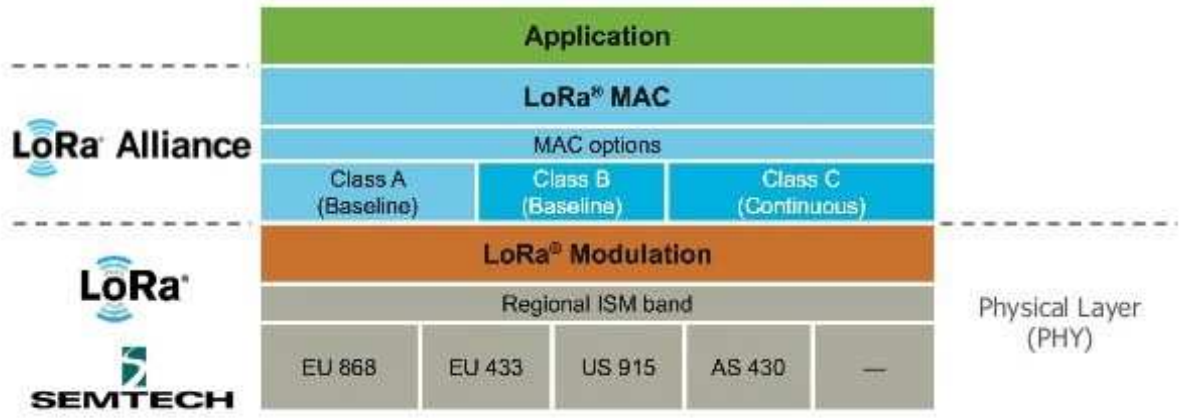
LoRaWAN est donc un réseau de type LPWAN (*Low Power Wide Area Network*) basé sur la technologie radio LoRa qui ne spécifie que le niveau physique du modèle OSI.

La figure suivante montre la place de LoRa par rapport aux autres réseaux sans fil.



LoRa dans la galaxie des protocoles sans fil

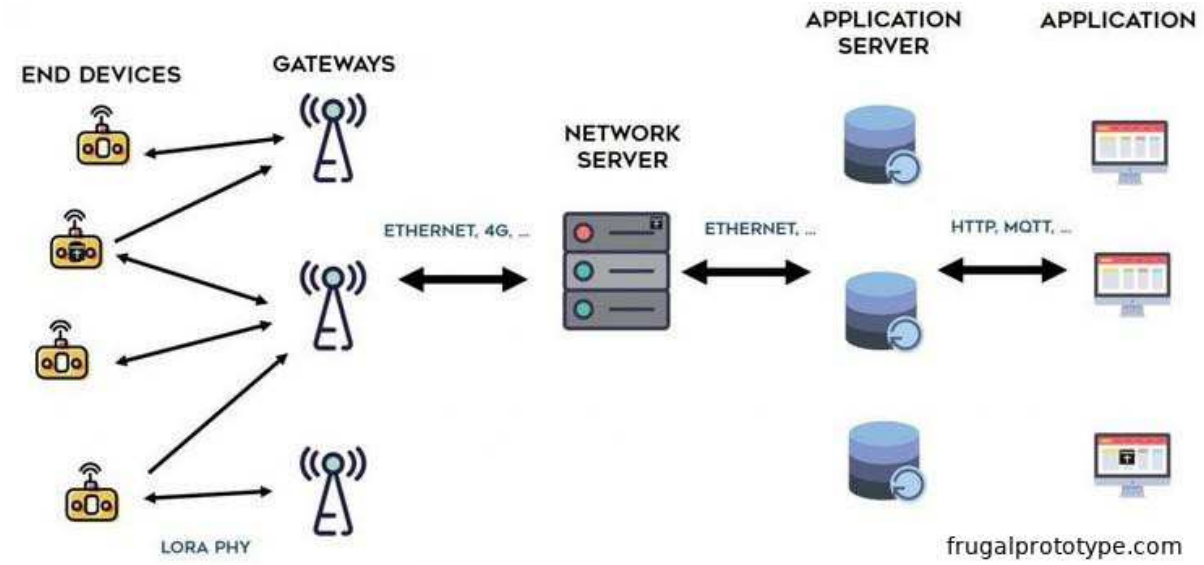
La figure suivante présente l'architecture en couches de LoRaWAN.



LoRaWAN et modèle OSI

La couche MAC de LoRaWAN gère le réseau et le transport des données.

L'architecture typique d'un réseau LoRaWAN est présentée sur la figure suivante.



Architecture d'une infrastructure LoRaWAN

On retrouve ainsi une topologie en étoile d'étoiles pour les équipements terminaux généralement des capteurs (*end device*) qui communiquent avec des passerelles (*gateway*) qui centralisent les données reçues avant de les retransmettre par Internet vers un serveur réseau (*network server*) et qui seront exploitées par des applications.

