Lab 6

Mise en ouvre des liens radio Long Range (LoRa) et d'une architecture IoT basée sur les liens LoRa.

Dans le TP précédant nous avons introduit une architecture IoT avec la communication radio basée sur les modules HC-12. Ces modules permettent de bâtir un système multi-terminaux avec une passerelle basée sur une carte SBC type RPIZ-W ou autre.
Pour construire une architecture IoT à grande échelle nous avons besoin de modems plus performant avec de liens radio sur **plusieurs kilomètres**, avec les codes de sécurité et de protocoles multi-accès.
Actuellement (2017) , la solution la plus ouverte et la plus« standard » est l'exploitation de la technologie et des modems LoRa (Long Range).
La technologie LoRa , inventé en France, a été achetée par une société Canadienne Semtech. La fabrication en masse de circuits (SX1276,SX1278, ..) conçus par cette société a permit une large diffusion de cette technologie.
Le prix d'un module LoRa intégrant ces circuit varie entre 4 et 8€ selon le modèle.

Part Number	Frequency Range	Spreading Factor	Bandwidth	Effective Bitrate	Est. Sensitivity
SX1276	137 - 1020 MHz	6 - 12	7.8 - 500 kHz	.018 - 37.5 kbps	-111 to -148 dBm
SX1277	137 - 1020 MHz	6 - 9	7.8 - 500 kHz	0.11 - 37.5 kbps	-111 to -139 dBm
SX1278	137 - 525 MHz	6- 12	7.8 - 500 kHz	.018 - 37.5 kbps	-111 to -148 dBm
SX1279	137 - 960MHz	6- 12	7.8 - 500 kHz	.018 - 37.5 kbps	-111 to -148 dBm

Figure 6.1 Paramètres de modulation LoRa des circuits SX1276-SX1279.

6.1 LoRa modulation

Les circuits opèrent sur une large bande passante. Pour l'instant France seulement deux sous-bandes sont autorisées pour le déploiement des systèmes IoT – 433- 435 et 868-870 MHz.
Dans la théorie de l'information, le théorème établit la capacité de canal de Shannon pour un lien de communication et définit le débit binaire maximal pouvant être transmis dans une bande passante spécifiée en présence du bruit:

$$C = B^* \log_2(1+S/N)$$

Où :
C = capacité du canal (bit/s)
B = bande passante du canal (Hz)
S = puissance moyenne du signal reçu (Watts)
N = puissance moyenne de bruit ou d'interférence (Watts)
S/N = rapport signal/bruit (SNR)

En transformant dans l'équation ci-dessus le logarithme à base de base 2 vers le logarithme naturel, et en notant que ln = $\log_e$, nous pouvons écrire :

$$C/B = 1.433 * (S/N)$$

Pour les applications à spectre étalé, le rapport S/N est petit, car la puissance du signal est souvent inférieure à la puissance du bruit. En supposant un niveau de bruit tel que S/N << 1, la dernière équation peut être réécrite comme suit:

$$C/B = S/N \quad => \quad C = B^*S/N$$

On peut voir que pour augmenter la capacité du canal , avec S/B fixé, seule la bande passante du signal transmis doit être augmentée.

6.1.1 Principe du spectre étalé

En augmentant la bande passante B du signal, nous pouvons compenser le dégradation du signal au bruit d'un canal radio.

Dans le spectre étalé avec une séquence directe d'étalement (DSSS), la phase de la porteuse de l'émetteur change en fonction de la séquence du code d'étalement .

Ce processus est généralement atteint en multipliant le signal de données avec un code d'étalement, aussi connue comme ***chip sequence***. La ***chip sequence*** évolue de façon beaucoup plus rapide que le signal de données et, par conséquent, a besoin d'une bande passante largement plus grande que la bande passante occupée par le simple signal original.

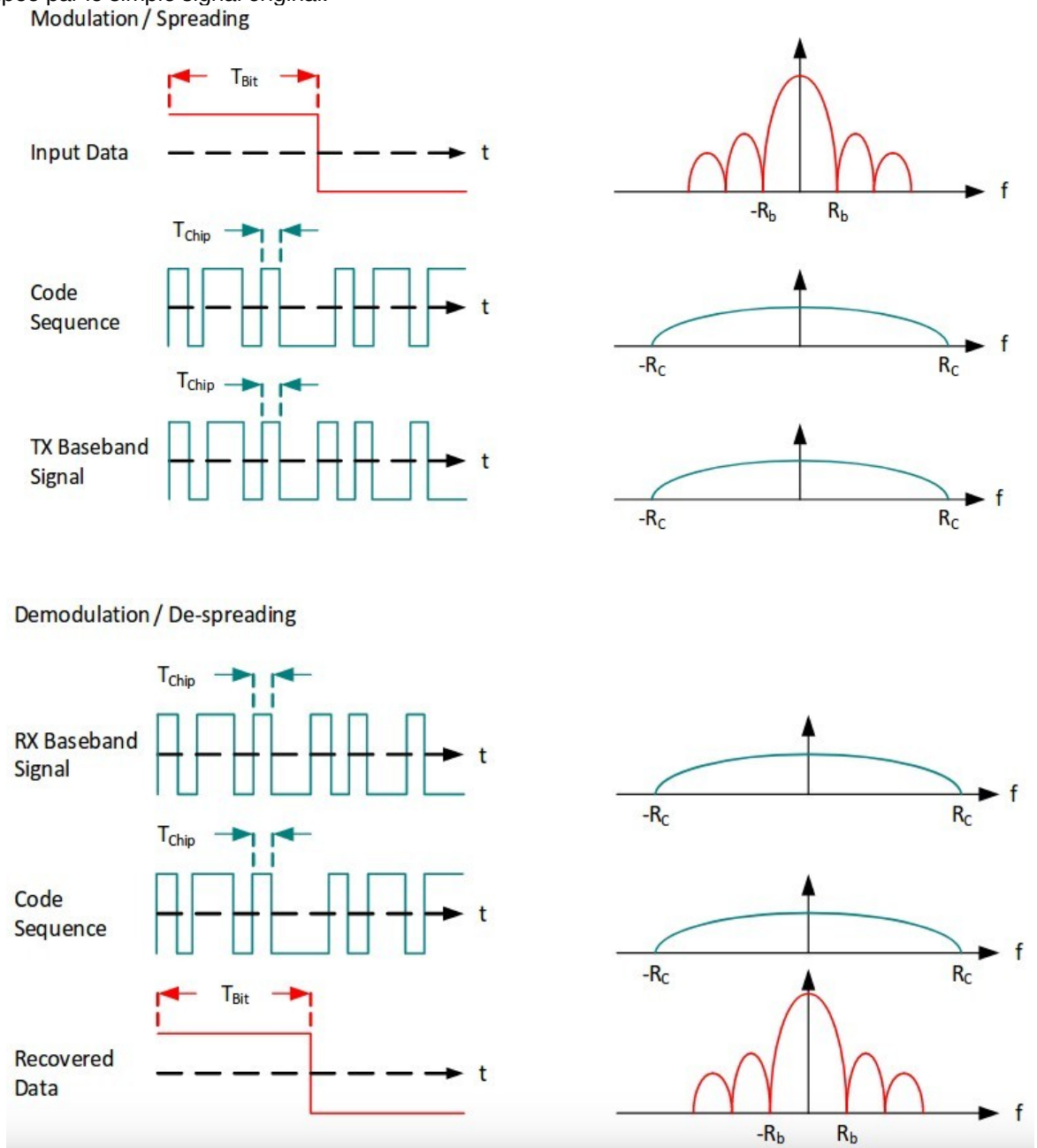

Figure 6.2 Modulation (LoRa) avec le spectre étalé : ***spreading*** et ***de-spreading***

A la réception, le signal de données désiré est récupérable en multipliant de nouveau par la même séquence d'étalement.

La valeur d'étalement, pour la séquence directe, est dépendant du rapport ratio "chips par bit" – il est appelé gain de traitement (**Gp**) communément exprimé en dB.

$$G_p = 10*\log_{10}(R_c/R_b) \ \ (dB)$$

où :

R_c/- chip rate
R_b - bit rate

Cette technique fournit un gain de traitement inhérent pour une transmission (qui permet au récepteur à récupérer correctement le signal de données même lorsque le S/R du canal est une valeur négative).
Les signaux d'interférence sont également réduits par le gain de traitement du récepteur. Ceux-ci se propagent au-delà de la bande passante d'information et peuvent être facilement supprimés par filtration.

6.1.2 Spectre étalé de LoRa

La modulation LoRa exploite la technique DSSS pour fournir une solution de bas-coût et faible puissance Elle offre une alternative robuste aux techniques de communications traditionnelles basées sur spectre étalé.
Dans la modulation LoRa, l'étalement du spectre est obtenu en générant un signal de ***chirp*** qui varie continuellement en fréquence.
Un avantage de cette méthode est que les décalages de synchronisation et de fréquence entre l'émetteur et le récepteur sont équivalents, ce qui réduit considérablement la complexité du récepteur. La bande passante de fréquence d'un **chirp** est équivalente à la bande passante du signal.

Le signal de données est étalé et modulé avec la fréquence de **chirp**. La relation entre le débit binaire désiré, le taux de symboles et le taux d'étalement pour la modulation LoRa peuvent être exprimé comme suit:

$$R_b = SF* \ 1/(2^{SF}/Bw) \quad (bit/s)$$

où :

R_b – débit binaire
SF – facteur d'étalement (7,..,12)
Bw – fréquence de modulation (Hz)

Les données binaires envoyées sont protégées par un code de correction , à chaque demi-octet (4-bit) est associé un code de correction qui peut être de taille de 1 à 4 bits. En prenant en compte ce code (code rate – CR) nous obtenons le débit binaire utile comme suit :

$$R_b = SF* \ 1/(2^{SF}/Bw) * 4/(4+CR) \quad (bit/s)$$

Les circuits SX1276,SX1278.. implémentent intégralement les techniques de modulation LoRa. Ils sont la base de production de modules RFM9X.

