Lab 11 - Protocole et passerelles LoRa TS (ThingSpeak)

Table des matières

11.1 Introduction

Dans ce laboratoire, nous allons créer une double (émetteur-récepteur) passerelle IoT pour la communication LoRa avec les serveurs de type **ThingSpeak** (**TS**). Les passerelles fournissent le relais entre les liaisons LoRa et les modems WiFi vers/depuis le serveur (ThingSpeak).
Une architecture IoT correspondante comprend deux cartes ESP32 fonctionnant comme des passerelles LoRa-TS. Une carte passerelle relaie les messages des nœuds terminaux vers le serveur TS , la seconde carte reçoit les données demandées de TS et les relaie vers les nœuds terminaux.

Les nœuds terminaux fonctionnant en tant qu'expéditeur et récepteur de données ont déjà été présentés en détail dans le laboratoire précédent.
Dans ce laboratoire, nous allons compléter les fonctionnalités des passerelles en émission et en réception des données vers/à partir du serveur ThingSpeak. Plus précisément, nous allons ajouter la partie *back-end* communiquant par **WiFi** avec le serveur ThingSpeak.

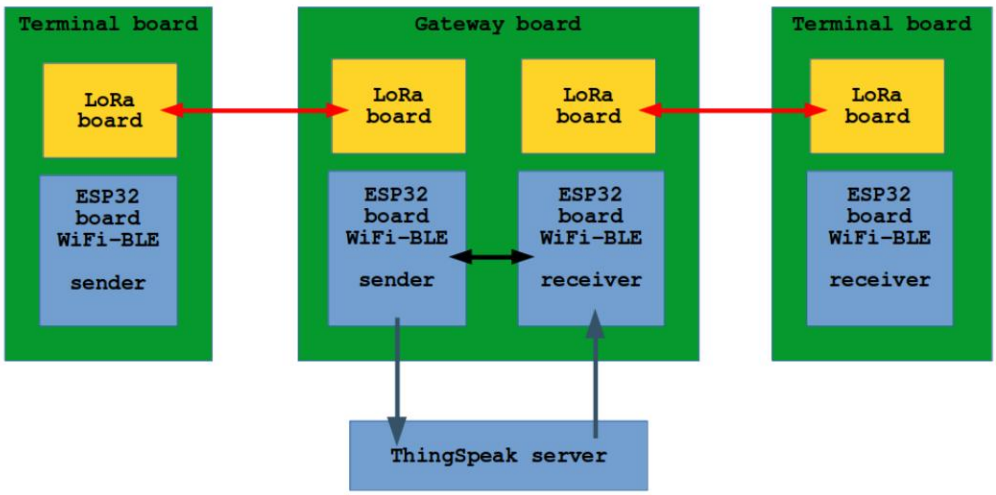

Figure 11.1 Architecture IoT avec deux passerelles vers le serveur ThingSpeak (envoi-réception) et les nœuds terminaux

Dans le Lab.10 précédent, nous avons présenté les nœuds Terminal et Gateway communiquant avec les paquets de contrôle et de données prédéfinis. Dans ce laboratoire, nous utilisons les mêmes bibliothèques et les mêmes fonctions **`send_XXXX()`** et **`onReceive()`**.
La principale différence est que nous complétons nos passerelles avec les tâches de communication basées sur le **WiFi**. Ces tâches nous permettent d'envoyer/recevoir les données vers/depuis le serveur ThingSpeak.

Comme dans les configurations présentées dans le laboratoire précédant, les terminaux et les passerelles fonctionnent en deux étapes.

- la première étape permet aux terminaux de se présenter à la passerelle via un paquet **`IDREQ`** (**`IDREQ`** porte le mot de passe) et de recevoir l'identifiant de la passerelle (**`gwID`**) et la clé AES (**`CKEY`**) via un paquet **`IDACK`**. Cette étape est réactivée périodiquement pour tester la disponibilité de la passerelle.

- la deuxième étape est utilisée pour envoyer les données avec un paquet **`DTSND`** ou pour demander les données avec un paquet **`DTREQ`**. Les paquets répondantes sont les accusés de réception portant la valeur de temporisation (**`DTACK`**) ainsi que les données pour **`DTREQ`** - **`DTRCV`**. La valeur de temporisation peut être utilisée par le nœud terminal pour imposer la période d'attente avant d'envoyer les trames **`DTSND`** ou **`DTREQ`** suivantes.

Comme nous l'avons déjà vu dans le laboratoire précédent, la première étape opérationnelle avec les paquets **`IDREQ/IDACK`** permet de configurer le fonctionnement au niveau de la couche **liaison/réseau** LoRa.

Cette couche **liaison/réseau** est la même pour tous les protocoles applicatifs, y compris: envoyer vers TS (MODE 1), recevoir depuis TS (MODE 2), publier vers MQTT (MODE 3) et souscrire à MQTT (MODE 4).

Le tableau suivant montre les codes utilisés pour identifier les paquets dans le champ **`con[0]`** des paquets de contrôle et de données.

```
1 - TS send, 2 - TS receive, 3 - MQTT
PUBlish, 4 - MQTT Suscribe

Link
con[0]=11:IDREQ(1),con[0]=12:IDACK(1)
con[0]=21:IDREQ(2),con[0]=22:IDACK(1)
con[0]=31:IDREQ(3),con[0]=32:IDACK(1)
con[0]=41:IDREQ(4),con[0]=42:IDACK(1)

ThingSpeak
con[0]=13:DTSND(1)
con[0]=14:DTACK(1)
con[0]=23:DTREQ(2)
con[0]=24:DTRCV(2)

MQTT
con[0]=33:DTPUB(3)
con[0]=34:DTACK(3)
con[0]=43:DTSUB(4)
con[0]=44:DTRCV(4)
```

Tableau 11.1 Les codes pour les **types** et **modes** des paquets LoRa

Le deuxième champ de contrôle **`con[1]`** est utilisé pour marquer les champs ou identifiants des capteurs/actionneurs pour les canaux TS.
L'union/la structure suivante définit les paquets de contrôle. Elle commence par les identificateurs de destination/source (**`did`**, **`sid`**) dérivés du **`chipID`** de chaque nœud.

```
typedef union
{
  uint8_t frame[32];
  struct
  {
    uint32_t did;       // destination identifier chipID (4 lower bytes)
    uint32_t sid;       // source identifier chipID (4 lower bytes)
    uint8_t  con[2];    // control field: con[0]
    char     pass[16];  // password or AES key - 16 characters
    int tout;           // timeout
    uint8_t  pad[2];    // future use
  } pack;               // control packet
} conframe_t;              // send control frame , receive control frame
```

Etape 1

Le paquet **IDACK** qui répond au **IDREQ** contient l'identifiant de destination - **did**, c'est-à-dire l'identifiant du terminal qui a envoyé la trame **IDREQ**. Le champ **sid** contient la partie inférieure (4 octets) de l'identifiant de la passerelle (**gwID**).