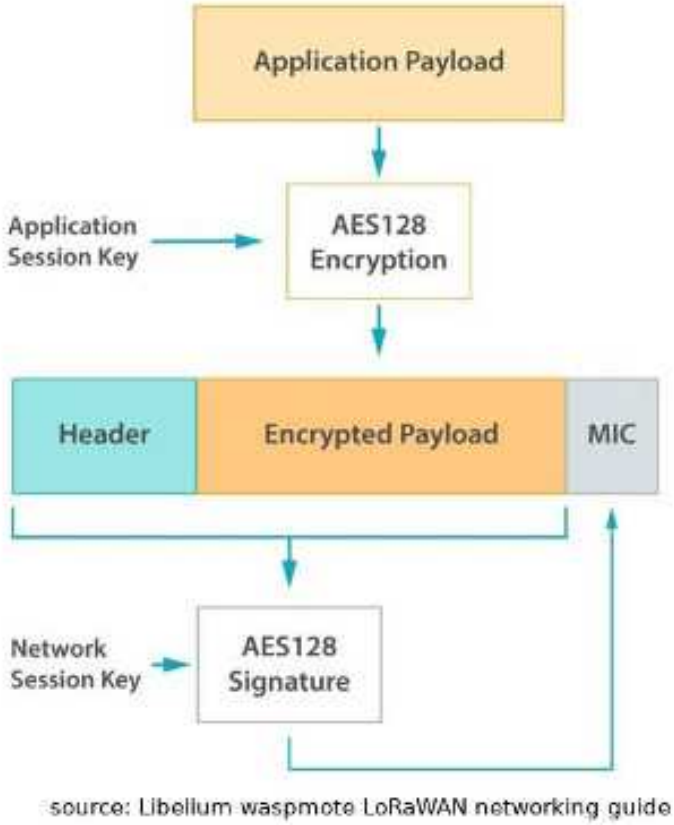
Le protocole LoRaWAN utilise plusieurs identificateurs pour les équipements du réseau :

- DevEUI : identificateur unique de l'équipement terminal (*end device*). Format EUI-64.
- AppEUI : identificateur unique de l'application. Format EUI-64.
- GatewayEUI : identificateur unique de la passerelle. Format EUI-64.
- DevAddr : adresse de l'équipement terminal. 32 bits et pas forcément unique.

Concernant la confidentialité des communications, LoRaWAN définit 3 clefs de chiffrement AES-128 (128 bits) :

- NwkSKey : clef de session réseau qui est utilisée lors des échanges entre l'équipement terminal et le réseau. Elle assure l'authenticité des équipements terminaux en calculant et en vérifiant un MIC (*Message Integrity Code*) calculé à partir de l'entête et du *payload* chiffré du message.
- AppSKey : clef de session applicative propre à un équipement terminal qui est utilisée pour chiffrer et déchiffrer le *payload*.
- AppKey : clef applicative connue seulement par l'application et par l'équipement terminal et qui permet de déduire les deux clefs précédentes.

La figure suivante montre l'usage des clés de chiffrement NwkSKey et AppSKey.



Principe du chiffrement avec les clés AppSKey et NwkSKey

Pour faire partie d'un réseau LoRaWAN, chaque équipement terminal doit avoir ses deux clefs de session NwkSKey et AppSKey. C'est l'étape d'activation. Il existe 2 types d'activation :

- Activation dynamique OTAA (*Over-The-Air Activation*).
- Activation statique ABP (*Activation By Personalization*).

Avec la méthode OTAA, l'équipement terminal transmet au réseau une demande d'accès (*Join Request*). Il envoie la requête qui contient les valeurs de DevEUI, AppEUI ainsi qu'un MIC calculé avec la clé de chiffrement AppKey. Cette requête est transmise au serveur d'enregistrement qui vérifie le MIC via la clé AppKey qu'il connaît au préalable. Si tout est conforme alors la requête d'acceptation (*Join Accept*) est envoyée à l'équipement terminal.

A partir de la réponse, l'équipement terminal va calculer les clés de session NwkSKey et AppSKey mais aussi connaître son adresse DevAddr.

L'équipement terminal a au final ses clés de session NwkSKey, AppSKey et son adresse DevAddr.

Avec la méthode ABP, les clés de session NwkSKey et AppSKey ainsi que l'adresse DevAddr sont enregistrés au préalable de façon statique dans l'équipement terminal qui peut alors directement communiquer et n'a plus besoin d'une demande d'accès préalable. C'est plus simple à mettre en œuvre mais cela reste moins solide pour la sécurité globale du réseau LoRaWAN.

