Lab 10

Liaison LoRa et couche réseau pour les services TS et MQTT

Dans ce laboratoire, nous allons étudier les couches **réseau** et **application** développées pour la transmission LoRa et l'association avec des serveurs/brokers IoT traditionnels tels que **ThingSpeak** ou **MQTT**.
Notre travail commence par l'analyse et l'explication des bibliothèques préparées. Cette explication est suivie de la présentation de 4 exemples essentiels fonctionnant avec 4 services: **envoyer** à **ThingSpeak**, **recevoir** de **ThingSpeak**, **publier** sur **MQTT**, **souscrire** et **recevoir** message **MQTT**.

Table des matières

10.1 Nos bibliothèques LoRa supplémentaires

Dans cette section, nous allons analyser les bibliothèques LoRa prêtes à travailler sur: la couches **physique-liens** et les couches **réseau-applications** :

`LoRa_Para.h` - paramètres et fonctions de la couche physique-liens LoRa
`LoRa_Packets.AES.h` – formats des paquets et fonctions pour envoyer les paquets avec AES
`LoRa_onReceive.AES` - fonctions pour recevoir les paquets

10.1.1 `Lora_Para.h`

Commençons par la bibliothèque **`LoRa_Para.h`** qui a été introduite et étudiée dans **Lab.7** (couche physique LoRa). Dans cette bibliothèque nous fournissons la définition des connexions - broches au modem LoRa (**SX1276/8**) et la définition des paramètres de modulation avec leurs valeurs par valeurs par défaut.
Nous avons 3 **fonctions d'initialisation** :

1. **`set_LoRa()`** - pour définir **tous les paramètres par défaut**
2. **`set_LoRa_Para()`** - pour définir les broches spécifiques et les paramètres de modulation
3. **`set_LoRa_Radio_Para()`** - pour définir les paramètres de modulation et les **broches par défaut**

`LoRa_rxMode()`, **`LoRa_txMode()`** et **`onTxDone()`** sont les fonctions permettant de spécifier le **mode IQ** normal ou inverse. Notez que ces modes sont **symétriquement différents** pour les nœuds **Terminal** et **Gateway**.

```
#define SS       5      // NSS
#define RST     15      // RST
#define DI0     26      // INTR
#define SCK     18      // CLK
#define MISO    19      // MISO
#define MOSI    23      // MOSI

#define BAND    868E6   // set frequency
#define SF      7       // set spreading factor
#define SBW     125E3   // set signal bandwidth
#define SW      0xF3    // set Sync Word
#define BR      8       // set bit rate (4/5,5/8)

void set_LoRa()  // all default settings
{
char buff[128];
  SPI.begin(SCK, MISO, MOSI, SS);  // SCK, MISO, MOSI, SS
  LoRa.setPins(SS, RST, DI0);
  Serial.begin(9600);
  delay(1000);
  Serial.println();
  if (!LoRa.begin(BAND)) {
    Serial.println("Starting LoRa failed!");
    while (1);
  }
  sprintf(buff,"BAND=%f,SF=%d,SBW=%f,SW=%X,BR=%d\n",BAND,SF,SBW,SW,BR);
  Serial.println(buff);
  LoRa.setSpreadingFactor(SF);
  LoRa.setSignalBandwidth(SBW);
  LoRa.setSyncWord(SW);
  }
void set_LoRa_Para(int sck,int miso,int mosi,int ss,int rst, int dio0,unsigned long freq,unsigned
sbw, int sf, uint8_t sw)  // user set pins and parameters
{
  SPI.begin(sck, miso, mosi, ss);  // SCK, MISO, MOSI, SS
  LoRa.setPins(ss, rst, dio0);
  Serial.begin(9600);
  delay(1000);
  Serial.println();
  if (!LoRa.begin(freq)) {
    Serial.println("Starting LoRa failed!");
    while (1);
  }
  LoRa.setSpreadingFactor(sf);
  LoRa.setSignalBandwidth(sbw);
  LoRa.setSyncWord(sw);
  }
```

```
void set_LoRa_Radio_Para(unsigned long freq,unsigned sbw, int sf, uint8_t sw)  // radio settings
{
  SPI.begin(SCK, MISO, MOSI, SS);  // SCK, MISO, MOSI, SS
  LoRa.setPins(SS, RST, DI0);
  Serial.begin(9600);
  delay(1000);
  Serial.println();
  if (!LoRa.begin(freq)) {
    Serial.println("Starting LoRa failed!");
    while (1);
  }
  LoRa.setSpreadingFactor(sf);
  LoRa.setSignalBandwidth(sbw);
  LoRa.setSyncWord(sw);
  }

#ifdef TERMINAL
void LoRa_rxMode(){
  LoRa.enableInvertIQ();                // active invert I and Q signals
  LoRa.receive();                       // set receive mode
}

void LoRa_txMode(){
  LoRa.idle();                          // set standby mode
  LoRa.disableInvertIQ();               // normal mode
}
#endif

#ifdef GATEWAY
void LoRa_rxMode(){
  LoRa.disableInvertIQ();               // normal mode
  LoRa.receive();                       // set receive mode
}

void LoRa_txMode(){rdf.pack.channel
  LoRa.idle();                          // set standby mode
  LoRa.enableInvertIQ();                // active invert I and Q signals
}
#endif

void onTxDone() {
  Serial.println("TxDone");
  LoRa_rxMode();
}

boolean runEvery(unsigned long interval)
{
  static unsigned long previousMillis = 0;
  unsigned long currentMillis = millis();
  if (currentMillis - previousMillis >= interval)
  {
    previousMillis = currentMillis;
    return true;
  }
  return false;
}
```

10.1.2 `Lora_Packets.AES.h`

Ce fichier décrit les types de **paquets LoRa** utilisés pour communiquer entre les terminaux LoRa et les nœuds de passerelle.
Les identificateurs, source : **`sid`** et destination : **`did`** , sont dérivés des identificateurs du circuit ESP32. Seuls 4 octets inférieurs sont pris en compte (**`uint32_t`**)

Les **paquets de contrôle** contiennent **32 octets** et les **paquets de données** sont construits avec **64** pour les services **ThingSpeak** ou **112** octets pour la communication avec les brokers **MQTT**.
Les 2 octets du **champ de contrôle** **`con[0]`** et **`con[1]`** permettent d'identifier le type du paquet et l'organisation des données (**masque**).

