Université de Mons
Faculté des sciences
Département d'Informatique
Service de réseaux et télécommunications

Réseau Wi-Fi multi-sauts sur plateforme ESP

Directeur :
Bruno Quoitin

Auteur :
Arnaud Palgen

Rapporteurs :
Alain Buys
Jeremy Dubrulle

Année académique 2019-2020

Introduction

Un réseau Wi-Fi traditionnel (voir Fig. 1) est composé d'un noeud central, le point d'accès (AP), qui est directement connecté à tous les autres noeuds (stations) du réseau. L'AP a alors pour rôle d'acheminer les paquets d'une station à une autre mais aussi des paquets vers des adresses IP externes au réseau. Un inconvénient de ce type réseau est qu'il a une couverture limitée car chaque station doit se trouver à portée de l'AP.

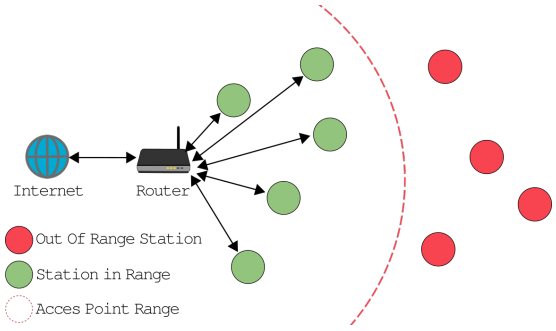

FIGURE 1 – Réseau Wi-Fi traditionnel.

L'objectif de ce projet est de mettre en place un réseau MESH multi-sauts (voir Fig. 2) [1] qui n'a pas ce problème. Un réseau MESH multi-sauts est un réseau où tous les noeuds peuvent communiquer avec tous les autres noeuds à la portée de leur radio. Chaque noeud peut ainsi acheminer les paquets de données de ses voisins vers le noeud suivant et ainsi de suite, jusqu'à ce qu'ils atteignent leurs destinations. Les routes utilisées pour acheminer les paquets sont obtenus à l'aide d'un protocole de routage.

1. Notez que sur la figure, un seul noeud fait office d'interface entre le réseau MESH et le réseaux IP externe. Ce n'est cependant pas toujours le cas.

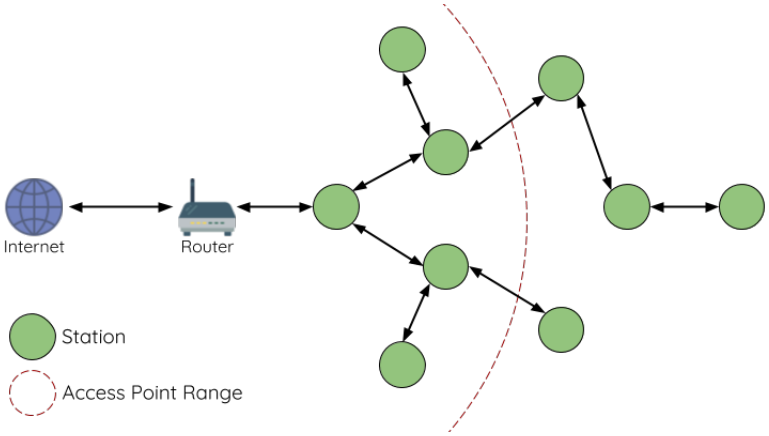

FIGURE 2 – Réseau MESH [1].

Pour ce projet, les noeuds du réseau MESH seront des microcontrôleurs Wi-Fi. L'ESP32 d'Espressif sera utilisé en raison de son très faible coût et ses mécanismes permettant d'obtenir une consommation électrique ultra-faible.

La première partie de ce rapport traite du choix de l'environnement de développement. Ensuite, la deuxième partie discute de certains protocoles de routage. Enfin, la dernière partie étudie et décrit la mise en oeuvre le protocole de routage ESP-MESH.

Table des matières

Chapitre 1

Etat de l'Art

1.1 Plateforme ESP32

Aperçu

Comme dit dans l'introduction, les noeuds du réseau sont des ESP32. Pour ce projet, une carte de développement (Fig. 1.1) équipé d'un ESP32-WROOM32, développé par Espressif est utilisée. L'ESP32-WROOM32 est un System-on-Chip (SoC), c'est à dire un circuit intégré rassemblant plusieurs composants comme des entrées/sorties, de la mémoire RAM, microprocesseurs, microcontrôleurs, etc.

