

SAÉ 32 : Développer des applications communicantes

**THOMAS, TÉVANÉ
AZÉ, LEVEIL**

À l'attention de M. VAL

Année universitaire 2024/2025

Sommaire :

I. Couche 2	7
A. Capsule 1 : Simple transmission	7
1. Présentation générale et objectif de l'exercice	7
2. Format des trames	7
3. Diagramme de séquence	8
4. Automates	9
5. Code source	10
6. Jeux d'essai	15
7. Démo	16
8. Compétences acquises	16
B. Capsule 2 : Data-ack	17
1. Présentation générale et objectif de l'exercice	17
2. Format des trames	18
3. Diagramme de séquence	19
4. Automates	20
5. Code source	21
6. Jeux d'essai	28
7. Démo	30
8. Compétences acquises	30
C. Capsule 3 : Erreur de transmission	31
1. Présentation générale de l'exercice	31
2. Détection d'erreur	31
3. Correction d'erreur	45
4. Compétences acquises	52
D. Capsule 4 : Couche MAC en accès aléatoire	53
1. Présentation des objectifs de l'exercice	53
2. Format des trames	53
3. Chronographie protocolaire	54
4. Automates	55
5. Code source	56
6. Démo	59
7. Compétences acquises	60
II. Routage par inondation	60
A. Travail demandé	60
B. Sprint 1 : Création d'un en-tête de couche 3	60
1. Objectif de cette partie	60
2. Format des paquets	61
3. Algorithmes	61
4. Tests réalisés	62
5. Avantages et limites	62
C. Sprint 2 : Routage et tempête de broadcast	63
1. Objectif de la partie traitée	63
2. Format des messages	63

3. Algorithmes conçus	63
4. Tests réalisés	64
5. Avantages et limites de cette étape	65
D. Sprint 3 : Création d'un TTL pour limiter la vie des paquets	65
1. Objectif de cette partie	65
2. Format des paquets	65
3. Algorithme	66
4. Test réalisé	67
5. Avantages et limites	68
E. Sprint 4 : Champ d'adresse source et de numéro de séquence	68
1. Objectif de cette partie	68
2. Format des paquets	68
3. Algorithme	69
4. Test réalisé	70
5. Avantages et limites	70
F. Sprint 5 : Mémorisation de paquet	71
1. Objectif de cette partie	71
2. Format des paquets	71
3. Algorithme	71
4. Test réalisé	72
5. Avantages et limitations	72
G. Sprint 6 : Mémorisation de N paquet	73
1. Objectif de cette partie	73
2. Format des paquets	73
3. Algorithme	73
4. Code source du projet	74
5. Tests réalisés	87
6. Avantages et limites	87
III. Conclusion	88

I. Couche 2

A. Capsule 1 : Simple transmission

1. Présentation générale et objectif de l'exercice

Le premier exercice demandé lors de la SAE 32 consiste à programmer un nœud sans fil pour qu'il émettent des trames (Tx) en point à point vers un nœud récepteur (Rx), on supposera donc que les deux nœuds sont à portée radio, que le trafic n'est pas trop perturbé et que la durée entre les trames émises soit suffisant pour laisser le temps au récepteur de traiter la dernière trame arrivée.

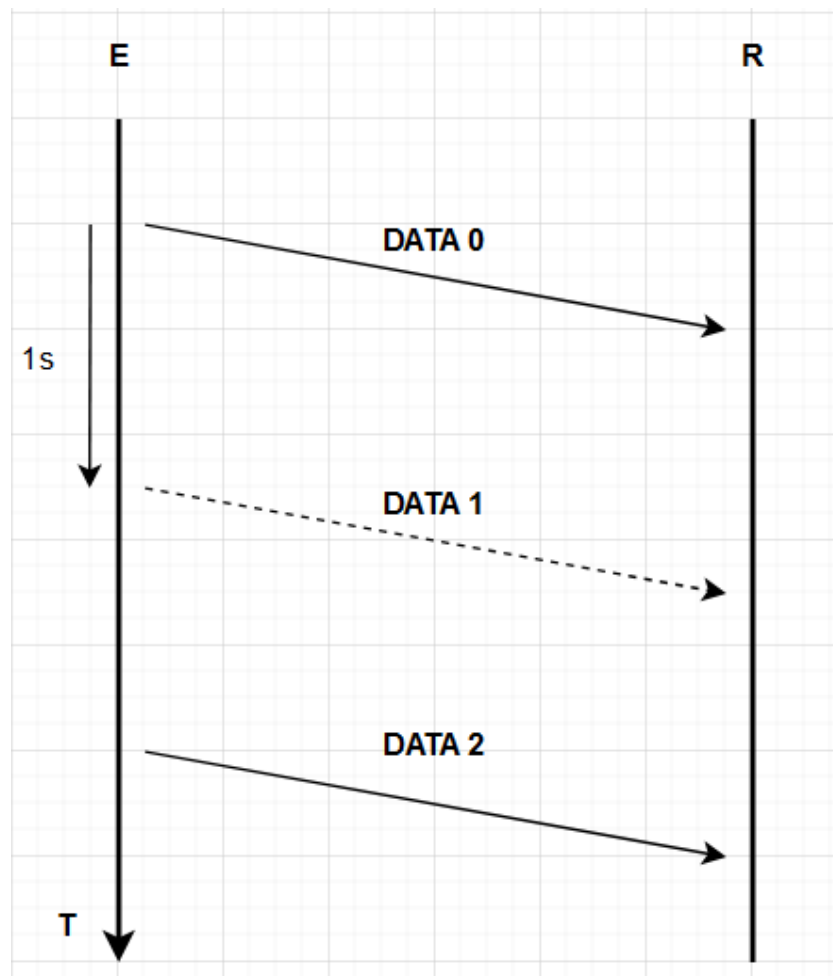
L'objectif de cet exercice est d'être capable de programmer en C un nœud émetteur d'une trame et programmer ensuite le récepteur pour qu'il reçoive et affiche cette trame.

2. Format des trames

- C'est un cas simple, il n'y a qu'une seule trame, donc pas nécessaire de différencier les trames par un champ de commande.
- Nous allons utiliser les fonctionnalités du transceiver radio, donc , pas nécessaire de définir un début et une fin de trame
 - Le transceiver radio formate lui-même la trame en rajoutant aux données utiles (que le programme va lui donner) des informations comme : début de la trame ; fin de la trame ; préambule initial (synchronisation de l'horloge de réception DLL).
- Longueur fixe connu de E et R (pas de champ de longueur) : 2 octets (16bits)
- Pas de détection ni de correction d'erreur
 - En supposant que le médium est de bonne qualité, son taux d'erreur bits (TEB) est bon. Le TEB permet de déduire le taux d'erreur trame. Si il y a erreur de réception: soit il affiche les erreurs de réception; soit rien ne s'affiche.
- Il n'y a qu'un E et qu'un R qui est le seul à pouvoir recevoir la trame, donc pas de champ d'adresse destination (ni Source)
- Pas de numéro de Trame. (le récepteur peut compter les trames reçu)

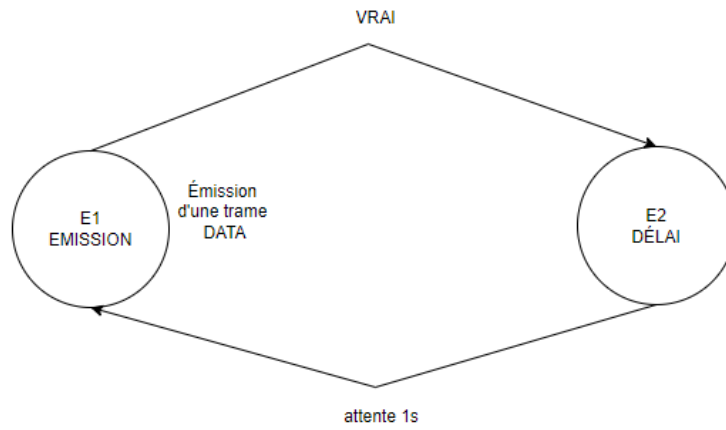
0X0AA	0X55
-------	------

3. Diagramme de séquence

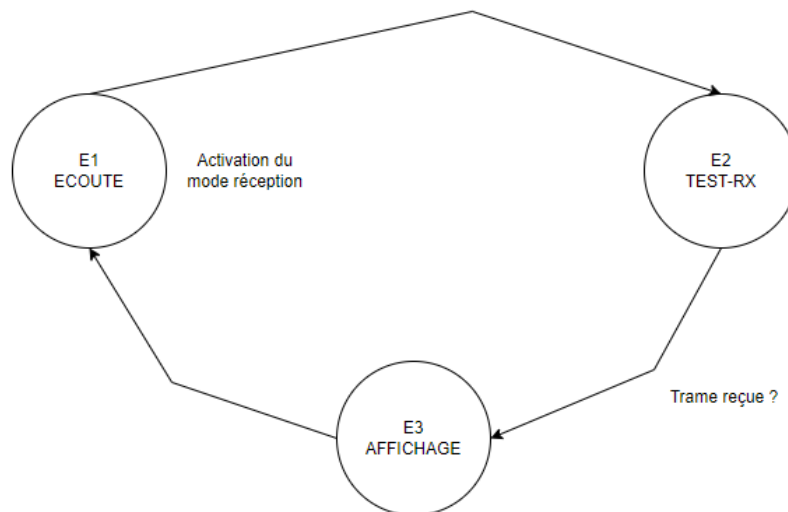


4. Automates

automate émetteur Tx



automate récepteur Rx



5. Code source

a) Structure générale

Les codes source utilisent une structure générale

```
#include <SPI.h>
#include <RH_RF95.h>
#include <M5Stack.h>

//Communication avec le module LoRa
#define RFM95_CS 5 //M5 LoRa
#define RFM95_DIO0 36 //M5 LoRa
RH_RF95 rf95(RFM95_CS, RFM95_DIO0);
```

On utilise les modules SPI.h, RH_RF95.h et M5Stack.h tout au long de la SAÉ.

De plus, on configure les constantes permettant de configurer les broches afin d'interfacer la bibliothèque RH_RF95.h avec le module LoRa du M5Stack.

```
void setup() {
    M5.begin();
    M5.Power.begin();
    M5.Lcd.println("Emetteur CAP1");

    delay(1000);
    if(!rf95.init()){
        M5.Lcd.println("Echec de la mise en place");
    }
    else{
        M5.Lcd.println("Mise en place reussie !!!");
        //Configuration de la radio
        rf95.setModemConfig(RH_RF95::Bw125Cr45Sf128);
        rf95.setFrequency(867.8);
        state=...;
    }
}
```

La fonction setup permet pour tous les codes source de la SAÉ d'initialiser le M5 Stack et ses composants avec un délai pour leur laisser le temps de s'allumer correctement, ainsi que le module LoRa avec la configuration de la radio puis de l'état d'émission.

```
void loop() {  
    switch(state){  
        case état 1:  
            . . . . .  
            break;  
        case état 2:  
            . . . . .  
            break;  
        default:  
            break;  
    }  
}
```

La fonction loop utilise un switch case permettant de naviguer entre chaque état sans avoir à surcharger les codes de if, else if.

b) Code source émetteur

```
//Constantes d'état  
#define EMISSION 0  
#define DELAI 1  
  
uint8_t state; //État courant  
uint8_t data[16]; //Buffer d'émission
```

On définit pour l'émetteur, les constantes pour l'état d'émission et de délai.
Puis on définit la variable state qui représente l'état courant dans le switch case.
Enfin, on définit un buffer d'émission data de 16 octets qui nous permettra d'envoyer les données.

```
void setup() {  
    . . . . .  
    state=EMISSION;  
    . . . . .  
}
```

Dans la fonction setup, on définit l'état courant à l'état EMISSION.

```
case EMISSION: //Code source à l'état d'émission  
    Serial.println("EMISSION");  
    M5.Lcd.println("EMISSION");  
    data[0]=0xAA;  
    data[1]=0x55;  
    rf95.send(data,2); //Envoi des données  
    rf95.waitPacketSent();  
    state=DELAI;  
    break;
```

Dans le cas EMISSION, on remplit les deux premiers octets avec 0xAA et 0x55.
Puis on envoie le buffer ainsi que sa taille par radio avec la fonction rf95.send(data,2).
Enfin, on change d'état pour passer à un délai.

```
case DELAI:  
    Serial.println("DELAI");  
    M5.Lcd.println("DELAI");  
    delay(1000);  
    state=EMISSION;  
    break;
```

Dans le cas DELAI, on met en place un délai d'une seconde avant de repasser dans l'état d'émission.

c) Code source récepteur

```
//Déclaration des constantes
uint8_t state; //État courant
uint8_t rxbuf[RH_RF95_MAX_MESSAGE_LEN]; //buffer en reception
uint8_t rxbuflen = sizeof(rxbuf); //taille du buffer en réception
int rxFrames; //Numéro de la trame reçue
```

Dans code récepteur, on définit le buffer rxbuf ainsi la taille rxbuflen.

Enfin, on définit la variable rxFrames qui correspond au nombre de trames reçues par le récepteur.

```
//Constantes d'état
#define ECOUTE 1
#define TEST_RX 2 //Les différents états
#define AFFICHAGE 3
```

Pour le récepteur, on définit les états ECOUTE, TEST_RX et AFFICHAGE.

```
void setup() {
    . . . . .
    state=ECOUTE;
    rxFrames = 0;
    . . . . .
}
```

Dans la fonction setup, on initialise l'état à ECOUTE et le numéro de trame à 0.

```
case ECOUTE:
    Serial.println("ECOUTE");
    M5.Lcd.println("ECOUTE");
    rf95.setModeRx();
    state=TEST_RX;
    break;
```

Dans l'état ECOUTE, on prépare la radio du module LoRa à recevoir des informations avec rf95.setModeRx().

Puis on passe à l'état TEST_RX.

```

case TEST_RX:
    Serial.println("TEST-RX");
    M5.Lcd.println("TEST-RX");
    if(rf95.available()){
        state=AFFICHAGE;
    }
    //délai permettant la récupération
    delay(2000);
    break;

```

Dans l'état TEST_RX, on vérifie si la radio a reçu des informations avec rf95.available(). Si c'est le cas, on passe à l'état AFFICHAGE pour extraire les données. Sinon, on reste dans l'état TEST_RX.

Il a été nécessaire de mettre un délai pour laisser le temps à l'émetteur d'envoyer des informations avant que le récepteur ne sature l'écran LCD avec "TEST_RX".

```

case AFFICHAGE:
    if(rf95.recv(rxbuf, &rxbuflen)){
        rxFrames+=1;
        Serial.println();
        M5.Lcd.println();

        Serial.printf("AFFICHAGE : Trame[%d] de ", rxFrames);
        M5.Lcd.printf("AFFICHAGE : Trame[%d] de ", rxFrames);

        Serial.printf("%d octets : |",rxbuflen);
        M5.Lcd.printf("%d octets : |",rxbuflen);

        for(int j=0;j<rxbuflen;j++){
            Serial.printf("%02x|", rxbuf[j]);
            M5.Lcd.printf("%02x|", rxbuf[j]);
        }
        Serial.println("\n");
        M5.Lcd.println("\n");
    }
    state=ECOUTE;
    break;

```

Dans l'état AFFICHAGE, on extrait le buffer et sa taille à l'aide de rf95.recv() et on incrémente le nombre de trames reçues.

Par la suite, on affiche les informations contenues dans le buffer à l'aide d'une boucle for.