Figure 11.2 Format des paquets **IDREQ**, **IDACK**

Ce sont les paramètres du paquet de contrôle **IDREQ**.

```
scf.pack.did=(uint32_t)0;     // destination is not known !!!
scf.pack.sid=(uint32_t)termID;
scf.pack.con[0]=0x11; scf.pack.con[1]=0x00;  // IDREQ frame for TS sender
```

Notez qu'en fonction des services requis, le champ de contrôle contient le MODE de service :

```
scf.pack.con[0]=0x11;scf.pack.con[1]=0x00; // IDREQ packet for TS sender (TSS)
scf.pack.con[0]=0x21;scf.pack.con[1]=0x00; // IDREQ packet for TS receiver (TSR)
```

Après l'envoi du paquet **IDREQ**, le terminal attend qu'un paquet **IDACK** soit générée par la passerelle.

Notez que la passerelle n'enverra le paquet **IDACK** correspondant que si le champ mot de passe - **pass[16]** fourni par le terminal correspond à la valeur du mot de passe enregistré dans la passerelle et au service demandé (émetteur/récepteur).

```
  scf.pack.did=(uint32_t)termID;  // destination is the requesting terminal
  scf.pack.sid=(uint32_t)gwD;     // gateway identifier
  scf.pack.con[0]=0x12; scf.pack.con[1]=0x00;  // IDACK - MODE 1
```

Etape 2

Après la réception de l'identifiant de passerelle, le terminal peut envoyer son paquet de données – **DTSND** ou demander les données via un paquet **DTREQ**.

Les données sont envoyées dans le type d'union/structure suivante :

Figure 11.3 Format de paquet de données (**DTSND**, **DTACK**)

```
typedef union
{
  uint8_t frame[64];        // TS frame to send/receive data
  struct
  {
    uint32_t did;           // destination identifier chipID (4 lower bytes)
    uint32_t sid;           // source identifier chipID (4 lower bytes)
    uint8_t  con[2];        // control field: lower byte is used as mask
    int      channel;       // TS channel number
    char     keyword[16];   // write (or read) keyword
    float    sens[8];       // max 8 values - data fields
    uint16_t tout;          // optional timeout
  } pack;                   // data packet
} TSframe_t;
```

La valeur de masque envoyée dans l'octet **con[1]** est un masque qui indique les valeurs de capteur valides. A chaque valeur de capteur **sens[8]** correspond un **bit** dans le **masque** de capteur/actionneur.

Par exemple, si le terminal envoie deux valeurs Température et Humidité dans les deux premiers champs, la valeur du masque doit être **0xC0** (en binaire: **11000000**)

```
sdf.pack.did=(uint32_t)gwID;
sdf.pack.sid=(uint32_t)termID;
sdf.pack.con[0]=0x13; sdf.pack.con[1]=mask;  // DTSND packet - MODE 1
```

Ensuite, le terminal attend un paquet **DTACK**.

```
sdf.pack.did=(uint32_t)termID;
sdf.pack.sid=(uint32_t)sendID;
sdf.pack.con[0]=0x14; sdf.pack.con[1]=mask;  // DTACK packet - MODE 1
```

Ce paquet peut contenir une information supplémentaire qui porte la valeur de temporisation pour la trame de données suivante. Elle sera enregistré dans le champ de numéro du canal (**channel**).
Voici la séquence de code requise pour initialiser l'état **deep_sleep** dans le nœud terminal :

```
esp_sleep_enable_timer_wakeup(1000*1000*timeout);
esp_sleep_pd_config(ESP_PD_DOMAIN_RTC_PERIPH, ESP_PD_OPTION_OFF);
LoRa.end(); delay(200);  // necessary to stop the LoRa modem
esp_deep_sleep_start();
```

La valeur de temporisation reçue est convertie en le nombre de microsecondes pendant lesquelles le circuit ESP32 passe en état de sommeil profond. Notez que pendant cette période, toutes les variables volatiles liées à deux processeurs principaux sont perdues.
Si nous devons les conserver, ils peuvent être stockés dans la RAM statique intégrée dans le bloc RTC ou dans la mémoire EEPROM locale.

Notez que les valeurs stockées dans le bloc RTC sont effacées si nous réinitialisons le SoC ESP32 avec le bouton **RST**.
Côté réception des paquets, **onReceive()** est une fonction définie dans le fichier **LoRa_onReceive.AES.h** et utilisée pour capturer les interruptions IO du modem LoRa et pour recevoir les paquets de contrôle et des données envoyées par le terminal et les nœuds de passerelle (réception **GATEWAY**: **IDREQ**, **DTSND**; réception **TERMINAL**:**IDACK**, **DTRCV**).

Les indicateurs de réception **stage1_flag** et **stage2_flag** indiquent le type du paquet reçu.
Dans le même fichier (**LoRa_onReceive.AES.h**), nous déclarons la file d'attente de réception utilisée pour communiquer le contenu des paquets reçus à d'autres tâches.

Notez que **onReceive()** est une fonction **ISR** exécutée de manière asynchrone et ne doit pas contenir d'opérations de traitement supplémentaires. La fonction **decrypt()** est ici exécutée par l'accélérateur matériel interne et peut être incluse dans les opérations de l'ISR.

Voici le contenu du fichier **LoRa_onReceive.AES.h** (le même que celui utilisé dans le Lab.10)

```
// this file contains onReceive() ISR that receives and decrypts the received data packets
// the analysis of the paquets is done in the main task after their reception in the corresponding
queues

QueueHandle_t tsrcv_queue, mqrcv_queue, con_queue;  // receive queues for data and control packets
int queueSize = 32;

void onReceive(int packetSize)
{
if(packetSize==32)
  {
  conframe_t rcf; int i=0;
  while (LoRa.available()) { rcf.frame[i] = LoRa.read();i++;}
  xQueueReset(con_queue);  // reset queue to keep only the last packet
  xQueueSendFromISR(con_queue, rcf.frame, NULL); Serial.println("Received IDREQ_MODE");
  }

if(packetSize==64)
  {
  TSframe_t rdf,rdcf; int i=0;
  char ckey[17];
  while (LoRa.available()) { rdcf.frame[i] = LoRa.read();i++;}
```

```
  strncpy(ckey,CKEY,16);ckey[16]='\0';
  decrypt(rdcf.frame,ckey,rdf.frame,4);
  xQueueReset(tsrcv_queue);  // reset queue to keep only the last packet
  xQueueSend(tsrcv_queue, rdf.frame, portMAX_DELAY);
  }

if(packetSize==112)
  {
  MQTTframe_t rdf,rdcf; int i=0;
  char ckey[17];
  strncpy(ckey,CKEY,16);ckey[16]='\0';
  while (LoRa.available()) { rdcf.frame[i] = LoRa.read();i++;}
  decrypt(rdcf.frame,ckey,rdf.frame,7);
  xQueueReset(mqrcv_queue);  // reset queue to keep only the last packet
  xQueueSend(mqrcv_queue, rdf.frame, portMAX_DELAY);
  }
}
```

Voici le diagramme de communication entre un terminal et la passerelle en MODE1.