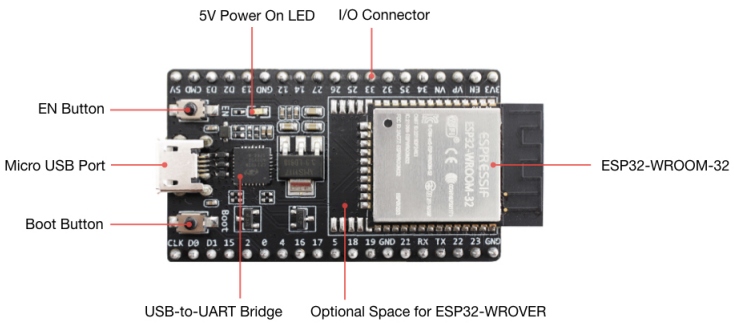

FIGURE 1.1 – ESP32-DevKitC V4 with ESP32-WROOM-32 module [10].

L'ESP32-WROOM32 a été choisi pour son faible coût (entre 3.50€ et 4€) et sa conception adaptée à l'Internet des Objets (IoT). En effet, en plus de supporter le Wi-Fi 2.4GHz et le Bluetooth, sa consommation en énergie est

faible et il possède des mécanismes permettant de l'économiser. La table 1.1 fournit ses spécifications.

Element	Spécification
WiFi	802.11 b/g/n (802.11n jusqu'à 150 Mbps)
Bluetooth	Bluetooth v4.2 BR/EDR and BLE specification
CPU	2 microprocesseurs Xtensa ® 32-bit LX6
Interfaces	SD card, UART, SPI, SDIO, I2C, LED PWM, Motor PWM, I2S, IR, pulse counter, GPIO, capacitive touch sensor, ADC, DAC
Tension de fonctionnement	$3.0V \sim 3.6V$

TABLE 1.1 – Spécification de l'ESP32-WROOM32 [3].

La figure 1.2 reprend le schéma-bloc des composants de l'ESP32.

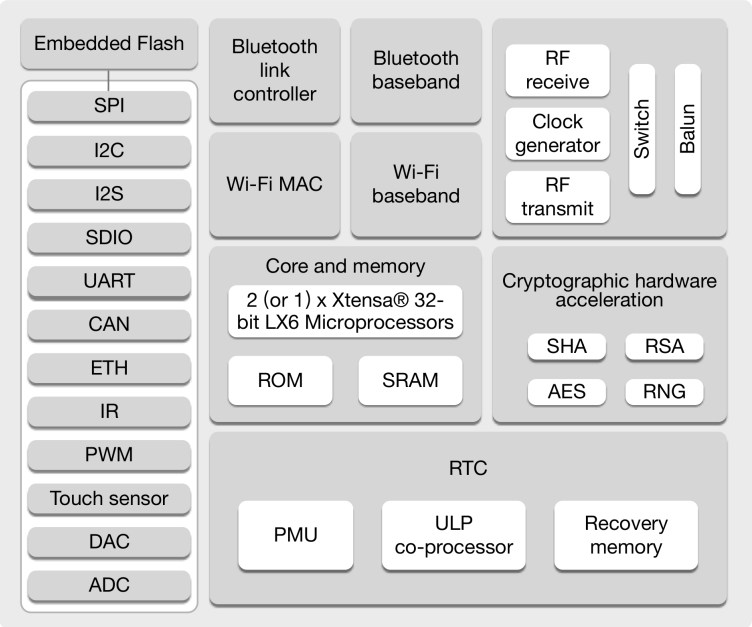

FIGURE 1.2 – Schéma-bloc [4].

Mémoire [3]
La mémoire interne inclut 448 KB de ROM pour le démarrage et les fonctions de base, et 520 KB de SRAM pour les données et les instructions.
L'ESP32 prend aussi en charge de la mémoire externe via le bus SPI.

Gestion de l'énergie
Comme dit plus haut, l'ESP32 a une consommation d'énergie faible. De plus, il possède plusieurs modes de fonctionnement repris dans la Table 1.2, permettant de la diminuer.