5.2. Infrastructure LoRaWAN mis en œuvre et réseau TTN

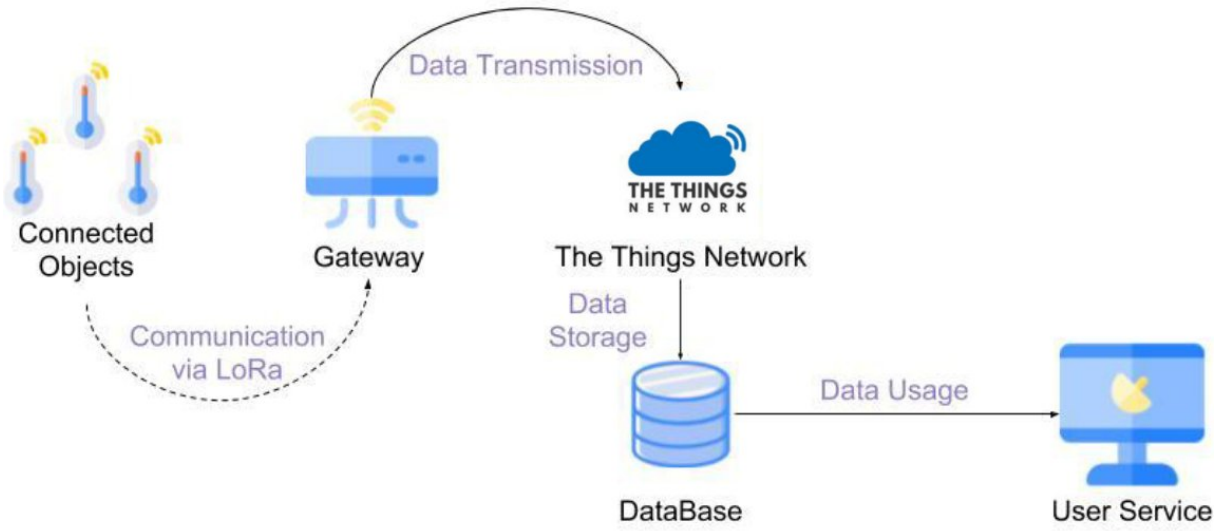
Pour ce TP, une infrastructure LoRaWAN a été déployée et est basée sur le réseau communautaire TTN (*The Things Network*) pour la partie serveur réseau et serveur d'applications.

On peut définir TTN comme « un réseau LoRaWAN communautaire et *open source* pour l'Internet des Objets. Actuellement le réseau se compose de plus de 40000 contributeurs regroupés en plus de 400 communautés dans 90 pays ayant déployé plus de 4000 passerelles. Il est possible pour les particuliers, universités, entreprises, ou encore les communes de contribuer au déploiement ou d'utiliser gratuitement The Things Network ».

On trouvera les équipements suivants :

- Les équipements terminaux (*end device*) : il s'agit de cartes RPi équipées pour ce TP d'un *HAT* supplémentaire par rapport aux TP précédents qui possède un émetteur/récepteur LoRa (carte Draguino avec un composant SX1276 RFM92 pour une modulation LoRa dans la bande ISM 868 MHz).
- Une passerelle LoRaWAN : il s'agit d'une carte RPi équipée d'un *HAT* possédant un émetteur/récepteur LoRa (carte ChisteraPi) exécutant un programme de passerelle vers le réseau TTN. Il faut noter que cette passerelle a des limitations : elle est monocanal pour LoRa et ne permet pas d'envoyer des données en aval vers les équipements terminaux. En conséquence, **on ne pourra utiliser que la méthode d'activation ABP**.
- Un serveur réseau et aussi serveur d'applications : c'est le réseau TTN.

La figure suivante montre l'infrastructure LoRaWAN déployée.



Infrastructure LoRaWAN du TP

La console de gestion du compte TTN créé pour ce TP permet de déclarer :

- Des applications LoRaWAN.
- Des passerelles LoRaWAN.
- Des équipements terminaux LoRaWAN.

Cette partie de déclaration de l'infrastructure a été réalisée au préalable. Nous allons ainsi focaliser sur l'application à déployer dans l'équipement terminal, c'est-à-dire ici la carte RPi et sur les méthodes d'exploitation des données des cartes RPi collectées par TTN.

5.3. Bibliothèque Cayenne

L'application à écrire en langage C/C++ et à compiler pour l'équipement terminal (carte RPi) utilise la bibliothèque IBM LMIC. Cette bibliothèque permet d'utiliser facilement le composant LoRa du *HAT* Draguino de la carte RPi.

LMIC met en place un environnement d'exécution de type événementiel qui permet d'émettre et recevoir des données et de gérer des temporisateurs. Cette approche est assez classique dans les réseaux de télécommunications.

La structure générale d'un programme C utilisant la bibliothèque LMIC est la suivante :

```
void main () {
  osjob_t initjob;

  // initialize run-time env
  os_init();
  // setup initial job
  os_setCallback(&initjob, initfunc);
  // execute scheduled jobs and events
  os_runloop();// (not reached)
}
```

L'environnement d'exécution est initialisé par la fonction `os_init()`et la fonction `os_runloop()` permet de lancer l'ordonnanceur de jobs à exécuter (pas de retour de cette fonction). Afin de démarrer le système, un job initial doit être généré. Ce job initial est programmé par la fonction `os_setCallback()`.

Le code du job initial défini dans la fonction `initfunc()` permet ci-après d'initialiser l'interface LoRa et de joindre ici le réseau LoRAWAN par la méthode OTAA.

```
// initial job
static void initfunc (osjob_t* j) {
  // reset MAC state
  LMIC_reset();
  // start joining
  LMIC_startJoining();
  // init done –onEvent() callback will be invoked...
}
```

La fonction `initfunc()` retourne immédiatement et la fonction de *callback* `onEvent()` sera invoquée par l'ordonnanceur d'événements sur la réception des événements OTAA `EV_JOINING`, `EV_JOINED` ou `EV_JOIN_FAILED`…

Pour ce TP, nous utiliserons la méthode d'activation ABP et non pas la méthode OTAA.