Le premier octet - **`con[0]`** indique le **type du paquet** et le **MODE** (service) à utiliser.
La **première valeur hexadécimale** de cet octet indique le **MODE** de la passerelle/du service:

```
 1 - ThingSpeak(TS) send, 2 -ThingSpeak(TS) receive
 3 - MQTT - MQP publish, 4 - MQTT - MQS subscribe,
```

La **deuxième valeur hexadécimale** indique le **type du paquet** :

```
0x01 - IDREQ : ID request for any service and ID acknowledge
0x11 - IDREQ, 0x12 - IDACK: ID request and ID acknowledge - TS send - TSS
0x21 - IDREQ, 0x22 - IDACK: ID request and ID acknowledge - TS receive -TSR
0x31 - IDREQ, 0x32 - IDACK: ID request and ID acknowledge - MQTT publish - MQP
0x41 - IDREQ, 0x42 - IDACK: ID request and ID acknowledge - MQTT subscribe - MQS

0x13 - DTSND, 0x14 - DTACK: DATA send and DATA acknowledge - TS send  - TSS
0x23 - DTREQ, 0x24 - DTRCV: DATA request and DATA receive - TS receive - TSR
0x33 - DTPUB, 0x34 - DTPUBACK: DATA publish and DATA publish acknowledge - MQTT publish - MQP
0x43 - DTSUB, 0x44 - DTSUBRCV: DATA subscribe and DATA subscribe receive - MQTT subscribe - MQS
```

Le deuxième octet du champ de contrôle **`con[1]`** est utilisé pour transporter le **masque de champs de données** nécessaire pour marquer les champs valides dans le **canal ThingSpeak**. Chaque bit de ce champ correspond à une valeur de données ou un champ du canal ThingSpeak.
Avec ce masque, les champs peuvent être spécifiés lors de **l'envoi** ou de la **demande** des données.
Le champ **`pass`** contient le mot de passe sur 16 octets à utiliser par les paquets de contrôle type **`IDREQ`** afin de protéger le premier accès au nœud passerelle. Seule une trame avec un mot de passe correct doit être confirmée par le paquet **`IDACK.`** Un paquet de confirmation **`IDACK`** porte le **code de chiffrement AES** à utiliser pour les **paquets de données**.
Le nœud de passerelle peut envoyer un délai calculé (valeur de temporisation) en fonction de son ordonnanceur (agenda) des nœuds Terminal.

Les paquets de données sont cryptés avec l'algorithme AES qui est implémenté/intégré directement dans ESP32. Les fonctions AES **`encrypt()`** et **`decrypt()`** suivantes qui sont définies pour protéger les paquets avec un mot-clé symétrique.

```
#include "mbedtls/aes.h"

void encrypt(unsigned char *plainText,char *key,unsigned char *outputBuffer, int nblocks)
{
  mbedtls_aes_context aes;
  mbedtls_aes_init( &aes );
  mbedtls_aes_setkey_enc(&aes,(const unsigned char*)key,strlen(key)*8);
  for(int i=0;i<nblocks;i++)
    {
    mbedtls_aes_crypt_ecb(&aes,MBEDTLS_AES_ENCRYPT,
                          (const unsigned char*)(plainText+i*16), outputBuffer+i*16);
    }
  mbedtls_aes_free(&aes);
}

void decrypt(unsigned char *chipherText,char *key,unsigned char *outputBuffer, int nblocks)
{
  mbedtls_aes_context aes;
  mbedtls_aes_init( &aes );
  mbedtls_aes_setkey_dec( &aes, (const unsigned char*) key, strlen(key) * 8 );
    for(int i=0;i<nblocks;i++)
    {
    mbedtls_aes_crypt_ecb(&aes,MBEDTLS_AES_DECRYPT,
                         (const unsigned char*)(chipherText+i*16), outputBuffer+i*16);
    }
  mbedtls_aes_free(&aes );
}
```

Le format des paquets de contrôle est le même pour tous les types de services (**TSS**, **TSR**, **MQP**, **MQS**).
Il est représenté par l'union/structure suivante.

```
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 - 16 characters
    int tout;           // timeout
    uint8_t  pad[2];    // future use
  } pack;               // control packet
} conframe_t;
```

Le format des paquets de données est différent pour les services **ThingSpeak** (TSS/TSR) et **MQTT** (MQP,MQS).
Ils sont représentés par les unions/structures suivants.

```
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 - fields
    uint16_t  tout;       // optional timeout
  } pack;                 // data packet
} TSframe_t;

typedef union
{
  uint8_t frame[112];    // MQTT frame to publish on the given topic
  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
    char     topic[48];  // topic name - e.g. /esp32/Term1/Sens1
    char     mess[48];   // message value
    int      tout;       // optional timeout for publish frame
    uint8_t  pad[2];     // future use
  } pack;                // data packet
} MQTTframe_t;
```

10.2 Fonctions de contrôle pour différents services - MODE

La fonction **`send_IDREQ )`** est utilisée par le nœud Terminal pour démarrer la communication avec un nœud passerelle. L'identifiant de l'adresse de destination n'est pas connu (**`scf.pack.did=(uint32_t) 0;`**).
Le code de contrôle hexadécimal est calculé à partir de la valeur de code (**MODE**).
Par exemple pour **TSS** - **T**hing**S**peak **S**end (MODE = 1), la valeur calculée est **`0x11`**.
Le paquet **`IDREQ`** doit porter un mot de passe à valider par la passerelle.

```
#ifdef TERMINAL              // this packet is sent only by the TERMINAL nodes

void send_IDREQ(char *pass)  // TERM: request ID for all modes: MODE, termID - global
{
conframe_t scf,sccf;
  LoRa_txMode();
  LoRa.beginPacket();
  scf.pack.did=(uint32_t)0;
  scf.pack.sid=(uint32_t)termID;
  scf.pack.con[0]=(uint8_t)(MODE*16+1);    // IDREQ_MODE - type 1 control TERM to GW
  scf.pack.con[1]=0x00;
  if(pass!=NULL)
     strncpy(scf.pack.pass,pass,16);       // password to validate by gateway
  LoRa.write(scf.frame,32);
  LoRa.endPacket(true);
}
```