Power mode	Description	Power consumption
Active	radio and CPU are on	95mA $\sim$ 240 mA
Modem-sleep	radio is off, CPU is on at 80MHz	20mA $\sim$ 31mA
Light-sleep	CPU is paused, RTC memory and peripherals are running. Any wake-up events like MAC events will wake up the chip.	0.8mA
Deep-sleep	RTC memory and RTC peripherals are powered on	$10\mu A \sim 150\mu A$
Hibernation	RTC timer only	$5\mu A$
Power off	-	$0.1\mu A$

TABLE 1.2 – Consommation par mode [4].

1.2 Environnement de développement

Plusieurs environnements de développement peuvent être utilisés avec l'ESP32. Les sections suivantes présentent les plus courants.

1.2.1 MicroPython

Selon le site officiel de MicroPython [8], MicroPython est une implémentation simple et efficace de Python 3 incluant un petit sous-ensemble de la bibliothèque standard Python. Il est optimisé pour fonctionner sur des microcontrôleurs, open source et facile à utiliser. La documentation est complète et de nombreux tutoriels sont disponible et facilement compréhensible. Cependant, MicroPython n'expose pas les fonctions de bas niveau utiles pour ce projet. Par exemple, il semble difficile d'envoyer des paquets au niveau de la couche liaison de données ou encore, d'avoir accès aux tables de routages IP.
L'exemple suivant [1] illustre la simplicité du langage. Cet exemple permet de faire clignoter une LED branchée au GPIO 14.

1. Exemple provenant de https://micropython-on-esp8266-workshop.readthedocs.io/en/latest/basics.html (consulté le 07-06-2020).

```
from machine import Pin
import time

led = Pin(14, Pin.OUT)
for i in range(10):
    led.high()
    time.sleep_ms(500)
    led.low()
    time.sleep_ms(500)
```

1.2.2 IoT Development Framework (IDF)

IDF [2] est l'environnement du constructeur de l'ESP32 (Espressif). La documentation est complète mais le code source n'est pas entièrement disponible. Pour certaines parties du framework, seuls les fichiers d'entête sont disponibles. Ce framework étant développé par le constructeur de l'ESP32, il apporte une plus grande fidélité et stabilité à l'ESP32. Cet environnement donne aussi accès à des fonctionnalités de FreeRTOS (free real-time operating system) [13], un système d'exploitation temps réel open source pour microcontrôleurs. Ses fonctionnalités sont utiles pour ce projet. Des protocoles tels que ESP-MESH ou ESP-NOW sont également disponibles. Ils facilitent la mise en place d'un réseau MESH.
Exemple[2] de code permettant de faire clignoter une LED branchée au GPIO *CONFIG_BLINK_GPIO* :

```
#include <stdio.h>
#include "freertos/FreeRTOS.h"
#include "freertos/task.h"
#include "driver/gpio.h"
#include "sdkconfig.h"

#define BLINK_GPIO CONFIG_BLINK_GPIO

void app_main(void)
{
```

2. Exemple provenant de https://github.com/espressif/esp-idf/blob/master/examples/get-started/blink/main/blink.c (consulté le 07-06-2020).

```
    /*Configure the IOMUX register for pad BLINK_GPIO*/
    gpio_pad_select_gpio(BLINK_GPIO);
    /* Set the GPIO as a push/pull output */
    gpio_set_direction(BLINK_GPIO, GPIO_MODE_OUTPUT);
    while(1) {
        /* Blink off (output low) */
        printf("Turning off the LED\n");
        gpio_set_level(BLINK_GPIO, 0);
        vTaskDelay(1000 / portTICK_PERIOD_MS);
        /* Blink on (output high) */
    printf("Turning on the LED\n");
        gpio_set_level(BLINK_GPIO, 1);
        vTaskDelay(1000 / portTICK_PERIOD_MS);
    }
}
```

