Telecommunications

IT372

Rapport

CONCEPTION D'OBJETS CONNECTES PAR PROTOTYPAGE RAPIDE

Etudiants :
Alexandre Baudry (abaudry10@gmail.com)
Kylian Ferron (kylian.ferron@outlook.fr)

Tuteur pédagogique :
Patrice KADIONIK

Décembre 2022

Table des matières

1 Présentation du projet **2**

2 TP 1 : Raspberry pi et langage python **2**
2.1 Application hello world 2
2.2 Capteurs de température et de pression 2
2.3 Threads et classes python 3
2.4 Threads et capteurs 3
2.5 Protocole HTTP Miniserveur Web 4
2.6 Protocole HTTP Miniserveur Web et capteurs 5
2.7 Miniprojet 6
2.8 Conclusion TP1 8

3 TP 2 : Protocole MQTT **8**
3.1 Question 1 : Introduction à MQTT 8
3.2 Question 2 : Paho MQTT 8
3.3 Question 3 : Ping 8
3.4 Question 4 : Qos et sessions persistantes 9
3.5 Question 5 : Retained Message 9
3.6 Question 6 : Last Will and Testament 9
3.7 Question 7 : Remontée d'informations 9
3.8 Question 8 : Arborescence et joker 9
3.9 Question 9 : Authentification 9
3.10 Code pour l'utilisation du protocole MQTT : Publisher 10
3.11 Code pour l'utilisation du protocole MQTT : Suscriber 10
3.12 Conclusion du TP2 11

4 TP 3 : Conception d'un objet connecté à LoRaWAN **11**
4.1 Introduction 11
4.2 EX 1 : écriture de l'application de l'équipement terminal 11
4.3 Ex 4 : Récupération des données avec un programme Python 14
4.4 Conclusion 15

1 Présentation du projet

L'objectif de ces travaux pratiques est de prolonger les connaissances acquises sur les systèmes d'exploitations par celles spécifiques aux systèmes embarqués très utilisés pour la conception de terminaux mobiles de télécommunications mais aussi des objets connectés plus récemment. Ce rapport propose une synthèse des connaisances aquises par le binome à travers la réalisation des différents exercices des 3 travaux pratiques effectués cette semaine.

2 TP 1 : Raspberry pi et langage python

De nombreuses bibliothèques Python permettent de développer sur la carte RPi. Il existe notamment la bibliothèque SenseHat pour piloter la carte Sense HAT. Pour utiliser celle-ci nous avons manipulé l'IDE IDLE 3 à travers 7 exercices.

2.1 Application hello world

La première manipulation consiste à réécrire en langage Python l'application "Hello World". Pour cela la struture du fichier Python est la suivante :

```
1 from sense_hat import SenseHat
2
3 sense = SenseHat()
4 pixel_list = sense.get_pixels()
5 print("Hello World")
6 sense.show_message("Hello World step for Pi!",text_colour=[255,100,0])
```

2.2 Capteurs de température et de pression

Cette seconde manipulation à pour objectif d'écrire en language python un programme "tp2.py" affichant à l'écran la pression ainsi que la température de la carte Sense HAT puis d'afficher cette température (à 10^{-1} près) sur l'afficheur matriciel à led. Pour cela la structure du fichier tp2 est donnée ci-dessous.

```
1  from sense_hat import SenseHat
2
3  sense = SenseHat()
4  temperature = sense.get_temperature()
5  pression = sense.get_pressure()
6
7  print("TEMP= %s. C"  % temperature)
8  print("Press = %s Millibars" % pression)
9
10 sense.show_message("TEMP %02.1f C" % temperature,text_colour=[255,255,0])
11 sense.show_message("Pression %02.1f Millibars" % ...
       pression,text_colour=[255,255,0])
```

2.3 Threads et classes python

Dans cet exercice on s'intéresse aux threads ainsi qu'aux classes python. Le programme implémenté ci-dessous à pour objectif de créer 2 threads ainsi qu'une classe de threads task_affiche qui dans sa méthode run affiche toutes les 2 secondes le nom de l'object créé à partir de cette classe puis au bout de 10 secondes les 2 threads sont arrêtés.

```
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:
        #-------------#
            sleep(2)
            print(self.nom)
        #-------------#
    def stop(self):
        self.Terminated = True
t1 = task_affiche("Task t1")
t1.start()
t2 = task_affiche("Task t2")
t2.start()
sleep(10)
t1.stop()
t2.stop()
```