6. Jeux d'essai

Émetteur	Récepteur
<pre> =====Emetteur CAP1===== Mise en place reussie !!! EMISSION DELA EMISSION DELA EMISSION DELA EMISSION DELA </pre>	<pre> =====Recepteur CAP1===== Mise en place reussie !!! Boucle principale ECOUTE TEST-RX AFFICHAGE : Trame[1] de 5 octets : aa 55 25 24 98 ECOUTE TEST-RX AFFICHAGE : Trame[2] de 5 octets : aa 55 25 24 98 ECOUTE TEST-RX AFFICHAGE : Trame[3] de 5 octets : aa 55 25 24 98 ECOUTE TEST-RX AFFICHAGE : Trame[4] de 5 octets : aa 55 25 24 98 </pre>

Le récepteur affiche bien les informations de la trame et s'adapte à la taille du message.

7. Démo

Émetteur	Récepteur
<pre> =====Emetteur CAP1===== Mise en place reussie !!! EMISSION DELA EMISSION DELA EMISSION DELA EMISSION DELA </pre>	<pre> =====Recepteur CAP1===== Mise en place reussie !!! Boucle principale ECOUTE TEST-RX AFFICHAGE : Trame[1] de 2 octets : aa 55 ECOUTE TEST-RX AFFICHAGE : Trame[2] de 2 octets : aa 55 ECOUTE TEST-RX AFFICHAGE : Trame[3] de 2 octets : aa 55 ECOUTE TEST-RX AFFICHAGE : Trame[4] de 2 octets : aa 55 </pre>

Le Récepteur envoie bien les deux octets au récepteur.

8. Compétences acquises

Lors de cette première capsule, nous étions capables d'aborder une structure de programme en C pour de la programmation événementielle.

De plus, nous sommes capables de transmettre des informations entre deux nœuds.

B. Capsule 2 : Data-ack

1. Présentation générale et objectif de l'exercice

Programmer un nœud sans fil émetteur pour qu'il envoie des trames (Tx) vers un nœud sans fil récepteur (Rx). Lorsque Rx reçoit les trames de Tx, il émet une trame d'acquittement pour confirmer qu'il a bien reçu la dernière trame envoyée par Tx.

Objectif :

- Fiabiliser le transfert de donnée par un acquittement (LLC3)
- programmer l'émission l'attente de L'ACK et l'éventuelle réémission de la DATA
- programmer la réception de la DATA et l'émission de l'ACK

Objectif de l'exercice :

Transmettre une trame de données (DATA) au nœud récepteur afin qu'il renvoie une trame d'acquittement (ACK).

L'émetteur attend une trame d'acquittement de la part du récepteur avant de renvoyer de nouvelles données. Cependant, il ne s'agit pas d'un mode connecté : le récepteur ne sait pas à quel moment il recevra les données. Il n'y a donc pas d'anticipation de la part du récepteur.

Scénario sans erreur :

Lorsque l'émetteur (E) envoie une trame de données, il attend que le récepteur (R) la reçoive et envoie une trame d'acquittement en retour. Une fois l'acquittement reçu, l'émetteur peut continuer à envoyer des données. Cela montre que l'émetteur et le récepteur sont bien à portée radio.

Scénario avec perte totale de la trame :

À chaque envoi de données, l'émetteur active un "chien de garde" (CdG) pour surveiller la réception de l'acquittement. Si la trame est perdue, l'absence d'acquittement déclenchera une retransmission. Le CdG limite le temps d'attente pour l'acquittement. Si le CdG expire, l'émetteur renvoie la trame initiale, mais avec un nombre limité de tentatives pour éviter les envois infinis en cas d'échec. Si la trame est bien reçue, le CdG se réinitialise pour la trame suivante.

Scénario avec perte de la trame d'acquittement :

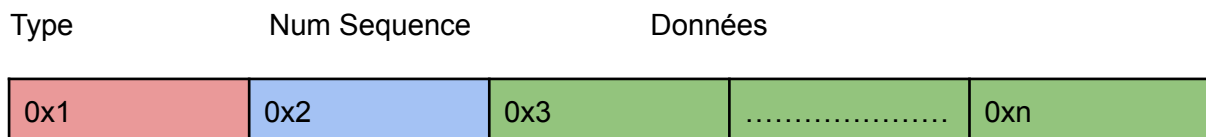
L'émetteur envoie une trame de données, mais l'acquittement est perdu ou n'arrive pas correctement. Dans ce cas, l'émetteur renvoie simplement la trame de données, pensant que le problème peut provenir de la trame de données ou de l'acquittement. Cela entraîne une duplication de données, que le récepteur gérera en absorbant le contenu sans le dupliquer.

Bilan des apprentissages :

- Programmation d'un échange data-ack type LLC3
- fiabilisation de transmission de data même si perte de trame
- utilisation attente non bloquante
- programmation chien de garde
- gestion duplication de trames de data

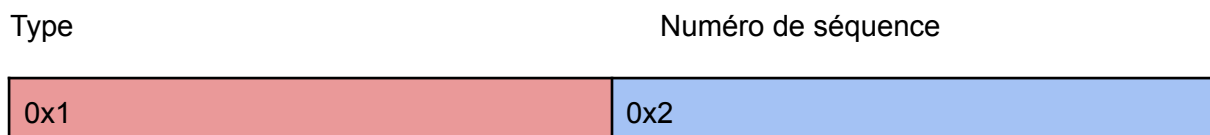
2. Format des trames

○ Données



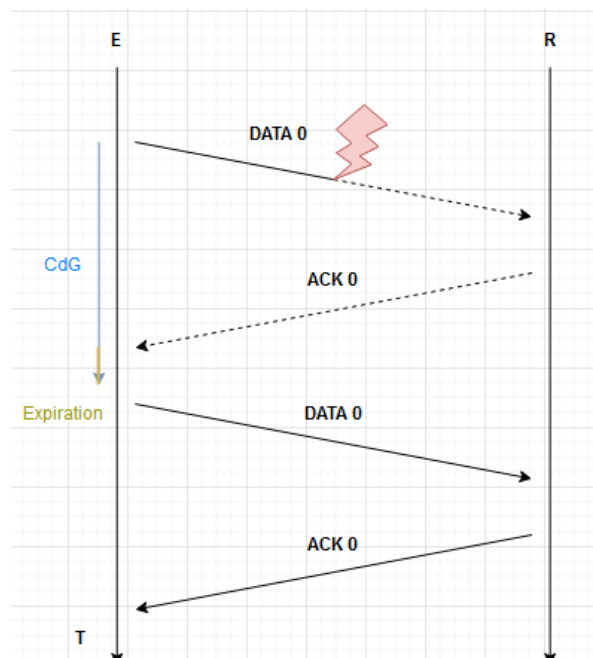
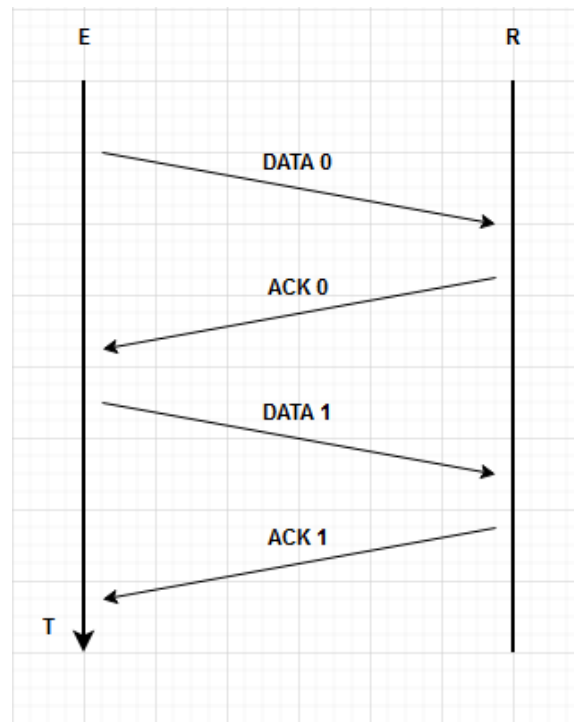
La trame de données est composée d'un champ type, d'un champ de numéro de séquence et d'octets de payload.

○ Acquittement



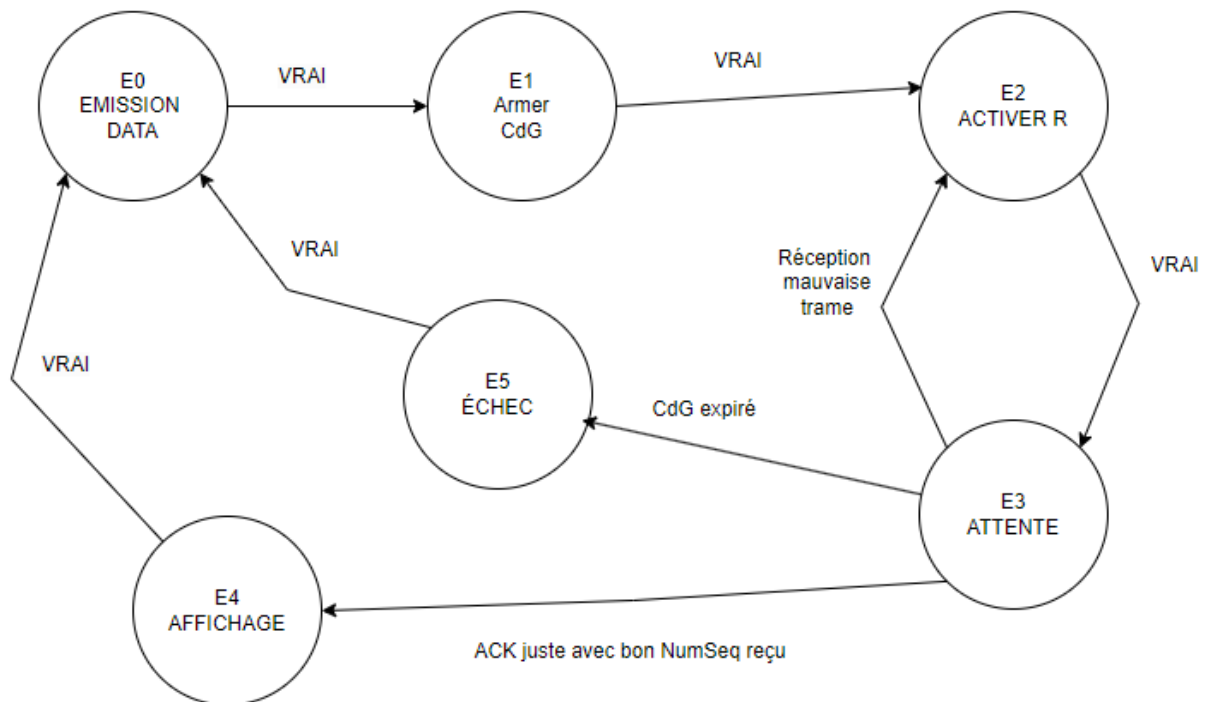
La trame d'acquittement est composée d'un champ type et d'un numéro de séquence.

3. Diagramme de séquence

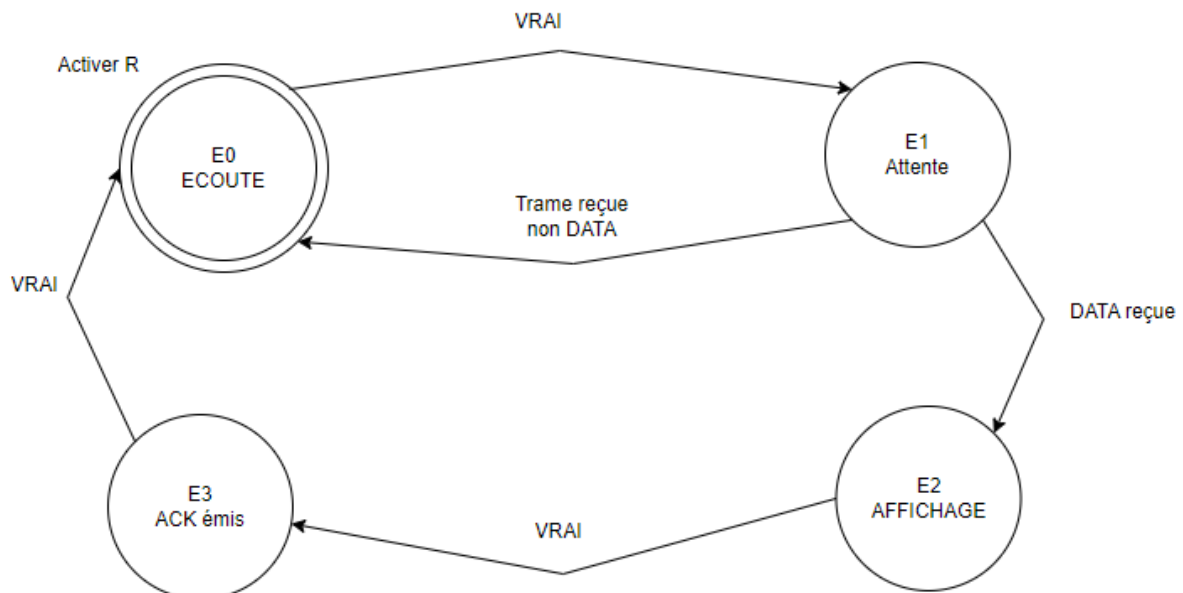


4. Automates

automate émetteur Tx



automate récepteur Rx



5. Code source

a) Code émetteur

```
uint8_t txbuf[RH_RF95_MAX_MESSAGE_LEN]; //Buffer d'emission
uint8_t rxbuf[RH_RF95_MAX_MESSAGE_LEN], rxbuflen=sizeof(rxbuf);
//Buffer de réception et longueur
uint8_t state, TxSeq, credit;
uint32_t attente;
```

On crée deux buffers de réception et d'émission dans afin de pouvoir recevoir les acquittements et envoyer les données.

On définit une variable de numéro de séquence en envoi et en réception ainsi qu'un crédit qui nous permettra de poursuivre le transfert si un message n'arrive pas à être acquitté.

```
#define E0 0
#define E1 1
#define E2 2
#define E3 3
#define E4 4
#define E5 5

#define TYPE_DATA 1
#define TYPE_ACK 2
#define TIMEOUT_ACK 40

#define MAX_CREDIT 5
```

On définit les constantes relatives à nos six états, puis les constantes de type de trame (DATA ou ACK) et enfin un timer qui va servir de chien de garde.

```

void setup() {
    . . . . .
    state=E0;
    TxSeq=0;
    credit=MAX_CREDIT;
    . . . . .
}

```

Dans le setup, on initialise l'état à E0.

Puis le numéro de séquence à 0.

Enfin, on initialise la variable de crédit à 5. Si la transmission d'un message échoue plus de 5 fois alors on passe au message suivant.

```

case E0: //Code source à l'état d'émission
    txbuf[0]=TYPE_DATA;
    txbuf[1]=TxSeq;
    txbuf[2]=0xAA;
    txbuf[3]=0x55;

    rf95.send(txbuf,4); //Envoi des données
    rf95.waitPacketSent();

    credit -=1;
    state=E1;
    break;

```

Dans l'état E0, on crée la trame avec le champ de type, le numéro de séquence ainsi que les deux champs de données.

Par la suite on envoie les données avec la taille correspondante.