1.2.3 Arduino

L'environnement Arduino [15] se base sur IDF. Il est donc possible que certaines fonctionnalités d'IDF ne soient pas disponibles. La documentation est moins complète qu'IDF mais tout le code source Arduino est disponible. Comme MicroPython, il semble difficile d'envoyer des paquets au niveau de la couche liaison de données ou d'avoir accès aux tables de routages IP. L'exemple suivant permet de faire clignoter une LED branchée au GPIO 2.

```
#define LED  2

 void setup() {
   pinMode(LED,OUTPUT);
 }

 void loop() {
   delay(1000);
   digitalWrite(LED,HIGH);
   delay(100);
   digitalWrite(LED,LOW);
 }
```

1.2.4 Choix de l'environnement de développement

Notons que les solutions évoquées ci-dessus sont gratuites. Il existe des solutions commerciales payantes non évoquées ici, car les solutions gratuites s'avèrent suffisantes pour ce projet.

IDF est choisi pour sa documentation complète, sa fiabilité et pour son ensemble plus exhaustif de fonctionnalités que les autres environnements. Néanmoins, les extraits de code montrent qu'il est plus difficile à prendre en mains.

1.3 Protocoles de routage

Cette section, discute et classe différents protocoles de routage. Les protocoles de routage MESH peuvent être divisés en deux grandes catégories :

1. **Proactifs** : Les noeuds maintiennent une/des table(s) de routage qui stockent les routes vers tous les noeuds du réseau. Ils envoient régulièrement des paquets de contrôle à travers le réseau pour échanger et mettre à jour l'information de leurs voisins.
2. **Réactifs** : Ces protocoles établissent une route uniquement quand des paquets doivent être transférés.

Il existe une multitude de protocoles de routage MESH. La liste suivante énumère certains des protocoles les plus souvent cités dans la littérature :

— Ad-hoc On-demand Distance Vector (**AODV**) [6]
Protocole réactif à vecteur de distance que nous décrirons en détail dans le chapitre 2.1.

— Dynamic Source Routing (**DSR**) [16]
Similaire à AODV à l'exception que les paquets servant à la découverte d'un chemin (*RREQ*) contiennent tous les sauts de ce chemin.

— Optimized Link State Routing (**OLSR**) [11]
Protocole proactif à état de liens. Dans ce protocole, certains noeuds servent de relais pour effectuer le broadcasting des paquets servant à la découverte de chemins. L'ensemble de ces noeuds forme un arbre couvrant du réseau.

— Better Approach to Mobile Adhoc Networking (**B.A.T.M.A.N**) [5]
Protocole proactif à état de liens. Le protocole ne calcule pas le chemin pour atteindre un noeud mais le meilleur saut dans la bonne direction.

Pour cela, pour chaque destination il va sélectionner son voisin qui lui a transmis le plus de messages de cette destination.

— Destination Sequence Distance Vector (**DSDV**) [12]
Protocole à vecteur de distance basé sur l'algorithme de Bellman-Ford.

Nous pouvons donc classer ces protocoles de la manière suivante :

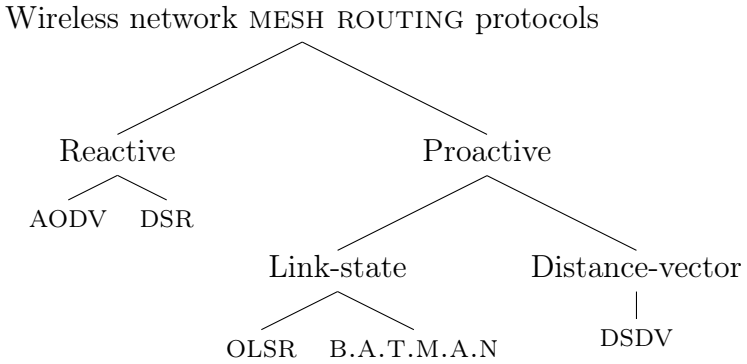

Diagram 1.1 – Classifications des protocoles de routages

Les protocoles proactifs gardent beaucoup d'information en mémoire. Ils ne passent donc pas à l'échelle. Les protocoles réactifs sont plus économes en ressources, mais nécessitent parfois un délai plus long pour établir une route, car elles sont établies à la demande.

A titre de comparaison avec ESP-MESH, le chapitre 2.1 détaille le protocole de routage AODV.