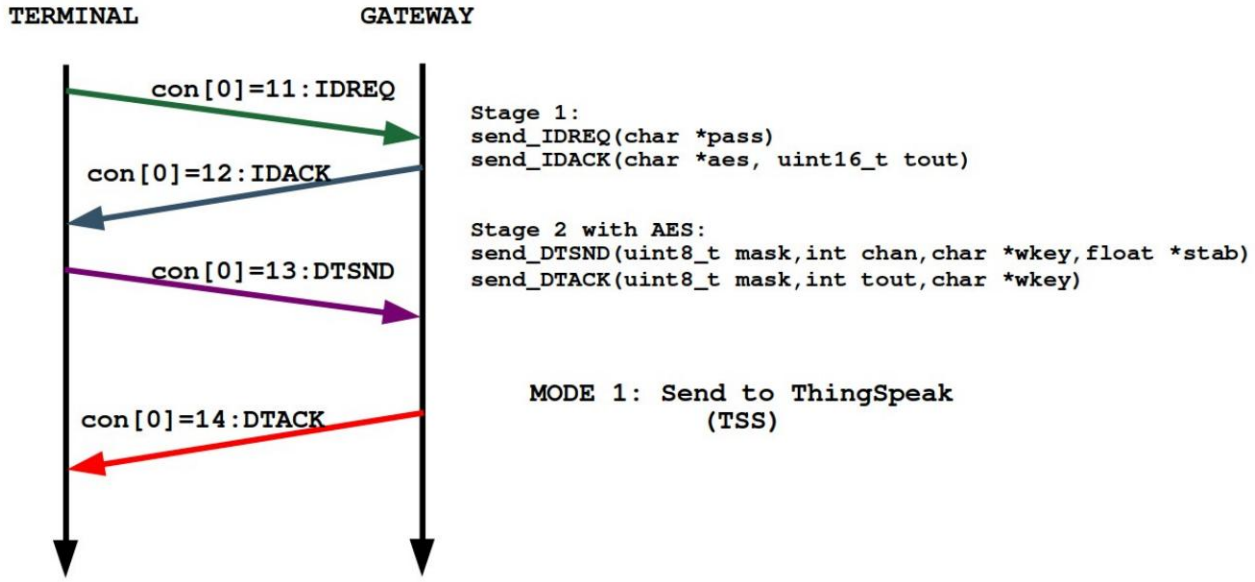

Figure 11.4 Diagramme de temps pour la communication entre un terminal et une passerelle TSS

11.2 Terminal émetteur (*sender*) - MODE 1

Dans cette section, nous présentons deux versions du terminal émetteur, l'une fonctionnant toujours en mode actif, la seconde fonctionne avec la fonction de sommeil profond (mode **deep_sleep**) et le **timeout** fourni par le nœud de passerelle.

11.2.1 Code du terminal émetteur – toujours actif et avec état `deep_sleep`

Le code suivant implémente un terminal qui envoie les données à la passerelle pour être relayées vers le serveur ThingSpeak.
La partie du code commenté suivant peut être utilisée pour mettre le mode terminal en mode sommeil profond après chaque itération (cycle).

```
esp_sleep_enable_timer_wakeup(1000*1000*cycle);  // cycle in seconds
esp_sleep_pd_config(ESP_PD_DOMAIN_RTC_PERIPH, ESP_PD_OPTION_OFF);
Serial.println("Going to sleep now");
Serial.flush(); LoRa.end();delay(160);
esp_deep_sleep_start();
Serial.println("This will never be printed");
```

11.2.1.1 Code complet du terminal avec SHT21 - capteur de température/humidité

```
#define TERMINAL  // TERMINAL or GATEWAY node
#define MODE 1    // sender (1), receiver (2) , publisher (3), subscriber (4) mode
#include <Wire.h>

#include "SHT21.h"
SHT21 SHT21;

#include <SPI.h>
#include <LoRa.h>
#include "LoRa_Para.h"
#include "LoRa_Packets.AES.h"
#include "LoRa_onReceive.AES.h"  // to capture the packets with MODE and server/broke

union
{
  uint8_t para[24];
  struct
  {
    int chan;      // channel number
    char key[16];  // write key
    uint8_t zero;
    uint8_t mask;  // sensor mask
    uint8_t pad[2];
  } pack;
} ts;  // TS send and receive para


float stab[8];

void get_sens()
{
    SHT21.begin();
    stab[0]=SHT21.getTemperature();
    delay(100);
    stab[1]=SHT21.getHumidity();
    Serial.printf("T:%2.2f, H:%2.2f\n",stab[0],stab[1]);
}

void setup() {
  Serial.begin(9600); delay(100);
  Wire.begin(12,14);
  termID=(uint32_t)ESP.getEfuseMac(); delay(100);
  set_LoRa();
  LoRa.onReceive(onReceive);
  LoRa.onTxDone(onTxDone);
  LoRa_rxMode();
  tsrcv_queue = xQueueCreate(queueSize, 64);  // to receive data paquets from onReceive() ISR
  con_queue = xQueueCreate(queueSize, 32);    // to receive control paquets from onReceive() ISR
  // the ThingSpeak server parameters - prepared once for all data packets
```

```
  ts.pack.chan=1243348; strncpy(ts.pack.key,"J4K8ZIWAWE8JBIX7",16);
  ts.pack.zero=0x00;ts.pack.mask=0xC0;
  ts.pack.pad[0]=0x00;ts.pack.pad[1]=0x00;
  // the above parameters may be read from external/internal EEPROM
  stage1_flag=1; stage2_flag=0;
}

int cycle=10;   // 10 seconds
char mess[17];

void loop() {
  if (runEvery(cycle*1000)) // main cycle may be modified dynamically by the Gateway node
    {
    if(cycle_cnt>10) { stage1_flag=1; stage2_flag=0; cycle_cnt=0; }
    else { Serial.printf("cycle=%d\n",cycle_cnt); cycle_cnt++; }

    if(stage1_flag)  // runs until IDACK sets stage1_flag to 1 - in RTC memory
      {
      conframe_t rcf;
      send_IDREQ("passwordpassword"); // send IDACK for service 1
      Serial.printf("IDREQ MODE=%x sent from TERMINAL: %08X\n",MODE,(uint32_t)termID);
      if(xQueueReceive(con_queue, rcf.frame, 10000)== pdPASS)  // set delay parameter
        {
        if(rcf.pack.con[0]==(uint8_t)(MODE*16+2) && rcf.pack.con[1]==0x77)
          {
          strncpy(CKEY,rcf.pack.pass,16); gwID=(uint32_t) rcf.pack.sid;
          stage1_flag=0; stage2_flag=1;
          }
        }
      }

    if(stage2_flag) // runs until DTACK sets stage2_flag to 1
      {
      TSframe_t rdf;
      Serial.println("sensor");
      get_sens();  // get stab[] values
      send_DTSND(ts.pack.mask,ts.pack.chan,ts.pack.key,stab);
      Serial.printf("DTSND MODE=%x from TERM:%08X\n",MODE,(uint32_t)termID);
      if(xQueueReceive(tsrcv_queue, rdf.frame, 10000)== pdPASS) // set delay parameter
        {
          Serial.printf("\nDTACK: cycle=%d, message=%s\n",rdf.pack.channel,rdf.pack.keyword);
          if(1000>rdf.pack.channel && rdf.pack.channel>10) cycle=rdf.pack.channel;
strncpy(mess,rdf.pack.keyword,16);
         // stage2_flag=0;   // stage2_flag set to 1 - resetting to 0
//        esp_sleep_enable_timer_wakeup(1000*1000*cycle);  // cycle in seconds
//        esp_sleep_pd_config(ESP_PD_DOMAIN_RTC_PERIPH, ESP_PD_OPTION_OFF);
//        Serial.println("Going to sleep now");
//        Serial.flush(); LoRa.end();delay(160);
//        esp_deep_sleep_start();
//        Serial.println("This will never be printed");
       }
    }

  }
}
```