Enfin, on diminue le crédit et on passe à l'état E1.

```

case E1: //CDG
    attente = millis()+TIMEOUT_ACK; //Armement du Chien de
    Garde
    state = E2;
    break;

```

Dans l'état E1, on utilise la fonction millis() pour récupérer le temps en millisecondes qui s'est écoulé depuis le démarrage de la carte ainsi que de la constante TIMEOUT_ACK afin de générer un chien de garde non bloquant.

Par la suite, on passe à l'état E2.

```
case E2: //ECOUTE
    rf95.setModeRx();
    state=E3;
    break;
```

Dans l'état E2, on utilise setModeRx afin de préparer l'émetteur à recevoir des informations dont l'Acquittement.

On passe à l'état E3.

```
case E3: //TEST_CDG_DEPASSE
    if(millis() > attente){ //Test expiration CdG
        Serial.println("CDG expire");
        M5.Lcd.println("CDG expire");
        state=E5; //Délais expiré
    }
    else{
        if(rf95.recv(rxbuf, &rxbuflen)){
            if((rxbuf[0]==TYPE_ACK) && (rxbuf[1]==TxSeq)){
                //Reception d'un ACK avec le bon num de séquence
                state=E4;
            }
            else{
                state=E2;
            }
        }
    }
    break;
```

Dans l'état E3, on vérifie d'abord que la durée du chien de garde n'est pas dépassée.

Si c'est le cas, on passe à l'état E5.

Sinon, on récupère le buffer d'émission ainsi que la taille.

Si le type reçu est bien un acquittement et que le numéro de séquence correspond à celui attendu alors on passe à l'état E4.

```
case E4: //ACK_RECUE
    Serial.println("ACK_RECUE");
    M5.Lcd.println("ACK_RECUE");
    state=E0, TxSeq+=1, credit=MAX_CREDIT;
    break;
```

Dans l'état E4, on incrémente le numéro de séquence et on renouvelle le crédit.

Puis on retourne dans l'état d'émission E0.

```

case E5: //PROBLEME
    Serial.printf("Probleme\n");
    M5.Lcd.printf("Probleme\n");
    if (credit==0){
        Serial.print("ECHEC\n");
        M5.Lcd.print("ECHEC\n");
        state=E0;
        credit=MAX_CREDIT;
        TxSeq+=1;
    }
    else{
        Serial.printf("Nouvelle tentative n %d\n", MAX_CREDIT-credit);
        M5.Lcd.printf("Nouvelle tentative n %d\n", MAX_CREDIT-credit);
        state=E0;
        Serial.println();
        M5.Lcd.println();
    }
    break;

```

Dans l'état E5, on vérifie au préalable que le crédit n'est pas expiré.

Si c'est le cas (passage au message suivant) alors on retourne à l'état d'émission E0, on renouvelle les crédits pour le message suivant et on incrémente le numéro de séquence pour le message suivant.

Sinon, on retourne à l'état d'émission E0 pour renvoyer le message.

b) Code récepteur

```

uint8_t state;
uint8_t txbuf[RH_RF95_MAX_MESSAGE_LEN];
uint8_t rxbuf[RH_RF95_MAX_MESSAGE_LEN], rxbuflen=sizeof(rxbuf);

uint8_t RxSeq;

```

On définit les buffers d'envoi et de réception ainsi qu'un numéro de séquence.

```
#define E0 0
#define E1 1
#define E2 2
#define E3 3

#define TYPE_DATA 1
#define TYPE_ACK 2
```

Par la suite, on définit les constantes correspondant aux quatre états puis on définit des constantes de type (données ou acquittement).

```
void setup() {
    . . . . .
    state=E0;
    RxSeq=255;
    . . . . .
}
```

Dans le setup, on initialise l'état à E0 puis la variable de numéro de séquence à 255. Puisqu'on code sur un octet en non signé, alors on commence à l'octet à 255 juste avant 0.

```
case E0: //ECOUTE
    Serial.println("Mode Reception");
    M5.Lcd.println("Mode Reception");
    rf95.setModeRx();
    state=E1;
    break;
```

Dans l'état E0, on prépare le récepteur à recevoir des données puis on passe à l'état E1.

```

case E1: //TEST_RECEPTION
    if(rf95.recv(rxbuf, &rxbuflen)){
        if(rxbuf[0]==TYPE_DATA){
            state=E2;
        }
        else{
            state=E0;
        }
    }
    break;

```

Dans l'état E1, si le champ type du message reçu correspond à TYPE_DATA alors on passe à l'état E2, sinon on retourne à E0.

```

case E2: //TEST_DUPLICATION
    if(RxSeq!=rxbuf[1]){
        RxSeq=rxbuf[1];
        Serial.printf("[%d] ", RxSeq);
        M5.Lcd.printf("[%d] ", RxSeq);
        Serial.printf("DATA de %d octets :", rxbuflen);
        M5.Lcd.printf("DATA de %d octets :", rxbuflen);
        Serial.println();
        M5.Lcd.println();
        for(int i=0;i<rxbuflen;i++){
            Serial.printf("%02x|", rxbuf[i]);
            M5.Lcd.printf("%02x|", rxbuf[i]);
        }
        Serial.println();
        M5.Lcd.println();
    }
    else{
        Serial.printf("Duplication\n");
        M5.Lcd.printf("Duplication\n");
    }
    state=E3;
    break;

```

Dans l'état E2, on vérifie s'il y a duplication. Si ce n'est pas le cas, alors on affiche le message.

Sinon, on indique qu'il y a eu duplication. Enfin, on passe à l'état E3.

```
case E3: //ENVOI_ACK

    Serial.println("Envoi ACK");
    M5.Lcd.println("Envoi ACK");

    txbuf[0]=TYPE_ACK;
    txbuf[1]=rxbuf[1];

    rf95.send(txbuf,2);
    rf95.waitPacketSent();

    state=E0;
    break;
```

Dans le cas E3, on formate la trame d'acquittement composée du type et du numéro de séquence attendu puis on l'envoie.

6. Jeux d'essai

a) Perte de la trame à l'envoi

Émetteur	Récepteur
====Émetteur CAP2==== Mise en place réussie !!! Boucle principale. EMISSION 0 ACK_RECU [0] EMISSION 1 ACK_RECU [1] EMISSION 2 Perte de la trame CDG expire Probleme Nouvelle tentative n 1 EMISSION 2 ACK_RECU [2] EMISSION 3 ACK_RECU [3]	[0] DATA de 4 octets : 01 00 aa 55 Envoi ACK Mode Reception [1] DATA de 4 octets : 01 01 aa 55 Envoi ACK Mode Reception [2] DATA de 4 octets : 01 02 aa 55 Envoi ACK Mode Reception [3] DATA de 4 octets : 01 03 aa 55 Envoi ACK

Lorsque la trame est perdue, le CDG expire et l'émetteur renvoie la trame. De son côté, le récepteur ne voit pas le problème et acquitte normalement les trames.

b) Expiration du crédit

Émetteur	Récepteur
====Emetteur CAP2==== Mise en place reussie !!! Boucle principale. EMISSION 0 Perte de la trame CDG expire Probleme Nouvelle tentative n 4 EMISSION 0 Perte de la trame CDG expire Probleme ECHEC EMISSION 1 ACK_RECUC [1]	====Recepteur CAP2==== Mise en place reussie !!! Boucle principale Mode Reception [1] DATA de 4 octets : 01 01 aa 55 Envoi ACK

Si la trame 0 est perdue plus de cinq fois de suite alors elle passe en "ECHEC" et l'émetteur envoie alors la trame suivante 1.

De son côté, le récepteur ne voit que la trame 1 qui lui est parvenue.

c) Perte de l'acquittement

Émetteur	Récepteur
EMISSION 7 ACK_RECUC [7] EMISSION 8 CDG expire Probleme Nouvelle tentative n 1 EMISSION 8 ACK_RECUC [8] EMISSION 9	Mode Reception [7] DATA de 4 octets : 01 07 aa 55 Envoi ACK Mode Reception [8] DATA de 4 octets : 01 08 aa 55 Envoi ACK Perte acquittement Mode Reception Duplication Envoi ACK Mode Reception [9] DATA de 4 octets : 01 09 aa 55 Envoi ACK

L'acquittement de la trame 8 est perdu ce qui entraîne l'expiration du CDG sur l'émetteur.

Ce dernier renvoie alors la trame 8, le récepteur détecte alors une duplication et renvoie l'acquittement correspondant.

7. Démo

a) Fonctionnement normal

Émetteur	Récepteur
====Émetteur CAP2==== Mise en place reussie !!! Boucle principale. EMISSION 0 ACK_RECU [0] EMISSION 1 ACK_RECU [1] EMISSION 2 ACK_RECU [2] EMISSION 3 ACK_RECU [3]	Mode Reception [0] DATA de 4 octets : 01 00 aa 55 Envoi ACK Mode Reception [1] DATA de 4 octets : 01 01 aa 55 Envoi ACK Mode Reception [2] DATA de 4 octets : 01 02 aa 55 Envoi ACK Mode Reception [3] DATA de 4 octets : 01 03 aa 55 Envoi ACK

Chaque message envoyé est bien acquitté.

8. Compétences acquises

À l'issue de cette capsule, nous sommes capables d'assurer le contrôle de la transmission en mettant en place une logique d'acquittement des trames. Nous pouvons gérer la perte de l'information.

C. Capsule 3 : Erreur de transmission

1. Présentation générale de l'exercice

L'objectif de cette troisième vidéo est de réaliser des détecteurs et correcteurs d'erreurs de transmissions sans fil. Il faut ajouter une méthode de redondance aux données utiles transmises.

Identifications des erreurs courants :

- émission de 0 ou de 1 et aucune réception (perte de données transmises)
- réception de données en +
- transformation de bit 0 devient 1 et inversement.

BER =TEB = Le taux d'erreur BIT 1 bit faux / 10 p x bits transmis
 les mauvais sont : 10 puissance -2, 10 puissance -3
 bon sont : 10 p -6, 10 p -7
 très bon sont : 10 p -10 et 10 p -12

Les deux principales méthodes de gestions d'erreurs de transmissions sont :

Rajout d'une redondance aux données utiles transmises

Faible redondance pour détecter l'erreur de transmission dans la trame

2. Détection d'erreur

a) Format des trames

(1) Trame DATA

La trame de data présente 1 octet de type DATA, 1 octet de numéro de séquence et 18 octets de charges utiles. Le OU-EXCLUSIF(XOR) est calculé à partir de ces 20 octets par l'émetteur et est ajouté à la fin de la trame émise.

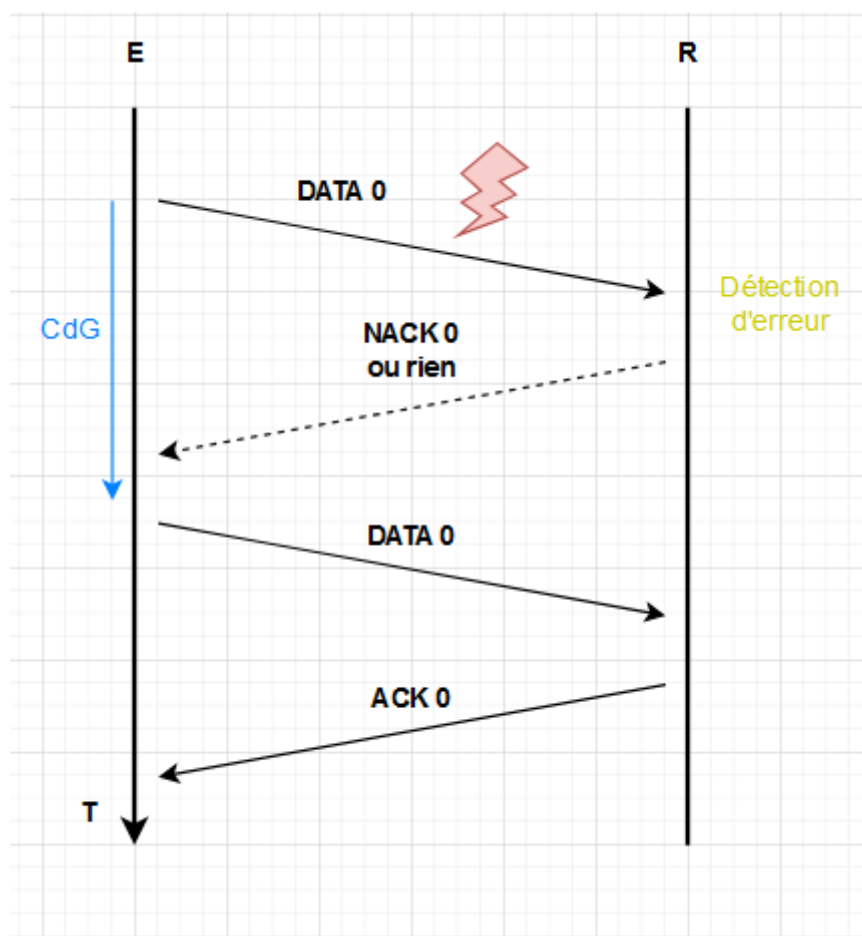
type	NumSeq					FCS
0X1	0X0...	DATA 0	DATA 1	informations	DATA 17	XOR

(2) Trame ACK

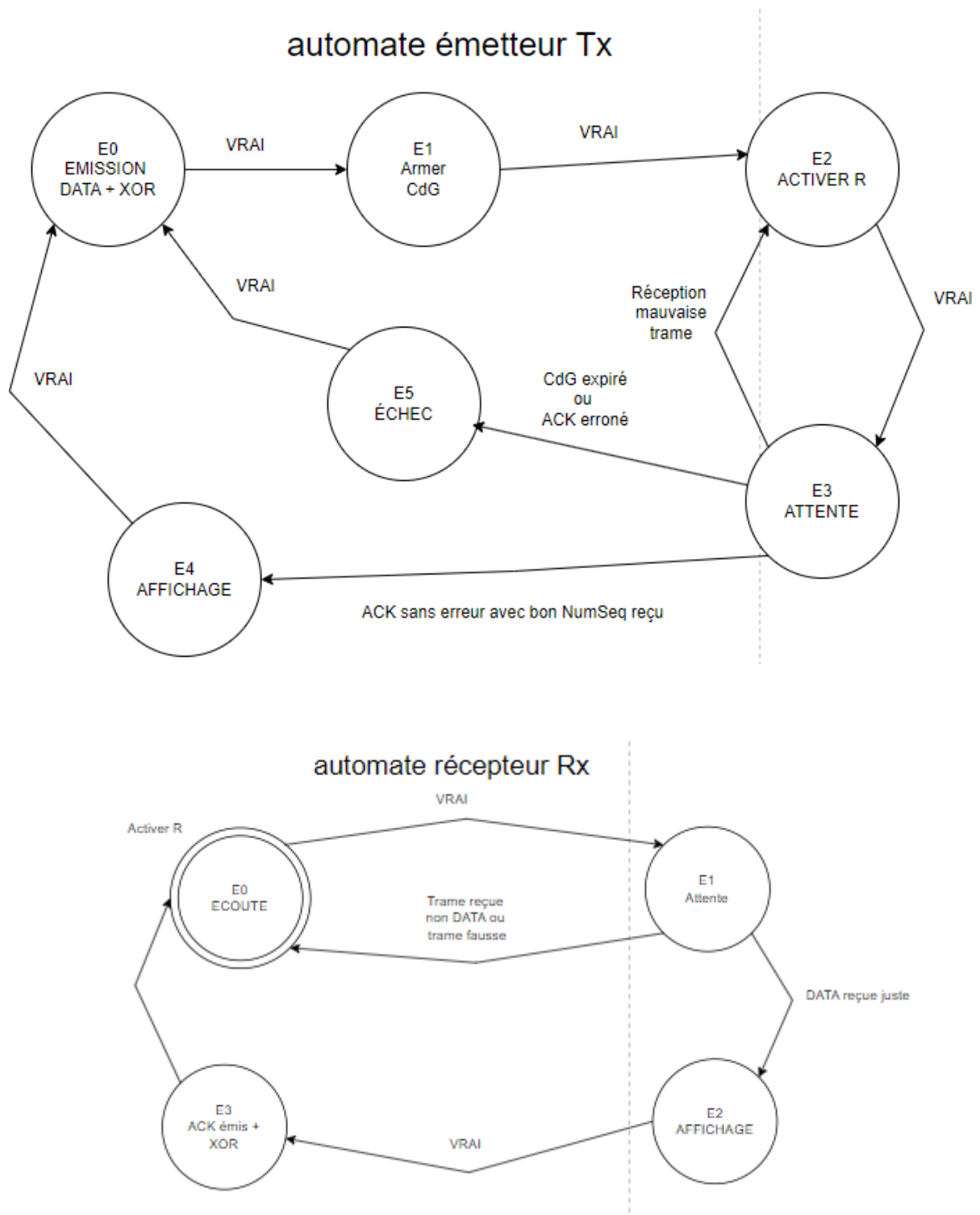
1 octet pour le type ACK, 1 octet de numéro de séquence et le XOR est calculé à partir de ces deux octets et forme le dernier champ de la trame ACK émise.