Chapitre 2

AODV

Ad-hoc On-demand Distance Vector (AODV) est un protocole réactif à vecteur de distance. Les 3 types de messages définis dans AODV sont les Route requests (RREQ*s*), les Route Replies (RREP*s*) et les Route Errors (RERR*s*). Comme illustré sur la figure 2.2, le premier sera envoyé en flooding par un noeud désirant obtenir une nouvelle route pour une destination donnée. Le second sert de réponse au premier. Il est envoyé à l'émetteur du RREQ. Et le dernier sert notamment lorsqu'un lien d'une route active est brisé. Les sections suivantes détaillent ces mécanismes.

Le flooding (inondation en français) consiste, pour un noeud donné, à transmettre chaque paquet entrant vers tous ses voisins à l'exception du voisin par lequel il a reçu ce paquet. Cette transmission n'a pas lieu si le noeud a déjà reçu le paquet auparavant.

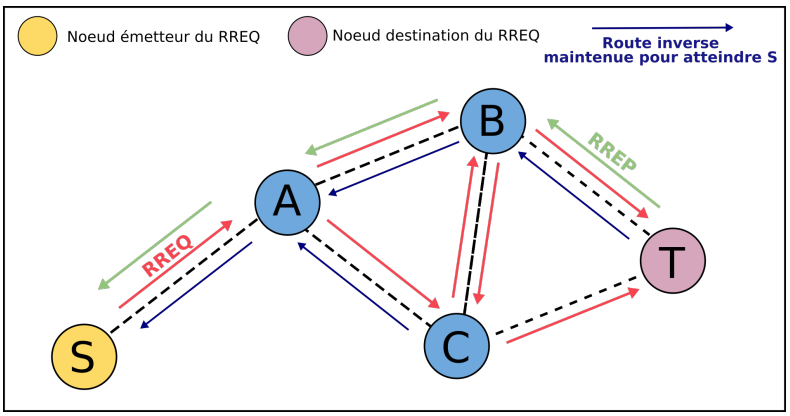

FIGURE 2.1 – Illustration du fonctionnement d'AODV.

2.1 Format des paquets

Cette section décrit le format des RREQ et RREP utilisés dans AODV.

RREQ

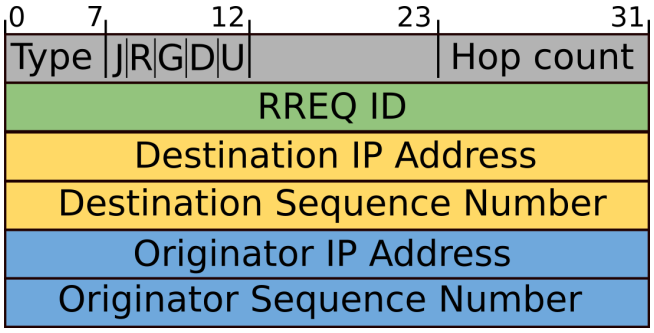

FIGURE 2.2 – Format d'un paquet RREQ [6].

Le format d'un RREQ est illustré sur la figure 2.2. Il contient les champs repris dans la table 2.1 :

type	= 1
J R G	flags
D	flag indiquant que seule la destination peut répondre au RREQ
U	flag indiquant que le numéro de séquence de la destination est inconnu
Hop count	nombre de sauts depuis le noeud source
RREQ ID	numéro de séquence identifiant le RREQ
Destination IP Address	adresse IP du noeud pour lequel la route est demandée
Destination Sequence Number	le dernier numéro de séquence connu pour une route vers la destination
Originator IP Address	adresse ip de l'émetteur du RREQ
Originator Sequence Number	numéro de séquence à utiliser pour une route pointant vers l'émetteur du RREQ

TABLE 2.1 – Champs d'un RREQ [6].

RREP

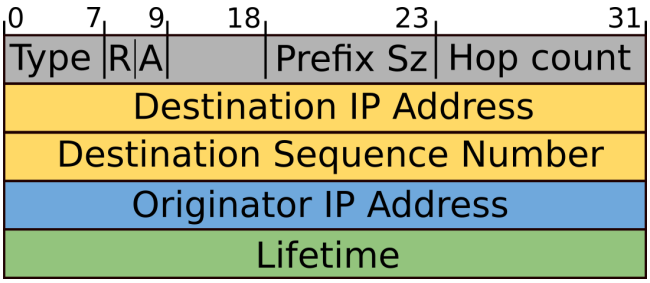

FIGURE 2.3 – Format d'un RREP [6].