Il s'agit d'envoyer vers TTN la température et la pression mesurées par le *HAT Sense HAT*. On n'utilisera pas un encodage des données de type XML ou JSON trop verbeux pour un réseau LPWAN. On utilisera plutôt l'encodage Cayenne qui est supporté par TTN.

Cayenne permet d'optimiser la taille des données. Il est du type (Canal, Type, Valeur) et reprend la philosophie de l'encodage binaire TLV (Type, Longueur, Valeur) que l'on utilise dans les réseaux de télécommunications.

Par exemple, Type vaut 0x67 pour la température et 0x73 pour la pression. Si la température correspond au canal de données 1 et la pression au canal de données 2, alors :

- La température de 27,2 °C sera encodée en Cayenne (multiple de 0.1 °C signé sur 2 octets pour Valeur) comme : 0x01 0x67 0x0110.
- La pression de 1020 hPa sera encodée en Cayenne (multiple de 0.1 hPa non signé sur 2 octets pour Valeur) comme : 0x02 0x73 0x27D8.

On utilise une bibliothèque Cayenne en langage C++.

Les méthodes intéressantes de cette bibliothèque sont :

- `CayenneLPP lpp(uint8_t size)` : taille du buffer utilisé par Cayenne en octets.
- `lpp.reset()` : mise à zéro du buffer.
- `lpp.getSize()` : taille consommée dans le buffer.
- `lpp.getBuffer()` : pointeur sur le début du buffer.
- `lpp.addTemperature(uint8_t channel, float celsius)` : ajout d'une température `celcius` dans le canal `channel`.
- `lpp.addBarometricPressure(uint8_t channel, float hpa)` : ajout d'une pression `hpa` dans le canal `channel`.

Un exemple d'usage de Cayenne est donné ci-après :

```
CayenneLPP lpp(51);

lpp.reset();
lpp.addTemperature(1, 22.5);
lpp.addBarometricPressure(2, 1030);

LMIC_setTxData2(1, (xref2u1_t)lpp.getBuffer(), lpp.getSize(), 0);
```

Le buffer Cayenne fait 51 octets. Le buffer est mis à zéro puis on définit le canal 1 pour une température de 22,5 °C et le canal 2 pour une pression de 1030 hPa. Le buffer est ensuite émis à l'aide de la fonction LMIC `LMIC_setTxData2()`.

5.4. EX 1 : écriture de l'application de l'équipement terminal

- Se placer dans son répertoire de travail :
  ```
  host% cd
  host% cd mon_nom
  ```

- Dans son répertoire à son nom, recopier le fichier `tp-draguino.tgz` sous `~kadionik/` :
  ```
  host% cp /home/kadionik/tp-draguino.tgz .
  ```

- Décompresser et installer le fichier `tp-draguino.tgz` :
  ```
  host% tar -xvzf tp-draguino.tgz
  ```

- Se placer ensuite dans le répertoire `draguino/src/`. **L'ensemble du travail sera réalisé à partir de ce répertoire ! Les chemins seront donnés par la suite en relatif par rapport à ce répertoire...**
  ```
  host% cd draguino/src
  ```

- Editer le fichier `abp.cpp` et étudier son code. Le contenu essentiel du fichier est le suivant :

```
#include <stdio.h>
#include <time.h>
#include <wiringPi.h>
#include <lmic.h>
#include <hal.h>
#include <local_hal.h>

#include "CayenneLPP.h"
#include "pressure.c"

CayenneLPP lpp(51);

// LoRaWAN Application identifier (AppEUI)
// Not used in this example
static const u1_t APPEUI[8]  = { 0x02, 0x00, 0x00, 0x00, 0x00, 0x00, 0xFF,
0xC0 };

// LoRaWAN DevEUI, unique device ID (LSBF)
// Not used in this example
static const u1_t DEVEUI[8]  = { 0x42, 0x42, 0x45, 0x67, 0x89, 0xAB, 0xCD,
0xEF };

// LoRaWAN NwkSKey, network session key
// Use this key for The Things Network
static const u1_t DEVKEY[16] = { 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00 };

// LoRaWAN AppSKey, application session key
// Use this key to get your data decrypted by The Things Network
static const u1_t ARTKEY[16] = { 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00 };

// LoRaWAN end-device address (DevAddr)
// See http://thethingsnetwork.org/wiki/AddressSpace
static const u4_t DEVADDR = 0x????????;

. . .


static void do_send(osjob_t* j){
  int i;

  // Show TX channel (channel numbers are local to LMIC)
  // Check if there is not a current TX/RX job running
  if (LMIC.opmode & (1 << 7)) {
    fprintf(stdout, "OP_TXRXPEND, not sending");
  } else {
    // Prepare upstream data transmission at the next possible time.
    lpp.reset();

    // Acquisition température et pression
    ???

    printf("TEMP=%.1f oC\n", ???);
    printf("PRESSION=%.1f hPa\n", ???);

    // Encodage Cayenne température canal 1 et pression canal 2
    ???

    for (i=0; i < lpp.getSize(); i++) {
      printf("%02X ", *(lpp.getBuffer() + i));
    }
    puts("\n"); fflush(stdout);
```

```
    LMIC_setTxData2(1, (xref2u1_t)lpp.getBuffer(), lpp.getSize(), 0);
  }
  // Schedule a timed job to run at the given timestamp (absolute system
time)
  os_setTimedCallback(j, os_getTime()+sec2osticks(60), do_send);
}

void setup() {
  // LMIC init
  wiringPiSetup();

  os_init();
  // Reset the MAC state. Session and pending data transfers will be
discarded.
  LMIC_reset();

. . .

  // Set data rate and transmit power (note: txpow seems to be ignored by
the library)
  LMIC_setDrTxpow(DR_SF7,14);
}

void loop() {
  do_send(&sendjob);

  while(1) {
    os_runloop();
  }
}

int main() {
  setup();

  while (1) {
    loop();
  }
  return 0;
}
```