11.2.1.2 Le code du terminal avec BH1750 - capteur de luminosité

La différence essentielle entre le code précédent et le code du terminal avec capteur de luminosité BH1750 réside dans le choix et l'utilisation du capteur :

```
#include <BH1750.h>
BH1750 lightMeter;
..

void get_sens()  // the sensor function
{
    lightMeter.begin();
    stab[2]=lightMeter.readLightLevel();
    delay(100);
    Serial.printf("L:%2.2f\n",stab[2]);
}
```

```
void setup() {
..
  ts.pack.zero=0x00;ts.pack.mask=0x20;  // luminosity sensor as 3rd sensor
  ts.pack.pad[0]=0x00;ts.pack.pad[1]=0x00;
..
}
```

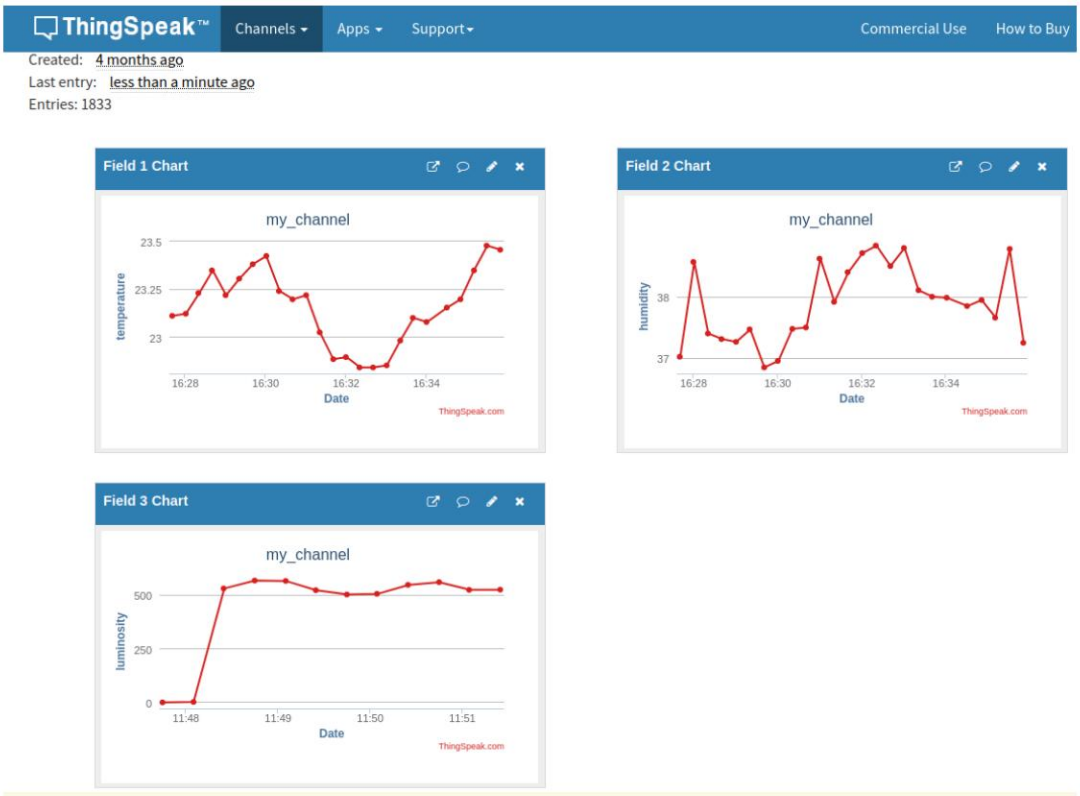

Figure 11.5 Diagramme ThingSpeak avec les données envoyées au même canal à partir de deux cartes séparées

11.3 Passerelle émetteur vers ThingSpeak - MODE 1

La passerelle d'envoi pour ThingSpeak relaie les trames LoRa reçues (**DTSND**) vers le serveur ThingSpeak (par exemple: **ThingSpeak.com**). La passerelle extrait la clé d'écriture, le numéro de canal et décode l'octet de masque - **con[1]** afin de préparer les champs du canal.

Toutes ces opérations sont exécutées séparément dans la tâche **TS_Task** présentée ci-dessous. La tâche attend le paquet de données sur **tsrcv_queue** :

```
xQueueReceive(tsrcv_queue,&rdf,portMAX_DELAY);
```

La file **tsrcv_queue** est écrite par les lignes de code correspondantes:

```
xQueueReset(tsrcv_queue);  // reset queue to keep only the last packet
xQueueSend(tsrcv_queue, rdf.frame, portMAX_DELAY);
```

exécuté dans dans la routine **onReceive()**.

La tâche **TS_Task** prend les données de la **tsrcv_queue** et les envoie au serveur ThingSpeak via une connexion WiFi. Pendant cette période de communication, les terminaux peuvent envoyer leurs paquets LoRa à la passerelle.

Afin de protéger la communication avec le serveur ThingSpeak, **TS_Task** met le modem LoRa en état inactif (**LoRa.idle)**), cet état est maintenu jusqu'à la fin du cycle **TS_Task** où l'on trouve la fonction **LoRa_rxMode()**. Cette fonction désactive le mode **InvertIQ** et met le modem LoRa en état de réception (côté passerelle).

```
void TS_Task( void * pvParameters ){
float stab[8]; char kbuff[32];
TSframe_t rdf;
while(true)
  {
  xQueueReceive(tsrcv_queue, &rdf, portMAX_DELAY);
  Serial.printf("channel:%d, mask=%x, wkey:%16.16s\n",rdf.pack.channel,
                rdf.pack.con[1],rdf.pack.keyword);

  LoRa.idle();
  if(rdf.pack.con[1] & 0x80) ThingSpeak.setField(1, rdf.pack.sens[0]);
  if(rdf.pack.con[1] & 0x40) ThingSpeak.setField(2, rdf.pack.sens[1]);
  if(rdf.pack.con[1] & 0x20) ThingSpeak.setField(3, rdf.pack.sens[2]);
  if(rdf.pack.con[1] & 0x10) ThingSpeak.setField(4, rdf.pack.sens[3]);
  if(rdf.pack.con[1] & 0x08) ThingSpeak.setField(5, rdf.pack.sens[4]);
  if(rdf.pack.con[1] & 0x04) ThingSpeak.setField(6, rdf.pack.sens[5]);
  if(rdf.pack.con[1] & 0x02) ThingSpeak.setField(7, rdf.pack.sens[6]);
  if(rdf.pack.con[1] & 0x01) ThingSpeak.setField(8, rdf.pack.sens[7]);
  memset(kbuff,0x00,32); strncpy(kbuff,rdf.pack.keyword,16);
  // write to the ThingSpeak channel
  int x = ThingSpeak.writeFields((uint32_t)rdf.pack.channel,kbuff);
  if(x == 200)
    { Serial.println("Channel update successful.");tss_flag=1;}
  else
    { Serial.println("Problem updating channel. HTTP error code " + String(x));tss_flag=2;}
  LoRa_rxMode();
  }
}
```