Le nœud passerelle répond avec le paquet **IDACK** envoyé par la fonction suivante.
Le paquet **IDACK** porte l'identifiant du nœud passerelle - **`gwID`** comme adresse source – **`scf.pack.sid`**.
Il contient également la clé AES à utiliser dans les paquets de données.

```
#ifdef GATEWAY                                  // this packet is sent only by gateway node

void send_IDACK(char *aes, uint16_t tout)       // GW: ID ACK for all modes: code
{
conframe_t scf,sccf;
  LoRa_txMode();
  LoRa.beginPacket();
  scf.pack.did=(uint32_t)termID;
  scf.pack.sid=(uint32_t)gwID;
  scf.pack.con[0]=(uint8_t)16*MODE+2;  // IDACK_MODE - type 2 - control GW to TERM
  scf.pack.con[1]=0x77;                // control byte required for validation
  strncpy(scf.pack.pass,aes,16);       // AES key for data packets
  scf.pack.tout=tout;                  // optional timeout for the next packet
  LoRa.write(scf.frame,32);
  LoRa.endPacket(true);
}
```

Notez que l'argument **`tout`** peut fournir la valeur du délai d'expiration au terminal. Ce délai peut être utilisé pour planifier le moment où la première trame de données doit être envoyée au nœud passerelle.

10.2.1 Fonctions d'envoi des paquets de données pour TSS - MODE 1

Commençons par présenter 2 paquets de données et envoyés par les fonctions utilisées pour implémenter notre premier service (**MODE = 1**) qui est l'**envoi des paquets** (valeurs des capteurs) **au serveur ThingSpeak** via un nœud Gateway (LoRa-WiFi Gateway).

Elles sont:

Étape 1 - le même pour tous les services ou MODES vient d'être présenté ci-dessus.
- Terminal: **`IDREQ`** avec **`send_IDREQ()`**
- Gateway: **`IDACK`** avec **`send_IDACK()`**

Étape 2
- Terminal: **`DTSND`** avec **`send_DTSND()`**
- Gateway: **`DTACK`** avec **`avec_DTACK()`**

Dans cette section nous allons présenter les deux fonctions permettant d'envoyer les données de capteurs à la passerelle et de recevoir la confirmation de leur envoie sur le serveur ThingSpeaK.

10.2.1.1 Fonction `send_DTSND (TERMINAL)`

La fonction **`data_DTSND()`** envoie un paquet vers la passerelle (**`gwID`**). Le type de la fonction est 3 et le mode 1 (**`0x13`**) est enregistré dans l'octet de contrôle **`con[0]`**.
Le paquet porte le numéro du canal (**`chan`**) et la clé d'écriture dans ce canal. Le byte **`mask`** indique les champs de capteurs valides dans cet envoie.
L'ensemble du paquet est encrypté avec la clé (**`CKEY`**) précédemment reçu dans le paquet **`IDACK`**.
Notez que les variables **`gwID`**, **`termID`**, et **`CKEY`** sont globales et enregistrées dan la mémoire RTC non modifiable par **deep_sleep**.

```
void send_DTSND(uint8_t mask,int chan,char *wkey,float *stab)  // TERM: send data to TS, gwID,
termID - global
{
TSframe_t sdf,sdcf;
  LoRa_txMode();
  LoRa.beginPacket();
  sdf.pack.did=(uint32_t)gwID;
  sdf.pack.sid=(uint32_t)termID;
  sdf.pack.con[0]=(uint8_t)(MODE*16+3);              // MODE 1 - type 3  data TERM to GW
  sdf.pack.channel= chan;
  strncpy(sdf.pack.keyword,wkey,16);
  sdf.pack.con[1]=mask;
  for(int i=0;i<8;i++) sdf.pack.sens[i]=stab[i];
  sdf.pack.tout=0;
  encrypt(sdf.frame,CKEY,sdcf.frame,4);
  LoRa.write(sdcf.frame,64);
  LoRa.endPacket(true);
}
```

10.2.1.2 Fonction `send_DTACK (GATEWAY)`

Après la réception du paquet **`DTSND`** du côté du nœud de passerelle par un ISR **`onReceive()`**, le nœud passerelle répond par un paquet **`DTACK`** (*Data Acknowledge*).
Dans ce paquet, le nœud passerelle peut envoyer une valeur de temporisation et un message de contrôle supplémentaire. Le paquet est crypté avec la clé AES (**`CKEY`**).
Le message de contrôle indique l'état de réception des données envoyées sur ThingSpeak.

```
void send_DTACK(uint8_t mask,int tout,char *wkey)  // GW: ACK data to TERM
{
TSframe_t sdf,sdcf;
  LoRa_txMode();
  LoRa.beginPacket();
  sdf.pack.did=(uint32_t)termID;
  sdf.pack.sid=(uint32_t)gwID;
  sdf.pack.con[0]=(uint8_t)16*MODE+4;
  sdf.pack.con[1]=mask;

  sdf.pack.channel= tout;               // calculated timeout for the next data frame
  if (wkey!=NULL)
    strncpy(sdf.pack.keyword,wkey,16);  // message indicating the reception state at ThingSpeak
  for(int i=0;i<8;i++) sdf.pack.sens[i]=0.0;
  sdf.pack.tout=0;
  encrypt(sdf.frame,CKEY,sdcf.frame,4);
  LoRa.write(sdcf.frame,64);
  LoRa.endPacket(true);
}
```

10.2.2 Fonctions d'envoi des paquets de données pour TSR - MODE 2

Dans le MODE=2 le Terminal envoie la demande des données par la fonction **`send_DTREQ()`** et il attend le résultat à la réception d'un paquet **`DTRCV`** envoyé par la passerelle.