2.4 Threads et capteurs

Dans cette partie on cherche à implémenter un programme python permettant la création de 2 threads affichant à l'ecran la pression de la carte toutes les 2 secondes et la température toute les secondes. Pour cela on créé les 2 classes de thrads task_pression et task_temp. Le code implémenté pour répondre à cette demande est le suivant :

```
from sense_hat import SenseHat
from time import *
import threading
sense = SenseHat()

class temp_affiche(threading.Thread):
     def __init__(self):
         threading.Thread.__init__(self)
         self.Terminated = False
```

```
10      def run(self):
11          while not self.Terminated:
12              sleep(1)
13              temperature = sense.get_temperature()
14              print("TEMP= %02.1f. C\n"  % temperature)
15      def stop(self):
16          self.Terminated = True
17  class pression_affiche(threading.Thread):
18      def __init__(self):
19          threading.Thread.__init__(self)
20          self.Terminated = False
21      def run(self):
22          while not self.Terminated:
23              sleep(2)
24              pressure = sense.get_pressure()
25              print("Pression= %02.1f. Millibars\n"  % pressure)
26      def stop(self):
27          self.Terminated = True
28  temp = temp_affiche()
29  pression = pression_affiche()
30  temp.start()
31  pression.start()
32  sleep(10)
33  temp.stop()
34  pression.stop()
```

2.5 Protocole HTTP Miniserveur Web

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 est composée d'une entête HTTP classique, d'un saut de ligne et des données codées en HTML correspondant à la page d'accueil du miniserveur Web.

L'objecetif de cet exercice est d'ecrire en langage Python le programme tp5.py de création d'un miniserveur Web en écoute sur le port 8080 et de tester celui-ci avec le navigateur Web. Le code utilisé pour ce miniserver Web est présenté ci-dessous :

```
1  import socket
2
3  from sense_hat import SenseHat
4  from time import *
5  import threading
6  import socket
7  sense = SenseHat()
8
9  socket = socket.socket(socket.AF_INET, socket.SOCK_STREAM    )
10 socket.bind(('', 8666))
11 socket.listen(1)
12 while True:
```

```
13      temperature = sense.get_temperature()
14      pressure = sense.get_pressure()
15
16      client, address = socket.accept()
17      request = client.recv(1024)
18
19      client.send(str.encode("HTTP/1.1 200 OK\n"))
20      client.send(str.encode("Content-Type: text/html \n"))
21      client.send(str.encode("\n"))
22      client.send(str.encode("<br> </br>"))
23      client.send(str.encode("<CENTER><H1>Welcome to the rpi-python Web ...
            Server</H1></CENTER>"))
24      client.send(str.encode("Hello World! \n"))
25      client.close()
```

2.6 Protocole HTTP Miniserveur Web et capteurs

Pour ajouter à notre miniserver un renvoie de la température ainsi que de la pression, on peut ajouter la balise HTML suivante pour forcer le rafraîchissement de la page Web toutes les secondes : <meta http-equiv="Refresh" content="1">. Le code du miniserver est donc modifié pour une nouvelle version :

```
1  import socket
2
3  from sense_hat import SenseHat
4  from time import *
5  import threading
6  import socket
7  sense = SenseHat()
8
9  socket = socket.socket(socket.AF_INET, socket.SOCK_STREAM    )
10 socket.bind(('', 8666))
11 socket.listen(1)
12 while True:
13      temperature = sense.get_temperature()
14      pressure = sense.get_pressure()
15
16      client, address = socket.accept()
17      request = client.recv(1024)
18
19      client.send(str.encode("HTTP/1.1 200 OK\n"))
20      client.send(str.encode("Content-Type: text/html \n"))
21      client.send(str.encode("\n"))
22      client.send(str.encode("<br> </br>"))
23      client.send(str.encode("<CENTER><H1>Welcome to the rpi-python Web ...
            Server</H1></CENTER>"))
24      client.send(str.encode("Hello World! \n"))
25      ##----Ajout----##
26      client.send(str.encode("<br> </br>"))
```

```
27      client.send(str.encode("Pression= %02.1f. Millibars \n"  % pressure))
28
29      client.send(str.encode("<br> </br>"))
30      client.send(str.encode("TEMP= %02.1f. C\n"  % temperature))
31      client.send(str.encode("<meta http-equiv=\"Refresh\" content=\"1\">"))
32      ##-------------##
33      client.close()
```