11.3.1 Code complet de la passerelle d'envoi TS - MODE 1

```
#define GATEWAY  // to TERMINAL or GATEWAY node
#define MODE  1  // sender (1), receiver (2) , publisher (3), subscriber (4) mode
#define CKEY "abcdefghijklmnop"  // AES key - 16 bytes
#define PASS "passwordpassword"
#include <WiFi.h>
#include "ThingSpeak.h"
#include <SPI.h>
#include <LoRa.h>
#include <Wire.h>
#include "LoRa_Para.h"
#include "LoRa_Packets.AES.h"
#include "SSD1306Wire.h"
SSD1306Wire display(0x3c, 12, 14);   // ADDRESS, SDA, SCL

const char* ssid = "Livebox-08B0";
const char* pass = "G79ji6dtE..VTPWmZP";
WiFiClient client;

int tss_flag=0;
int taskCore =0;

void TS_Task( void * pvParameters ){
float stab[8]; char kbuff[32];
TSframe_t rdf;
while(true)
  {
  xQueueReceive(tsrcv_queue, &rdf, portMAX_DELAY);
  Serial.printf("channel:%d,mask=%x,wkey:%16.16s\n",rdf.pack.channel,rdf.pack.con[1],
                rdf.pack.keyword);
  LoRa.idle();
  if(rdf.pack.con[1] & 0x80) ThingSpeak.setField(1, rdf.pack.sens[0]);
  if(rdf.pack.con[1] & 0x40) ThingSpeak.setField(2, rdf.pack.sens[1]);
  if(rdf.pack.con[1] & 0x20) ThingSpeak.setField(3, rdf.pack.sens[2]);
  if(rdf.pack.con[1] & 0x10) ThingSpeak.setField(4, rdf.pack.sens[3]);
  if(rdf.pack.con[1] & 0x08) ThingSpeak.setField(5, rdf.pack.sens[4]);
  if(rdf.pack.con[1] & 0x04) ThingSpeak.setField(6, rdf.pack.sens[5]);
  if(rdf.pack.con[1] & 0x02) ThingSpeak.setField(7, rdf.pack.sens[6]);
  if(rdf.pack.con[1] & 0x01) ThingSpeak.setField(8, rdf.pack.sens[7]);
  memset(kbuff,0x00,32); strncpy(kbuff,rdf.pack.keyword,16);

  // write to the ThingSpeak channel
  int x = ThingSpeak.writeFields((uint32_t)rdf.pack.channel,kbuff); // "J4K8ZIWAWE8JBIX7"
  if(x == 200)
    { Serial.println("Channel update successful.");tss_flag=1;}
  else
    { Serial.println("Problem updating channel. HTTP error code " + String(x));tss_flag=2;}
  LoRa_rxMode();
  }
}

void connect() {
  Serial.print("checking wifi...");
  while (WiFi.status() != WL_CONNECTED) {
    Serial.print(".");
    delay(1000);
  }
  Serial.println("\nIoT.GW1 - connected!");
}
```

Figure 11.5 Diagramme ThingSpeak avec les données envoyées au même canal à partir de deux cartes séparées

```
void CON_Task( void * pvParameters )
{
conframe_t rcf; int rec=0;
while(true)
  {
  if(xQueueReceive(con_queue, rcf.frame, portMAX_DELAY)== pdPASS)
    {
    termID=rcf.pack.sid;
    if(rcf.pack.con[0]==(uint8_t)(MODE*16+1) && !strncmp(rcf.pack.pass,PASS,16))
      {
       Serial.printf("recv IDREQ MODE=%x from TERM: %08X, pass=%16.16s\n",MODE,
                     termID,rcf.pack.pass);
       send_IDACK(CKEY,0x0000); // send IDACK for service 1 with CKEY
```

```
       Serial.printf("IDACK service=%x sent\n",MODE);
      }
    }
  }
}

void DATA_Task(void * pvParameters)
{
TSframe_t rdf;  int tout=10; char mess[24];
strcpy(mess,"control message");
while(true)
  {
  while(tss_flag==0)
    {
      Serial.println("wating for TS return");
      delay(1000);
    }
  if(tss_flag==1) {Serial.println("got TS return OK");   strcpy(mess,"TS update OK    ");}
  if(tss_flag==2) {Serial.println("got TS return ERROR");strcpy(mess,"TS update    ERR");}
  tss_flag=0; tout=5+random(10,20);
  send_DTACK(rdf.pack.con[1],tout,mess); // send DTACK: mask, channel from DTSND
  Serial.printf("DTACK to TERMINAL: %08X\n",termID);
  }
}

void setup() {
  Serial.begin(9600); delay(100);
  gwID=(uint32_t)ESP.getEfuseMac(); // we are in MASTER
  delay(100);

  WiFi.begin(ssid, pass);
  if (!client.connected()) { connect(); }
  Serial.println();Serial.println();
  Serial.println("WiFi connected");
  ThingSpeak.begin(client);  // Initialize ThingSpeak
  set_LoRa();
  LoRa.onReceive(onReceive);
  LoRa.onTxDone(onTxDone);
  LoRa_rxMode();
  tsrcv_queue = xQueueCreate(queueSize, 64);
  con_queue = xQueueCreate(queueSize, 32);

  xTaskCreatePinnedToCore(
                    TS_Task,   /* Function to implement the task */
                    "TS_Task", /* Name of the task */
                    10000,      /* Stack size in words */
                    NULL,       /* Task input parameter */
                    0,          /* Priority of the task */
                    NULL,       /* Task handle. */
                    taskCore);  /* Core where the task should run */
  Serial.println("TS_Task created...");

  xTaskCreatePinnedToCore(
                    CON_Task,   /* Function to implement the task */
                    "CON_Task", /* Name of the task */
                    10000,      /* Stack size in words */
                    NULL,       /* Task input parameter */
                    0,          /* Priority of the task */
                    NULL,       /* Task handle. */
                    taskCore);  /* Core where the task should run */
  Serial.println("CON_Task created...");

  xTaskCreatePinnedToCore(
                    DATA_Task,   /* Function to implement the task */
                    "DATA_Task", /* Name of the task */
                    10000,      /* Stack size in words */
                    NULL,       /* Task input parameter */
                    0,          /* Priority of the task */
                    NULL,       /* Task handle. */
                    taskCore);  /* Core where the task should run */
  Serial.println("DATA_Task created...");
}

void loop() {}
```

11.3.2 Passerelle - planificateur des terminaux

Le paramètre **timeout** à envoyer par la passerelle dans le numéro du canal (**`channel`**) peut être contrôlé par la passerelle afin de planifier le prochain paquet d'envoi depuis le nœud terminal.

L'ordonnanceur peut prendre en compte le nombre de nœuds terminaux enregistrés et décomposer le cycle global en tranches de temps, une tranche par terminal. La durée de chaque créneau doit être plus longue que la durée de la transaction, y compris la communication avec le terminal (LoRa) et le serveur ThingSpeak (WiFi).

Avec cette solution, le numéro du terminal correspond au numéro de su créneau.