10.2.2.1 Fonction `send_DTREQ() (TERMINAL)`

La fonction **`send_DTREQ()`** exécutée au niveau du nœud Terminal envoie le paquet **`DTREQ`** incluant le champ de masque pour les valeurs requises, le numéro du canal et la clé de lecture permettant de récupérer les données du serveur.

```
#ifdef TERMINAL
void send_DTREQ(uint8_t mask,int chan,char *rkey)  // TERM: send data request to TS
{
TSframe_t sdf,sdcf;
```

```
  LoRa_txMode();
  LoRa.beginPacket();
  sdf.pack.did=(uint32_t)gwID;
  sdf.pack.sid=(uint32_t)termID;
  sdf.pack.con[0]=(uint8_t)(MODE*16+3);               // MODE  2 - type 3  data TERM to GW
  sdf.pack.con[1]=mask;
  sdf.pack.channel= chan;
  strncpy(sdf.pack.keyword,rkey,16);
  for(int i=0;i<8;i++) sdf.pack.sens[i]=0.0;
  sdf.pack.tout=0;
  encrypt(sdf.frame,CKEY,sdcf.frame,4);
  LoRa.write(sdcf.frame,64);
  LoRa.endPacket(true);
}
```

10.2.2.2 Fonction `send_DTRCV()` `(GATEWAY)`

Du côté de la passerelle, après la réception d'un paquet **`DTREQ`**, nous demandons les données (à recevoir du serveur ThingSpeak). Après leurs réception nous les envoyons dans un paquet **`DTRCV`** (*Data Received*) par la fonction **`send_DTRCV()`**.
Ce paquet transporte le champ de données obtenues marqué par l'octet de masque. Il peut également transmettre la nouvelle valeur de temporisation dans l'argument de canal ainsi qu'un message de contrôle de 16 octets) à la place de la valeur de clé de lecture.

```
#ifdef GATEWAY
void send_DTRCV(uint8_t mask,int tout,char *mess,float stab[])  // GW: send received data in stab[]
{
TSframe_t sdf,sdcf;
  LoRa_txMode();
  LoRa.beginPacket();
  sdf.pack.did=(uint32_t)termID;
  sdf.pack.sid=(uint32_t)gwID;
  sdf.pack.con[0]=(uint8_t)16*MODE+4;      // MODE 2 - type 4: GW to TERM
  sdf.pack.con[1]=mask;

  sdf.pack.channel= tout;                  // calculated timeout for the next data frame
  strncpy(sdf.pack.keyword,mess,16);
  for(int i=0;i<8;i++) sdf.pack.sens[i]=stab[i];
  sdf.pack.tout=0;
  encrypt(sdf.frame,CKEY,sdcf.frame,4);
  LoRa.write(sdcf.frame,64);
  LoRa.endPacket(true);
}
```

10.2.3 Fonctions d'envoi des paquets de données pour MQP (MQTT Publish) – MODE 3

Les 2 fonctions suivantes nous permettent de créer un service de publication **MQTT** avec des paquets **`DTPUB`** et **`DTPUBACK`**.

10.2.3.1 Fonction `send_DTPUB()` `(TERMINAL)`

Du côté du terminal, la fonction **`send_DTPUB()`** prend le sujet (**`topic`**) proposé et le message (**`mess`**) associé et l'envoie dans un paquet crypté AES de **112 octets**.

```
#ifdef TERMINAL
void send_DTPUB(char *topic, char *mess)      // TERM: send DTPUB packet
{
MQTTframe_t sdf,sdcf;
  LoRa_txMode();
  LoRa.beginPacket();
  sdf.pack.did=(uint32_t)gwID;
  sdf.pack.sid=(uint32_t)termID;
  sdf.pack.con[0]=(uint8_t)(MODE*16+3);        // MODE  3 - type 3  data TERM to GW
  sdf.pack.con[1]=0x00;
  strcpy(sdf.pack.topic,topic);
  strcpy(sdf.pack.mess,mess);
  sdf.pack.tout=0;
  encrypt(sdf.frame,CKEY,sdcf.frame,7);       // 7*16=112 - 7 16-byte blocks
  LoRa.write(sdcf.frame,112);
  LoRa.endPacket(true);
}
```

10.2.3.2 Fonction `send_DTPUBACK() (GATEWAY)`

Du côté passerelle, nous confirmons la réception du paquet **DTPUB** en envoyant le paquet **DTPUBACK** correspondant.

```
#ifdef GATEWAY
void send_DTPUBACK(char *topic,char *mess, int tout)  // GW: send ACK for PUB message
{
MQTTframe_t sdf,sdcf;
  LoRa_txMode();
  LoRa.beginPacket();
  sdf.pack.did=(uint32_t)termID;
  sdf.pack.sid=(uint32_t)gwID;
  sdf.pack.con[0]=(uint8_t)16*MODE+4;               // MODE 3 - type 4: GW to TERM
  sdf.pack.con[1]=0x00;
  strncpy(sdf.pack.topic,topic,48);
  strncpy(sdf.pack.mess,mess,48);
  sdf.pack.tout=tout;
  encrypt(sdf.frame,CKEY,sdcf.frame,7);
  LoRa.write(sdcf.frame,112);
  LoRa.endPacket(true);
}
```

10.2.4 Fonctions d'envoi des paquets de données pour MQS (MQTT Subscribe) – MODE 4

Dans le MODE=4 le Terminal envoie un paquet d'abonnement au sujet donné à la passerelle, puis il attend les données (messages) envoyés vers ce sujet par d'autre terminaux.

10.2.4.1 Fonction `send_DTSUB() (TERMINAL)`

Le nœud Terminal envoie un paquet **DTSUB** pour s'abonner à un sujet (topic) MQTT. Ceci est fait par la fonction **`send_DTSUB()`** suivante :

```
void send_DTSUB(char *topic)                        // TERM:send DTSUB frame - subscribe topic
{
MQTTframe_t sdf,sdcf;
  LoRa_txMode();
  LoRa.beginPacket();
  sdf.pack.did=(uint32_t)gwID;
  sdf.pack.sid=(uint32_t)termID;
  sdf.pack.con[0]=(uint8_t)(MODE*16+3);           // MODE  4 - type 3  data TERM to GW
  sdf.pack.con[1]=0x00;
  strncpy(sdf.pack.topic,topic,48);
  memset(sdf.pack.mess,0x00,48);
  sdf.pack.tout=0;
  encrypt(sdf.frame,CKEY,sdcf.frame,7);
  LoRa.write(sdcf.frame,112);
  LoRa.endPacket(true);
}
```