2.7 Miniprojet

Pour ce miniprojet on cherche à écrire en langage Python le programme miniprojet.py de synthèse des exercices précédents. Il a pour objectif la création d'une classe de thread task_sensor d'acquisition des capteurs de température et de pression, d'une classe de thread task_led d'affichage de la température courante sur l'afficheur matriciel à leds et d'une classe de thread task_www d'implémentation d'un miniserveur Web qui renvoie la température courante et la pression courante. Le code répondant à ce miniprojet est donné ci-dessous.

```
1  temp = 0
2  press = 0
3  from sense_hat import SenseHat
4  from time import *
5  import threading
6  import socket
7  sense = SenseHat()
8
9  class task_sensor(threading.Thread):
10     def __init__(self):
11         threading.Thread.__init__(self)
12         self.Terminated = False
13     def run(self):
14         global temp
15         global press
16         while not self.Terminated:
17             temp = sense.get_temperature()
18             press = sense.get_pressure()
19     def stop(self):
20          self.Terminated = True
21 class task_led(threading.Thread):
22     def __init__(self):
23         threading.Thread.__init__(self)
24         self.Terminated = False
25     def run(self):
26         global temp
27         global press
28         while not self.Terminated:
29             sense.show_message("TEMP %02.1f  C " % temp,text_colour=[0,0,255])
30             sense.show_message("Pression %02.1f MB" % ...
                   press,text_colour=[255,0,0])
```

```
31      def stop(self):
32           self.Terminated = True
33  class task_www(threading.Thread):
34      def __init__(self):
35          threading.Thread.__init__(self)
36          self.socket = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
37          self.socket.bind(('', 8666))
38          self.socket.listen(1)
39          self.Terminated = False
40      def run(self):
41          global temp
42          global press
43          while not self.Terminated:
44              while (True):
45                  client, address = self.socket.accept()
46                  request = client.recv(1024)
47
48                  client.send(str.encode("HTTP/1.1 200 OK\n"))
49                  client.send(str.encode("Content-Type: text/html \n"))
50                  client.send(str.encode("\n"))
51                  client.send(str.encode("<br> </br>"))
52                  client.send(str.encode("<CENTER><H1>Welcome to the rpi-python ...
                        Web Server</H1></CENTER>"))
53                  client.send(str.encode("Hello World! \n"))
54
55                  client.send(str.encode("<br> </br>"))
56                  client.send(str.encode("Pression= %02.1f. Millibars \n"  % ...
                        press))
57
58                  client.send(str.encode("<br> </br>"))
59                  client.send(str.encode("TEMP= %02.1f. C\n"  % temp))
60                  client.send(str.encode("<meta http-equiv=\"Refresh\" ...
                        content=\"1\">"))
61
62                  client.close()
63      def stop(self):
64           self.Terminated = True
65  sensor=task_sensor()
66  www = task_www()
67  led=task_led()
68
69  sensor.start()
70  www.start()
71  led.start()
72
73  sleep(10)
74  sensor.stop()
75  www.stop()
76  led.stop()
```

2.8 Conclusion TP1

Utiliser Python avec Linux Raspbian sur la carte RPi pour développer une application IoT possède plusieurs avantages que nous allons détailler. Dans un premier temps, La facilité de prise en main du language de programmation Python est ce qui en fait un choix populaire pour tout types de developpeurs. Dans un second temps, La grande communauté de développeurs Linux Raspbian est un fort atout. De plus, le contrôle des capteurs et de l'affichage de la carte Sense HAT est très simple via Python. Enfin, le contrôle à distance par internet de la carte RPi via le server web en Python est très intéressant notamment dans un optique de contrôle d'objets connectés à distance.