11.3.3 Paramètres de canal ThingSpeak stockés dans l'EEPROM

Le terminal et les nœuds de passerelle ont besoin d'un certain nombre de paramètres pour communiquer entre eux et entre la passerelle et le serveur ThingSpeak. Par exemple, le terminal a besoin de connaître les paramètres du canal ThingSpeak avant d'envoyer un paquet **`DTSND`**.

Afin de fournir ces paramètres depuis l'extérieur, nous pouvons utiliser l'**EEPROM externe** et **interne**. Tout d'abord, l'EEPROM programmée en externe peut fournir les paramètres requis à la carte qui lit ces paramètres et les stocke dans la mémoire EEPROM interne.
La prochaine fois que le terminal ou la passerelle est activé il utilisera les paramètres stockés en interne pour démarrer les opérations.

Ce qui suit est le code (fonctions) dans le fichier **`EEPROM_TS.h`** qui peut être utilisé pour écrire et lire les paramètres du canal ThingSpeak vers et depuis les mémoires EEPROM externes et internes.

```
// EEPROM_TS.h
#include "EEPROM.h"
#include "Wire.h"
#include "I2C_eeprom.h"
I2C_eeprom ee(0x50, I2C_DEVICESIZE_24LC256);
union
{
  uint8_t para[24];
  struct
  {
    int chan;      // channel number
    char key[16];  // write key
    uint8_t zero;
    uint8_t mask;  // sensor mask
    uint8_t pad[2];
  } pack;
} ts,ts_test;  // TS send and receive

void write_ee_ts(int channel,char *key,uint8_t mask)
{
  Serial.begin(9600);
  Wire.begin(12,14);
  ee.begin();
  if (! ee.isConnected())
  {    Serial.println("ERROR: Can't find eeprom\nstopped..."); while (1);  }
  ts.pack.chan=channel;
  strncpy(ts.pack.key,key,16);ts.pack.zero=0x00;
  ts.pack.mask=mask;
  ts.pack.pad[0]=0x00;  ts.pack.pad[1]=0x00;
  Serial.println(); Serial.println("Write TS parameters to external EEPROM");
  for(unsigned int i=0;i<24;i++)    { ee.writeByte(i,ts.para[i]); }
}
void read_ee_ts()
{
  Serial.begin(9600);  // included for completeness
  Wire.begin(12,14);
  ee.begin();
  if (! ee.isConnected())
  {
    Serial.println("ERROR: Can't find eeprom\nstopped..."); while (1);
  }
  Serial.println("TS parameters in external EEPROM");
  for(unsigned int i=0;i<24;i++)
    {  ts_test.para[i]=ee.readByte(i); }
}
```

```
void write_ie_ts(int channel,char *key,uint8_t mask)
{
  Serial.begin(9600);
  if(!EEPROM.begin(24))
  {
    Serial.println("failed to initialise internal EEPROM"); while (1);
  }
  ts.pack.chan=channel;
  strncpy(ts.pack.key,key,16);ts.pack.zero=0x00;
  ts.pack.mask=mask;
  ts.pack.pad[0]=0x00;  ts.pack.pad[1]=0x00;
  Serial.println();
  Serial.println("Write TS parameters to internal EEPROM");
  for(unsigned int i=0;i<24;i++)
     { EEPROM.write(i,ts.para[i]); } // write to EEPROM buffer
  EEPROM.commit();  // send the EEPROM buffer to memory
}

void read_ie_ts()
{
  Serial.begin(9600);  // included for completeness
  if(!EEPROM.begin(24))
  {
    Serial.println("failed to initialise internal EEPROM");  while (1);
  }
  Serial.println("TS parameters in internal EEPROM");
  for(unsigned int i=0;i<24;i++)  {  ts_test.para[i]=byte(EEPROM.read(i)); }
  Serial.println();
}
```

11.3.3.Code pour tester les fonctions de la bibliothèque `EEPROM_TS.h`

```
#include "EEPROM_TS.h"

void setup()
{
  Serial.begin(9600);
  Wire.begin(12,14);
  Serial.println(" ");
  write_ee_ts(1243348,"J4K8ZIWAWE8JBIX7",0xC0);
  delay(1000);
  read_ee_ts();
  Serial.printf("Channel number: %d\n",ts_test.pack.chan);
  Serial.printf("Write/read key: %s\n",ts_test.pack.key);
  Serial.printf("mask value: %0X\n",ts_test.pack.mask);
  Serial.println(" ");
  delay(3000);
  write_ie_ts(1243348,"J4K8ZIWAWE8JBIX7",0xC0);
  delay(1000);
  read_ie_ts();
  Serial.printf("Channel number: %d\n",ts_test.pack.chan);
  Serial.printf("Write/read key: %s\n",ts_test.pack.key);
  Serial.printf("mask value: %0X\n",ts_test.pack.mask);
}

void loop() { }
```

11.3.4 A faire

1. Utilisez l'EEPROM externe et interne pour charger les paramètres ThingSpeak.
   Commencez avec une EEPROM externe
   - s'il est disponible, charger les paramètres dans l'EEPROM interne
   - s'il n'est pas disponible prendre les paramètres enregistrés dans l'EEPROM interne

2. Expérimentez avec la valeur du délai d'expiration pour «planifier» l'activation du terminal émetteur
3. Utilisez **`WiFiManager`** pour définir le point d'accès WiFi dans la passerelle

11.4 Terminal et Passerelle de réception (ThingSpeak) - MODE 2

Comme le terminal émetteur, le terminal récepteur fonctionne en 2 étapes:

1. la première étape permet aux terminaux de s'enregistrer à la passerelle et de recevoir la clé EAS et l'identifiant et de la passerelle via les paquets de contrôle **`IDREQ`** et **`IDACK.`** Il s'agit de la même transaction qu'en MODE 1
2. la deuxième étape est utilisée pour envoyer la requête - **`DTREQ`** et pour recevoir les données demandées du canal ThingSpeak via un paquet **`DTRCV`**

Le terminal récepteur (**TSR**) envoie les paquets de demande de données (**`DTREQ`**) et attend les paquets de réception de données (**`DTRCV`**). Chaque paquet de requête porte le **numéro du canal**, le mot-**clé de lecture** du canal et le **masque des champs** de données (8 bits) indiquant quels sont les champs de données d'intérêt.

Notez que les paquets **`DTRCV`** contiennent les données demandées ainsi qu'un nouveau **timeout** ou valeur de cycle dans **`rdf.pack.channel`** et un message de contrôle dans **`rdf.pack.keyword`** envoyé par la passerelle.