10.2.4.2 Fonction `send_DTSUBRCV() (GATEWAY)`

L'arrivée d'une nouvelle donnée au sujet souscrit et la réception de ces données dans le nœud passerelle est retransmis au nœud Terminal par la fonction send_DTSUBRCV () présentée ci-dessous. Cette fonction prépare et envoie un paquet DTSURCV contenant le nom du sujet (**topic**) et la valeur du message (**mess**).

```
void send_DTSUBRCV(char *topic,char *mess,int tout)  // GW:send received message
{
MQTTframe_t sdf,sdcf;
  LoRa_txMode();
  LoRa.beginPacket();
  sdf.pack.did=(uint32_t)termID;
  sdf.pack.sid=(uint32_t)gwID;
  sdf.pack.con[0]=(uint8_t)16*MODE+4;   // MODE 4 - type 4: GW to TERM
  sdf.pack.con[1]=0x00;
  strncpy(sdf.pack.topic,topic,48);
  strncpy(sdf.pack.mess,mess,48);
  sdf.pack.tout=tout;
  encrypt(sdf.frame,CKEY,sdcf.frame,7);
  LoRa.write(sdcf.frame,112);
  LoRa.endPacket(true);
}
```

10.3 Fichier des fonctions de réception : `Lora_onReceive.AES.h`

La réception des paquets se fait via une ISR (*Interrupt Service Routine*) **`onReceive()`**.
Le fichier suivant contient les données requises et l'**ISR global `onReceive()`** préparé pour la réception de tous les paquets de contrôle et de données.
Les données déclarées sont stockées dans la mémoire RTC qui persistent pendant l'opération deep_sleep de l'ESP32.

Les paquets reçus dans ISR sont envoyés dans la file d'attente et envoyés vers la destination par fonction **`xQueueSendFromISR()`**.
La réception des paquets type **`IDREQ`**, **`DTSND`**, **`DTREQ`**, **`DTPUB`**, et **`DTSUB`** dans les nœuds type Gateway est réalisée par les tâches spécifiques travaillant avec les paquets de taille 32 (contrôle), 64 (ThingSpeak), et 112 (MQTT) bytes. Les paquets qui portent les données (**`DTSND`**, **`DTPUB`**) ou les demandes des données (**`DTREQ`**,**`DTSUB`**) sont encryptés.

```
// 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 queue to get out the data or control
packets form onReceive() ISR
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);
  }
}
```

10.5 Implémentation des services avec le *front-end* de passerelles

Dans cette deuxième partie du Lab.10, nous présentons les codes simples pour 4 services. Chaque exemple comprend le code pour le terminal et le code passerelle. Côté passerelle contient uniquement la partie frontale (***front-end***), qui compris l'interface LoRa, la partie arrière (***back-end***) est implémentée par la simulation de la communication avec le service réel (serveur ThingSpeak, broker MQTT).

10.5.1 Code du terminal et de la passerelle - MODE 1

Nous commençons par le code du **TERMINAL** fonctionnant en **MODE 1**, il fonctionne comme code émetteur LoRa vers le serveur ThingSpeak par le biais de la passerelle.

Le code commence par la définition de :

le type de nœud : **`TERMINAL`** [TERMINAL, GATEWAY]
le **`MODE : 1`** [1- TSS, 2-TSR, 3-MQP, 4-MQS], et

Ensuite, nous incluons un certain nombre de bibliothèques :

- **`SPI.h`** - pour importer les opérations sur le bus SPI
- **`LoRa.h`** - pour inclure les fonctions de contrôle du modem LoRa
- **`LoRa_Para.h`** - pour utiliser les fonctions d'initialisation des broches et du modem LoRa
- **`LoRa_Packets.h`** - pour utiliser les formats de paquets prédéfinis et les fonctions d'envoi
- **`LoRa_onReceive.h`** - pour utiliser la fonction **`onReceive()`** prédéfinie pour capturer différents types de paquets. **`LoRa_onReceive.h`** intègre les déclarations des identifiants du terminal et de la passerelle: **`termID`**, **`gwID`** qui sont stockés dans la **mémoire RTC** et 2 drapeaux (**`stage1_flag`**, **`stage2_flag`**) indiquant l'étape de communication : contrôle (**stage1**) et données (**stage2**).
  Le fichier **`LoRa_onReceive.h`** contient les déclarations des files d'attente pour la communication de différents types de paquets : **`QueueHandle_t tsrcv_queue, mqrcv_queue, con_queue;`** et le **password** à envoyer dans le paquet **`IDREQ`**.

10.5.1.1 Code complet du terminal en MODE 1

Le code suivant intègre une union/structure permettant de grouper les paramètres à envoyer vers le serveur ThingSpeak pour pouvoir y enregistrer des nouvelles données : numéro du canal, clé d'écriture, masque des champs à écrire.
La boucle principale fonctionne en deux étapes. Elle est chronométrée par la variable **`cycle`**. La valeur de **`cycle`** est initialisée dans le code mais elle peut être modifiée pendant l'exécution par le paramètre **`timeout`** envoyé par la passerelle.
Le code peut fonctionner en mode toujours actif ou en mode **deep-sleep** (voir la partie commentée).

```
#define TERMINAL  // to choose TERMINAL or MASTER (GATEWAY) node
#define MODE 1  // to choose 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;
  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"); // the test password
      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);

//        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");

       }
    }

  }
}
```

10.5.1.2 Code complet de la passerelle (partie frontale) en MODE 1

Voici le code de la passerelle fonctionnant en MODE 1 , c'est à dire comme un relai entre les terminaux LoRa et le serveur type ThingSpeak dans le sens d'écriture.