6.2 Modems LoRa RFM95/96/97/98 et cartes d'extension

Les émetteurs-récepteurs RFM95 / 96/97/98 (W) sont les modem à longue portée (long range) qui offre une communication basée sur le spectre étalé qui donne une forte immunité aux interférences tout en minimisant la consommation de courant.

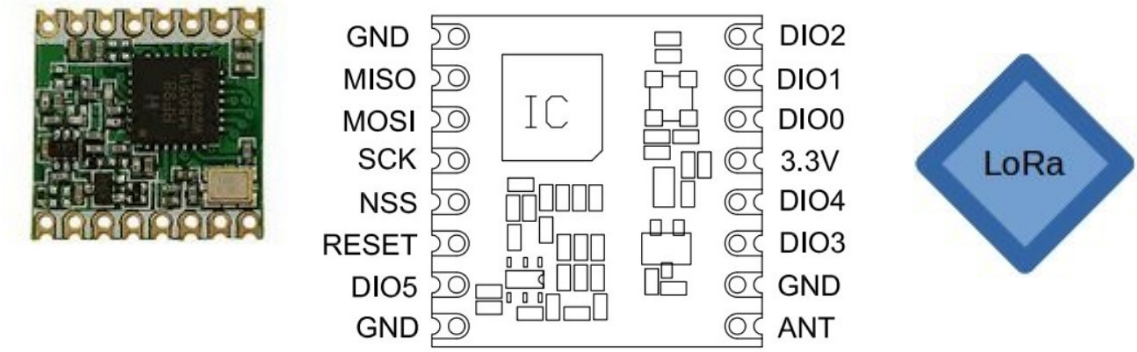

Figure 6.3 Le module RFM95/6/7/8 son schéma de connecteurs et le symbole.

La haute sensibilité combinée avec +20 dBm l'amplificateur de puissance à +20dBm (100 mW) rend ce module optimal pour toute application nécessitant une bonne portée ou robustesse.
Le module se connecte avec une carte MCU ou SBC via le bus SPI. Le disponibilité de PCBs pour les cartes Wemos D1 facilite intégration des modules RFM9X dans les architectures IoT fonctionnant comme objet terminal ou comme une passerelle LoRa – WiFi (***gateway***).

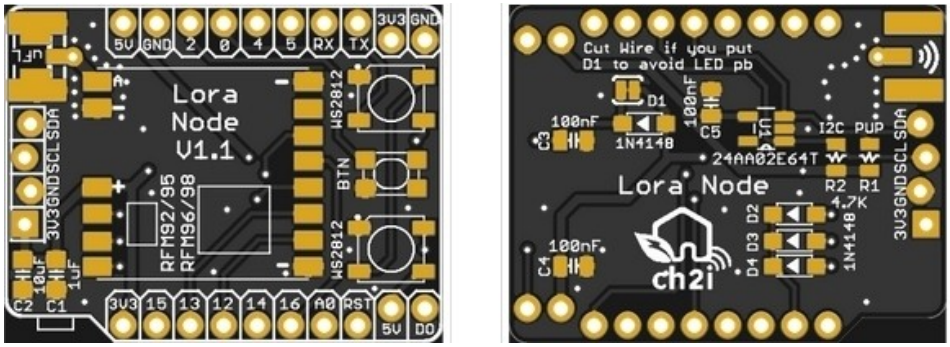

Figure 6.4 Une carte d'extension Wemos D1 pour le circuit RFM9X

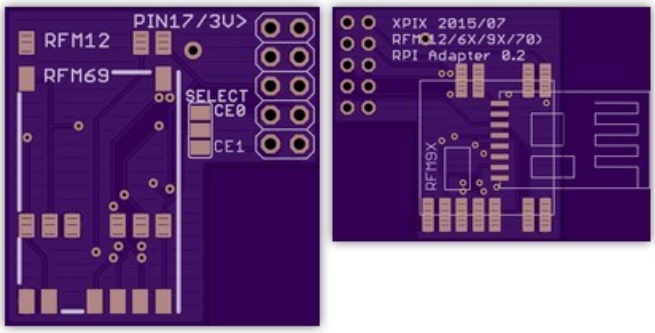

Figure 6.5 Une carte (PCB) d'extension pour le module RFM9X et RPIZ-W

6.3 Programmation avec des modems LoRa RFM95/96/97/98

Il existe plusieurs plateformes de programmation permettant d'exploiter les fonctionnalités des module RFM9X. Pour les raison de disponibilité et de la compatibilité nous avons choisi la bibliothèque **`RadioHead`**. La bibliothèque **`RadioHead`** offre un nombre important de drivers pour les modules radio connectables aux cartes MCU et SBC sur leurs interfaces SPI, dont **`RH_RF95.h`**, **`RH_RF95.cpp`** nécessaires pour l'utilisation de circuits type RFM9X.

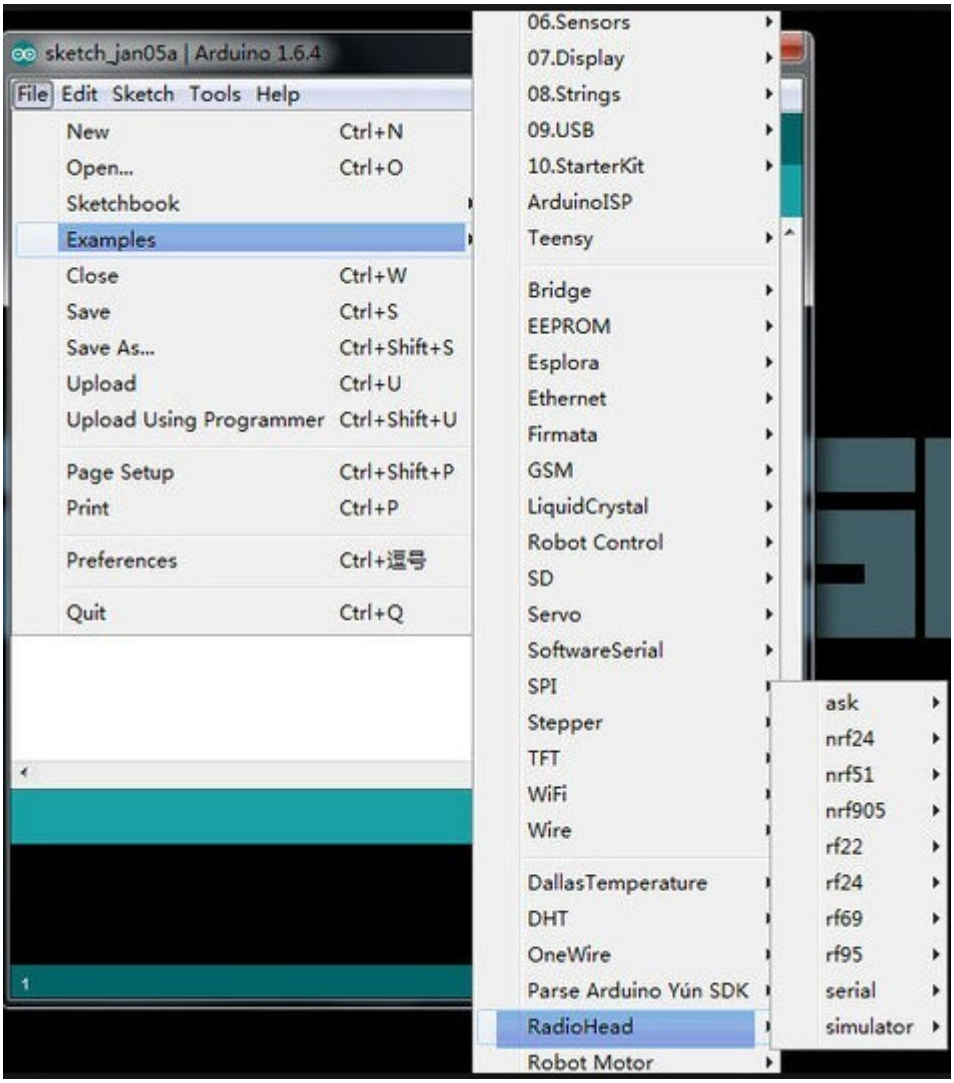

Figure 6.6 Utilisation de RadioHead dans l'environnement Arduino IDE

6.4 Une architecture avec des modems LoRa et cartes Wemos D1

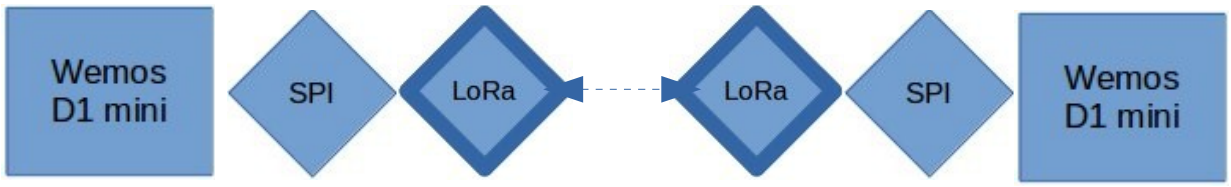

Figure 6.7 Une architecture IoT client-serveur avec un simple lien LoRa

L'exemple du code pour la carte Wemos D1: **`rf95_client puis rf95_server sont fournis ci-dessous.`**