- A quoi sert la fonction `do_send()` ?

- A quoi sert la fonction `getsensors()` ? Dans quel fichier est-elle définie ? Dans quelles variables récupère-t-on la température et la pression du *Sense HAT* ?

- Avec la méthode d'activation ABP, on doit ajuster la valeur des clés de session NwkSKey et AppSKey ainsi que l'adresse DevAddr de l'équipement terminal. Quelles sont les 3 variables du programme à ajuster ? On se réfèrera à l'annexe 6 pour fixer la valeur de ces 3 variables. On pourra aussi trouver la configuration LoRa dans le fichier `config_lora.txt` dans le répertoire `draguino/src/`. On notera que le premier octet correspond au MSB.

- On désire encoder avec Cayenne la température sur le canal 1 et la pression sur le canal 2. Remplacer les **???** du programme par les bonnes expressions.

- Editer le fichier `Makefile` et comprendre son fonctionnement.

- Compiler les sources :
  `host% make`

- Lancer l'exécutable `temp` :
  `host% sudo ./temp`

- Par analyse des traces d'exécution de `temp`, vérifier la bonne lecture de la température et de la pression ainsi que leur encodage Cayenne. **On laissera le programme `temp` s'exécuter pour la suite des exercices.**

5.5. EX 2 : récupération des données avec Mosquitto

Mosquitto est un projet libre vu au TP3 qui permet d'interroger et de publier sur un *broker* MQTT. L'approche ici est en mode ligne de commandes.

Le réseau TTN possède un *broker* MQTT qui va nous permettre par souscription de récupérer les données émises par l'équipement terminal. TTN étant un réseau communautaire, il ne permet une remontée des données que toutes les 12 minutes environ. En décembre 2021, TTN est passé à la version 3 et l'infrastructure LoRaWAN de l'école a du être migrée.

Un *topic* géré par le *broker* TTN version 3 est de la forme :
`v3/AppID/devices/+/up`

Une requête Mosquitto de souscription MQTT est de la forme :

```
host% mosquitto_sub -h nom_broker -t 'topic' -u AppID -P passwd -v -d
```

- Sachant que `nom_broker` vaut `eu1.cloud.thethings.network`, forger la commande `mosquitto_sub` qui permet de s'abonner aux données émises par son objet connecté. On pourra s'aider pour cela de l'annexe 6 et du fichier `config_lora.txt`.