Le code de la passerelle contient 3 tâches spécifique (plus une tâche de fond – **`loop()`**) qui sont :

1. tâche **`TS_Task`** – le rôle de cette tâche est d'imiter la communication avec un serveur type ThingSpeak en lui envoyant les données des capteurs assoviés aux terminaux.
2. tâche **`CON_Task`** – cette tâche capte les paquets de contrôle – **`IDREQ`** reçus initialement par la fonction ISR **`onReceive()`**. Après l'analyse du paquet reçu dans la file d'attente, la tâche **`CON_Task`** envoie une paquet **`IDACK`** vers le terminal concerné. Ce paquet porte la clé **AES**.
3. tâche **`DATA_Task`** – cette tâche attend le résultat de communication avec le serveur ThingSpeak et en fonction de la valeur du retour envoie un paquet **`DTACK`** vers le terminal concerné.

```
#define GATEWAY  // to choose TERMINAL or GATEWAY node
#define MODE  1  // to choose sender (1), receiver (2) , publisher (3), subscriber (4) mode
#define CKEY "abcdefghijklmnop"  // AES key - 16 bytes
#define PASS "passwordpassword"

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

int tss_flag=0;
int taskCore = 0;

void TS_Task( void * pvParameters ){
float stab[8]; char kbuff[32];
TSframe_t rdf; int r,x,d;
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);
  Serial.printf("f1:%2.2f,f2:%2.2f,f3:%2.2f,f4:%2.2f\n", rdf.pack.sens[0], rdf.pack.sens[1],
                rdf.pack.sens[2],rdf.pack.sens[3]);
  LoRa.idle();

  d=1000+random(2000,8000);  // this code section imitates the behaviour of ThingSpeak server
  delay(d);
  r=random(1,100);
  if(r>50) x=200; else x=100;

  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 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);
    }
  //tsrcv_queue in TS task
  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);

  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() {
}
```

10.5.2 Nœuds terminaux et passerelle - MODE 2 (TSR)

MODE 2 est conçu pour envoyer les paquets de demandes de données au serveur ThingSpeak et à attendre les données demandées.
A l'étape 1, le nœud Terminal commence par établir la liaison logique avec la passerelle (IDREQ,IDACK), puis il envoie les paquets **DTREQ**. Après la réception des données demandées à partir d'un serveur ThingSpeak, le nœud passerelle les envoie au nœud terminal dans un paquet de type **DTRCV**.

10.5.2.1 Code du Terminal en MODE 2

Comme dans l'exemple précédant (MODE 1), côté du terminal, nous commençons par l'étape1 pour obtenir l'identifiant de la passerelle - **gwID** (si un tel service est disponible!).
Puis, à l'étape 2, le terminal envoie les paquets de demande de données - **DTREQ** à la passerelle détectée.

```
#define TERMINAL  // to choose TERMINAL or MASTER (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;
  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=%dsec,%s,%x\n", rdf.pack.channel, rdf.pack.keyword,
                    rdf.pack.con[0]);
      if(10<rdf.pack.channel && rdf.pack.channel< 3600) // new cycle eliminates error
        {
        disp_sens(ts.pack.mask,rdf.pack.sens);
        cycle=rdf.pack.channel;
        }
      }
    }
  }
}
```

A noter que le champ **`rdf.pack.channel`** dans le paquet **`DTRCV`** peut porter la valeur du **délai d'expiration**. Par conséquent, cette valeur peut être utilisée pour planifier l'émission du prochain paquet **`DTREQ`**.

10.5.2.2 Code de la passerelle en MODE 2

Du côté de la passerelle, le nœud attend d'abord le paquet **`IDREQ`**. Si le service est disponible, le nœud répond avec le paquet **`IDACK`**.
Dans l'étape suivante, le nœud passerelle attend un paquet **`DTREQ`**. Après sa réception, il peut relayer la demande au serveur ThingSpeak ou fournir directement les données demandées comme dans l'exemple suivant.
Comme dans le MODE 1, la passerelle fonctionnant en MODE 2 contient 3 tâches spécifiques :

1. tâche **`TS_Task`** – le rôle de cette tâche est d'imiter la communication avec un serveur type ThingSpeak en lui envoyant une demande des données dont les champs sont indiqués par le code de masque.
2. tâche **`CON_Task`** – cette tâche capte les paquets de contrôle – **`IDREQ`** reçus initialement par la fonction ISR **`onReceive()`**. Après l'analyse du paquet reçu dans la file d'attente, la tâche **`CON_Task`** envoie une paquet **`IDACK`** vers le terminal concerné. Ce paquet porte la clé **AES**.
3. tâche **`DATA_Task`** – cette tâche attend le résultat de communication avec le serveur ThingSpeak et envoie les données reçues dans un paquet **`DTRCV`** vers le terminal concerné.

```
#define GATEWAY  // to choose TERMINAL or MASTER (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 <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
```

```
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; int x,r;
while(true)
  {
  xQueueReceive(tsrcv_queue, &rdf, portMAX_DELAY);
  Serial.println(rdf.pack.channel);Serial.println(rdf.pack.keyword);
  Serial.println(rdf.pack.con[1],HEX);
  LoRa.idle();

  if(rdf.pack.con[1] & 0x80) { stab[0]=10.2+random(10,20);delay(TSdelay); }
  if(rdf.pack.con[1] & 0x40) { stab[1]=10.2+random(10,30);delay(TSdelay); }
  if(rdf.pack.con[1] & 0x20) { stab[2]=10.2+random(10,40);delay(TSdelay); }
  if(rdf.pack.con[1] & 0x10) { stab[3]=10.2+random(10,50);delay(TSdelay); }
  if(rdf.pack.con[1] & 0x08) { stab[4]=10.2+random(10,60);delay(TSdelay); }
  if(rdf.pack.con[1] & 0x04) { stab[5]=10.2+random(10,70);delay(TSdelay); }
  if(rdf.pack.con[1] & 0x02) { stab[6]=10.2+random(10,80);delay(TSdelay); }
  if(rdf.pack.con[1] & 0x01) { stab[7]=10.2+random(10,90);delay(TSdelay); }

  r=random(1,100);
  if(r>50) x=200; else x=100;
  if(x == 200) tss_flag=1;
  else tss_flag=2;

  LoRa_rxMode();
  }
}

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); // sends IDACK  with CKEY
       Serial.printf("IDACK service=%x sent\n",MODE);
      }
    }
  }
}

void DATA_Task(void * pvParameters)  // imitation of the real communication with ThingSpeak
{
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,s=%2.2f,mess=%16.16s\n",termID,tout,stab[0],mess);
  }
}
```

```
void setup() {
  Serial.begin(9600); delay(100);
  gwID=(uint32_t)ESP.getEfuseMac();
  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() {
}
```

10.5.3 Nœuds de terminal et de passerelle – MODE 3 (MQP)

MODE 3 est développé pour envoyer les paquets de données au courtier (*broker*) **MQTT**.
A l'étape 1 le nœud Terminal commence par établir la liaison logique avec la passerelle, puis il envoie les paquets **DTPUB** avec le sujet (**topic**) et les données de message (**mess**).
A l'étape 2, après la réception du paquet **DTPUB**, le nœud passerelle envoie le paquet **DTPUBACK**. Ce dernier paquet informe le Terminal concerné que les nouveaux message a était publié.