type	NumSeq	FCS
0X2	0x0...	XOR

b) Diagramme de séquence



c) Automates



d) Codes source

(1) Émetteur

```
//Variables
uint8_t txbuf[RH_RF95_MAX_MESSAGE_LEN];
uint8_t rxbuf[RH_RF95_MAX_MESSAGE_LEN], rxbuflen=sizeof(rxbuf);

uint8_t state, TxSeq, credit;
uint32_t attente;
uint8_t FCS; //Champ de contrôle
```

On définit des buffers d'émission et de réception, un temps d'attente ainsi que d'un champs de contrôle.

```
//Constantes d'états
#define E0 0
#define E1 1
#define E2 2
#define E3 3
#define E4 4
#define E5 5

//Constantes de données et d'acquieement
#define TYPE_DATA 1
#define TYPE_ACK 2
#define TIMEOUT_ACK 40
#define MAX_CREDIT 5
```

On définit les constantes correspondantes à nos six états puis une constante de temps pour le chien de garde et enfin, un maximum pour les crédits.

```
//Constantes de champs de trame
#define CHAMP_TYPE 0
#define CHAMP_NUM_SEQ 1
#define CHAMP_MIN_PAYLOAD 2
#define CHAMP_MAX_PAYLOAD 19
#define CHAMP_FCS_DATA 20
#define CHAMP_FCS_ACK 2
```

Pour plus de lisibilité et pour une meilleure évolutivité du format de la trame, nous avons défini des constantes pour les différents champs correspondants aux indices dans les buffers.

```
void setup() {
    . . . . .
    TxSeq=0;
    credit=MAX_CREDIT;
    . . . . .
}
```

Dans le setup de l'émetteur, on initialise le numéro de séquence à 0 et le crédit au maximum défini.

```
case E0: //Code source à l'état d'émission
    txbuf[CHAMP_TYPE]=TYPE_DATA;
    txbuf[CHAMP_NUM_SEQ]=TxSeq;

    //On remplit les 18 octets de payload au max (255)
    for (int i=CHAMP_MIN_PAYLOAD;i<=CHAMP_MAX_PAYLOAD;i++){
        txbuf[i]=255;
    }
    //Calcul du FCS : XOR
    FCS=0; //Champ sur un octet
    for(int i=0;i<=CHAMP_MAX_PAYLOAD;i++){
        //XOR des 20 octets de la trame.
        FCS=FCS ^ txbuf[i];
    }
    //On ajoute le FCS calculé à la fin de la trame
    txbuf[CHAMP_FCS_DATA] = FCS;
    //Envoi des données
    rf95.send(txbuf,21);
```

```

rf95.waitPacketSent();

credit-=1;
state=E1;
break;

```

Dans l'état E0, nous formons la trame avec d'abord le champ type et celui de séquence. Par la suite, on rentre les 18 octets de données à l'aide d'une boucle for puis on effectue un XOR sur les 20 premiers octets de la trame. On entre le résultat du XOR dans le champ FCS et on émet la trame.

```

case E1: //Armement CDG
    attente = millis()+TIMEOUT_ACK; //Armement du Chien de
    Garde
    state = E2;
    break;

```

Dans l'état E1, on arme le chien de garde.

```

case E2: //ECOUTE
    rf95.setModeRx();
    state=E3;
    break;

```

Dans l'état E2, on se met en mode réception.

```

case E3:
    if(millis() > attente){ //Test expiration CdG
        state=E5; //Délais expiré
    }
    else{
        if(rf95.recv(rxbuf, &rxbuflen)){
            //Réception d'un ACK avec le bon num de séquence
            if((rxbuf[CHAMP_TYPE]==TYPE_ACK) &&
(rxbuf[CHAMP_NUM_SEQ]==TxSeq)){
                //Trame reçue sans erreur
                if((rxbuf[CHAMPS_TYPE] ^
rxbuf[CHAMP_NUM_SEQ])==rxbuf[CHAMP_FCS_ACK]){
                    //Affichage de la trame reçue
                    state=E4;
                }
            }
            else{

```

```

        //Affichage de l'échec
        state=E5;
    }
}
else{

```

Dans l'état E3, vérifie tout d'abord si le délai d'attente est expiré. Si c'est le cas, on entre dans l'état E5.

Sinon, on vérifie que le champ type et le numéro de séquence de la trame reçue correspondent. Si c'est le cas, on vérifie si la trame est erronée afin de déterminer si on passe à la trame suivante ou s'il faut émettre à nouveau.

Sinon, on réécoute.

```

case E4: //Reception ACK
    Serial.print("ACK_REC\n");
    M5.Lcd.print("ACK_REC\n");
    state=E0, TxSeq+=1, credit=MAX_CREDIT;
    break;

```

Dans l'état E4, on indique que le bon acquittement est bien reçu et on passe à l'émission de la trame suivante.

```

case E5: //Problème
    Serial.print("E5  ");
    M5.Lcd.print("E5  ");
    if (credit==0){
        Serial.print("ECHEC\n");
        M5.Lcd.print("ECHEC\n");
        credit=MAX_CREDIT;
        TxSeq+=1;
    }
    else{
        Serial.printf("Nouvelle tentative n %d\n",
MAX_CREDIT-credit);
        M5.Lcd.printf("Nouvelle tentative n %d\n",
MAX_CREDIT-credit);
        Serial.println();
        M5.Lcd.println();
    }
    state=E0;
    break;

```

L'état E5 permet de vérifier si les crédits d'envoi de la trame sont épuisés, si c'est le cas on passe à la trame suivante, sinon on réémet en décrémentant les crédits.

(2) Récepteur

```
//Variables
uint8_t state;
uint8_t txbuf[RH_RF95_MAX_MESSAGE_LEN];
uint8_t rxbuf[RH_RF95_MAX_MESSAGE_LEN], rxbuflen=sizeof(rxbuf);
uint8_t RxSeq;
uint8_t FCSc, FCSr; //Champs de contrôle d'un octet calculé et
reçu
```

Pour le récepteur, on définit des buffers de réception et d'émission, un numéro de séquence ainsi que un FCS reçu et calculé.

```
//Constantes d'état
#define E0 0
#define E1 1
#define E2 2
#define E3 3

#define TYPE_DATA 1
#define TYPE_ACK 2
```

On définit les constantes pour nos quatre états ainsi que pour les types de trames.

```
//Constantes de champs de trames
#define CHAMP_TYPE 0
#define CHAMP_NUM_SEQ 1
#define CHAMP_MIN_PAYLOAD 2
#define CHAMP_MAX_PAYLOAD 19
#define CHAMP_FCS_DATA 20
#define CHAMP_FCS_ACK 2
```

Comme pour l'émetteur, on définit des constantes pour les champs de la trame.

```
case E0: //ECOUTE
    Serial.println("E0 : Attente Reception \n");
    M5.Lcd.println("E0 : Attente Reception \n");
    rf95.setModeRx();
    state=E1;
    break;
```

Dans l'état E0, on se met en mode d'écoute puis on passe à l'état E1.

```
case E1:
    if(rf95.recv(rxbuf, &rxbuflen)){
        if(rxbuf[CHAMP_TYPE]==TYPE_DATA){
            //Test si FCS juste
            FCSr=rxbuf[CHAMP_FCS_DATA]; //Champ de contrôle reçu
            //Calcul du FCS
            FCSc=0;
            for(int i=0;i<=CHAMP_MAX_PAYLOAD;i++){
                FCSc= FCSc ^ rxbuf[i];
            }
        }
    }
```

Dans l'état E1, on vérifie d'abord si le message reçu est de type DATA.
Si c'est le cas, on extrait le FCS reçu (FCSr) et on calcule celui de la trame (FCSc).

```

//Mêmes FCS : Trame juste
    if(FCSr==FCSc){
        state=E2;
    }
    //Sinon trame fausse
    else{
        Serial.println();
        M5.Lcd.println();
        FCSr, FCSc);
        M5.Lcd.printf("Erreur de Trame, FCSr=%d FCSc=%d",
        FCSr, FCSc);
        Serial.println();
        M5.Lcd.println();
        state=E0;
    }

```

Par la suite, on vérifie si les FCS reçu et calculé sont les mêmes, si c'est le cas, on passe à l'état E2 sinon on indique l'erreur et on ignore la trame.

```

    }
    else{
        state=E0;
    }
}
break;

```

Si la trame n'est pas de type DATA, alors on l'ignore.

```

case E2:
    if (RxSeq != rxbuf[CHAMP_NUM_SEQ]) {
        RxSeq = rxbuf[CHAMP_NUM_SEQ];
        M5.Lcd.printf("Num Seq : [%d] \n", RxSeq);
        M5.Lcd.printf("DATA de %d octets :", rxbuflen);
        M5.Lcd.println();
        for (int j = 0; j < rxbuflen; j++) {
            M5.Lcd.printf("%02x|", rxbuf[j]);
        }
        M5.Lcd.println();
    }
    else {
        M5.Lcd.printf("Duplication\n");
    }
    state = E3;
    break;

```

Pour l'état E2, on vérifie s'il y a eu duplication, si c'est le cas, on ignore la trame, sinon on affiche le contenu.

```

case E3:
    Serial.println("E3 : Envoi ACK");
    M5.Lcd.println("E3 : Envoi ACK");
    txbuf[CHAMP_TYPE] = TYPE_ACK;
    txbuf[CHAMP_NUM_SEQ] = rxbuf[CHAMP_NUM_SEQ];
    //Ajout du FCS du ACK
    txbuf[CHAMP_FCS_ACK] = txbuf[CHAMP_TYPE] ^
txbuf[CHAMP_NUM_SEQ];
    rf95.send(txbuf, 3);
    rf95.waitPacketSent();
    state = E0;
    break;

```

Dans le cas E3, on formate la trame d'acquittement avec un TYPE_ACK, un numéro de séquence et un FCS sur les deux premiers champs.

e) Jeux d'essai

(1) Erreur à l'envoi

Émetteur	Récepteur
EMISSION 8 01 08 FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF 09 ACK_RECUE [8] EMISSION 9 01 09 FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF 08 ACK_RECUE [9] EMISSION 10 01 0A FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF 0B Erreur E5 Nouvelle tentative n 1 EMISSION 10 01 0A FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF 0B ACK_RECUE [10] EMISSION 11	E0 : Attente Reception Num Seq : [8] DATA de 21 octets : 01 08 ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff 09 E3 : Envoi ACK E0 : Attente Reception Num Seq : [9] DATA de 21 octets : 01 09 ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff 08 E3 : Envoi ACK E0 : Attente Reception Erreur de Trame, FCSr=11 FCSs=244 seq=10 E0 : Attente Reception Num Seq : [10] DATA de 21 octets : 01 0a ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff 0b E3 : Envoi ACK E0 : Attente Reception

Nous pouvons voir qu'il y a une erreur sur l'envoi de la trame [10], le récepteur détecte cette erreur et ignore la trame. Le CDG de l'émetteur pour la trame [10] expire alors et l'émetteur envoie la trame à nouveau qui, cette fois-ci, arrive sans erreur et est acquittée par le récepteur.

(2) Erreur à l'acquittement

Émetteur	Récepteur
EMISSION 11 01 0B FF 0A ACK_RECUC [11] EMISSION 12 01 0C FF 0D E5 Nouvelle tentative n 1 EMISSION 12 01 0C FF 0D ACK_RECUC [12] EMISSION 13 01 0D FF 0C ACK_RECUC [13]	Num Seq : [11] DATA de 21 octets : 01 0b ff 0a E3 : Envoi ACK E0 : Attente Reception Num Seq : [12] DATA de 21 octets : 01 0c ff 0d E3 : Envoi ACK E0 : Attente Reception Duplication E3 : Envoi ACK E0 : Attente Reception Num Seq : [13] DATA de 21 octets : 01 0d ff 0c E3 : Envoi ACK

Ici, le premier acquittement de la trame 12 arrive erroné, il est donc ignoré par l'émetteur et le chien de garde expire.