Notons que les paramètres par défaut sont :

434.0MHz,
13dBm,
Bw = 125 kHz,
Cr = 4/5,
Sf = 128chips/symbol, CRC on

La puissance d'émission pour le module 434MHz est 13dBm. Si on utilise les modules RFM9X fonctionnant en fréquence 868MHz avec l'option **`PA_BOOST`** on peut configurer la puissance d'émission antre 5 et 23 dBm (200mW) par la fonction :

```
driver.setTxPower(23, false);
```

```
#include <SPI.h>
#include <RH_RF95.h>
RH_RF95 rf95(D4,D8);   //  RFM9X connected  Wemos D1
float frequency = 868.00;

void setup()
{

  Serial.begin(9600);
  if (!rf95.init())
    Serial.println("init failed");
     rf95.setFrequency(frequency);
     // Setup Power,dBm
     rf95.setTxPower(23,false)
}

void loop()
{
  Serial.println("Sending to rf95_server");
  // Send a message to rf95_server
  uint8_t data[] = "Hello World!";
  rf95.send(data, sizeof(data));

  rf95.waitPacketSent();
  // Now wait for a reply
```

```
  uint8_t buf[RH_RF95_MAX_MESSAGE_LEN];
  uint8_t len = sizeof(buf);

  if (rf95.waitAvailableTimeout(3000))
  {
    // Should be a reply message for us now
    if (rf95.recv(buf, &len))
   {
      Serial.print("got reply: ");
      Serial.println((char*)buf);
      Serial.print("RSSI: ");
      Serial.println(rf95.lastRssi(), DEC);
    }
    else
    {
      Serial.println("recv failed");
    }
  }
  else
  {
    Serial.println("No reply, is rf95_server running?");
  }
  delay(400);
}
```

Le programme du serveur **rf95_server** fonctionnant avec une autre carte Wemos D1

```
#include <SPI.h>
#include <RH_RF95.h>
RH_RF95 rf95(D4,D8);   //  driver connected to for Wemos D1
float frequency = 868.00;

void setup()
{
   Serial.begin(9600);
  if (!rf95.init())
    Serial.println("init failed");
     rf95.setFrequency(frequency);
     // Setup Power,dBm
     rf95.setTxPower(23,false)
}

void loop()
{
  if (rf95.available())
  {
    // Should be a message for us now
    uint8_t buf[RH_RF95_MAX_MESSAGE_LEN];
    uint8_t len = sizeof(buf);
    if (rf95.recv(buf, &len))
```

```
    {
      digitalWrite(led, HIGH);
      RH_RF95::printBuffer("request: ", buf, len);
      Serial.print("got request: ");
      Serial.println((char*)buf);
      Serial.print("RSSI: ");
      Serial.println(rf95.lastRssi(), DEC);

      // Send a reply
      uint8_t data[] = "And hello back to you";
      rf95.send(data, sizeof(data));
      rf95.waitPacketSent();
      Serial.println("Sent a reply");
       digitalWrite(led, LOW);
    }
    else
    {
      Serial.println("recv failed");
    }
  }
}
```

6.5 Une architecture des modems LoRa et cartes RPIZ(W), Wemos D1

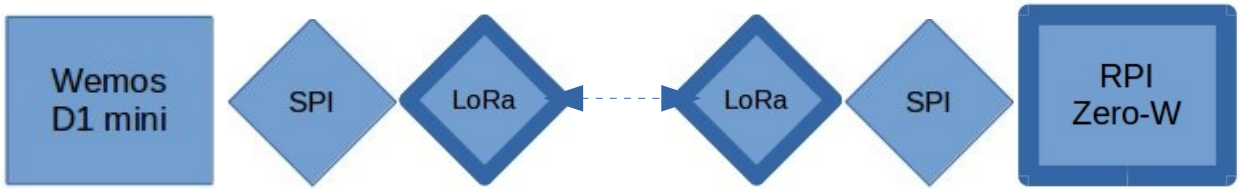

Figure 6.8 Une architecture IoT client-serveur avec un simple lien LoRa entre Wemos D1 et RPIZ(W)

6.5.1 Code C du client sur RPIZ(W) :

Dans ce programme nous pouvons choisir les valeurs de plusieurs paramètres incluant **mode** et **frequency**.

Le mode contient plusieurs éléments :

BW – bande passante,
CR – code rate,
SF – facteur d'étalement

Ces paramètres sont enregistrés dans le fichier **`RH_RF95.h`** .Il est possible de les modifier ou ajouter les nouveaux modes.

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <signal.h>
#include <RH_RF95.h>
RH_RF95 rf95(8,25); // LoraSpiShield - small shield
/* The address of the node which is 10 by default */
uint8_t devnum = 10;
uint8_t msg[32]; // {10, 0}
int mode=1; // default mode
float freq=868.0; // default frequency
uint8_t buf[RH_RF95_MAX_MESSAGE_LEN];
uint8_t len = sizeof(buf);
int mlen,i=0;
char mesbuff[32];

void setup(int sc, char *sa[])
{
    wiringPiSetupGpio();
    if (!rf95.init())
    {
        fprintf(stderr, "Init failed\n");
        exit(1);
    }
    else
        printf("Init ok\n");
    printf("Available modes with default frequency\n");
    printf("Mode 1 (default) : BW=125KHz, CR=45, SF=128 - frequency:%f\n",freq);
    printf("Mode 2: BW=31.25KHz, CR=48, SF=512 - frequency:%f\n",freq);
    printf("Mode 3: BW=125KHz, CR=48, SF=4096 - frequency:%f\n",freq);
```

```
    printf("Mode 4: BW=500KHz, CR=45, SF=128 - frequency:%f\n",freq);
    printf("Mode 5: BW=125KHz, CR=48, SF=1024 - frequency:%f\n",freq);
    printf("Use:\n sudo ./bakolora mode_number frequency\n");
    if(sc==1) { mode=1; freq=868.0; }
    if(sc==2) { mode=atoi(sa[1]); freq=868.0; }
    if(sc==3) { mode=atoi(sa[1]); freq=atof(sa[2]); }
    if(mode==1)
        {
        rf95.setModemConfig(RH_RF95::Bw125Cr45Sf128);
        printf("Mode 1: BW=125KHz, CR=45, SF=128 - frequency:%f\n",freq);
        }
    if(mode==2)
        {
        rf95.setModemConfig(RH_RF95::Bw31_25Cr48Sf512);
        printf("Mode 2: BW=31.25KHz, CR=48, SF=512 - frequency:%f\n",freq);
        }
    if(mode==3)
        {
        rf95.setModemConfig(RH_RF95::Bw125Cr48Sf4096);
        printf("Mode 3: BW=125KHz, CR=48, SF=4096 - frequency:%f\n",freq);
        }
    if(mode==4)
        {
        rf95.setModemConfig(RH_RF95::Bw500Cr45Sf128);
        printf("Mode 4: BW=500KHz, CR=45, SF=128 - frequency:%f\n",freq);
        }
    if(mode==5)
        {
        rf95.setModemConfig(RH_RF95::Bw125Cr48Sf1024);
        printf("Mode 5: BW=125KHz, CR=48, SF=1024 - frequency:%f\n",freq);
        }
rf95.setFrequency(freq);
if(freq>=433.0 && freq<435.0) rf95.setTxPower(14); //RFM98
if(freq>=868.0 && freq<=870.0) rf95.setTxPower(23); //RFM95
if(freq<433.0 || (freq>435.0 && freq<868.0) || freq>870)
        {
        printf("wrong frequency\n");
        printf("must be between 433.0 and 435.0 or 868.0 and 870.0 !\n");
        exit(1); }
msg[0]=devnum;
}
void loop()
{
while(1)
  {
    printf("Write your message (max 31 char):\n");
    memset(mesbuff,0x00,32);
    scanf("%s",mesbuff);
    mlen=strlen(mesbuff);
    for(i=0;i<mlen;i++) msg[i+1]=(uint8_t)mesbuff[i];
```

```
    rf95.send(msg, sizeof(msg));
    rf95.waitPacketSent();
    printf("Send!\n");
    usleep(10);
  if( rf95.waitAvailableTimeout(30000) )
  {
   memset(buf,0x00,len);
   if (rf95.recv(buf, &len))
    {
    printf("got reply RSSI= %d\n", rf95.lastRssi());
    printf("%s\n", buf);
    }
   else printf("rcv failed\n");
  }
  else printf("no reply in 30 sec\n");
  sleep(3);
  }
}
```

6.5.2 Code Arduino sur la carte Wemos D1 - serveur

```
#include <SPI.h>
#include <RH_RF95.h>
RH_RF95 rf95(D4,D8);   //  RFM9X connected  Wemos D1
float frequency = 868.00;

uint8_t data[64];
int i;

void setup()
{
  Serial.begin(9600);
  if (!rf95.init())
    Serial.println("init failed");
  else
    Serial.println("init ok");
     rf95.setFrequency(frequency);
     // Setup Power,dBm
     rf95.setTxPower(23,false);
}

void loop()
{
  if (rf95.available())
  {
    // Should be a message for us now
    uint8_t buf[RH_RF95_MAX_MESSAGE_LEN];
    uint8_t len = sizeof(buf);
```

```
    if (rf95.recv(buf, &len))
    {
//      RH_RF95::printBuffer("request: ", buf, len);
      Serial.print("got request: ");
      Serial.println((char*)buf);
//      Serial.print("RSSI: ");
//      Serial.println(rf95.lastRssi(), DEC);