10.5.3.1 Code complet du nœud Terminal en MODE 3 (MQP)

Ce qui suit est le code du nœud terminal pour MODE 3 (**MQ**TT Publish - MQ**P**)

```
#define TERMINAL  // to choose TERMINAL or GATEWAY
#define MODE  3  // to choose 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"
char topic[48];
char message[48];
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);
  strcpy(topic,"/esp32/my_sensors/");
  set_LoRa();
  LoRa.onReceive(onReceive);
  LoRa.onTxDone(onTxDone);
  LoRa_rxMode();
  con_queue = xQueueCreate(queueSize, 32);
  mqrcv_queue = xQueueCreate(queueSize,112);
  stage1_flag=1;stage2_flag=0;
}

int cycle=10;   // 10 seconds
RTC_DATA_ATTR int stage1_init=1;

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;
          Serial.printf("Received IDACK from GATEWAY: %08X\n",(uint32_t)gwID);
          stage1_flag=0; stage2_flag=1;
          }
        }
      }
```

```
  if(stage2_flag) // runs until DTPUBACK sets stage2_flag to 1
    {
    MQTTframe_t rdf;
    get_sens();  // get stab[] values
    sprintf(message, "T:%2.2f, H:%2.2f",stab[0],stab[1]);
    Serial.printf("DTPUB MODE=%x sent from TERM: %08X\n",MODE,(uint32_t)termID);
    send_DTPUB(topic,message);
    if(xQueueReceive(mqrcv_queue,rdf.frame,10000)== pdPASS)
      {
      Serial.printf("DTPUBACK from GW:%08X,cycle=%d sec,topic=%s\n",(uint32_t)rdf.pack.sid,
                    (int)rdf.pack.tout,rdf.pack.topic);
      if(rdf.pack.tout<1000 && rdf.pack.tout>10)
        {
          cycle=rdf.pack.tout; Serial.printf("New cycle set to: %d sec",cycle);
        }
      }
    }
  }
}
```

10.5.3.1 Nœud de passerelle – MODE 3 (MQP)

Ce qui suit est le code du nœud de passerelle pour MODE 3 (MQTT Publish - MQP).

```
#define GATEWAY
#define MODE 3  // to choose sender (1), receiver (2) , publisher (3), subscriber (4) mode
#define CKEY "abcdefghijklmnop"  // AES key - 16 bytes
#define PASS "passwordpassword"
#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

int mqpub_flag=0;  // MQTT publish flag
int taskCore = 0;

void MQTT_Task( void * pvParameters ) // this task imitates the real communication with MQTT broker
{
char kbuff[32];
MQTTframe_t rdf;
while(true)
  {
  xQueueReceive(mqrcv_queue, &rdf.pack, portMAX_DELAY);
  LoRa.idle();
  Serial.printf("Publish topic:%s\n",rdf.pack.topic);
  Serial.printf("Publish message:%s\n",rdf.pack.mess);
  delay(2000);mqpub_flag=1;
  LoRa_rxMode();
  }
}

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 with CKEY
       Serial.printf("IDACK service=%x sent\n",MODE);
      }
    }
  }
}
void DATA_Task(void * pvParameters)  // sending DTPUBACK packet
{
TSframe_t rdf;  int tout=10; char mess[48];char control[48];
strcpy(mess,"control message");
strcpy(control,"control message");
```

```
while(true)
  {
  while(mqpub_flag==0)
    {
      Serial.println("wating for DTPUB packet and MQTT return");
      delay(1000);
    }
  if(mqpub_flag==1) {Serial.println("got MQTT PUB return OK");   strcpy(mess,"TS update OK    ");}
  mqpub_flag=0; tout=(int)(10+random(10,50));
  send_DTPUBACK(control,mess,tout);  mqpub_flag=0;
  Serial.printf("DTPUBACK to TERMINAL: %08X ; timeout=%d\n",termID,tout);
  stage2_flag=0;

  }
}

void setup() {
  Serial.begin(9600); delay(100);
  gwID=(uint32_t)ESP.getEfuseMac();
  set_LoRa();
  LoRa.onReceive(onReceive);
  LoRa.onTxDone(onTxDone);
  LoRa_rxMode();
  con_queue = xQueueCreate(queueSize, 32);
  mqrcv_queue = xQueueCreate(queueSize, 112);
  xTaskCreatePinnedToCore(
                    MQTT_Task,   /* Function to implement the task */
                    "MQTT_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("MQTT_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() {

}
```

10.5.4 Nœuds de terminal et de passerelle – `MODE 4` (MQS)

Le MODE 4 est conçu pour envoyer la demande de données via la commande d'abonnement au sujet (topic) du courtier MQTT. Dans notre cas, le nœud Terminal, après la réception de l'identifiant de la passerelle (**`gwID`**) et de la clé AES, envoie le paquet **`DTSUB`**. Ce paquet contient le sujet à souscrire.
Le nœud passerelle fait appel à cette demande au courtier MQTT et attend les nouvelles données publiées sur le sujet spécifié. Ces données sont ensuite envoyées au nœud Terminal avec le paquet **`DTSUBACK`**.
Dans notre cas, nous générons les données localement dans le nœud de passerelle.