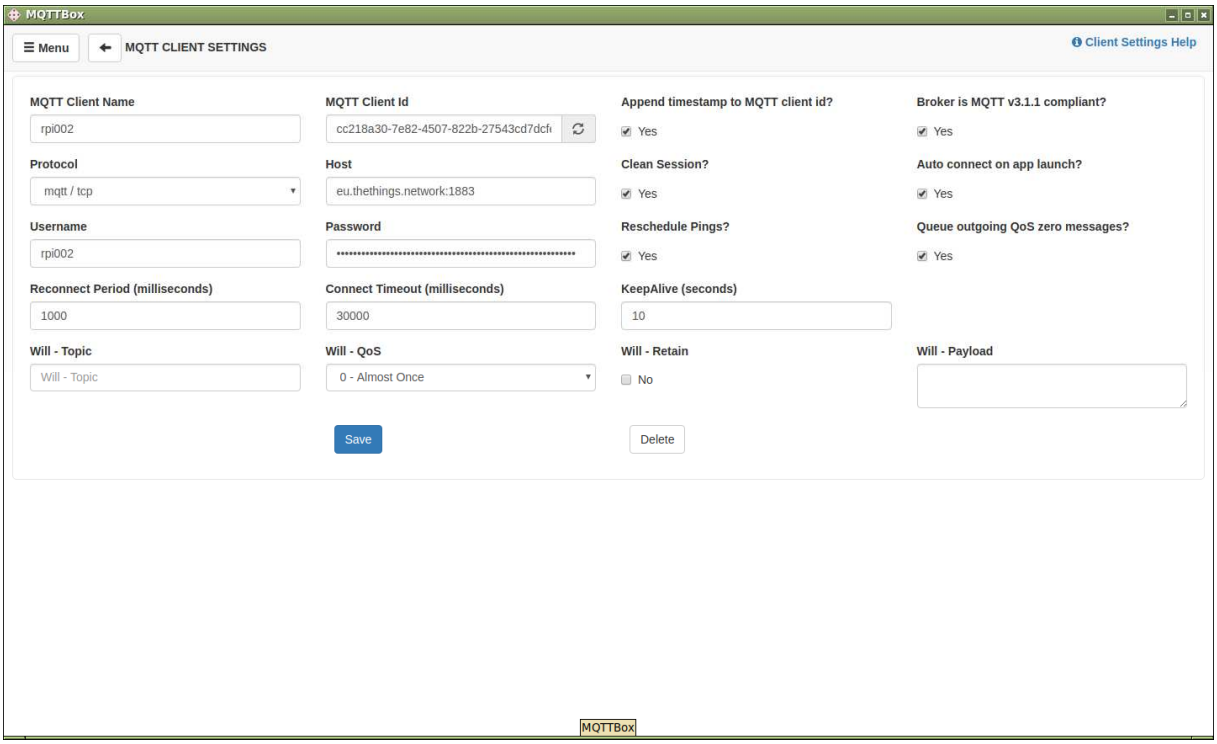
- Observer et analyser les traces MQTT capturées.

5.6. EX 3 : récupération des données avec MQTTBox

On utilisera pour cela l'application MQTTBox sous le navigateur Google Chrome.

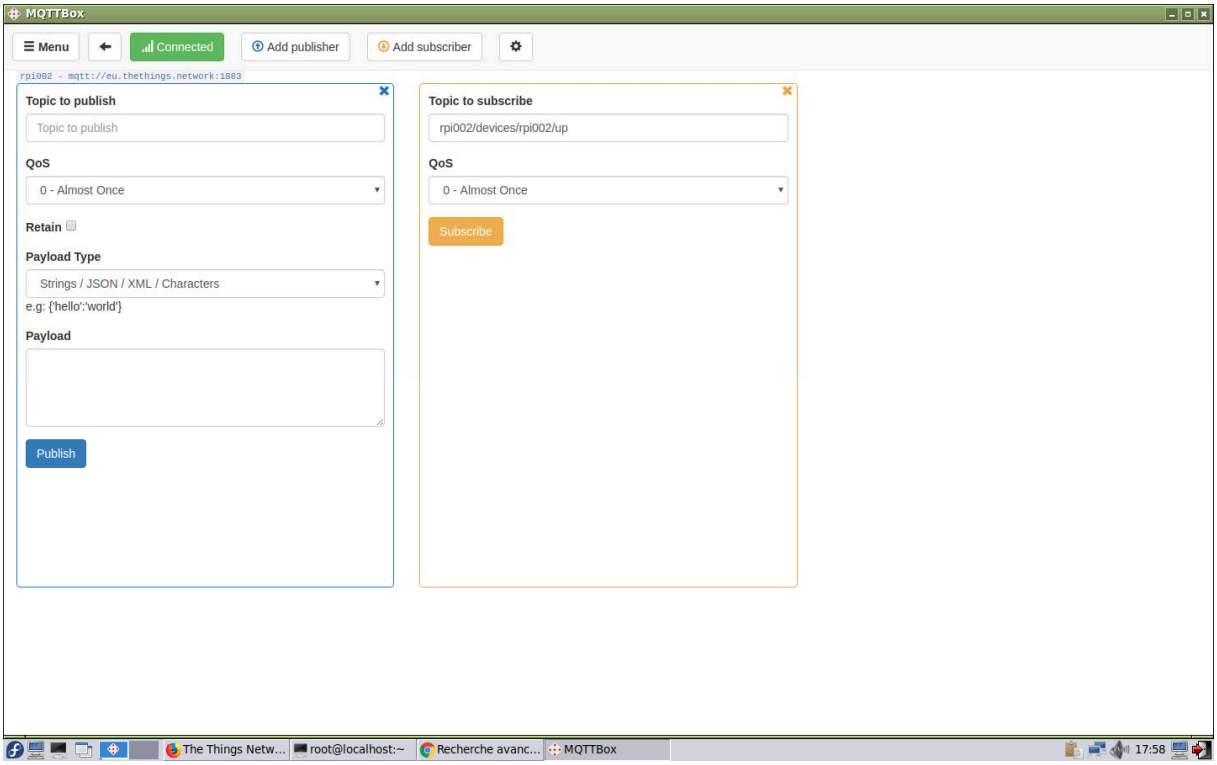
- Lancer Google Chrome puis l'application MQTTBox.

- Créer un profil client MQTT comme montré sur la figure suivante. On remplira les champs suivants en s'aidant de l'annexe 6 et du fichier `config_lora.txt`:
  - *MQTT Client Name* : `AppID`.
  - *Username* : `AppID`.
  - *Password* : mot de passe MQTT.
  - *Protocol* : `mqtt/tcp`.
  - *Host* : `eu1.cloud.thethings.network`.
  - *Will - QoS* : `0 - Almost Once`.