      // Send a reply
      i=0; memset(data,0x00,64);
      while(!Serial.available());
      while(Serial.available()>0)
      { data[i]=(uint8_t)Serial.read();i++; }
      rf95.send(data, sizeof(data));
      rf95.waitPacketSent();
      Serial.println("Sent a reply");
    }
    else
    {
      Serial.println("recv failed");
    }
  }
}
```

6.6 L'envoi des données de capteurs sur un lien LoRa

Dans cette section nous allons passer à la transmission des donnée captées par les liens LoRa entre un terminal Wemos D1 et le serveur (passerelle) sur la carte RPIZ(W).

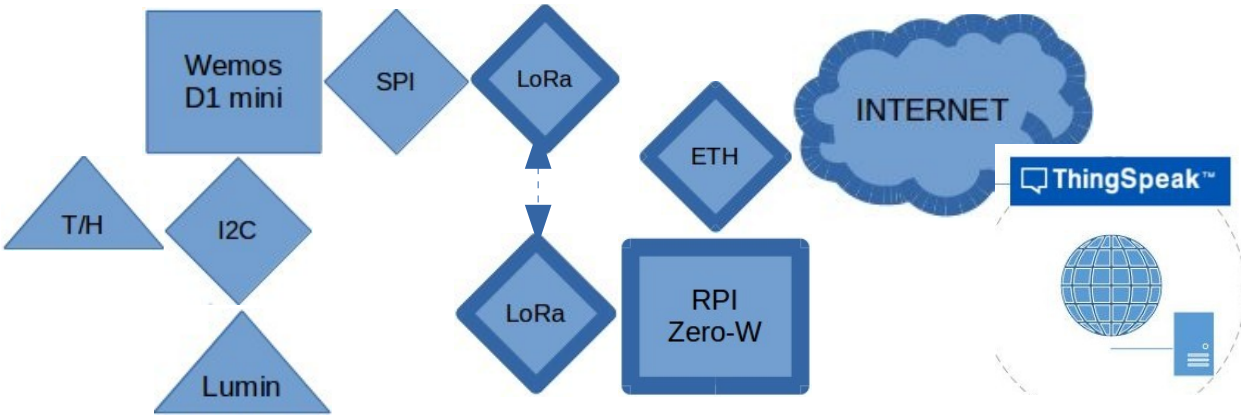

Figure 6.9 Une architecture IoT client-serveur avec un simple lien LoRa entre Wemos D1 et RPIZ(W)

L'architecture ci-dessus exploite deux capteurs connectés sur le bus I2C, SHT30 (température-humidité) et BH1750 (luminosité).
Les données captées par sont préparées dans une trame illustrée ci-dessous. :

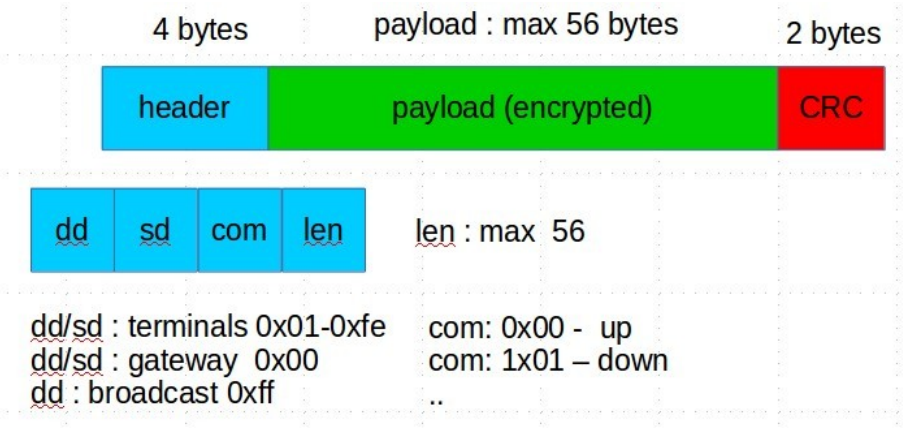

Figure 6.10 Trame **LoRaTS** avec les données encryptées et identifiées par un code CRC.

Une trame LoRaTS comporte une entête (header) avec 4 octets:

`dd` - adresse de destination
`sd` – adresse source
`com` – commande
`len` – taille de données (payload)

L'adresse **`0x00`** est réservée pour identifier la passerelle, l'adresse **`0xff`** est une adresse de diffusion.

Attention :

Dans le premier exemple nous transmettons les trames sans l'encryptage et sans le code CRC.

6.6.1 Protocole LoRaTS

La version la plus simple du protocole (version test) consiste à envoyer les trames en format **LoRaTS** dans les deux sens de communication :
Le serveur (passerelle) attend une trame de données à partir d'un terminal. Les données sont envoyées comme chaînes de caractères , par exemple.

```
1=67.98&2=45.87&3=123.00&4=98.00&t=25
```

Le champs (***fields***) sont marqués par un numéro `1=`,`2=`,`3=`,.., qui identifie le numéro du champs. Un canal peut comporter maximum 8 champs. Les champs sont séparés par le caractère `&`. Les lettres avant le caractère `=`, par exemple `t=`, indiquent les paramètres de gestion entre le terminal et la passerelle.
La lettre `t` est réservée pour le `time-out` mémorisé sur le terminal.

Une trame de retour, envoyée par la passerelle, sert à confirmer la bonne réception de la trame transmise par le terminal et à communiquer la nouvelle valeur de **time-out** à mettre en œuvre par le terminal. Le terminal est autorisé d'envoyer les données seulement au moment de l'expiration du **time-out**.
La forme d'une donnée de réponse est simplement `t=29`, par exemple.

L'identification du terminal vis-à-vis de la passerelle est effectuée grâce à l'identificateur (numéro) envoyé dans la trame de données (deuxième octet). La passerelle repond avec l'adresse de destination du terminal.
Le cycle de réponse est d'ordre de 2 sec maximum, le temps nécessaire d'envoyer la requête `POST` vers le serveur `thingspeak`.

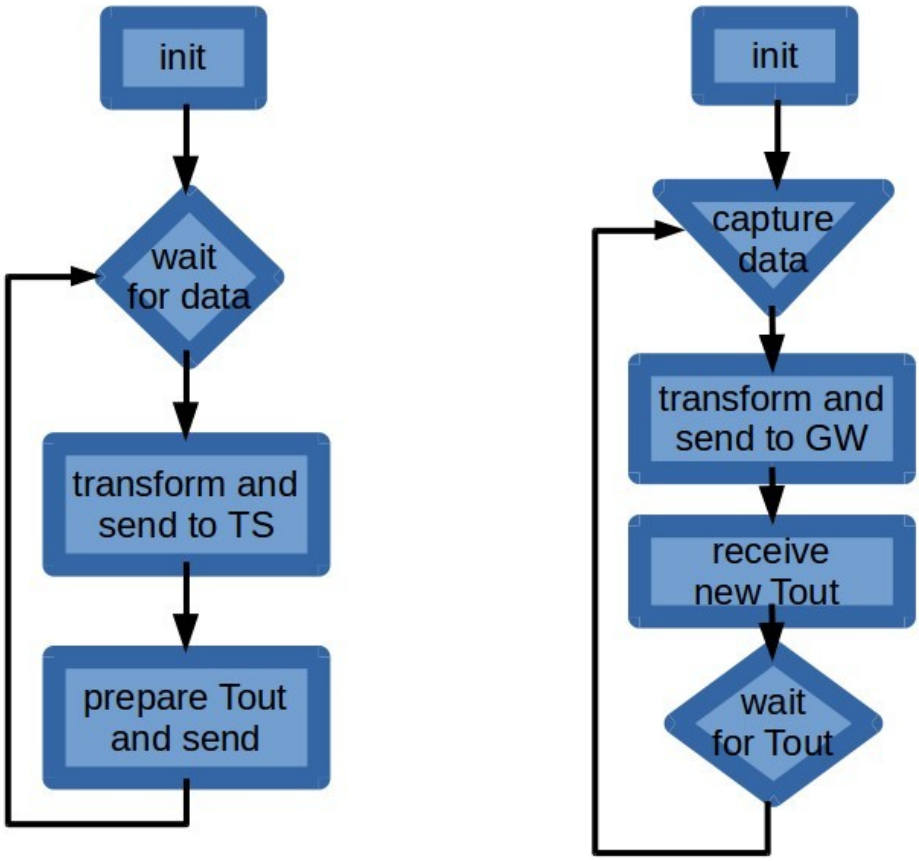

Figure 6.11 Protocole **LoRaTS**, coté serveur (passerelle) et coté terminal

6.6.1 Code Arduino pour le terminal (Wemos D1) :