Le format d'un RREP est illustré sur la figure 2.2. Il contient les champs repris dans la table 2.2 :

type	= 2
R	flag utilisé pour le multicast
Prefix size	utilisé pour les aggrégations de routes
Hop Count	nombre de sauts de l'*originator* à la destination
Destination IP address	adresse IP de du noeud pour qui l'adresse est demandée
Destination Sequence Number	numéro de séquence de destination associé à la route
Originator IP address	adresse IP du noeud émetteur du RREQ
Lifetime	temps (en ms) pendant lequel le noeud qui reçoit le RREP va considérer la route valide

TABLE 2.2 – Champs d'un RREP [6]

2.2 Découverte d'un chemin

La découverte d'un chemin est initiée par un noeud voulant envoyer des paquets à une destination pour laquelle il n'a aucune route active. Chaque noeud possède deux compteurs, le *sequence_number*, permettant d'éviter les boucles et de maintenir la consistance des informations de routage, et le *rreq_id*, qui combiné à une adresse IP, permet d'identifier un RREQ.

La découverte d'un chemin commence par la génération du RREQ. Le noeud source incrémente ses compteurs *sequence_number* et *rreq_id* de 1. Il envoie ensuite un RREQ en broadcast à ses voisins.

Elle est suivie par la propagation du RREQ se déroulant différemment pour un noeud intermédiaire et le noeud de destination :

- Noeud intermédiaire
  A la réception d'un RREQ, un noeud intermédiaire va pouvoir ajouter ou mettre à jour les routes vers son prédécesseur et vers le noeud source du RREQ(route inverse).
  Ensuite deux situations sont possibles :
  1. Le noeud courant possède une route active vers la destination et le numéro de séquence de la route est plus grand ou égal au numéro de séquence de la destination dans le RREQ. Dans ce cas, il peut envoyer par unicast un RREP à la source du RREQ.
  2. Sinon, le noeud va incrémenter le nombre de sauts du RREQ et le propager à ses voisins.
- Noeud destination
  A la réception d'un RREQ lui étant destiné, un noeud va, comme un noeud intermédiaire, rajouter ou mettre à jour les routes vers son prédécesseur et vers le noeud source du RREQ. Si le *Destination Sequence Number* du RREQ est égal à son *sequence_number*, il va incrémenter ce dernier et envoyer un RREP en unicast vers la source du RREQ.

Enfin, à la réception d'un RREP, un noeud va pouvoir rajouter ou mettre à jour les routes vers le noeud source du RREP et vers son successeur.
Il va ensuite incrémenter le nombre de sauts du RREP et le propager en unicast vers la destination de ce RREP. Cette propagation en unicast vers la source du RREQ est possible par l'apprentissage de la route inverse (destination du RREQ vers l'émetteur du RREQ) réalisée lors du flooding du RREQ.

2.3 Table de routage

Chaque entrée d'une table de routage contient les informations reprises dans la table 2.3.

dest	Adresse IP de destination
dest_SN	Numéro de séquence de destination
flag	Indicateur de numéro de séquence de destination valide
out	Interface réseau
hops	Comptage de sauts (nombre de sauts nécessaires pour atteindre la destination)
next-hop	Prochain saut
precursors	Liste des précurseurs
lifetime	temps d'expiration ou de suppression de l'itinéraire

TABLE 2.3 – entrée d'une table de routage AODV [6]

Mise à jour de la table de routage
Soit N une nouvelle route et O la route existante.
O est mise à jour si :

$$O.SN \leq N.SN$$
$$\textbf{ou } (O.SN = N.SN \textbf{ et } O.hop\_count > N.hop\_count)$$

Gestion du *lifetime*
Le temps de vie d'une route dans la table de routage est initialisé à *active_route_timeout* (3 secondes).
Quand ce timer expire, la route passe de active à inactive. Une route inactive ne pourra plus être utilisée pour transférer des données mais pourra fournir des informations pour de futurs RREQ et la réparation de routes.
Quand une route est utilisée, son temps de vie est actualisé à : $currenttime + active\_route\_timeout$

2.4 Défaillance d'un lien

Un noeud faisant partie d'une route active envoie en broadcast des messages *hello* (RREP) régulièrement. Si un noeud ne reçoit pas de message durant un certain temps pour un voisin, il va considérer que le lien avec ce voisin est perdu. Dans ce cas, il va en informer ses voisins impactés par un RERR.