Création d'un profil client sous MQTTBox

- Se connecter puis renseigner le *topic* de souscription à son objet connecté de la forme : `v3/AppID/devices/+/up`



Souscription à un *topic* sous MQTTBox

- Observer et analyser les traces MQTT capturées.

5.7. EX 4 : récupération des données avec un programme Python

On désire écrire un programme Python qui s'abonne à un *topic* du *broker* TTN pour récupérer les informations émises par son objet connecté.
On utilisera pour cela la bibliothèque Paho et ce qui a été vu durant le TP 3.

- Se placer dans son répertoire de travail :
  `host% cd`
  `host% cd mon_nom`

- Dans son répertoire à son nom, recopier le fichier `mqtt_subscriber.py` sous `~kadionik/` :
  `host% cp /home/kadionik/mqtt_subscriber.py .`

- Editer le fichier `mqtt_subscriber.py` et étudier son code. Le contenu essentiel du fichier est le suivant :

```
import paho.mqtt.client as mqtt

MQTT_SERVER = "eu1.cloud.thethings.network"     # MQTT server address
MQTT_PATH = "???"                               # Topic name

def on_connect(client, userdata, flags, rc):
   print("Connection code : " + str(rc))
   # Subscribe to the topic
   client.subscribe(MQTT_PATH)

# A publish message is received from the server
def on_message(client, userdata, msg):
   print("Sujet : " + msg.topic + " Message : " + str(msg.payload))

client = mqtt.Client()

client.username_pw_set(username="???", password="???")

client.on_connect = on_connect
client.on_message = on_message
client.connect(MQTT_SERVER, 1883, 60)

client.loop_forever()
```

Le principal ajout est l'appel de la méthode `client.username_pw_set()` pour préciser la valeur de l'AppID et le mot de passe MQTT.

- Remplacer les **???** du programme par les bonnes expressions.

- Lancer le script Python `mqtt_subscriber`.

- Observer et analyser les traces MQTT capturées.

5.8. Conclusion

Nous avons pu voir la conception logicielle d'un objet connecté possédant une interface sans fil LoRa.

Nous avons pu aussi l'insérer dans une infrastructure LoRaWAN par mise en œuvre du réseau communautaire TTN.

Nous avons pu ainsi récupérer de différentes manières les informations de température et de pression de l'objet connecté LoRa en utilisant le protocole MQTT.

Quels sont les avantages et les inconvénients de l'intégration d'un objet connecté LoRa dans une infrastructure LoRaWAN (complexité, temps de développement…) ?

6. CONCLUSION

On a pu voir une approche de conception logicielle d'un objet connecté par prototypage rapide basée sur l'usage d'une distribution standard (*Raspbian*) en utilisant le langage Python pour le développement de l'application IoT.

On a pu ensuite étudier et comprendre le protocole MQTT très utilisé dans la conception des objets connectés.

On a pu enfin étudier et intégrer une interface LoRa à son objet connecté que l'on a ajouté au réseau communautaire LoRaWAN TTN ce qui nous a permis de récupérer ensuite les données émises avec MQTT depuis différents environnements.

7. REFERENCES

- Carte Raspberry Pi : https://www.raspberrypi.org/
- Carte *Sense HAT* : https://www.raspberrypi.org/learning/addons-guide/sensehat/
- Distribution Linux *Raspbian* : https://www.raspberrypi.org/downloads/raspbian/
- *Sense HAT API Reference* : https://pythonhosted.org/sense-hat/api/
- Python et Raspberry Pi : https://www.raspberrypi.org/documentation/usage/python/
- Apprendre Python : http://apprendre-python.com/
- API Python *sockets* : http://apprendre-python.com/page-reseaux-sockets-python-port
- *Threads* Python : https://openclassrooms.com/courses/apprenez-a-programmer-en-python/la-programmation-parallele-avec-threading
- Serveur Web Python : http://apprendre-python.com/page-python-serveur-web-creer-rapidement
- Introduction à LoRa. Arnaud Pecoraro
- Introduction à LoRa. Ali Benfattoum
- Réseau TTN : https://www.thethingsnetwork.org/
- Bibliothèque IBM LMIC : https://github.com/ernstdevreede/lmic_pi

8. ANNEXE 1 : MEMENTO PYTHON DU *SENSE HAT*

9. ANNEXE 2 : API PYTHON DU SENSE HAT

10. ANNEXE 3 : CONFIGURATION RESEAU HOTES ET CIBLES

	POSTE PC01	
	NOM	**ADRESSE IP**
HOTE	rodirula	10.7.4.223
CIBLE	rpi001	10.7.2.118

	POSTE PC02	
	NOM	**ADRESSE IP**
HOTE	sarracenia	10.7.4.225
CIBLE	rpi002	10.7.2.119

	POSTE PC03	
	NOM	**ADRESSE IP**
HOTE	utricularia	10.7.4.227
CIBLE	rpi003	10.7.2.120