```
#include <SPI.h>
#include <RH_RF95.h>
#include <Wire.h>
#include <WEMOS_SHT3X.h>
#include <BH1750.h>
SHT3X sht30(0x45);
BH1750 lsens;
RH_RF95 rf95(D4,D8);   // Wemos Gateway small
float frequency = 868.00;
uint8_t id=0x01; // terminal identifier
uint8_t com=0;  // command value
uint8_t data[64];
char cbuf[64];
int tout;  // time-out value in seconds sent by gateway !
float f1,f2,f3, f4=8.98; // temperature, humidity, luminosity

int getdata()
{
  uint16_t lux = lsens.readLightLevel();
  f3 = (float) lux;
  delay(1000);
  sht30.get();
  delay(300);
  f2 = sht30.humidity;
  f1 = sht30.cTemp;
  delay(1000);
}

int senddata()
{
  int i=0;int evalue=30;
  char cf1[12],cf2[12],cf3[12],cf4[12];
  memset(cbuf,0x00,64);  // preparing payload buffer - more fields may be used !
  memset(data,0x00,64);  // preparing lora buffer

dtostrf(f1,2,2,cf1);dtostrf(f2,2,2,cf2);dtostrf(f3,2,2,cf3);dtostrf(f4,2,2,cf4);
  sprintf(cbuf,"1=%s&2=%s&3=%s&4=%s&t=%d",cf1,cf2,cf3,cf4,evalue);
  Serial.println(cbuf);
  data[0]=0x00; // gateway ID
  data[1]=id;   // terminal ID
  data[2]=com;  // command
  data[3]=strlen(cbuf); // size of payload
  while(cbuf[i]) data[i+4]= (uint8_t)cbuf[i++];
  rf95.send(data, sizeof(data));
  rf95.waitPacketSent();
  Serial.println("Packet sent");
}
```

```
int recvdata()
{
  //if (rf95.waitAvailableTimeout(3000))
   if (rf95.available())
  {
    uint8_t buf[RH_RF95_MAX_MESSAGE_LEN];
    uint8_t len = sizeof(buf);
    if (rf95.recv(buf, &len))
    {
      tout = atoi((char *)(buf+2)); // t=value for time-out
    }
    else
    {
      Serial.println("recv failed");
    }
  }
  else
  {
    Serial.println("no reply");
  }
}

void setup()
{
  Serial.begin(9600);
  Wire.begin();
  lsens.begin();
  delay(200);
  if (!rf95.init()) Serial.println("init failed");
  else Serial.println("init ok");
  rf95.setFrequency(frequency);
  rf95.setTxPower(23,false);  // RFM95 only, 868MHz ..
}

void loop()
{
    recvdata(); // eventually tout is set
    delay(tout*1000);  // delay is variable !!!
    getdata();
    delay(200);
    senddata();
  }
```

6.6.2 Code C de la passerelle RPIZ(W) entre LoRa et Ethernet/Internet

Le code C à exécuter sur le RPIZ(W) exploite la librairie **`RH_RF95.h`** pour la communication avec le modem LoRa et librairie **`thingspeak.h`** et ts_http.h pour la communication avec le serveur ThingSpeak.
A l'initialisation du programme l'utilisateur est demandé de proposer le mode de fonctionnement et la fréquence du module LoRa ainsi que le contexte de communication avec le serveur ThingSpeak comprenant la clé d'écriture (**`wkey`**) et le numéro du canal (**`chID`**) concerné.

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <signal.h>
#include "thingspeak.h"
#include "ts_http.h"
#include "ts_time.h"
#include <RH_RF95.h>
RH_RF95 rf95(8,25); // LoraSpiShield - small shield
int tout=15;  // initial timeout 15 seconds
int mode=1; // default mode
float freq=868.0; // default frequency
uint8_t buf[RH_RF95_MAX_MESSAGE_LEN];
uint8_t len = sizeof(buf);
char request[64];  // data sent by terminal via LoRa
char TSrequest[128]; // data to send to thingspeak server
ts_context_t *ctx   = NULL;  // contains wkey and channel id

int dataLRtoTS(char *LRdata, char *TSdata) // data LoRa to data ThingSpeak
{
int  i=0,j=0,termout;
strcpy(TSdata,"field");j=5;
while(LRdata[i])
        {
        TSdata[j]=LRdata[i];
        if(LRdata[i]=='&')
                if(LRdata[i+1]=='t')
                        {
                        termout=atoi(&LRdata[i+3]);
                        TSdata[j]=0x00;
                        return termout;
                        }
                else
                { strcat(&TSdata[j],"field");j+=5; }
        i++;j++;
        }
return 0;
}
void setup(int sc, char *sa[])
{
    wiringPiSetupGpio();
    if (!rf95.init())
    {
```

```
        fprintf(stderr, "Init failed\n");
        exit(1);
    }
    else
        printf("Init ok\n");
    printf("Available modes with default frequency, mode=1 default\n");
        printf("Mode 1 : BW=125KHz, CR=45, SF=128 - frequency:%f\n",freq);
        printf("Mode 2: BW=31.25KHz, CR=48, SF=512 - frequency:%f\n",freq);
        printf("Mode 3: BW=125KHz, CR=48, SF=4096 - frequency:%f\n",freq);
        printf("Mode 4: BW=500KHz, CR=45, SF=128 - frequency:%f\n",freq);
        printf("Mode 5: BW=125KHz, CR=48, SF=1024 - frequency:%f\n",freq);
    printf("Use:\n sudo ./bakolora mode_number frequency\n");
    if(sc==1) { mode=1; freq=868.0; }
    if(sc==2) { mode=atoi(sa[1]); freq=868.0; }
    if(sc==3) { mode=atoi(sa[1]); freq=atof(sa[2]); }
    if(mode==1)
        {
        rf95.setModemConfig(RH_RF95::Bw125Cr45Sf128);
        printf("Mode 1: BW=125KHz, CR=45, SF=128 - frequency:%f\n",freq);
        }
    if(mode==2)
        {
        rf95.setModemConfig(RH_RF95::Bw31_25Cr48Sf512);
        printf("Mode 2: BW=31.25KHz, CR=48, SF=512 - frequency:%f\n",freq);
        }
    if(mode==3)
        {
        rf95.setModemConfig(RH_RF95::Bw125Cr48Sf4096);
        printf("Mode 3: BW=125KHz, CR=48, SF=4096 - frequency:%f\n",freq);
        }
    if(mode==4)
        {
        rf95.setModemConfig(RH_RF95::Bw500Cr45Sf128);
        printf("Mode 4: BW=500KHz, CR=45, SF=128 - frequency:%f\n",freq);
        }
    if(mode==5)
        {
        rf95.setModemConfig(RH_RF95::Bw125Cr48Sf1024);
        printf("Mode 5: BW=125KHz, CR=48, SF=1024 - frequency:%f\n",freq);
        }
rf95.setFrequency(freq);
if(freq>=433.0 && freq<435.0) rf95.setTxPower(14); //RFM98
if(freq>=868.0 && freq<=870.0) rf95.setTxPower(23); //RFM95
if(freq<433.0 || (freq>435.0 && freq<868.0) || freq>870)
        {
        printf("wrong frequency\n");
        printf("must be between 433.0 and 435.0 or 868.0 and 870.0 !\n");
        exit(1); }
}
```

```
void loop()
{
  if( rf95.waitAvailableTimeout(3000) )
  {
  memset(buf,0x00,64);
  if (rf95.recv(buf, &len))
    {
    printf("got reply RSSI= %d\n", rf95.lastRssi());
    printf("%s\n", buf+4); memset(TSrequest,0x00,128);
    dataLRtoTS((char *)(buf+4), TSrequest);
    printf("%s\n", TSrequest);
    ts_http_post(ctx, HOST_API, "/update", TSrequest);
    delay(100);
    tout++;sprintf(request,"t:%d",tout);
      // timeout should be calculated from terminal identifier, the running
time,
      // and the system cycle
    rf95.send((uint8_t *)request, strlen(request)+1);
    rf95.waitPacketSent();
    printf("Send!\n");
    usleep(10);
    }
  else printf("rcv failed\n");
  }
  //else printf("no reply\n");
  }
}

int main(int argc, char **argv)
{
char wkey[]="WCGJK3ZIRW6CXUJG"; // these values depend on the terminal ID and ..
int chID=279015;
setup(argc,argv);
ctx = ts_create_context(wkey,chID);
}
```

6.7 Création d'un terminal à faible consommation électrique (<1mA)

Dans l'exemple étudié précédemment, le terminal (Wemos D1) fonctionne en boucle active avec le délai d'attente impliqué par la fonction dalay(tout).
Dans ce cas le circuit de Wemos D1, et le circuit du module LoRa consomment une centaine de mA, et pourtant le terminal a rien à faire.

Nous pouvons mettre le micro-contrôleur et le module radio en mode de sommeil profond (**deep_sleep**) à condition de pouvoir réveiller notre terminal par une interruption du **timer**.Dans l'état de deep_sleep la carte Wemos D1 consomme mois de 1mA. Ce qui permet de la faire fonctionner sur une simple batterie de 1200 mAh (5V) pendant plus d'un mois.

```
rf95.sleep();  // sleep mode for radio module
ESP.deepSleep(SECONDS_DS(tout));  // restarts with setup()
```

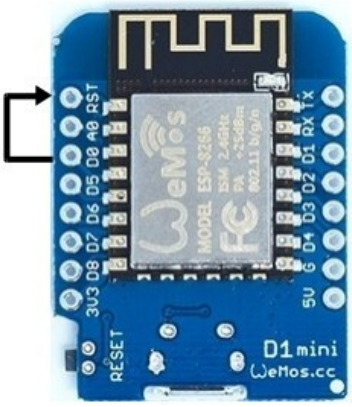

Figure 6.12 Le mécanisme de **deep_sleep** et **restart** par l'interruption du **timer**

A l'expiration du timer un signal d'interruption est envoyé par la broche **`D0`** de la carte Wemos D1 qui doit être connectée à l'entrée **`RST`** (restart).

Attention :

Un restart provoque la **perte totale des données** maintenues dans la mémoire **RAM** !