Néanmoins, pour citer quelques inconvénients de cette méthodologie de développement d'une application IoT nous avons relevé les problèmes suivants : La programmation des différents composants, tels que les capteurs, les afficheurs et les serveurs Web est couteuse en temps. De plus, il est complexe de réaliser une interface web digne de ce nom uniquement à partir de Python.

3 TP 2 : Protocole MQTT

MQTT (Message Queuing Telemetry Transport) est un protocole de messagerie utilisé pour l'internet des objets. Il s'agit d'un transport de messagerie de type publish/suscribe extrêmement léger. Il est idéal pour connecter des appareils distants avec une petite empreinte de code et une bande passante réseau minimale. Le but de ce deuxième TP est de découvrir le fonctionnement de MQTT et quelques utilisations de ce protocole.

3.1 Question 1 : Introduction à MQTT

MQTT est un protocole de messagerie simple conçu pour que la taille des données transmises soit limitées. Ce protocole est utilisé en Iot (Internet des objets) pour des appareils à faible bande passante notamment. Il fonctionne par un système de publications et d'abonnements avec la publication de messages sur des sujets précis. Le principe est simple, chaque appareil abonné à un sujet reçoit le message qui a été publié. Ces affectations de messages se font à travers un broker vérifiant l'abonnement afin de transmettre le message.

3.2 Question 2 : Paho MQTT

La librairie Paho-MQTT permet à des applications de se connecter à un broker MQTT à travers un client fait pour publier et s'abonner à des sujets (topics). Il propose aussi des fonctions simples facilitant la publication de messages à un serveur MQTT.

3.3 Question 3 : Ping

Après avoir mis en place un système de log jouant le rôle d'un ping, on constate qu'une fois les X secondes indiqués le suscriber envoie un ping pour vérifier que le publisher est toujours connecté. De plus, si le publisher envoie un message au suscriber et que les deux sont présents

sur le même topic alors le suscriber va tout simplement recevoir le message. Au contraire, si le broker MQTT ne reçoit pas de réponses, il ne va rien se passer.

3.4 Question 4 : Qos et sessions persistantes

En modifiant la variable clean_session et en lui assignant la valeur false, on remarque que la session est gardée en mémoire d'où le nom de session persitante.

Si on lui assigne la valeur true, cela entraîne la suppression de la session en cours et des données liées. C'est dans cette situation qu'il est nécessaire d'affecter un ID au client pour qu'il se souvienne de la session avec laquelle communiquer.

Enfin, on peut évoquer la qualité de service. Si celle ci est égale à 1, le suscriber reçoit au moins une fois le message. Si elle est égale à 2, le suscriber le reçoit exactement une fois.

Dans une session persistante, les informations stockées sont les IDs des clients et ceux des paquets afin d'avoir un historique des transmissions.

Si on prend l'exemple d'un détecteur de fumée, cela peut permettre de garder la trace d'un déclenchement en cas de fumée.

3.5 Question 5 : Retained Message

L'intérêt du mécanisme du message enregistré est de garder en mémoire les messages des publishers en leur assignant un index.

3.6 Question 6 : Last Will and Testament

A la fermeture du publisher, le message est transmis à nouveau. EN cas de panne ou de rupture du système, cela permet de garder cet évènement en mémoire et de prévenir les autres appareils. Il se déclenche à la terminaison d'un programme de façon souhaitée ou pas.

3.7 Question 7 : Remontée d'informations

Il a été possible de créer des topics concernant la température et la pression récupérées à partir de la remontée d'informations des capteurs (Sense Hat).

3.8 Question 8 : Arborescence et joker

Le joker "+" permet de s'abonner à toutes les branches correspondant au niveau mrécédent de l'arborescence et au niveau suivant spécifié.

Les jokers permettent donc de simplifier l'abonnement à plusieurs topics présents dans un ensemble de branches de l'arborescence des topics. Cela permet d'éviter de la redondance dans la volonté d'abonnement. Par exemple, autant s'abonner directement à tous les détecteurs de la maison pour connaître la température ambiante de chaque pièce.

3.9 Question 9 : Authentification

Nous n'avons pas eu le temps de réaliser cette utilisation du protocole lors de la séance.