L'émetteur envoie alors à nouveau la trame 12. Le récepteur détecte une duplication, ignore la trame et renvoie l'acquittement qui arrive cette fois-ci sans erreur.

f) Démo

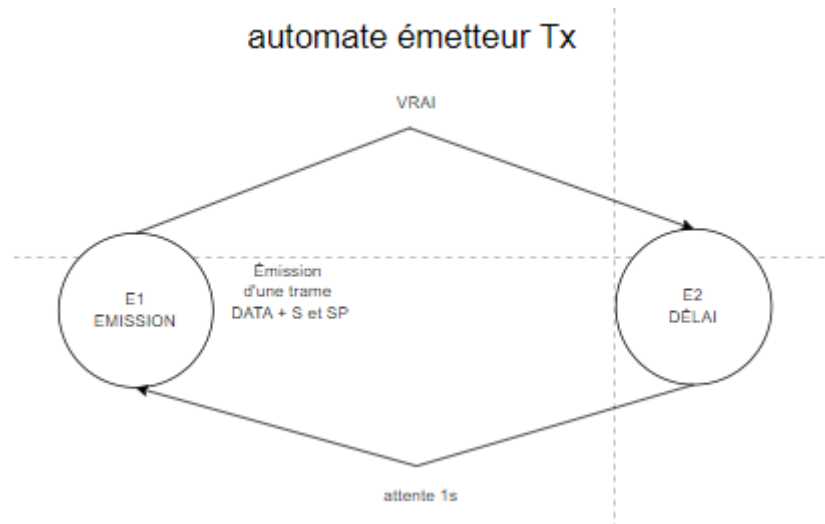
(1) Fonctionnement normal

Émetteur	Récepteur
EMISSION 0 01 00 FF 01 ACK_RECUE EMISSION 1 01 01 FF 00 ACK_RECUE EMISSION 2 01 02 FF 03 ACK_RECUE EMISSION 3 01 03 FF 02 ACK_RECUE	Num Seq : [0] DATA de 21 octets : 01 00 ff 01 E3 : Envoi ACK E0 : Attente Reception Num Seq : [1] DATA de 21 octets : 01 01 ff 00 E3 : Envoi ACK E0 : Attente Reception Num Seq : [2] DATA de 21 octets : 01 02 ff 03 E3 : Envoi ACK E0 : Attente Reception Num Seq : [3] DATA de 21 octets : 01 03 ff 02 E3 : Envoi ACK

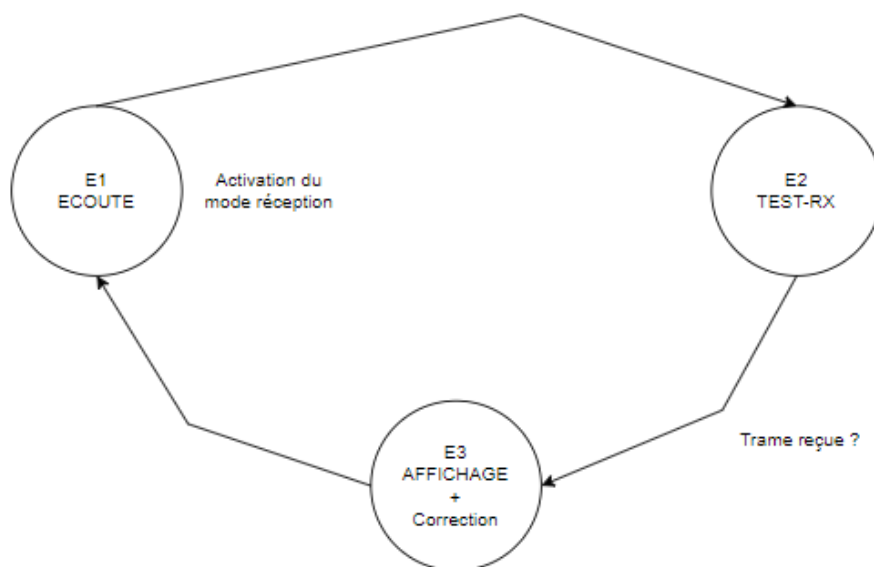
En fonctionnement normal, le récepteur acquitte bien les messages de l'émetteur.

3. Correction d'erreur

a) Automates



automate récepteur Rx



b) Code source

(1) Code émetteur

Dans cette partie, le code de l'émetteur est similaire à celui de la capsule 1. Nous allons uniquement nous attarder sur les différences.

```
//Variables pour la correction d'erreur
uint16_t S, SP;

//Constantes pour les champs de la trame
#define CHAMP_MIN_PAYLOAD 0
#define CHAMP_MAX_PAYLOAD 19
#define CHAMP_MIN_S 20
#define CHAMP_MIN_SP 22
#define LONGUEUR_TRAME 24
```

Pour l'émetteur, nous utilisons les variables S et SP afin de calculer les sommes et sommes pondérées.

Puis nous utilisons des constantes pour les champs de la trame.

```
case EMISSION: //Code source à l'état d'émission
    Serial.println("EMISSION"); //Trame de 20 octets de payload
    et 4 octets RS
    M5.Lcd.println("EMISSION");
    S=0;SP=0;

    for(int i=CHAMP_MIN_PAYLOAD;i<=CHAMP_MAX_PAYLOAD; i++){
        txbuf[i] = 255;
        S+=txbuf[i];
        SP+=txbuf[i]*(i+1);
        Serial.printf("|%02X", txbuf[i]);
        M5.Lcd.printf("|%02X", txbuf[i]);
    }
```

À l'état d'émission, on remplit la payload de la trame par le maximum.
En parallèle, on calcule les sommes et sommes pondérées.

```
//Injection des 8 bits de poids faible
//ET logique avec masque (0000 0000 1111 1111)
txbuf[CHAMP_MIN_S] = S & 0x00FF;

//Injection des 8 bits de poids fort
//ET logique avec masque (1111 1111 0000 0000)
//Puis décalage de 8 bits vers la droite pour pouvoir les
coder
//sur un octet
txbuf[CHAMP_MIN_S+1] = (S & 0xFF00) >> 8;

//Même principe que pour S
txbuf[CHAMP_MIN_SP] = SP & 0x00FF;
txbuf[CHAMP_MIN_SP+1] = (SP & 0xFF00) >> 8;
```

Comme S et SP sont codés sur 2 octets, il est nécessaire de les couper en deux pour les placer dans les deux champs de redondance.

```
rf95.send(txbuf, LONGUEUR_TRAME); //Envoi des données
rf95.waitPacketSent();

state=DELAI;
break;
```

Par la suite, on envoie la trame.

(2) Code récepteur

De la même manière, le code récepteur est très proche de celui de la capsule 1.

```
//Variables pour la correction d'erreur
int erreur, rang;
uint16_t Sr, SPr, Sc, SPc; //Sommes et Sommes pondérées calculées
et reçues
```

Nous utilisons les variables de rang et d'erreur ainsi que les sommes et sommes pondérées afin de corriger les éventuelles erreurs de la trame.

```

case AFFICHAGE:
    if(rf95.recv(rxbuf, &rxbuflen)){ //récupération de la trame
reçue dans rxbuf
        rxFrames+=1;
        if(rxbuflen != LONGUEUR_TRAME){
            Serial.printf("erreur de longueur de trame : %d !",
rxbuflen);
            M5.Lcd.printf("erreur de longueur de trame : %d !",
rxbuflen);
            Serial.println();
            M5.Lcd.println();
        }
    }

```

Dans l'état affichage, on extrait les données de la trame reçue puis on vérifie si la trame correspond à la bonne longueur normalisée. Si ce n'est pas bon, on affiche l'erreur et on ignore la trame.

```

else{
    Sr = (rxbuf[CHAMP_MIN_S] & 0x00FF) +
((rxbuf[CHAMP_MIN_S+1] & 0x00FF) << 8);
    SPr = (rxbuf[CHAMP_MIN_SP] & 0x00FF) +
((rxbuf[CHAMP_MIN_SP+1] & 0x00FF) << 8);
    Sc=0;
    SPc=0;
}

```

Dans le cas où la longueur correspond, alors on extrait la somme et la somme pondérée de la trame reçue.

```

for(int i=CHAMP_MIN_PAYLOAD; i<=CHAMP_MAX_PAYLOAD ;
i++){
    Sc=Sc+rxbuf[i];
    SPc=SPc+rxbuf[i]*(i+1);
}
Serial.printf("Sr=%d SPr=%d Sc=%d SPc=%d ", Sr, SPr,
Sc, SPc);
M5.Lcd.printf("Sr=%d SPr=%d Sc=%d SPc=%d ", Sr, SPr,
Sc, SPc);
Serial.println();
M5.Lcd.println();

```

Par la suite on calcule la somme et la somme pondérée sur la payload de la trame reçue.

```

    if(Sr != Sc){
        M5.Lcd.println("Une correction est utile ! ");
        erreur=Sc-Sr;
        M5.Lcd.printf("Valeur erreur = %d", erreur);
        M5.Lcd.println();

        rang=(SPc-SPr)/erreur;
        M5.Lcd.printf("Rang erreur = %d", rang);
        M5.Lcd.println();
    }

```

Si la somme reçue et calculée sont différentes, alors on calcule l'erreur et le rang de l'erreur.

```

        if(rang!=0){
            M5.Lcd.printf("Correction de %02X en %02X",
rxbuf[rang-1], rxbuf[rang-1]-erreur);
            M5.Lcd.println();
        }
        else{
            M5.Lcd.println("Erreur sur la redondance et non sur
les DATA utiles");
        }
    }
}

```

Si le rang est différent de 0 alors on corrige l'erreur. Sinon elle est sur la redondance.

```

        else{
            if(SPr != SPc){
                M5.Lcd.println("Erreur sur la redondance et non sur
les DATA utiles");
            }
        }
    }
}

state=ECOUTE;
break;

```

Dans le cas où ce sont les sommes pondérées qui sont différentes, alors l'erreur est sur la redondance.

c) Jeux de tests

(1) Erreur sur la longueur de la trame

Émetteur	Récepteur
===Emetteur CAP3 Corr=== Mise en place reussie !!! EMISSION FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF EC 13 2E D1 DELAI	ECOUTE TEST-RX [1] 23 bytes: ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ec 13 2 e erreur de longueur de trame : 23 ! ECOUTE TEST-RX

Le récepteur détecte l'erreur sur la longueur et ignore la trame.

(2) Erreur sur un octet

Émetteur	Récepteur
===Emetteur CAP3 Corr=== Mise en place reussie !!! EMISSION FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF EC 13 2E D1 DELAI	ECOUTE TEST-RX [1] 24 bytes: ff ff ff ff ff ff ff ff ff ff ff ff ff ff 15 ff ff ec 13 2e d1 Sr=5100 SPr=53550 Sc=4866 SPc=49572 Une correction est utile ! Valeur erreur = -234 Rang erreur = 17 Correction de 15 en FF

Le récepteur corrige l'erreur sur l'octet de rang 17.

(3) Erreur sur plusieurs octets

Émetteur	Récepteur
<pre> ===Emetteur CAP3 Corr=== Mise en place reussie !!! EMISSION FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF EC 13 2E D1 DELA </pre>	<pre> ECOUTE TEST-RX [4] 24 bytes: ff ff 08 ff ff ff ff ff ff ff ff ff ff ff 15 ff ff ff ec 13 2e d1 Sr=5100 SPr=53550 Sc=4619 SPc=48831 Une correction est utile ! Valeur erreur = -481 Rang erreur = 9 Correction de FF en 2E0 </pre>

Avec des erreurs sur plusieurs octets, le récepteur n'est pas en mesure de corriger correctement le code.

d) Démo

(1) Fonctionnement normal

Émetteur	Récepteur
<pre> EMISSION FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF EC 13 2E D1 DELA EMISSION FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF EC 13 2E D1 DELA </pre>	<pre> ECOUTE TEST-RX [1] 24 bytes: ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ec 13 2 e d1 Sr=5100 SPr=53550 Sc=5100 SPc=53550 ECOUTE TEST-RX [2] 24 bytes: ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ec 13 2 e d1 Sr=5100 SPr=53550 Sc=5100 SPc=53550 </pre>

En fonctionnement normal, le récepteur ne détecte pas l'erreur.

4. Compétences acquises

À l'issue de cette capsule, nous sommes capables de gérer les erreurs de transmission entre émetteurs. Nous sommes capables de détecter les erreurs et de les corriger à l'aide d'un code correcteur. Nous sommes également capables de comprendre le fonctionnement et le principe des codes correcteurs d'erreurs.

D. Capsule 4 : Couche MAC en accès aléatoire

1. Présentation des objectifs de l'exercice

Le but de cet exercice est de mettre en place un protocole d'accès au médium. Ce protocole permet de gérer les perturbations générées par plusieurs émetteurs vers un seul récepteur. Dans cet exercice, nous allons utiliser le protocole ALOHA.

2. Format des trames

Nous devons prévoir un format des trames qui va pouvoir prendre en compte les informations de contrôle nécessaire à la couche MAC dans une topologie multi-point.

Trame de données:

Tout d'abord, il faut faire porter dans les deux trames un champ d'adresse source afin d'indiquer leur identité unique pour différencier les deux émetteurs. Il est important aussi de faire porter dans chaque trame un champ d'adresse destination (1 seul récepteur R0 est présent donc ce n'est pas obligatoire dans ce cas là mais il permet d'être compatible plus tard sur une topologie multi-puits) afin d'identifier le destinataire de chaque trame.

@S	@D	Type DATA	NumSeq	Données utiles
----	----	-----------	--------	----------------

Type DATA : il va permettre de différencier les trames de données et celles d'acquittement.

NumSeq Il est incrémenté par chaque noeud émetteur

Données utiles = payload

On aurait pu prévoir un champ longueur pour pouvoir varier la taille des données.

Trame d'acquittement:

@S	@D	Type ACK	NumSeq
----	----	----------	--------

@S = adresse 0 de R0

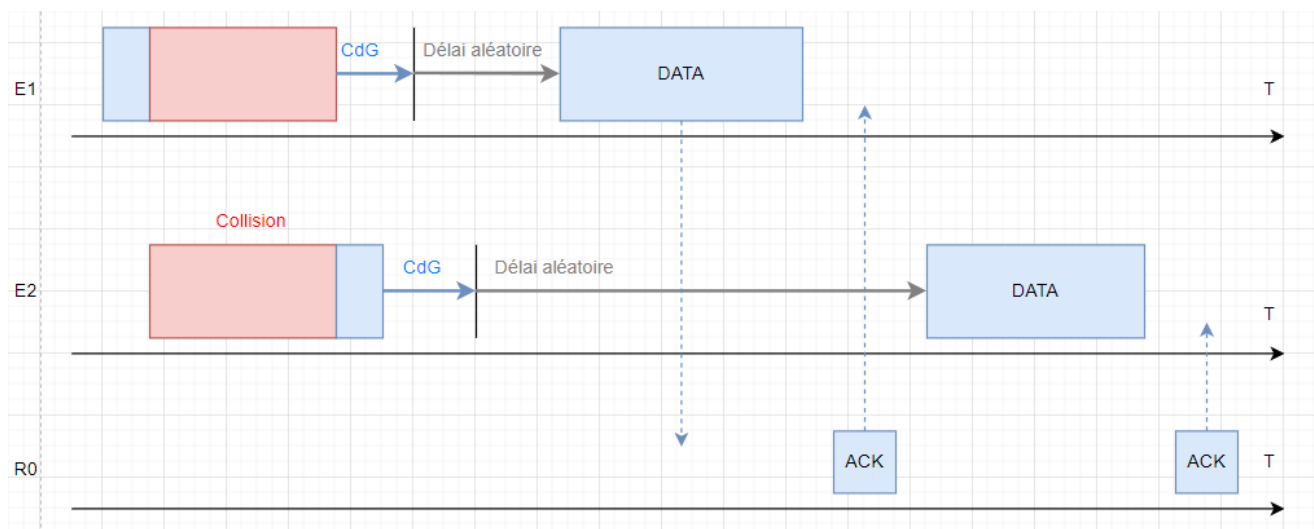
@D = adresse source de la précédente trame de data