Nous avons donc besoin de réinitialiser totalement nos circuits.

```
#include <SPI.h>
#include <RH_RF95.h>
#include <Wire.h>
#include <WEMOS_SHT3X.h>
#include <BH1750.h>
#define SECONDS_DS(seconds)  ((seconds)*1000000UL)
SHT3X sht30(0x45);
BH1750 lsens;
RH_RF95 rf95(D4,D8);   // Wemos Gateway small
float frequency = 868.00;

uint8_t id=0x01; // terminal identifier
uint8_t com=0;  // command value
uint8_t data[64];
char cbuf[64];
int tout=30;  // 30 sec default value
```

```
float f1,f2,f3, f4=8.98; // temperature, humidity, luminosity

int getdata()
{
  uint16_t lux = lsens.readLightLevel();
  f3 = (float) lux;
  delay(1000);
  sht30.get();
  delay(300);
  f2 = sht30.humidity;
  f1 = sht30.cTemp;
  delay(1000);
}
int senddata()
{
  int i=0;int evalue=30;
  char cf1[12],cf2[12],cf3[12],cf4[12];
  memset(cbuf,0x00,64);  // preparing payload buffer - more fields may be used !
  memset(data,0x00,64);  // preparing lora buffer

dtostrf(f1,2,2,cf1);dtostrf(f2,2,2,cf2);dtostrf(f3,2,2,cf3);dtostrf(f4,2,2,cf4);
  sprintf(cbuf,"1=%s&2=%s&3=%s&4=%s&t=%d",cf1,cf2,cf3,cf4,evalue);
  Serial.println(cbuf);
  data[0]=0x00; // gateway ID
  data[1]=id;   // terminal ID
  data[2]=com;  // command
  data[3]=strlen(cbuf); // size of payload
  while(cbuf[i]) data[i+4]= (uint8_t)cbuf[i++];
  rf95.send(data, sizeof(data));
  rf95.waitPacketSent();
  Serial.println("Packet sent");
}

int recvdata()
{
  if (rf95.waitAvailableTimeout(3000))
   //if (rf95.available())
  {
    uint8_t buf[RH_RF95_MAX_MESSAGE_LEN];
    uint8_t len = sizeof(buf);
    if (rf95.recv(buf, &len))
    {
      Serial.print("got request: ");
      Serial.println(len);
      Serial.println((char *)buf);
      tout = atoi((char *)(buf+2));
    }
    else
    {
      Serial.println("recv failed");
```

```
    }
  }
  else
  {
    Serial.println("no reply");
  }
}

void setup()
{
  Serial.begin(9600);
  Wire.begin();
  lsens.begin();
  delay(200);
  if (!rf95.init())
    Serial.println("init failed");
  else
    Serial.println("init ok");
  rf95.setFrequency(frequency);
     // Setup Power,dBm
  rf95.setTxPower(23,false);
    getdata();
    delay(200);
    senddata();
    recvdata(); // eventually tout is set
  delay(40);
  Serial.print("ESP8266 in sleep mode:");Serial.println(tout);
  rf95.sleep();  // sleep mode for radio module
  ESP.deepSleep(SECONDS_DS(tout));  // restarts with setup()
}

void loop()
{
// empty loop!!!
}
```

6.8 Stockage et lecture des données dans la mémoire EEPROM

Dans le paragraphe précédant nous avons utilisé une interruption **RST** permettant de garder le circuit en état **deep_sleep** pendant le temps désiré.

Ce mécanisme permet de ré-démarrer l'exécution à partir de **setup()** ce qui signifie que toutes les données de la mémoire RAM sont initialisées de nouveau.
Ci vous avez besoin de préserver les données RAM entre deux périodes d'exécution il faut les stocker puis lire dans la mémoire **EEPROM**.

6.8.1 Ecriture des données dans l'EEPROM

Voici un exemple d'écriture des données dans la mémoire **EEPROM**.

```
#include <EEPROM.h>
int addr = 0;
void setup()
{
  int i=0;
  byte id=2;
  byte tout_low=30;
  byte tout_high=0;
  byte
encryptkey[16]={0x01,0x23,0x45,0x67,0x89,0xAB,0xCD,0xEF,0x01,0x23,0x45,0x67,0x89
,0xAB,0xCD,0xEF};
  Serial.begin(9600);
  EEPROM.begin(512);
  EEPROM.write(addr, id);addr++;
  EEPROM.write(addr, tout_low);addr++;
  EEPROM.write(addr, tout_high);addr++;
  for(i=0;i<16;i++) {  EEPROM.write(addr, encryptkey[i]); addr++; }
  EEPROM.commit();
  Serial.println("init done");
}

void loop()
{

}
```

A noter que la mémoire **EEPROM** de la carte Wemos D1 a la capacité de **512 octets**. L'utilisation de cette mémoire nécessite l'initialisation par **EEPROM.begin(512)** et après l'écriture par les fonctions type **EEPROM.write(addr, id);** il faut confirmer l'écriture par **EEPROM.commit();** pour transférer effectivement le contenu des données à partir d'un tampon **RAM** vers la mémoire **EEPROM**.

6.2.2 Lecture des données dans l'EEPROM

La lecture des données enregistrées dans l'EEPROM doit s'effectuer dans la fonction de **setup().**
Voici un fragment du code à mettre dans la fonction de **setup().**

```
byte id;
int address = 0;
byte value;
int tout;
byte id,tout_high,tout_low;

int evalue=0,mvalue=30;  // default timeout

EEPROM.begin(64);  // asking for 64 bytes in eeprom data memory
value = EEPROM.read(address); id=value; address++;  // reading device identifier
value = EEPROM.read(address); tout_low=value; address++;  // low byte of timeout
value = EEPROM.read(address); tout_high=value; address++; // high byte of
timeout
evalue = (int) (tout_low + 256*tout_high); // calculating registered timeout
for(i=0;i<16;i++) { value = EEPROM.read(address);address++; encryptkey[i]=value;
}
Serial.print("id: ");Serial.println(id);
Serial.print("tout: ");Serial.println(evalue);
```

6.9 Création d'un canal de transmission sécurisé sur un lien radio LoRa

L'utilisation de la communication radio avec LoRa et d'autres modems offre un accès ouvert aux canaux de communication. Une telle situation peut ne pas être acceptable pour de nombreux types d'applications.
L'utilisation des les algorithmes de cryptage/décryptage permet un certain degré de sécurité
Dans cette section, nous illustrons comment utiliser un algorithme relativement simple pour assurer la la sécurité de la communication et la possibilité de vérifier les données reçues.

6.9.1 L'algorithme TEA corrigé

L'algorithme TEA corrigé (souvent appelé XXTEA) est un code à bloc. Le chiffrement n'est soumis à aucun brevet.
XXTEA fonctionne sur des blocs à longueur variable qui sont un multiple arbitraire de 32 bits (minimum 64 bits). Le nombre de cycles complets dépend de la taille du bloc, au moins six (jusqu'à 32 pour les petites tailles de blocs).
L'original Block TEA applique la fonction arrondue XTEA à chaque mot dans le bloc et le combine de façon additive avec son voisin le plus à gauche.

Voici le code C pour l'algorithme XTTEA. Le code **crypte** la chaîne d'octets donnée lorsque l'argument n est **positif**, sinon il le décrypte avec la clé fournie.

```
#define DELTA 0x9e3779b9
#define MX (((z>>5^y<<2) + (y>>3^z<<4))^((sum^y) + (key [(p&3)^e]^z)))
key [16] = {0x01,0x23,0x45,0x67,0x89,0xAB, 0xCD, 0xEF,
0x01,0x23,0x45,0x67,0x89 , 0xAB, 0xCD, 0xEF}; // clé de chiffrement-decryptage

void btea(uint8_t *v, int n, uint8_t const key[16]) {
    uint8_t y, z, sum;
    unsigned p, rounds, e;
    if (n > 1) {          /* Coding Part */
      rounds = 6 + 52/n;
      sum = 0;
      z = v[n-1];
      do {
        sum += DELTA;
        e = (sum >> 2) & 3;
        for (p=0; p<n-1; p++) {
          y = v[p+1];
          z = v[p] += MX;
        }
        y = v[0];
        z = v[n-1] += MX;
      } while (--rounds);
    } else if (n < -1) {  /* Decoding Part */
      n = -n;
      rounds = 6 + 52/n;
      sum = rounds*DELTA;
      y = v[0];
      do {
        e = (sum >> 2) & 3;
        for (p=n-1; p>0; p--) {
          z = v[p-1];
          y = v[p] -= MX;
```

```
        }
        z = v[n-1];
        y = v[0] -= MX;
        sum -= DELTA;
      } while (--rounds);
    }
  }
```

6.9.2 Création de la somme de test CRC16

Le processus de cryptage-décryptage fournit les moyens de protéger les données du message. Pour vérifier la bonne transmission, nous pouvons ajouter la somme de contrôle générée par l'algorithme **CRC16**.