3.10 Code pour l'utilisation du protocole MQTT : Publisher

```

#!/usr/bin/env python3
from time import *
from sense_hat import SenseHat
import paho.mqtt.client as mqtt


sense = SenseHat()
temperature = sense.get_temperature()
pressure = sense.get_pressure()
client = mqtt.Client()
client.connect("localhost",1883,60)
client.publish("temperature", "TEMP= %02.1f. C\n"  % temperature)
client.publish("pression", "Pression= %02.1f. Millibars\n"  % pressure);

client.publish("topic/3", "200 | 350 | 420 | 110");

client.disconnect();
```

3.11 Code pour l'utilisation du protocole MQTT : Suscriber

```
#!/usr/bin/env python3

import paho.mqtt.client as mqttClient
import time

def on_connect(client, userdata, flags, rc):

    if rc == 0:

        print("Connected to broker")

        global Connected                #Use global variable
        Connected = True                #Signal connection

    else:

        print("Connection failed")

def on_message(client, userdata, message):
    print("Message received : "  + str(message.payload) + " on " + message.topic)

Connected = False

broker_address= "localhost"
port = 1883
```

```
26
27 client = mqttClient.Client("Python")
28 client.on_connect= on_connect
29 client.on_message= on_message
30 client.connect(broker_address, port=port)
31 client.loop_start()
32
33 while Connected != True:
34     time.sleep(0.1)
35
36 client.subscribe([("temperature", 0), ("pression", 0), ("topic/3", 0)])
37 try:
38     while True:
39         time.sleep(1)
40
41 except KeyboardInterrupt:
42     print ("exiting")
43     client.disconnect()
44     client.loop_stop()
```

3.12 Conclusion du TP2

MQTT peut être utilisé pour de nombreux domaines même si la plupart des exemples données durant ce rapport sont dans celui de la domotique. Ce système de messagerie très léger pourrait être énormément utilisés et semble très pratique pour de nombreux objets connectés. Pour citer un autre domaine que celui de capteurs dans une maison, il pourrait très bien être utilisé en industrie pour connaître les différents états d'une chaîne de production ou encore directement dans le médical, pour transmettre la situation ou l'état d'une personne agée ou en situation de handicap.

4 TP 3 : Conception d'un objet connecté à LoRaWAN

4.1 Introduction

LoRaWAN (Long Range Radio Wide Area Network) est un protocole de communication dédié à l'IoT permettant des communications sans fil longue portée, faible débit et avec une faible consommation électrique. Il s'agit donc d'un réseau de type "Low Power Wide Area Network" basé sur la technologie radio LoRa.

4.2 EX 1 : écriture de l'application de l'équipement terminal

do_send() renvoies les données de température et de pression captés par les capteurs du Raspberry Pi.

getsensors() revoies les valeurs de température et de pression après avoir configurer le I2C_SLAVE.

```
  temp_out_l = i2c_smbus_read_byte_data(fd, TEMP_OUT_L);
  temp_out_h = i2c_smbus_read_byte_data(fd, TEMP_OUT_H);
  press_out_xl = i2c_smbus_read_byte_data(fd, PRESS_OUT_XL);
  press_out_l = i2c_smbus_read_byte_data(fd, PRESS_OUT_L);
  press_out_h = i2c_smbus_read_byte_data(fd, PRESS_OUT_H);
```

Le code permettant d'encoder avec Cayenne la température sur le canal 1 et la pression sur le canal 2 est doné ci-dessous :

```
#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);

static const u1_t DEVKEY[16] = { 0x00, 0x00, 0x00, 0x00, 0x00, ...
    0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, ...
    0x06 };
static const u1_t ARTKEY[16] = { 0x00, 0x00, 0x00, 0x00, 0x00, ...
    0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, ...
    0x06 };
static const u4_t DEVADDR = 0x2601151E;

// ////////////////////////////////////////////////
// APPLICATION CALLBACKS
// ////////////////////////////////////////////////

void os_getArtEui (u1_t* buf) {
  memcpy(buf, APPEUI, 8);
}

void os_getDevEui (u1_t* buf) {
  memcpy(buf, DEVEUI, 8);
}

void os_getDevKey (u1_t* buf) {
  memcpy(buf, DEVKEY, 16);
}

static osjob_t sendjob;