	POSTE PC04	
	NOM	**ADRESSE IP**
HOTE	ibicella	10.7.2.112
CIBLE	rpi004	10.7.2.121

	POSTE PC05	
	NOM	**ADRESSE IP**
HOTE	nepenthes	10.7.2.114
CIBLE	rpi005	10.7.2.122

	POSTE PC06	
	NOM	**ADRESSE IP**
HOTE	pinguicula	10.7.2.116
CIBLE	rpi006	10.7.2.123

Masque de sous réseau : **255.255.248.0**

Exemple : configuration réseau de la carte cible `rpi001` :

```
RPi3# ifconfig eth0 10.7.2.118 netmask 255.255.248.0
```

	POSTE PC07	
	NOM	**ADRESSE IP**
HOTE	genlisea	10.7.1.245
CIBLE	rpi007	10.7.2.124

	POSTE PC08	
	NOM	**ADRESSE IP**
HOTE	heliamphoria	10.7.2.110
CIBLE	rpi008	10.7.2.125

	POSTE PC09	
	NOM	**ADRESSE IP**
HOTE	darlingtonia	10.7.1.239
CIBLE	rpi009	10.7.7.57

	POSTE PC10	
	NOM	**ADRESSE IP**
HOTE	dionaca	10.7.1.241
CIBLE	rpi010	10.7.7.58

	POSTE PC11	
	NOM	**ADRESSE IP**
HOTE	drosera	10.7.1.243
CIBLE	rpi011	10.7.7.59

	POSTE PC12	
	NOM	**ADRESSE IP**
HOTE	catopsis	10.7.1.40
CIBLE	rpi012	10.7.7.60

Masque de sous réseau : **255.255.248.0**

Exemple : configuration réseau de la carte cible `rpi001` :

```
RPi3# ifconfig eth0 10.7.2.118 netmask 255.255.248.0
```

11. ANNEXE 4 : CONFIGURATION TTN DES CIBLES

	POSTE PC01
Device Address	26011A11
Network Session Key	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 01
App Session Key	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 01
DevID	rpi001
AppID	rpi001@ttn
Username MQTT	rpi001@ttn
Passwd MQTT	voir fichier `config_lora.txt`

	POSTE PC02
Device Address	260119D1
Network Session Key	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 02
App Session Key	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 02
DevID	rpi002
AppID	rpi002@ttn
Username MQTT	rpi002@ttn
Passwd MQTT	voir fichier `config_lora.txt`

	POSTE PC03
Device Address	26011DA1
Network Session Key	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 03
App Session Key	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 03
DevID	rpi003
AppID	rpi003@ttn
Username MQTT	rpi003@ttn
Passwd MQTT	voir fichier `config_lora.txt`

	POSTE PC04
Device Address	2601143B
Network Session Key	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 04
App Session Key	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 04
DevID	rpi004
AppID	rpi004@ttn
Username MQTT	rpi004@ttn
Passwd MQTT	voir fichier `config_lora.txt`

	POSTE PC05
Device Address	260116A2
Network Session Key	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 05
App Session Key	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 05
DevID	rpi005
AppID	rpi005@ttn
Username MQTT	rpi005@ttn
Passwd MQTT	voir fichier `config_lora.txt`

	POSTE PC06
Device Address	2601151E
Network Session Key	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 06
App Session Key	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 06
DevID	rpi006
AppID	rpi006@ttn
Username MQTT	rpi006@ttn
Passwd MQTT	voir fichier `config_lora.txt`

	POSTE PC07
Device Address	2601146C
Network Session Key	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 07
App Session Key	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 07
DevID	rpi007
AppID	rpi007@ttn
Username MQTT	rpi007@ttn
Passwd MQTT	voir fichier `config_lora.txt`

	POSTE PC08
Device Address	26011C1D
Network Session Key	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 08
App Session Key	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 08
DevID	rpi008
AppID	rpi008@ttn
Username MQTT	rpi008@ttn
Passwd MQTT	voir fichier `config_lora.txt`

	POSTE PC09
Device Address	26011191
Network Session Key	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 09
App Session Key	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 09
DevID	rpi009
AppID	rpi009@ttn
Username MQTT	rpi009@ttn
Passwd MQTT	voir fichier `config_lora.txt`

	POSTE PC10
Device Address	26011D6E
Network Session Key	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 10
App Session Key	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 10
DevID	rpi010
AppID	rpi010@ttn
Username MQTT	rpi010@ttn
Passwd MQTT	voir fichier `config_lora.txt`

	POSTE PC11
Device Address	2601113A
Network Session Key	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 11
App Session Key	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 11
DevID	rpi011
AppID	rpi011@ttn
Username MQTT	rpi011@ttn
Passwd MQTT	voir fichier `config_lora.txt`

	POSTE PC12
Device Address	26011A30
Network Session Key	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 12
App Session Key	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 12
DevID	rpi012
AppID	rpi012@ttn
Username MQTT	rpi012@ttn
Passwd MQTT	voir fichier `config_lora.txt`