Le code suivant montre comment calculer le code CRC16 pour la chaîne d'octets donnée.

```
uint16_t calcByte(uint16_t crc, uint8_t b)
{
  uint32_t i = 0;
  crc = crc ^ (uint32_t)b << 8;
  for ( i = 0; i < 8; i++)
  {
    if ((crc & 0x8000) == 0x8000) crc = crc << 1 ^ 0x1021;
    else crc = crc << 1;
  }
  return crc & 0xffff;
}
uint16_t CRC16(uint8_t *pBuffer,uint32_t length)
{
    uint16_t wCRC16 = 0;
    uint32_t i = 0;
    if (( pBuffer==0 )||( length==0 ))
    {
      return 0;
    }
    for ( i = 0; i < length; i++)
    {
      wCRC16 = calcByte(wCRC16, pBuffer[i]);
    }
    return wCRC16;
}
```

L'utilisation des fonctions ci-dessus pour vérifier l'exactitude du message de données est fournie ci-dessous. Notez que le les deux derniers octets portent la somme de contrôle ajoutée au message de données:

```
uint16_t recdata( unsigned char* recbuf,int Length)
{
    uint16_t crcdata = 0;
    uint16_t recCRCData = 0;
    crcdata = CRC16(recbuf,Length-2);// CRC for the received message
    recCRCData = recbuf[Length-1];//get the CRC high byte
```

```
    recCRCData = recCRCData<<8;// get the CRC low byte
    recCRCData |= recbuf[Length-2];//get the receive CRC
    if(crcdata == recCRCData)
    {
      CrcFlag = 1; crcdata = 0; recCRCData = 0;
      Serial.println("CRC++++");
    }
    else
    {
      CrcFlag = 0 ; crcdata = 0; recCRCData = 0;
    }
}
```

6.9.3 L'encryptage/décryptage et la protection d'une trame LoRa par un CRC

Les fonctionnalités présentés ci-dessus servent à produire les trames LoRaTS encryptées et protégées. Voici la fonction d'envoi d'une trame LoRaTS encryptée. **sendcrypt**(). **sendcrypt**() encrypte seulement la partie de données (payload). L'entête de la trame reste en clair. Après l'encryptage avec la clé fournie en argument, on ajoute la somme de test qui couvre la trame entière. La trame envoyé contient 64 octets.

```
int sendcrypt(uint8_t *head,uint8_t const *key) // address and crypt key
  {
  int len,i;
  uint8_t msg[64];
  int smsg=sizeof(msg);
    len=head[3];
    memset(msg,0x00,smsg);
    for(i=0;i<len+4;i++) msg[i]=head[i]; // all frame is in msg
    // encrypt mbuf part
    btea(msg+4,(int)len,ckey);  // only data part is encrypted
    crc=CRC16(msg,(uint32_t)(len+4));  // whole frame protected
    msg[len+5]=(uint8_t)(crc>>8); // get high byte - little endian
    msg[len+4]=(uint8_t)crc; // get low byte
    rf95.send(msg, smsg);
    rf95.waitPacketSent();
   delay(100);
  }
```

Le décryptage de la trame reçue est réalise dans le «sens» inverse ; la trame est d'abord vérifiée en comparant le code CRC16 calculé et le code CRC16 reçu dans la trame. Ensuite, si le code **CRC16** est correct, la trame est décryptée.

```
int recvcrypt(uint8_t *res, uint8_t const *key)
{
    int i;
    uint16_t rcrc; // recorded crc
    if( rf95.waitAvailableTimeout(3000) )  // waiting for ACK - 3 sec max
     {
     memset(buf,0x00,len);  // buf is declared in global
     if (rf95.recv(buf, &len))
       {
        Serial.println("Replay Packet received\n");
```

```
        crc=CRC16(buf,(uint32_t)(buf[3]+4));  // calculate received frame CRC
        rcrc=buf[buf[3]+4]; // get low byte
        rcrc = buf[buf[3]+5]; // get high byte
        rcrc= rcrc<<8; // move byte
        rcrc |= buf[buf[3]+4]; // get low byte
        if(crc!=rcrc)
          { Serial.println("Checksum error\n"); return -1; }
       else
          {
          Serial.println("Checksum ok\n");
          btea(buf+4,-(int)buf[3],key);
          if(buf[0]==id || buf[0]==0xff)
                  for(i=0;i<(int)buf[3];i++) res[i]=buf[i+4];
          else return -2;
          }
       }
     }
}
```

Le résultat retourné par cette fonction contient les données de la trame vérifiée et décryptée.

6.10 Architecture IoT avec liens sécurisés

6.10.1 code Arduino complet (sans l'initialisation de la EEPROM)

Voici le code complet permettant de mettre en œuvre les fonctionnalités d'un terminal. La communication au niveau d'un canal LoRa est vérifiée et protégée.
Les paramètres par défaut sont enregistrés au début du programme. Dans la version finale il faut les mettre dans la mémoire EEPROM. De telle façon on pourra utiliser le même code principal pour plusieurs terminaux avec différentes valeurs de paramètres initiales.

```
#include <SPI.h>
#include <RH_RF95.h>
#include <Wire.h>
#include <WEMOS_SHT3X.h>
#include <BH1750.h>
#define SECONDS_DS(seconds)  ((seconds)*1000000UL)
SHT3X sht30(0x45);
BH1750 lsens;
RH_RF95 rf95(D4,D8);   // Wemos Gateway small
float frequency = 868.00;
uint8_t buf[RH_RF95_MAX_MESSAGE_LEN];
uint8_t len = sizeof(buf);
#define DELTA 0x9e3779b9
#define MX (((z>>5^y<<2) + (y>>3^z<<4)) ^ ((sum^y) + (key[(p&3)^e] ^ z)))
unsigned char
ckey[16]={0x01,0x23,0x45,0x67,0x89,0xAB,0xCD,0xEF,0x01,0x23,0x45,0x67,0x89,0xAB,
0xCD,0xEF};
uint8_t id=0x01; // terminal identifier
uint8_t com=0;  // command value
uint8_t data[64];
char cbuf[64];
int tout=30;  // 30 sec default value
float f1,f2,f3, f4=8.98; // temperature, humidity, luminosity

uint16_t calcByte(uint16_t crc,uint8_t b)
{
  uint32_t i = 0;
  crc = crc ^ (uint32_t)b << 8;
  for ( i = 0; i < 8; i++)
  {
    if ((crc & 0x8000) == 0x8000)
      crc = crc << 1 ^ 0x1021;
    else
      crc = crc << 1;
  }
  return crc & 0xffff;
}
void btea(uint8_t *v, int n, uint8_t const key[16]) {
    uint8_t y, z, sum;
    unsigned p, rounds, e;
    if (n > 1) {          /* Coding Part */
      rounds = 6 + 52/n;
```

```
      sum = 0;
      z = v[n-1];
      do {
        sum += DELTA;
        e = (sum >> 2) & 3;
        for (p=0; p<n-1; p++) {
          y = v[p+1];
          z = v[p] += MX;
        }
        y = v[0];
        z = v[n-1] += MX;
      } while (--rounds);
    } else if (n < -1) {  /* Decoding Part */
      n = -n;
      rounds = 6 + 52/n;
      sum = rounds*DELTA;
      y = v[0];
      do {
        e = (sum >> 2) & 3;
        for (p=n-1; p>0; p--) {
          z = v[p-1];
          y = v[p] -= MX;
        }
        z = v[n-1];
        y = v[0] -= MX;
        sum -= DELTA;
      } while (--rounds);
    }
  }

uint16_t CRC16(uint8_t *pBuffer,uint32_t length)
{
    uint16_t wCRC16 = 0; uint32_t i = 0;
    if (( pBuffer==0 )||( length==0 )){return 0;}
    for ( i = 0; i < length; i++)
    {
      wCRC16 = calcByte(wCRC16, pBuffer[i]);
    }
    return wCRC16;
}
uint16_t crc;
int sendcrypt(uint8_t *head,uint8_t const *key) // address and crypt key
  {
  int len,i;
  uint8_t msg[64];
  int smsg=sizeof(msg);
    len=head[3];
    memset(msg,0x00,smsg);
    for(i=0;i<len+4;i++) msg[i]=head[i]; // all frame is in msg
    btea(msg+4,(int)len,ckey);  // only data part is encrypted
```

```
    crc=CRC16(msg,(uint32_t)(len+4));  // whole frame is protected
    msg[len+5]=(uint8_t)(crc>>8); // get high byte - little endian
    msg[len+4]=(uint8_t)crc; // get low byte
    rf95.send(msg, smsg);
    rf95.waitPacketSent();
   delay(100);
  }

int recvcrypt(uint8_t *res, uint8_t const *key)
{
    int i; uint16_t rcrc; // recorded crc
    if( rf95.waitAvailableTimeout(3000) )  // waiting for ACK - 3 sec max
     {
     memset(buf,0x00,len);  // buf is declared in global
     if (rf95.recv(buf, &len))
       {
        Serial.println("Replay Packet received\n");
        crc=CRC16(buf,(uint32_t)(buf[3]+4));  // calculate received frame CRC
        rcrc=buf[buf[3]+4]; // get low byte
        rcrc = buf[buf[3]+5]; // get high byte
        rcrc= rcrc<<8; // move byte
        rcrc |= buf[buf[3]+4]; // get low byte
        if(crc!=rcrc)
          { Serial.println("Checksum error\n"); return -1; }
       else
          {
          Serial.println("Checksum ok\n");
          btea(buf+4,-(int)buf[3],key);
          if(buf[0]==id || buf[0]==0xff)
                  for(i=0;i<(int)buf[3];i++) res[i]=buf[i+4];
          else return -2;
          }
       }
     }
}
int getdata()
{
  uint16_t lux = lsens.readLightLevel();
  f3 = (float) lux;
  delay(1000);
  sht30.get();
  delay(300);
  f2 = sht30.humidity;
  f1 = sht30.cTemp;
  delay(1000);
}

int senddata()
{
  int i=0;int evalue=30;
```

```
  char cf1[12],cf2[12],cf3[12],cf4[12];
  memset(cbuf,0x00,64);  // preparing payload buffer - more fields may be used !
  memset(data,0x00,64);  // preparing lora buffer

dtostrf(f1,2,2,cf1);dtostrf(f2,2,2,cf2);dtostrf(f3,2,2,cf3);dtostrf(f4,2,2,cf4);
  sprintf(cbuf,"1=%s&2=%s&3=%s&4=%s&t=%d",cf1,cf2,cf3,cf4,evalue);
  Serial.println(cbuf);
  data[0]=0x00; // gateway ID
  data[1]=id;   // terminal ID
  data[2]=com;  // command
  data[3]=strlen(cbuf); // size of payload
  while(cbuf[i]) data[i+4]= (uint8_t)cbuf[i++];
  sendcrypt(data,ckey);
  Serial.println("Packet sent");
}

uint8_t rval[16];
void setup()
{
  Serial.begin(9600);
  Wire.begin();
  lsens.begin();
  delay(200);
  if (!rf95.init())
    Serial.println("init failed");
  else
    Serial.println("init ok");
  rf95.setFrequency(frequency);
     // Setup Power,dBm
  rf95.setTxPower(23,false);
    getdata();
    delay(200);
    senddata();
    recvcrypt(rval,ckey); // eventually tout is set
    Serial.println((char*)rval);
    if((char)rval[0]=='t') tout=atoi((char*)(rval)+2);
  delay(40);
  Serial.print("ESP8266 in sleep mode:");Serial.println(tout);
  rf95.sleep();
  ESP.deepSleep(SECONDS_DS(tout));  // restarts with setup()
}

void loop()
{
}
```