NumSeq = celui de la data acquitté

3. Chronographie protocolaire

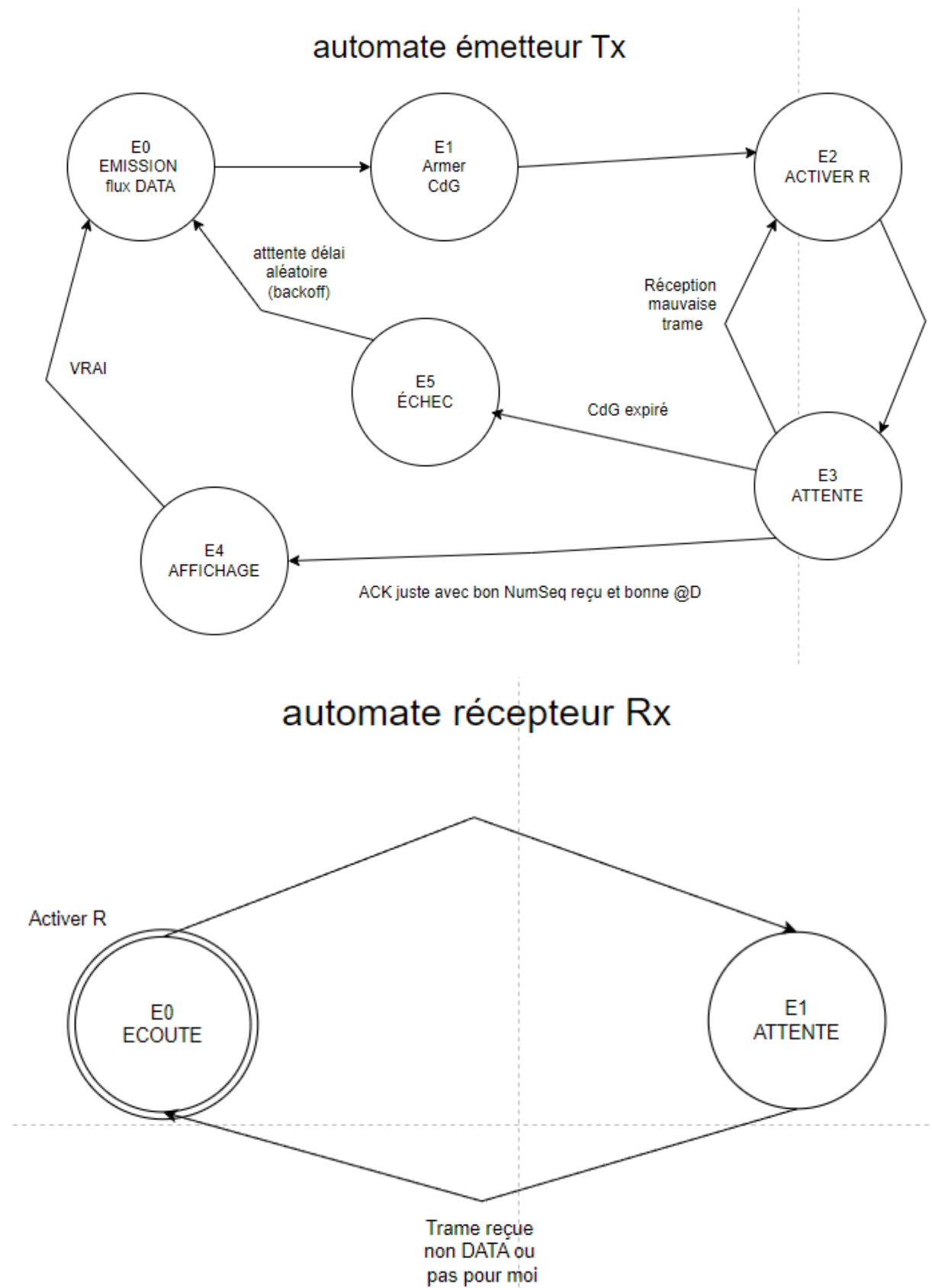
Le fonctionnement d'une couche MAC est présenté par un chronogramme protocolaire. On va pouvoir visualiser facilement les différentes étapes du protocoles entre les machines concurrentes à l'accès au médium (collisions, tentatives d'accès, ré-émissions, délais d'attente).

3 noeuds : 2 émetteurs et 1 récepteur



- Si expiration délais aléatoire des concurrents en même temps : nouvelle collision
- Nouvelle tentative (crédit de répétition : non déterministe)
- Peu de trafic : bonne efficacité
- Fort trafic > beaucoup de collision > nombreuses tentatives > effondrement

4. Automates



5. Code source

Seul le code source émetteur est modifié afin de répondre aux exigences de cette partie.

```
uint8_t NewFrame, EIT;
uint32_t attente;
uint32_t backoff;
```

Nous utilisons une variable NewFrame qui permet d'indiquer l'émission de la prochaine trame. La variable EIT correspond à un temps aléatoire tiré avant l'émission de la trame suivante.

Enfin, la variable backoff correspond à un temps aléatoire tiré après non retour d'acquiescement.

```
//Constante d'adresse
#define LOCAL_ADDR 1
#define SINK_ADDR 0
```

Pour les constantes, nous déclarons l'adresse du nœud et celle du puits vers lequel envoyer l'information.

```
void setup() {
    . . . . .
    NewFrame=1; //Initialisation drapeau
    randomSeed(analogRead(30)); //Initialisation sur la broche 30
    delay(10000);
}
```

Dans le setup, nous initialisons le module randomSeed permettant de générer des nombres aléatoires.

Ici, nous utilisons la broche 30 (libre).

```
case E0: //Code source à l'état d'émission

    if(NewFrame==1){
        EIT=random(80,120)*random(80,123);
        delay(EIT); //attente pour l'envoi de la nouvelle trame
        . . . . .
    }

    //Formatage des champs de la trame et envoi
    . . . . .
    break;
```

Si la valeur NewFrame est à un alors on tire un temps d'attente aléatoire avant d'envoyer la trame.

```
case E3:
    . . . . .
    if(rf95.recv(rxbuf, &rxbuflen)){
        if((rxbuf[F_TYPE]==TYPE_ACK) &&
(rxbuf[F_NUM_SEQ]==TxSeq) && (rxbuf[F_D_ADDR]==LOCAL_ADDR)){
            //Reception d'un ACK avec le bon num de séquence et
la bonne adresse de destination
        }
        . . . . .
    }
```

Si on reçoit une trame, on vérifie s'il s'agit d'un acquittement, du numéro de séquence attendu et l'adresse de destination de la trame correspond à celle du nœud.

```
case E5:
    if (credit==0){
        . . . . .
        NewFrame=1;
    }
    else{
        state=E0;
        NewFrame=0; //Toujours la même trame
        //Backoff qui evolue en fonction du nombre de tentatives
        backoff=pow(random(1,10),MAX_CREDIT-credit);
        delay(backoff); //Attente aléatoire ALOHA
        . . . . .
    }
    break;
```

Si le chien de garde expire, on génère un backup qui varie exponentiellement en fonction du nombre de tentatives écoulées.

6. Démo

Émetteur (1)	Émetteur (2)	Récepteur
====Emetteur CAP4==== Mise en place reussie !!! Boucle principale. EIT : 88 EMISSION 0 ===CDG expire=== Collision ? Nouvelle tentative n 1 backoff : [[1]] EMISSION 0 ===CDG expire=== Collision ? Nouvelle tentative n 2 backoff : [[16]] EMISSION 0 ===CDG expire=== Collision ? Nouvelle tentative n 3 backoff : [[343]] EMISSION 0 ===CDG expire=== Collision ? Nouvelle tentative n 4 backoff : [[81]] EMISSION 0 ===CDG expire=== EHEC EIT : 32 EMISSION 1 ACK_RECUI [1] EIT : 217 EMISSION 2 ACK_RECUI [2] EIT : 208 EMISSION 3 ACK_RECUI [3] EIT : 19 EMISSION 4 ACK_RECUI [4]	====Emetteur CAP4==== Mise en place reussie !!! Boucle principale. EIT : 59 EMISSION 0 ===CDG expire=== Collision ? Nouvelle tentative n 1 backoff : [[6]] EMISSION 0 ===CDG expire=== Collision ? Nouvelle tentative n 2 backoff : [[49]] EMISSION 0 ===CDG expire=== Collision ? Nouvelle tentative n 3 backoff : [[125]] EMISSION 0 ACK_RECUI [0] EMISSION 1 ===CDG expire=== Collision ? Nouvelle tentative n 1 backoff : [[6]] EMISSION 1 ACK_RECUI [1] EMISSION 2 ===CDG expire=== Collision ? Nouvelle tentative n 1 backoff : [[4]] EMISSION 2	====Recepteur CAP4==== Mise en place reussie !!! [0] DATA de 10 octets : 01 00 01 00 b8 dc ef 6e 63 c b ===Envoi ACK=== ===Duplication=== ===Envoi ACK=== ===Duplication=== ===Envoi ACK=== ===Duplication=== ===Envoi ACK=== [1] DATA de 10 octets : 02 00 01 01 8b 99 bd 69 94 9a ===Envoi ACK=== ===Duplication=== ===Envoi ACK=== [0] DATA de 10 octets : 01 00 01 00 79 ec 15 48 b6 21 ===Envoi ACK=== [1] DATA de 10 octets : 01 00 01 01 c1 5c 8d 9d ae 28 ===Envoi ACK=== [2] DATA de 10 octets : 01 00 01 02 73 33 98 e0 eb 08 ===Envoi ACK=== [3] DATA de 10 octets : 01 00 01 03 ce 7d 1a 18 36 2e ===Envoi ACK=== [4] DATA de 10 octets : 01 00 01 04 ae 77 b6 6d 39 56 ===Envoi ACK===

Nous pouvons voir que de multiples collisions apparaissent. Au fur et à mesure que les tentatives se multiplient, le backoff augmente rapidement.

Pour l'émetteur 1, la transmission échoue après les cinq tentatives puis réussit pour les trames suivantes.

Pour l'émetteur 2, la transmission réussit après 4 tentatives du fait du backoff important de l'émetteur 2 (343 contre 125).

7. Compétences acquises

À l'issue de cet exercice, nous sommes capables de gérer les perturbations liées aux transmissions de plusieurs nœuds en même temps.

Cette méthode d'accès est simple à mettre en œuvre mais est très peu applicable avec un grand nombre de machines communicantes sur un même support.

II. Routage par inondation

A. Travail demandé

Dans cette partie de la SAÉ, nous allons mettre en place un routage par inondation afin de faire communiquer plusieurs nœuds entre eux.

Cette partie est divisée en 6 sprints lors desquels nous allons optimiser ce routage à l'aide de mécanismes tels que le TTL ou la mémorisation de paquets.

B. Sprint 1 : Création d'un en-tête de couche 3

1. Objectif de cette partie

L'objectif de cette première opération consiste à créer un en-tête de niveau 3 où seul figurera, pour l'instant, un champ d'adressage 16 bits. La fonction `wino.encodeUi16()` sera utilisée pour placer l'adresse multicast, sur 16 bits, dans les deux premiers octets du paquet, qui forment l'en-tête.

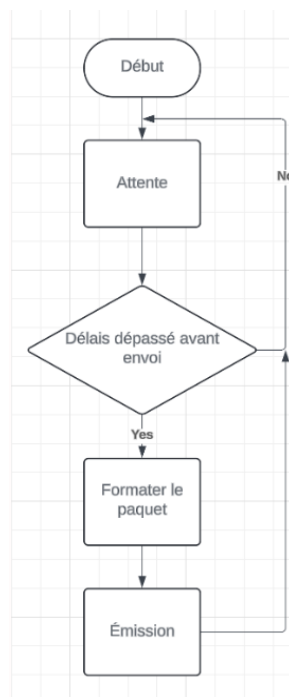
2. Format des paquets

@destination	Payload
--------------	---------

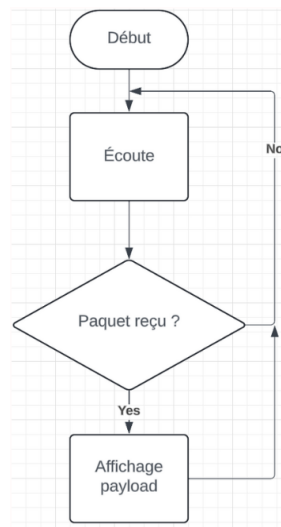
Un champ d'adresse de destination de deux octets et une payload qui varie.

3. Algorithmes

a) Émetteur



b) Récepteur



4. Tests réalisés

Émetteur	Récepteur
====Envoi Paquet==== 78 78 18 E4 F8 7F 84 6A 75 0B 00 89	=====Reception===== Noeud Source : 1 Adresse Multicast : 7878 Payload : 18 E4 F8 7F 84 6A 75 0B 00 89

Nous pouvons voir que la payload du paquet émis correspond à la payload du paquet reçu.

5. Avantages et limites

Les nœuds sont capables d'envoyer et de recevoir des paquets.

Il n'y a pas de réémission, si les nœuds sont hors de portée, ils ne peuvent pas communiquer entre eux sans passer par un nœud intermédiaire.

C. Sprint 2 : Routage et tempête de broadcast

1. Objectif de la partie traitée

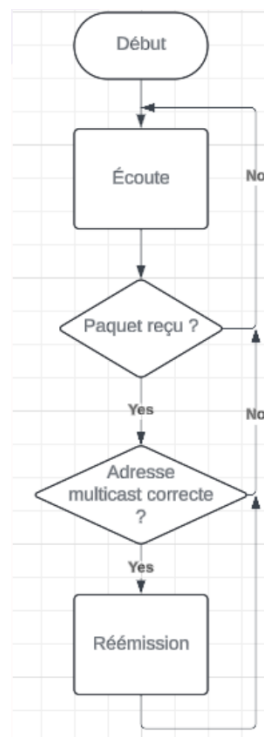
L'objectif de ce second sprint est de programmer une réémission du paquet par le routeur si l'adresse multicast correspond à celle du routeur.

2. Format des messages

@destination	Payload
--------------	---------

Même format que pour le sprint précédent.

3. Algorithmes conçus



4. Tests réalisés

Dans les tests suivants, nous partons du principe que les nœuds 1 et 2 sont hors de portée l'un de l'autre mais chacun à portée du nœud 2.

Émetteur (1)	Relais (2)	Récepteur (3)
<pre>====Envoi Paquet==== 78 78 EF FF 37 38 29 0E A7 E0 9A 52 ====Reception===== Noeud Source : 2 Adresse Multicast : 7878 Payload : EF FF 37 38 29 0E A7 E0 9A 52 ====Réémission Paquet===== ====Reception===== Noeud Source : 3 Adresse Multicast : 7878 Payload : EF FF 37 38 29 0E A7 E0 9A 52 ====Réémission Paquet===== ====Reception===== Noeud Source : 2 Adresse Multicast : 7878 Payload : EF FF 37 38 29 0E A7 E0 9A 52 ====Réémission Paquet===== ====Reception===== Noeud Source : 2 Adresse Multicast : 7878 Payload : EF FF 37 38 29 0E A7 E0 9A 52 ====Réémission</pre>	<pre>=====Reception===== Noeud Source : 1 Adresse Multicast : 7878 Payload : EF FF 37 38 29 0E A7 E0 9A 52 ====Réémission Paquet===== ====Reception===== Noeud Source : 3 Adresse Multicast : 7878 Payload : EF FF 37 38 29 0E A7 E0 9A 52 ====Réémission Paquet===== ====Reception===== Noeud Source : 3 Adresse Multicast : 7878 Payload : EF FF 37 38 29 0E A7 E0 9A 52 ====Réémission</pre>	<pre>=====Reception===== Noeud Source : 2 Adresse Multicast : 7878 Payload : EF FF 37 38 29 0E A7 E0 9A 52 ====Réémission Paquet===== ====Reception===== Noeud Source : 2 Adresse Multicast : 7878 Payload : EF FF 37 38 29 0E A7 E0 9A 52 ====Réémission Paquet===== ====Reception===== Noeud Source : 2 Adresse Multicast : 7878 Payload : EF FF 37 38 29 0E A7 E0 9A 52 ====Réémission</pre>

Nous pouvons voir dans ce test que le nœud récepteur est capable de recevoir le paquet de l'émetteur à l'aide du relais.

Nous pouvons également voir que le nœud relais échange indéfiniment des paquets avec les deux autres, nous pouvons voir ici un tempête de broadcast.

5. Avantages et limites de cette étape

Deux nœuds hors de portée sont capables de se transmettre des paquets en passant par un troisième nœud relais.