```

```
34 lmic_pinmap pins = {
35               .nss = 6,
36               .rxtx = UNUSED_PIN,
37               .rst = 0,
38               .dio = {7,4,5}
39 };
40
41 void onEvent (ev_t ev) {
42   fprintf(stdout, "Event received: %d\n", ev);
43   switch(ev) {
44   case EV_TXCOMPLETE:
45     fprintf(stdout, "Event EV_TXCOMPLETE, time: %d\n\n", millis() ...
        / 1000);
46     break;
47   default:
48     break;
49   }
50 }
51
52 static void do_send(osjob_t* j){
53   int i;
54   if (LMIC.opmode & (1 << 7)) {
55     fprintf(stdout, "OP_TXRXPEND,not sending");
56   } else {
57     lpp.reset();
58     getsensors();
59     printf("TEMP=%.1f oC\n", t_c);
60     printf("PRESSION=%.1f hPa\n", pressure);
61     lpp.addTemperature(1,t_c);
62     lpp.addBarometricPressure(2,pressure);
63
64     for (i=0; i < lpp.getSize(); i++)
65       printf("%02X ", *(lpp.getBuffer() + i));
66     puts("\n"); fflush(stdout);
67     LMIC_setTxData2(1, (xref2u1_t)lpp.getBuffer(), lpp.getSize(), 0);
68   }
69   os_setTimedCallback(j, os_getTime()+sec2osticks(60), do_send);
70 }
71
72 void setup() {
73   wiringPiSetup();
74   os_init();
75   LMIC_reset();
76   LMIC_setSession (0x1, DEVADDR, (u1_t*)DEVKEY, (u1_t*)ARTKEY);
77   LMIC_setAdrMode(0);
78   LMIC_setLinkCheckMode(0);
79   LMIC_disableTracking ();
```

```
80    LMIC_stopPingable();
81    LMIC_setDrTxpow(DR_SF7,14);
82  }
83
84  void loop() {
85    do_send(&sendjob);
86    while(1)
87      os_runloop();
88  }
89
90  int main() {
91    setup();
92    while (1)
93      loop();
94    return 0;
95  }
```

Commment fonctionne le Makefile ?

La façon de compiler le programme temp est décrite dans la section 'temp'. Le répertoire ../lib/lmic doit est compilé en premier. La commande $(CC) est utilisée pour compiler 'abp.cpp' avec les options de compilation 'CFLAGS' et les fichiers objets générés dans le répertoire ../lib/lmic ainsi que les options d'édition de liens 'LDFLAGS'.

La manière de compiler le programme CayenneLPP est décrite dans la section 'CayenneLPP'. On utilise la commande $(CC) pour compiler 'CayenneLPP.cpp' en utilisant les options de compilation 'CFLAGS' et les options d'édition de liens 'LDFLAGS'.

4.3 Ex 4 : Récupération des données avec un programme Python

```
1  import paho.mqtt.client as mqtt
2
3  MQTT_SERVER = "eu1.cloud.thethings.network" # MQTT server address
4  MQTT_PATH = "v3/rpi006@ttn/devices/rpi006/up"
5
6  def on_connect(client, userdata, flags, rc):
7      print("Connection code : " + str(rc))
8      # Subscribe to the topic
9      client.subscribe(MQTT_PATH)
10
11 def on_message(client, userdata, msg):
12     print("Sujet : " + msg.topic + " Message : " + str(msg.payload))
13
14 client = mqtt.Client()
15
16 client.username_pw_set(username="rpi006@ttn", password="...")
17
18 client.on_connect = on_connect
19 client.on_message = on_message
```

```
20 client.connect(MQTT_SERVER, 1883, 60)
21 client.loop_forever()
```

4.4 Conclusion

Les inconvénients de l'intégration d'un objet connecté LoRadans une infrastructure LoRaWAN sont la complexité de mise en oeuvre ainsi que la vitessse de transmission des données. Pour ce qui est des avantages, ceux-ci réside dans la faible consommation d'énergie et des coûts d'exploitation réduits sur le long terme ainsi que la longue portée de la communication sans fil.