6.10.2 Code C/C++ complet sur RPIZ-W

```
/*
 * smartcomputerlab.org
 * Example of a communication between different LoRa boards-shields for RPI2/3
 * with a RFM9X modules.
 */

#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <signal.h>
#include "teacrc.h"
#include "thingspeak.h"
#include "ts_http.h"
#include "ts_time.h"
#include <RH_RF95.h>

//RH_RF95 rf95(8,4); // module RFM95
// RH_RF95 rf95(8,4); // Lora HAT
RH_RF95 rf95(8,25); // LoraSpiShield - small shield
//RH_RF95 rf95; // chistera Pi and simple RFM95 module

/* The address of the node which is 10 by default */
int mode=1; // default mode
float freq=868.0; // default frequency
uint8_t buf[RH_RF95_MAX_MESSAGE_LEN];
uint8_t len = sizeof(buf);
int flen;
char request[64];
char data[64];
char TSrequest[128];
ts_context_t *ctx   = NULL;

unsigned char
ckey[16]={0x01,0x23,0x45,0x67,0x89,0xAB,0xCD,0xEF,0x01,0x23,0x45,0x67,0x89,0xAB,
0xCD,0xEF};

int tout=15;
int fromLoRa(uint8_t *frame, char *data, unsigned char *key)
{
int i=0;
    uint16_t crc,rcrc; // recorded crc
    crc=CRC16(frame,(uint32_t)(frame[3]+4));
    rcrc=frame[frame[3]+4]; // get low byte
    rcrc = frame[frame[3]+5]; // get high byte
    rcrc= rcrc<<8; // move byte
    rcrc |= frame[frame[3]+4]; // get low byte
    if(crc!=rcrc) return -1;  // CRC error
    else           // decrypt data
        {
```

```
        printf("frame with no error!: id=%d\n",frame[1]);
        if(frame[1]==0x01)  // decrypt frame
          {
          btea(frame+4,-(int)frame[3],key);
          }
        for(i=0;i<(int)frame[3];i++) data[i]=(char)frame[i+4];
        }
    return 0;
}

int toLoRa(int timeout,uint8_t *frame, unsigned char *key)
{
  int len,crypt,i;
  uint16_t crc;
  uint8_t td=frame[1];
  char buffer[64];
  frame[1]=frame[0]; frame[0]=td;
  memset(buffer,0x00,64);
  sprintf(buffer,"t=%d",timeout); len=strlen(buffer)+1;
  frame[3] =(uint8_t) len;
  for(i=0;i<len;i++) frame[i+4] = (uint8_t) buffer[i];
  btea(frame+4,len,key);
  crc=CRC16(frame,(uint32_t)(len+4));
  frame[len+5]=(uint8_t)(crc>>8); // get high byte - little endian
  frame[len+4]=(uint8_t)crc; // get low byte
  return len+6;
  }

int dataLRtoTS(char *LRdata, char *TSdata) // data LoRa to data ThingSpeak
{
int  i=0,j=0,termout;
strcpy(TSdata,"field");j=5;
while(LRdata[i])
        {
        TSdata[j]=LRdata[i];
        if(LRdata[i]=='&')
                if(LRdata[i+1]=='t')
                        {
                        termout=atoi(&LRdata[i+3]);
                        TSdata[j]=0x00;
                        return termout;
                        }
                else
                { strcat(&TSdata[j],"field");j+=5; }
        i++;j++;
        }
return 0;
}
```

```
void setup(int sc, char *sa[])
{
    wiringPiSetupGpio();
    if (!rf95.init())
    {
        fprintf(stderr, "Init failed\n");
        exit(1);
    }
    else
        printf("Init ok\n");
    printf("Available modes with default frequency\n");
        printf("Mode 1 (default) : BW=125KHz, CR=45, SF=128 - frequency:
%f\n",freq);
        printf("Mode 2: BW=31.25KHz, CR=48, SF=512 - frequency:%f\n",freq);
        printf("Mode 3: BW=125KHz, CR=48, SF=4096 - frequency:%f\n",freq);
        printf("Mode 4: BW=500KHz, CR=45, SF=128 - frequency:%f\n",freq);
        printf("Mode 5: BW=125KHz, CR=48, SF=1024 - frequency:%f\n",freq);
    printf("To chose the mode and frequency use:\n sudo ./bakolora mode_number
frequency\n");
    if(sc==1) { mode=1; freq=868.0; }
    if(sc==2) { mode=atoi(sa[1]); freq=868.0; }
    if(sc==3) { mode=atoi(sa[1]); freq=atof(sa[2]); }
    if(mode==1)
        {
        rf95.setModemConfig(RH_RF95::Bw125Cr45Sf128);
        printf("Mode 1: BW=125KHz, CR=45, SF=128 - frequency:%f\n",freq);
        }
    if(mode==2)
        {
        rf95.setModemConfig(RH_RF95::Bw31_25Cr48Sf512);
        printf("Mode 2: BW=31.25KHz, CR=48, SF=512 - frequency:%f\n",freq);
        }
    if(mode==3)
        {
        rf95.setModemConfig(RH_RF95::Bw125Cr48Sf4096);
        printf("Mode 3: BW=125KHz, CR=48, SF=4096 - frequency:%f\n",freq);
        }
    if(mode==4)
        {
        rf95.setModemConfig(RH_RF95::Bw500Cr45Sf128);
        printf("Mode 4: BW=500KHz, CR=45, SF=128 - frequency:%f\n",freq);
        }
    if(mode==5)
        {
        rf95.setModemConfig(RH_RF95::Bw125Cr48Sf1024);
        printf("Mode 5: BW=125KHz, CR=48, SF=1024 - frequency:%f\n",freq);
        }
rf95.setFrequency(freq);
if(freq>=433.0 && freq<435.0) rf95.setTxPower(14); //RFM98
if(freq>=868.0 && freq<=870.0) rf95.setTxPower(23); //RFM95
```

```
if(freq<433.0 || (freq>435.0 && freq<868.0) || freq>870)
        {
        printf("wrong frequency\n");
        printf("must be between 433.0 and 435.0 or 868.0 and 870.0 !\n");
        exit(1); }
}

void loop()
{
while(1)
  {
  if( rf95.waitAvailableTimeout(3000) )
  {
  memset(buf,0x00,64);
  if (rf95.recv(buf, &len))
    {
    if(!fromLoRa(buf,data,ckey))
      {
      printf("got reply RSSI= %d\n", rf95.lastRssi());
      printf("%s\n", buf+4); memset(TSrequest,0x00,128);
      dataLRtoTS(data, TSrequest);
      printf("%s\n", TSrequest);
      ts_http_post(ctx, HOST_API, "/update", Tsrequest);
      // in thingspeak.h, ts_http.h,ts_time.h libraries
      delay(100);
      tout++;
      flen=toLoRa(tout,buf,ckey);
      rf95.send(buf,flen);
      rf95.waitPacketSent();
      printf("Send!\n");
      usleep(10);
      }
    else printf("CRC error!\n");
    }
  else printf("rcv failed\n");
  }
  //else printf("no reply\n");
  }
}

int main(int argc, char **argv)
{
char wkey[]="WCGJK3ZIRW6CXUJG";
int chID=279015;
setup(argc,argv);
ctx = ts_create_context(wkey,chID);
loop();
return EXIT_SUCCESS;
}
```

La fonction **ts_http_post(ctx, HOST_API, "/update"**, Tsrequest); permet d'evoyer la requête HTTP de type POST vers le serveur indiqué dans le fichier **ts_http.h**. La ligne ci-dessous a été modifiée pour changer l'adresse IP du serveur (**thingspeak.com**) à l'adresse de notre serveur local.

```
..
//#define HOST_API        "api.thingspeak.com"
#define HOST_API        "172.19.65.62"
..
```

Le serveur **thingspeak.com** fonctionne sur le port numéro **80**. Cette valeur doit être remplacée par **3000** dans le fichier **ts_http.c**

```
..
sockfd = socket(AF_INET, SOCK_STREAM, 0);
    bzero(&servaddr, sizeof(servaddr));
    servaddr.sin_family = AF_INET;
//    servaddr.sin_port = htons(80);
    servaddr.sin_port = htons(3000);
..
```

Exercices :

Compléter le programme Arduino par l'introduction des paramètres enregistrés dans la EEPROM. Tels que l'identificateur du terminal – id , le time-out par défaut, le code de cryptage, la fréquence,..