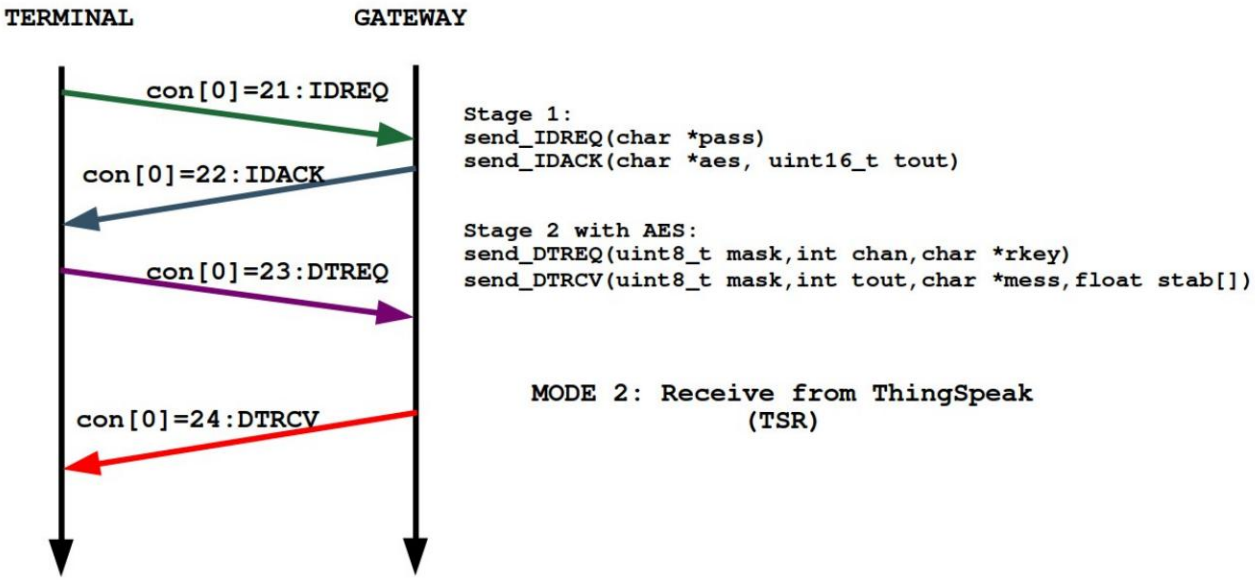

Figure 11.6 Diagramme de temps pour la communication entre un terminal et une passerelle TSR

11.4.1 Code complet d'un terminal - MODE 2

```
#define TERMINAL  // to choose TERMINAL or GATEWAY node
#define MODE 2  // to choose sender (1), receiver (2) , publisher (3), subscriber (4) mode
#include <SPI.h>
#include <LoRa.h>
#include "LoRa_Para.h"
#include "LoRa_Packets.AES.h"
#include "LoRa_onReceive.AES.h"  // to capture the packets with MODE and server/broker

#include <Wire.h>
#include "SSD1306Wire.h"

SSD1306Wire display(0x3c, 12, 14);

union
{
  uint8_t para[24];
  struct
  {
    int chan;      // channel number
    char key[16];  // write-read key
    uint8_t zero;
    uint8_t mask;  // sensor mask
    uint8_t pad[2];
  } pack;
} ts;  // TS send and receive para
```

```
void disp_sens(uint8_t mask, float *stab)
{
  char buff[32];
  display.init();
  display.flipScreenVertically();
  display.setFont(ArialMT_Plain_10);
  display.setTextAlignment(TEXT_ALIGN_LEFT);
  display.drawString(0,0,"Terminal TSR");  // first 16 lines are yellow
  if(mask&0x80) { sprintf(buff,"T:%2.2f",stab[0]);display.drawString(0,16,buff); }
  if(mask&0x40) { sprintf(buff,"H:%2.2f",stab[1]);display.drawString(0,28,buff); }
  if(mask&0x20) { sprintf(buff,"L:%4.2f",stab[2]);display.drawString(0,40,buff); }
  display.display();
}

void setup() {
  Serial.begin(9600); delay(100);
  Wire.begin(12,14);
  termID=(uint32_t)ESP.getEfuseMac(); delay(100);
  set_LoRa();
  LoRa.onReceive(onReceive);
  LoRa.onTxDone(onTxDone);
  LoRa_rxMode();
  con_queue = xQueueCreate(queueSize, 32);
  tsrcv_queue = xQueueCreate(queueSize, 64);
  stage1_flag=1;stage2_flag=0;
  ts.pack.chan=1243348; strncpy(ts.pack.key,"0XYA1MAWXFGVWDX9",16);
  ts.pack.zero=0x00;ts.pack.mask=0xE0;  // 3rd sensor
  ts.pack.pad[0]=0x00;ts.pack.pad[1]=0x00;
  // the above parameters may be read from external-internal EEPROM as 24 bytes
  stage1_flag=1; stage2_flag=0;
}

int cycle=10;   // 10 seconds

void loop()
{
if(runEvery(cycle*1000))
  {
  if(cycle_cnt>10) { stage1_flag=1; stage2_flag=0;cycle_cnt=0; }
  else { Serial.printf("cycle=%d\n",cycle_cnt); cycle_cnt++; }
  if(stage1_flag)  // runs until IDACK sets stage1_flag to 1 - in RTC memory
      {
      conframe_t rcf;
      send_IDREQ("passwordpassword"); // send IDACK for service 1
      Serial.printf("IDREQ MODE=%x sent from TERMINAL: %08X\n",MODE,(uint32_t)termID);
      if(xQueueReceive(con_queue, rcf.frame, 10000)== pdPASS)  // set delay parameter
        {
        if(rcf.pack.con[0]==(uint8_t)(MODE*16+2) && rcf.pack.con[1]==0x77)
          {
          strncpy(CKEY,rcf.pack.pass,16); gwID=(uint32_t) rcf.pack.sid;
          stage1_flag=0; stage2_flag=1;
          }
        }
      }
  if(stage2_flag)
    {
    TSframe_t rdf;
    Serial.printf("DTREQ MODE=%x sent to GW: %08X\n",MODE,(uint32_t)gwID);
    send_DTREQ(ts.pack.mask,ts.pack.chan,ts.pack.key);  // channel and read key
    if(xQueueReceive(tsrcv_queue, rdf.frame, 10000)== pdPASS)
      {
      Serial.printf("\nDTRCV: cycle=%d sec,%s,%x\n", rdf.pack.channel, rdf.pack.keyword,
                    rdf.pack.con[0]);
      if(10<rdf.pack.channel && rdf.pack.channel<3600) // to eliminates eventual error
        {
        disp_sens(ts.pack.mask,rdf.pack.sens);
        Serial.printf("field1=%2.2f,field2=%2.2f\n",rdf.pack.sens[0],rdf.pack.sens[1]);
        cycle=rdf.pack.channel;  // eliminates errors
        }
      }
    }
  }
}
```

11.5 Passerelle de réception (TSR) - MODE 2

Par rapport à un terminal la passerelle de réception fonctionne en 2 étapes:

1. la première étape permet aux terminaux de s'enregistrer (**IDREQ**) à la passerelle et de recevoir la clé AES et l'identifiant de la passerelle via un paquet **IDACK**
2. dans la deuxième étape, la passerelle reçoit les requêtes de données (**DTREQ**) qui sont relayées vers le serveur ThingSpeak; après la réception des données du serveur ThingSpeak, la passerelle les envoie au terminal via un paquet de type **DTRCV**.

La passerelle de réception reçoit les demandes de données par le biais de l'ISR **onReceive()** qui met les paquets la file d'attente (**tsrcv_queue**). Cette file d'attente est lue par le **TS_Task** et les données demandées sont extraites du serveur ThingSpeak. Ensuite, par le biais de la tâche **DATA_Task** le paquet de type **DTRCV** avec les données reçues est envoyée au terminal concerné.