Cependant, nous pouvons voir une tempête de broadcast générée par des réémissions successives de paquets.

D. Sprint 3 : Création d'un TTL pour limiter la vie des paquets

1. Objectif de cette partie

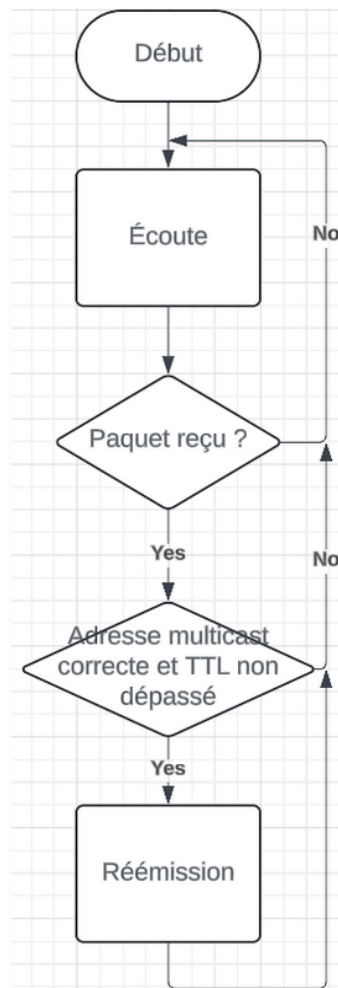
L'objectif de ce troisième sprint est de limiter la durée de vie des paquets sur le réseau afin d'empêcher la tempête de broadcast. Un TTL doit être placé dans l'en-tête de niveau 3 et incrémenté à chaque routage.

2. Format des paquets

@destination	TTL	Payload
--------------	-----	---------

Il s'agit du même format que pour le sprint précédent auquel on ajoute un champ TTL d'un octet entre le champ d'adresse et celui de la payload.

3. Algorithme



Le TTL est incrémenté avant le test.

4. Test réalisé

Émetteur (1)	Relais (2)	Récepteur (3)
<pre>====Envoi Paquet==== 78 78 00 4C 5C 0D B2 DA C8 9E 9E FE BF ====Reception===== Noeud Source : 2 Adresse Multicast : 7878 TTL : 1 Payload : 4C 5C 0D B2 DA C8 9E 9 E FE BF ====Réémission Paquet===== ====Reception===== Noeud Source : 2 Adresse Multicast : 7878 TTL : 3 Payload : 4C 5C 0D B2 DA C8 9E 9 E FE BF ====Réémission Paquet===== ====Reception===== Noeud Source : 2 Adresse Multicast : 7878 TTL : 5 Payload : 4C 5C 0D B2 DA C8 9E 9 E FE BF ====Réémission Paquet===== ====Reception===== Noeud Source : 2 Adresse Multicast : 7878 TTL : 7 Payload : 4C 5C 0D B2 DA C8 9E 9 E FE BF </pre>	<pre>====Reception===== Noeud Source : 1 Adresse Multicast : 7878 TTL : 0 Payload : 4C 5C 0D B2 DA C8 9E 9 E FE BF ====Réémission Paquet===== ====Reception===== Noeud Source : 1 Adresse Multicast : 7878 TTL : 2 Payload : 4C 5C 0D B2 DA C8 9E 9 E FE BF ====Réémission Paquet===== ====Reception===== Noeud Source : 1 Adresse Multicast : 7878 TTL : 4 Payload : 4C 5C 0D B2 DA C8 9E 9 E FE BF ====Réémission Paquet===== ====Reception===== Noeud Source : 1 Adresse Multicast : 7878 TTL : 6 Payload : 4C 5C 0D B2 DA C8 9E 9 E FE BF ====Réémission Paquet=====</pre>	<pre>====Reception===== Noeud Source : 2 Adresse Multicast : 7878 TTL : 1 Payload : 4C 5C 0D B2 DA C8 9E 9 E FE BF ====Réémission Paquet===== ====Reception===== Noeud Source : 2 Adresse Multicast : 7878 TTL : 3 Payload : 4C 5C 0D B2 DA C8 9E 9 E FE BF ====Réémission Paquet=====</pre>

Nous pouvons voir que la durée de vie du paquet est limitée par le TTL. Cependant, nous pouvons voir l'apparition d'une duplication de paquets dont les TTL sont indépendants.

5. Avantages et limites

Nous n'avons plus de tempête de broadcast du fait de la limitation de la durée de vie du paquet.

Cependant, s'il y a un grand nombre de nœuds, nous avons un risque de duplication des paquets dont les TTL sont indépendants.

E. Sprint 4 : Champ d'adresse source et de numéro de séquence

1. Objectif de cette partie

L'objectif de cette quatrième partie consiste à identifier le nœud source avec l'ajout d'un champ d'adressage de niveau 3. Nous ajoutons également un numéro de séquence.

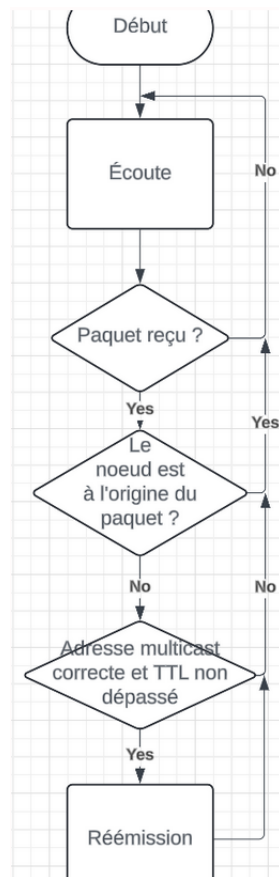
Les nœuds pourront alors identifier l'origine du paquet.

2. Format des paquets

@destination	TTL	@source	N° Séquence	Payload
--------------	-----	---------	-------------	---------

Même format que pour le sprint précédent mais avec l'ajout d'un champ d'adresse source sur deux octets et un numéro de séquence sur un octet entre le TTL et la payload.

3. Algorithme



4. Test réalisé

Émetteur (1)	Relais (2)	Récepteur (3)
<pre>====Envoi Paquet==== 78 78 00 01 00 00 E8 35 1 9 B3 02 FB D6 D3 C1 2C =====Reception===== Noeud Source : 1 Adresse Multicast : 7878 TTL : 1 Num Seq : 0 Payload : E8 35 19 B3 02 FB D6 D3 C1 2C === Paquet emis par le noeud === ====Envoi Paquet==== 78 78 00 01 00 02 7A 86 0 3 AA 29 AB 98 42 86 12 =====Reception===== Noeud Source : 1 Adresse Multicast : 7878 TTL : 1 Num Seq : 2 Payload : 7A 86 03 AA 29 AB 98 42 86 12 === Paquet emis par le noeud ===</pre>	<pre>=====Reception===== Noeud Source : 1 Adresse Multicast : 7878 TTL : 0 Num Seq : 0 Payload : E8 35 19 B3 02 FB D6 D3 C1 2C =====Reception===== Noeud Source : 1 Adresse Multicast : 7878 TTL : 2 Num Seq : 0 Payload : E8 35 19 B3 02 FB D6 D3 C1 2C =====Reception===== Noeud Source : 1 Adresse Multicast : 7878 TTL : 2 Num Seq : 2 Payload : 7A 86 03 AA 29 AB 98 42 86 12 </pre>	<pre>=====Reception===== Noeud Source : 1 Adresse Multicast : 7878 TTL : 1 Num Seq : 0 Payload : E8 35 19 B3 02 FB D6 D3 C1 2C =====Reception===== Noeud Source : 1 Adresse Multicast : 7878 TTL : 1 Num Seq : 2 Payload : 7A 86 03 AA 29 AB 98 42 86 12 </pre>

Nous pouvons voir que le nœud émetteur est capable d'identifier qu'il est à l'origine du paquet et qu'il ne doit pas le réémettre en conséquence.

5. Avantages et limites

Il est possible d'identifier le nœud source du paquet ainsi que le numéro de séquence.

Le nœud peut ignorer la réémission si le paquet provient de ce dernier.

Mais la limitation du nombre de paquets ne se fait toujours qu'au travers du TTL.

F. Sprint 5 : Mémorisation de paquet

1. Objectif de cette partie

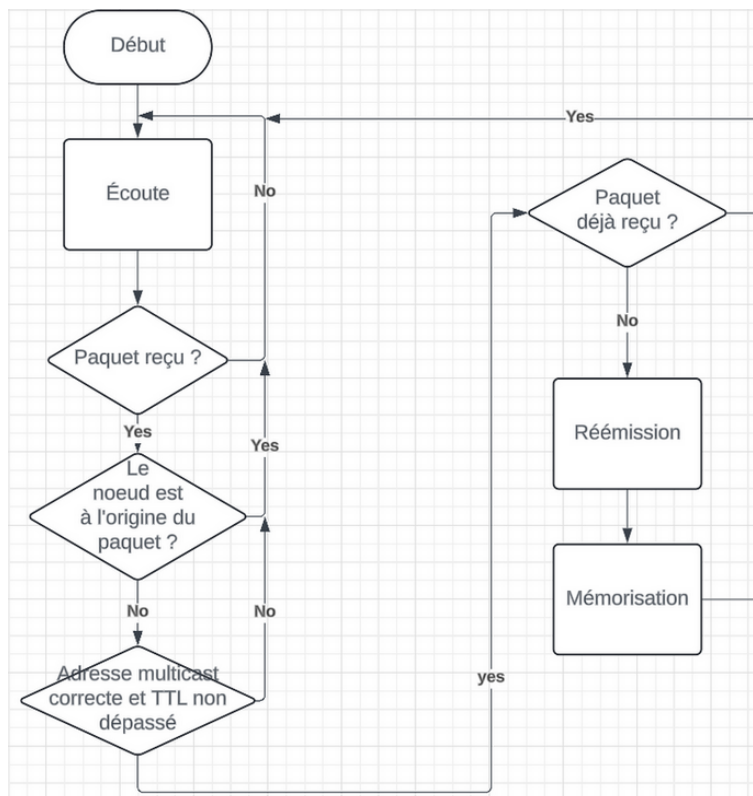
Pour éviter la répétition du routage de paquet, nous mémorisons maintenant l'adresse source et le numéro de séquence du paquet reçu. Lorsque nous recevons un paquet, nous regardons l'adresse source et le numéro de séquence afin de voir s'ils correspondent à celui du paquet émis précédemment. Si c'est le cas, il n'y a pas de réémission sinon on réémet.

2. Format des paquets

@destination	TTL	@source	N° Séquence	Payload
--------------	-----	---------	-------------	---------

Même format que pour le sprint précédent.

3. Algorithme



4. Test réalisé

Émetteur (1)	Relais (2)	Récepteur (3)
<pre>====Envoi Paquet==== 78 78 00 01 00 00 F2 8F 0 E 75 82 EC 06 48 4F 1A =====Reception===== Noeud Source : 1 Adresse Multicast : 7878 TTL : 1 Num Seq : 0 Payload : F2 8F 0E 75 82 EC 06 48 4F 1A ===Paquet émis par le noeud===</pre>	<pre>=====Reception===== Noeud Source : 1 Adresse Multicast : 7878 TTL : 0 Num Seq : 0 Payload : F2 8F 0E 75 82 EC 06 48 4F 1A =====Réémission Paquet===== =====Reception===== Noeud Source : 1 Adresse Multicast : 7878 TTL : 2 Num Seq : 0 Payload : F2 8F 0E 75 82 EC 06 48 4F 1A ===Déjà Reçu===</pre>	<pre>=====Reception===== Noeud Source : 1 Adresse Multicast : 7878 TTL : 1 Num Seq : 0 Payload : F2 8F E 75 82 EC 06 48 4 F 1A =====Réémission Paquet=====</pre>

Nous pouvons voir que le relais est capable de mémoriser le paquet et de ne pas le réémettre en conséquence.

5. Avantages et limitations

Permet aux nœuds de mémoriser le dernier paquet et de ne pas le réémettre s'il revient.

Permet de sauvegarder uniquement un seul paquet à la fois.

Si un paquet est réémis et qu'un autre est reçu entre-temps, la mémorisation change (si le premier paquet revient, il sera réémis).

G.Sprint 6 : Mémorisation de N paquet

1. Objectif de cette partie

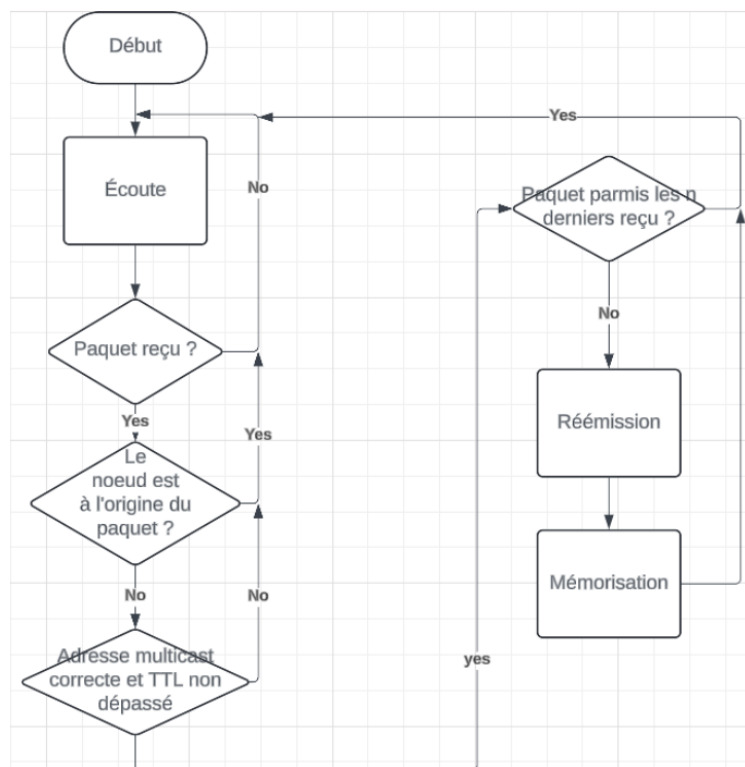
Nous suivons le même principe que pour le sprint 5 sauf que nous mémorisons n couples d'adresse et de numéro de séquence.

2. Format des paquets

@destination	TTL	@source	N° Séquence	Payload
--------------	-----	---------	-------------	---------

Même format que pour le sprint précédent.

3. Algorithme



4. Code source du projet

a) Structure et configuration M5

```
#include <Arduino.h>
#include <SPI.h>
#include "SimpleWiNo.h"
#include <M5Stack.h>
```

Nous utilisons principalement les librairies SimpleWino.h afin de gérer la couche 2 du projet ainsi que la librairie M5Stack.h permettant de gérer l’affichage sur l’écran LCD.

```
//Objet Wino
SimpleWiNo wino;

//Constantes de configuration des noeuds
#define ADRESSE_CLUSTER 0x4A89
#define ADRESSE_NOEUD 0x0001
#define CANAL 7
```

Par la suite, il convient d’instancier l’objet wino permettant d’envoyer et de recevoir les trames.