10.5.4.1 Code complet du nœud Terminal en MODE 4

Voici le code du Terminal fonctionnant en MODE 4 avec abonnement au sujet donné.

```
#define TERMINAL  // to choose TERMINAL or GATEWAY
#define MODE  3  // to choose 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/broker
char topic[48];   // this variable may be loaded from an EEPROM (external-internal)
char message[48];  // this variable to be loaded with sensor message
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);
  strcpy(topic,"/esp32/my_sensors/");
  set_LoRa();
  LoRa.onReceive(onReceive);
  LoRa.onTxDone(onTxDone);
  LoRa_rxMode();
  con_queue = xQueueCreate(queueSize, 32);
  mqrcv_queue = xQueueCreate(queueSize,112);
  stage1_flag=1;stage2_flag=0;
}

int cycle=10;

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)
      {
      conframe_t rcf;
      send_IDREQ("passwordpassword");
      Serial.printf("IDREQ MODE=%x sent from TERMINAL: %08X\n",MODE,(uint32_t)termID);
      if(xQueueReceive(con_queue, rcf.frame, 10000)== pdPASS)
        {
        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;
          Serial.printf("Received IDACK from GATEWAY: %08X\n",(uint32_t)gwID);
          stage1_flag=0; stage2_flag=1;
          }
        }
      }
```

```
  if(stage2_flag) // runs until DTPUBACK sets stage2_flag to 1
    {
    MQTTframe_t rdf;
    get_sens();  // get stab[] values
    sprintf(message, "T:%2.2f, H:%2.2f",stab[0],stab[1]);
    Serial.printf("DTPUB MODE=%x sent from TERM: %08X\n",MODE,(uint32_t)termID);
    send_DTPUB(topic,message);
    if(xQueueReceive(mqrcv_queue,rdf.frame,10000)== pdPASS)
      {
      Serial.printf("DTPUBACKfrom GW:%08X,cycle=%d sec,topic=%s\n",(uint32_t)rdf.pack.sid,
                    (int)rdf.pack.tout,rdf.pack.topic);
      if(rdf.pack.tout<1000 && rdf.pack.tout>10)
        {
          cycle=rdf.pack.tout; Serial.printf("New cycle set to: %d sec",cycle);
        // we can also put the node into deep_sleep mode during the time of cycle
        }
      }
    }
  }
}
```

10.5.4.2 Code complet du nœud passerelle en MODE 4

Voici le code du nœud de passerelle fonctionnant en MODE 4 avec abonnement au sujet donné. Notez que ce nœud fonctionne avec 4 tâches spécifiques, dont une pour la simulation du service de réception des massages sur le sujet abonné.

```
#define GATEWAY  // to choose TERMINAL or MASTER (GATEWAY) node
#define MODE 4  // to choose sender (1), receiver (2) , publisher (3), subscriber (4) mode
#define CKEY "abcdefghijklmnop"  // AES key - 16 bytes
#define PASS "passwordpassword"
#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
int mqsub_flag=0;  // MQTT subscribe get message flag
int taskCore = 0;
QueueHandle_t mqtt_queue;   // receive queue to get out the data packets from callback function

typedef struct
{
  char topic[48];
  char mess[48];
} SUB_t;  // structure for topic and message

void SUB_Task( void * pvParameters) {  // message reception for the subscribed topic
SUB_t mqtt_sub;
while(true)
  {
    float temp,humi;
  delay(4000); temp=10.2+random(1,40);humi=10.2+random(1,70);
  sprintf(mqtt_sub.mess,"T:%2.2f,H:%2.2f",temp,humi);
  strcpy(mqtt_sub.topic,"incoming topic");
  xQueueReset(mqtt_queue);  // reset queue to keep only the last packet
  xQueueSend(mqtt_queue, &mqtt_sub, 100000);
  mqsub_flag=1;
  }
}

void MQTT_Task( void * pvParameters ){  // receiveng subscribe packet and subscription task
char kbuff[32];
MQTTframe_t rdf;
while(true)
  {
  xQueueReceive(mqrcv_queue, &rdf.pack, portMAX_DELAY);
  Serial.println("Received DTSUB");
  LoRa.idle();
  Serial.printf("%s\n",rdf.pack.topic);
  Serial.println("topic subscribed");
  delay(100);
  LoRa_rxMode();
  }
}
```

```
void CON_Task( void * pvParameters )  // sending IDACK packet
{
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 with CKEY
       Serial.printf("IDACK service=%x sent\n",MODE);
      }
    }
  }
}

void DATA_Task(void * pvParameters)  // sending DTPUBACK packet
{
SUB_t mqtt_sub; int tout=10;

while(true)
  {
  if(xQueueReceive(mqtt_queue, &mqtt_sub, 80000)== pdTRUE)
    {
    tout=10+random(10,50);
    send_DTSUBRCV(mqtt_sub.topic,mqtt_sub.mess,tout);
    Serial.printf("DTSUBRCV to TERMINAL: topic:%s, message:%s\n",mqtt_sub.topic,mqtt_sub.mess);
    }
  else  Serial.println("mqtt_sub queue timeout");
  stage2_flag=0;
  }
}

void setup() {
  Serial.begin(9600); delay(100);
  gwID=(uint32_t)ESP.getEfuseMac();
  set_LoRa();
  LoRa.onReceive(onReceive);
  LoRa.onTxDone(onTxDone);
  LoRa_rxMode();
  con_queue = xQueueCreate(queueSize, 32);
  mqtt_queue = xQueueCreate(queueSize, 96);
  mqrcv_queue = xQueueCreate(queueSize, 112);
  xTaskCreatePinnedToCore(
                    MQTT_Task,   /* Function to implement the task */
                    "MQTT_Task", /* Name of the task */
                    10000,       /* Stack size in words */
                    NULL,        /* Task input parameter */
                    0,           /* Priority of the task */
                    NULL,        /* Task handle. */
                    0);  /* Core where the task should run */
  Serial.println("MQTT_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. */
                    0);  /* Core where the task should run */
  Serial.println("DATA_Task created...");
  xTaskCreatePinnedToCore(
                    SUB_Task,   /* Function to implement the task */
                    "SUB_Task", /* Name of the task */
                    10000,       /* Stack size in words */
                    NULL,        /* Task input parameter */
                    0,           /* Priority of the task */
                    NULL,        /* Task handle. */
                    0);  /* Core where the task should run */
  Serial.println("SUB_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. */
                    0);  /* Core where the task should run */
  Serial.println("CON_Task created...");


}


void loop()
{
}
```

10.6 A faire

1. Testez les 4 modes de services
2. Utilisez
   `set_LoRa_Radio_Para(unsigned long freq,unsigned sbw,int sf, uint8_t sw)`
   pour modifier les paramètres de la radio
3. Au nœud Terminal, ajoutez une fonction ou tâche de capteur (à la place d'une fonction) pour fournir les données pour les MODES 1 et 3 (**`DTSND`**, **`DTPUB`**)
4. Ajouter un écran OLED dans le nœud de passerelle pour afficher les données des paquets **`DTSND`** et **`DTPUB`**

Table des matières