11.5.1 Le code de la passerelle du récepteur

```
#define GATEWAY  // to choose TERMINAL or GATEWAY node
#define MODE 2  // to choose sender (1), receiver (2) , publisher (3), subscriber (4) mode
#define CKEY "abcdefghijklmnop"  // AES key - 16 bytes
#define PASS "passwordpassword"
#include <WiFi.h>
#include "ThingSpeak.h"
#include <SPI.h>
#include <LoRa.h>
#include "LoRa_Para.h"
#include "LoRa_Packets.AES.h"
#include "LoRa_onReceive.AES.h"  // to capture the packets with MODE and server/broker
const char* ssid = "Livebox-08B0";
const char* pass = "G79ji6dtEptVTPWmZP";
WiFiClient client;
int taskCore = 0;
float stab[8];
int statusCode=0,tss_flag=0;

void TS_Task( void * pvParameters ){
char kbuff[32];
TSframe_t rdf; int TSdelay=1000;
while(true)
  {
  xQueueReceive(tsrcv_queue, &rdf, portMAX_DELAY);
  Serial.println(rdf.pack.channel);Serial.println(rdf.pack.keyword);
  LoRa.idle();
    memset(kbuff,0x00,32); strncpy(kbuff,rdf.pack.keyword,16);
  if(rdf.pack.con[1] & 0x80)
    { stab[0]=ThingSpeak.readFloatField(rdf.pack.channel,1,kbuff);delay(TSdelay); }
  if(rdf.pack.con[1] & 0x40)
    { stab[1]=ThingSpeak.readFloatField(rdf.pack.channel,2,kbuff);delay(TSdelay); }
  if(rdf.pack.con[1] & 0x20)
   { stab[2]=ThingSpeak.readFloatField(rdf.pack.channel,3,kbuff);delay(TSdelay); }
  if(rdf.pack.con[1] & 0x10)
   { stab[3]=ThingSpeak.readFloatField(rdf.pack.channel,4,kbuff);delay(TSdelay); }
  if(rdf.pack.con[1] & 0x08)
   { stab[4]=ThingSpeak.readFloatField(rdf.pack.channel,5,kbuff);delay(TSdelay); }
  if(rdf.pack.con[1] & 0x04)
   { stab[5]=ThingSpeak.readFloatField(rdf.pack.channel,6,kbuff);delay(TSdelay); }
  if(rdf.pack.con[1] & 0x02)
   { stab[6]=ThingSpeak.readFloatField(rdf.pack.channel,7,kbuff);delay(TSdelay); }
  if(rdf.pack.con[1] & 0x01)
   { stab[7]=ThingSpeak.readFloatField(rdf.pack.channel,8,kbuff);delay(TSdelay); }
  statusCode = ThingSpeak.getLastReadStatus();
  if(statusCode == 200) tss_flag=1;
  else tss_flag=2;
  LoRa_rxMode();
  }
}

void connect() {
  Serial.print("checking wifi...");
  while (WiFi.status() != WL_CONNECTED) {
    Serial.print(".");  delay(1000);
  }
  Serial.println("\nIoT.GW1 - connected!");
}
```

```
void CON_Task( void * pvParameters )
{
conframe_t rcf; int rec=0;
while(true)
  {
  if(xQueueReceive(con_queue, rcf.frame, portMAX_DELAY)== pdPASS)
    {
    termID=rcf.pack.sid;
    if(rcf.pack.con[0]==(uint8_t)(MODE*16+1) && !strncmp(rcf.pack.pass,PASS,16))
      {
       Serial.printf("recv IDREQ MODE=%x from TERM: %08X, pass=%16.16s\n",MODE,
termID,rcf.pack.pass);
       send_IDACK(CKEY,0x0000); // send IDACK for service 1 with CKEY
       Serial.printf("IDACK service=%x sent\n",MODE);
      }
    }
  }
}

void DATA_Task(void * pvParameters)
{
TSframe_t rdf;  int tout=10; char mess[24];
strcpy(mess,"control message");
while(true)
  {
  while(tss_flag==0)
    {
      Serial.println("wating for TS return");
      delay(1000);
    }
  tout=10+random(10,20);
  // we send the time-out in the channel field
  if(tss_flag==1) strcpy(mess,"last read OK   ");
  if(tss_flag==2) strcpy(mess,"last read error");

  send_DTRCV(rdf.pack.con[1],tout,mess,stab);  tss_flag=0;
  Serial.printf("DTRCV to TERM:%08X;tout=%d,stab[0]=%2.2f,mess=%16.16s\n",termID,tout,stab[0],mess);
  }
}

void setup() {
  Serial.begin(9600); delay(100);
  gwID=(uint32_t)ESP.getEfuseMac();
  WiFi.begin(ssid, pass);
  if (!client.connected()) { connect(); }
  Serial.println();Serial.println();
  Serial.println("WiFi connected");
  ThingSpeak.begin(client);  // Initialize ThingSpeak
  set_LoRa();
  LoRa.onReceive(onReceive);
  LoRa.onTxDone(onTxDone);
  LoRa_rxMode();
  con_queue = xQueueCreate(queueSize, 32);
  tsrcv_queue = xQueueCreate(queueSize, 64);
  xTaskCreatePinnedToCore(
                    TS_Task,   /* Function to implement the task */
                    "TS_Task", /* Name of the task */
                    10000,      /* Stack size in words */
                    NULL,       /* Task input parameter */
                    0,          /* Priority of the task */
                    NULL,       /* Task handle. */
                    taskCore);  /* Core where the task should run */
  Serial.println("TS_Task created...");
    xTaskCreatePinnedToCore(
                    CON_Task,   /* Function to implement the task */
                    "CON_Task", /* Name of the task */
                    10000,      /* Stack size in words */
                    NULL,       /* Task input parameter */
                    0,          /* Priority of the task */
                    NULL,       /* Task handle. */
                    taskCore);  /* Core where the task should run */
  Serial.println("CON_Task created...");

  xTaskCreatePinnedToCore(
                    DATA_Task,   /* Function to implement the task */
                    "DATA_Task", /* Name of the task */
```

```
                    10000,      /* Stack size in words */
                    NULL,       /* Task input parameter */
                    0,          /* Priority of the task */
                    NULL,       /* Task handle. */
                    taskCore);  /* Core where the task should run */
  Serial.println("DATA_Task created...");
}

void loop() { }
```

11.6 A faire

1. Utilisez deux ou trois terminaux pour tester les codes ci-dessus.
2. Écrivez une application pour programmer l'EEPROM externe avec différents mots-clés et paramètres, notamment:
    - pour terminal et passerelle: paramètres de liaison LoRa - fréquence, bande d'onde du signal, facteur d'étalement, code de synchronisation et mots de passe de liaison
    - pour terminal: paramètres ThingSpeak - numéro de canal, mots-clés d'écriture / lecture, masque de capteur
    - pour passerelle: identifiants WiFi

3. Modifiez les codes du terminal et de la passerelle, y compris l'utilisation de mémoires EEPROM externes et internes. Lors de la phase d'initialisation, le terminal / la passerelle essaie de lire l'EEPROM externe. Si elles sont présentes, les données sont transférées de l'EEPROM externe vers l'EEPROM interne. Sinon, les nœuds Terminal / Gateway sont activés directement avec les données de l'EEPROM interne.