Nous définissons également les constantes correspondant à la configuration du nœud (adresse du cluster, adresse du nœud dans le cluster, et canal utilisé).

```

void setup() {

    //Démarrage M5
    M5.begin();
    M5.Power.begin();
    delay(3000);

    //Demarage wino
    wino.init();
    //Adresse cluster
    wino.set(NODE_PANID, ADRESSE_CLUSTER);
    //Adresse noeud
    wino.set(NODE_SHORT_ADDRESS, ADRESSE_NOEUD);
    //Fréquence de travail
    wino.set(NODE_CHANNEL, CANAL);
    . . . . .
}

```

Dans la fonction setup, nous démarrons le M5 avec M5.begin() et M5.Power.begin(). Puis nous allumons l'objet wino et nous paramétrons les informations du noeud avec wino.set().

```

void loop() {
    wino.process();
    . . . . .
}

```

Dans la fonction loop, nous utilisons wino.process() afin de déléguer les ressources à la couche 2 pour chaque itération de loop.

b) Gestion des champs de la trame

```
//Propriétés du paquet
#define ADRESSE_MULTICAST 0x7878 //Adresse multicast du noeud
#define NB_PAYLOAD 10 //Nombre de données de payload
#define TTL_MAX 7 //Time To Live avant expiration
```

Nous utilisons des constantes pour définir l'adresse multicast du cluster, le nombre d'octets de payload et le TTL maximum du paquet.

```
//Indices des champs du paquet
#define CHAMP_ADRESSE_DST 0 //Position de l'adresse de destination
#define CHAMP_TTL 2 //Position du TTL dans le paquet
#define CHAMP_ADRESSE_SRC 3 //Position du champ d'adressage
#define CHAMP_NUM_SEQ 5 //Position du numéro de séquence
#define CHAMP_MIN_PAYLOAD 6 //Position du premier octet de la payload
```

Pour gérer les différents champs du paquet et afin de rendre le code plus compréhensible, nous utilisons des constantes d'indices correspondant à ceux du buffer représentant le paquet.

c) Gestion de l'envoi des paquets

```
#define DELAI_ENVOI 60000

//Variables pour le paquet

//Émission
uint8_t txbuf[40]; //Buffer d'émission
uint8_t packetLenTX; //Longueur du paquet émis
uint8_t numSeq=0; //Numéro de séquence incrémenté pour l'envoi
uint32_t attente = millis()+DELAI_ENVOI; //Condition pour
déclencher un envoi toute les
// DELAI_ENVOI secondes
```

Pour la gestion de l'envoi, nous utilisons une constante DELAI_ENVOI qui permet de définir un délai non bloquant entre chaque envoi de paquet.

Nous utilisons un buffer d'émission, une variable qui calcule la longueur du paquet, un numéro de séquence et la variable "attente" qui permet de calculer le moment à partir duquel il faudra envoyer le paquet.

```
//Procédures pour l'envoi

//Procédure de formatage de l'en tête et de la payload
//du paquet en émission
void formaterPaquet(uint8_t* longueur, uint8_t donnees[], uint8_t
nSeq) {
    *longueur=0;
    //Ajout En-tete
    //Adresse Destination
    wino.encodeUi16(ADRESSE_MULTICAST,
&donnees[CHAMP_ADRESSE_DST]);
    *longueur +=2;
    //Ajout du TTL
    donnees[CHAMP_TTL]=0;
    *longueur+=1;
    //Adresse Source
    wino.encodeUi16(ADRESSE_NOEUD, &donnees[CHAMP_ADRESSE_SRC]);
    *longueur +=2;
```

```

//Numéro de séquence
donnees[CHAMP_NUM_SEQ]=nSeq;
*longueur +=1;
//Ajout de la payload
for(int i=0; i<NB_PAYLOAD; i++){
    donnees[*longueur]=random(0,256);
    *longueur+=1;
}
}

```

Nous utilisons la procédure “formaterPaquet” qui permet de construire le paquet avec le champ destination, le champ ttl, le champ source, le numéro de séquence, ainsi que la payload (générée aléatoirement). Elle permet également de calculer la longueur du paquet indispensable pour l’émission.

```

void setup() {
    . . . . .
    //Générateur de payload aléatoire
    randomSeed(analogRead(30));
    . . . . .
}

```

Dans le setup, nous initialisons le module permettant de générer des nombres aléatoires pour la payload.

```

void loop() {
    wino.process();

    //Envoi d'un paquet toutes les DELAI_ENVOI secondes
    if(millis() > attente){
        M5.Lcd.println("===Envoi Paquet===");
        Serial.println("===Envoi Paquet===");
        formaterPaquet(&packetLenTX, txbuf, numSeq);
        wino.send(0xFFFF, txbuf, packetLenTX);
        numSeq+=1;
        attente=millis()+DELAIE_ENVOI;
    }
}

```

Dans le loop, nous vérifions si le délai non bloquant est dépassé, si c'est le cas alors on formate le paquet et on l'envoie à l'adresse de broadcast de couche 2 (0xFFFF). Par la suite, on incrémente le numéro de séquence et on recalcule un délai non bloquant pour le prochain envoi.

d) Gestion de la réception des paquets

```

//Réception
uint8_t rxbuf[40]; //Buffer de réception
uint8_t packetLenRX; //Longueur du paquet reçu
uint16_t adresseMacSource; //Adresse de couche 2 du noeud source
uint16_t adresseIpSource; //Adresse de couche 3 du noeud source
uint16_t adresseMulticast; //Adresse multicast reçue

```

Pour la réception des paquets, nous utilisons un buffer de réception, une variable de longueur de paquet, une variable pour stocker l'adresse de couche 2 de la source du paquet et celle de couche 3. Ainsi qu'une variable permettant d'extraire l'adresse multicast destination contenue dans le paquet afin d'identifier le réseau spécifique aux différents nœuds.

```
//Procédures pour la réception

//Procédure d'affichage de la payload reçue
//À partir de l'indice de début du paquet uniquement
void afficherPayload(uint8_t indiceDebut, uint8_t longueur,
uint8_t donnees[]){

    M5.Lcd.printf("Payload : ");
    for(int i=indiceDebut; i<longueur; i++){
        M5.Lcd.printf("|%02X", donnees[i]);
    }
    M5.Lcd.println("|");
}
```

Lors de la réception d'un paquet, nous affichons la payload de ce dernier; nous utilisons alors une procédure prévue à cet effet.

```

void loop() {
    . . . . .
    //Test Reception
    if(wino.recv(&adresseMacSource, rxbuf, &packetLenRX)){
        //Affichage des éléments du paquet reçu
        M5.Lcd.println("====Reception====");
        //Affichage du noeud source
        adresseIpSource=wino.decodeUi16(&rxbuf[CHAMP_ADRESSE_SRC]);
        M5.Lcd.printf("MAC SRC : %X | ", adresseMacSource);
        M5.Lcd.printf("IP SRC : %X \n", adresseIpSource);
        //Affichage Adresse Multicast
        adresseMulticast=wino.decodeUi16(&rxbuf[CHAMP_ADRESSE_DST]);
        M5.Lcd.printf("Adresse Multicast : %X \n", adresseMulticast);
        //Affichage TTL
        M5.Lcd.printf("TTL : %d \n", rxbuf[CHAMP_TTL]);
        //Incrémentation TTL
        rxbuf[CHAMP_TTL]+=1;
        //Affichage du numéro de séquence
        M5.Lcd.printf("Num Seq : %d \n", rxbuf[CHAMP_NUM_SEQ]);
        //Affichage Payload
        afficherPayload(CHAMP_MIN_PAYLOAD, packetLenRX, rxbuf);
        //Test pour la réémission
        . . . . .
    }
}

```

Dans le cas d'une réception d'un paquet, on extrait les différents éléments des champs du tableau avant de les afficher (adresse mac source et ip source, adresse multicast, le TTL, le numéro de séquence et la payload).

Le TTL est incrémenté après son affichage.

e) Mémorisation des n derniers paquets

```
//Nombre de paquets maximums mémorisables
#define NB_PAQUET_MEM 10
```

On utilise une constante permettant de définir le nombre d'identifiants mémorisables par le programme.

```
//Structure pour l'identification
struct identifiant{
    uint16_t srcAddr;
    uint8_t sqn;
};
typedef struct identifiant IDENTIFIANT;
```

Nous utilisons une structure identifiant dont les attributs sont une adresse codée sur 16 bits et un numéro de séquence sur un octet.

Nous utilisons typedef permettant de définir les objets de type struct identifiant avec uniquement le type IDENTIFIANT.

```
IDENTIFIANT tabId[NB_PAQUET_MEM]; //Tableau d'identifiants
uint8_t indiceMemId=0; //Indice pour la mise à jour dans le
tableau
IDENTIFIANT idRecu; //Identifiant courant du paquet reçu
```

Nous définissons une variable d'indice qui va permettre de mettre à jour les identifiants dans le tableau, un IDENTIFIANT correspondant au couple adresse source numéro de séquence. Enfin, nous définissons un tableau qui va nous permettre de stocker les NB_PAQUET_MEM derniers identifiants.

```
//Initialisation du tableau d'identifiants
void initTabId(IDENTIFIANT tab[]){
    for(int i=0; i<NB_PAQUET_MEM; i++){
        tab[i].srcAddr=0xFFFF;
        tab[i].sgn=255;
    }
}
```

Nous utilisons une procédure qui va nous permettre d'initialiser les éléments du tableau avec le couple 0xFFFF; 255 (qui ne peut pas arriver théoriquement). Cette initialisation permet d'avoir des valeurs cohérentes à l'affichage.

```
void setup() {
    . . . . .
    //Initialisation du tableau de mémoire des identifiants
    initTabId(tabId);
}
```

Cette procédure d'initialisation du tableau est appelée dans le setup.

```
//Ajout d'un nouvel identifiant dans le tableau
//L'indice du tableau suit un modulo pour reboucler dans les
mises à jour
//Afin de ne garder que les derniers NB_PAQUET_MEM paquets
void ajoutTabId(IDENTIFIANT id, IDENTIFIANT tab[], uint8_t*
indice){
    tab[*indice]=id;
    *indice=(*indice+1)%NB_PAQUET_MEM;
}
```

Nous utilisons également une procédure qui permet d'ajouter un identifiant dans le tableau, cet ajout se fait en utilisant un indice qui s'incrémente tout en bouclant (à l'aide du modulo) au début du tableau à chaque dépassement de la taille. Cette boucle permet d'écraser les premiers identifiants du tableau qui ne font plus partie des NB_PAQUET_MEM derniers paquets.

```
//Fonction vérifiant si l'id du paquet arrivé a déjà été routé
//Si c'est le cas la fonction renvoi 1 (VRAI)
//Sinon, elle renvoi 0 (FAUX)
int PaquetDejaRoute(IDENTIFIANT id, IDENTIFIANT tab[]){

    for(int i=0; i<NB_PAQUET_MEM; i++){
        //Un élément du tableau correspond
        if((tab[i].srcAddr==id.srcAddr) && (tab[i].sqn==id.sqn)){
            return 1;
        }
    }
    return 0;
}
```

Finalement, nous utilisons la procédure PaquetDejaRoute qui prend en paramètre un identifiant ainsi qu'un tableau d'identifiants et qui renvoie VRAI (1) si l'identifiant est déjà présent dans le tableau et FAUX (0) si ce n'est pas le cas.

f) Gestion de la réémission

```
//Test Reception
if(wino.recv(&adresseMacSource, rxbuf, &packetLenRX)){
    . . . . .
    //Test pour la réémission
    //Le paquet n'a pas été émis par le noeud lui-même
    if(adresseIpSource!=ADRESSE_NOEUD){
        . . . . .
    }
}
```

Le premier test de réémission consiste à vérifier si l'adresse de couche 3 du paquet correspond à celle du nœud. Si c'est le cas, le nœud ignore le paquet, sinon, on passe au test suivant.

```

//Test Reception
if(wino.recv(&adresseMacSource, rxbuf, &packetLenRX)){
    . . . . .

    //Même adresse multicast et TTL inférieur au max
    if((adresseMulticast==ADRESSE_MULTICAST) &&
(rxbuf[CHAMP_TTL]<=TTL_MAX)){

        //Récupération de l'identifiant du paquet reçu
        idRecu.srcAddr=adresseIpSource;
        idRecu.sqn=rxbuf[CHAMP_NUM_SEQ];
        . . . . .
    }
}
}

```

Si le test précédent réussit, on vérifie si l'adresse multicast dans le paquet correspond à celle du nœud et si le TTL n'est pas encore dépassé. Si ce n'est pas le cas alors le nœud ignore le paquet. Sinon, on stocke l'adresse source et le numéro de séquence dans un identifiant pour le test suivant.

```

//Test Reception
if(wino.recv(&adresseMacSource, rxbuf, &packetLenRX)){
    . . . . .
    //Test si le paquet n'a pas déjà été reçu
    if(!PaquetDejaRoute(idRecu, tabId)){

        //Ajout du nouvel id dans le tableau
        ajoutTabId(idRecu, tabId, &indiceMemId);

        //Réémission du paquet
        wino.send(0xFFFF, rxbuf, packetLenRX);

        //Affichage des id mémorisés
        AfficheId(tabId);
    }
}
}
}

```

À partir de l'identifiant extrait construit précédemment on vérifie si ce dernier n'a pas déjà été routé par le nœud à l'aide de `PaquetDejaRoute()`. Si il a déjà été routé, alors on l'ignore. Sinon, on ajoute l'identifiant dans le tableau à l'aide de `AjoutTabId()` et on réemet le paquet.

5. Tests réalisés

Émetteur (1)	Relais (2)	Récepteur (3)
<pre>====Envoi Paquet==== 78 78 00 01 00 00 13 6D 1 E 33 7E F5 4B 3C BC 81 =====Reception===== MAC SRC : 2 IP SRC : 1 Adresse Multicast : 7878 TTL : 1 Num Seq : 0 Payload : 13 6D 1E 33 7E F5 4B 3C BC 81 ==Paquet emis par le noeud==</pre>	<pre>=====Reception===== MAC SRC : 1 IP SRC : 1 Adresse Multicast : 7878 TTL : 0 Num Seq : 0 Payload : 13 6D 1E 33 7E F5 4B 3C BC 81 =====Reemission Paquet===== {01, 00}{FFFF, FF}{FFFF, FF}{FFFF, FF}{FFFF, FF}{FFFF, FF}{FFFF, FF}{FFFF, FF}{FFFF, FF}{FFFF, FF} =====Reception===== MAC SRC : 3 IP SRC : 1 Adresse Multicast : 7878 TTL : 2 Num Seq : 0 Payload : 13 6D 1E 33 7E F5 4B 3C BC 81 ===Deja Recu===</pre>	<pre>=====Reception===== MAC SRC : 2 IP SRC : 1 Adresse Multicast : 7878 TTL : 1 Num Seq : 0 Payload : 13 6D 1E 33 7E F5 4B 3C BC 81 =====Reemission Paquet===== {01, 00}{FFFF, FF}{FFFF, FF}{FFFF, FF}{FFFF, FF}{FFFF, FF}{FFFF, FF}{FFFF, FF}{FFFF, FF}{FFFF, FF}</pre>

Nous pouvons voir que les identifiants des paquets sont stockés après réémission et le paquet n'est pas réémis si son identifiant est dans le tableau.

6. Avantages et limites

Permet de mémoriser plusieurs paquets.

Il faut adapter le nombre de paquets mémorisables au nombre de nœuds.

III. Conclusion

Cette SAÉ nous a permis de mieux comprendre les différents mécanismes liés aux protocoles de couche 2 et 3. Nous avons une meilleure compréhension de ces couches.

De plus, nous avons pu améliorer nos compétences en programmation événementielle ainsi qu'en langage C. Un langage qui requiert rigueur et attention.

Ce fut une expérience très enrichissante tant sur le plan technique que personnel.