2.5 Discussion

A priori les numéros de séquences suffisent pour éviter les boucles. Cependant, d'après [14], il y a des ambiguités dans le RFC [6]. Dû à ces ambiguités, l'implémentation pourrait introduire des boucles. Malgré cela, ce protocole semble plus robuste que ESP-MESH, car en comparaison avec ce dernier, si un noeud tombe, les noeuds peuvent trouver une autre route pour envoyer des paquets d'un point à un autre. A priori cette robustesse dépend également de certains paramètres comme le temps de vie ou la fréquence d'émission des messages *hello*.

Chapitre 3

ESP MESH

Esp-mesh est le protocole du constructeur Espressif permettant d'établir un réseau mesh avec des esp32. Ce protocole a pour objectif la création d'un arbre recouvrant. Cet arbre contient différents types de noeuds :

1. **Racine** : seule interface entre le réseau esp-mesh et un réseau ip externe.
2. **Noeuds intermédiaires** : noeuds qui ont un parent et au moins un enfant. Ils transmettent leurs paquets et ceux de leurs enfants.
3. **Feuilles** : noeuds qui n'ont pas d'enfants et ne transmettent que leurs paquets.
4. **Noeuds idle** : noeuds qui n'ont pas encore rejoint un réseau esp-mesh.

La figure 3.1 illustre ces différents types de noeuds au sein du réseau. esp-mesh est un protocole proactif s'adaptant automatiquement aux changements de topologie. Il permet également la connectivité à un réseau ip classique via une connexion Wi-Fi établie entre la racine du réseau et un point d'accès. La construction du réseau minimise la profondeur de l'arbre et l'élection de la racine se base sur la qualité du lien avec le point d'accès Wi-Fi.

Ce chapitre décrit le fonctionnement d'esp-mesh en détaillant le routage des paquets, la construction du réseau, la maintenance du réseau et le format des paquets.

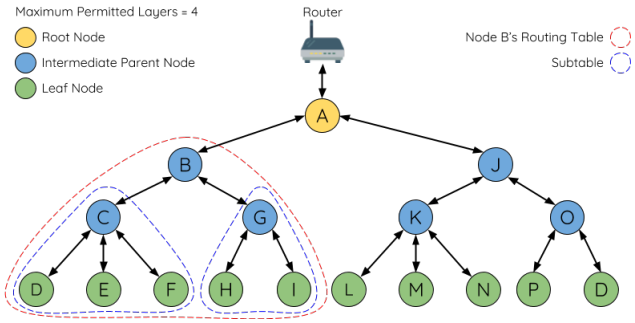

FIGURE 3.1 – Topologie d'un réseau ESP-MESH [1]

3.1 Routage

Chaque noeud possède sa table de routage. Soit p un noeud, sa table de routage contient les adresses MAC des noeuds du sous-arbre ayant p comme racine, et également celle de p.

Elle est partitionnée en sous-tables où chaque sous-table correspond à un sous-arbre de p. Par exemple, nous pouvons apercevoir sur la figure 3.1 que la table de routage du noeud B (en rouge) est partitionnée en 2 sous-table (en bleu) contenant respectivement le sous-arbre de racine C et le sous-arbre de racine G. A l'aide de cette table, l'acheminement des paquets est réalisé comme suit. Pour chaque paquet reçu :

- Si l'adresse MAC du paquet est dans la table de routage et si elle est différente de l'adresse du noeud l'ayant reçu, le paquet est envoyé à l'enfant correspondant à la sous-table contenant l'adresse.
- Si l'adresse n'est pas dans la table de routage, le paquet est envoyé au parent.

3.2 Construction d'un réseau

La première étape de construction du réseau consiste à sélectionner la racine. Cette étape est suivie par la formation de la deuxième couche et enfin par la formation des couches suivantes. La liste suivante détail ces trois étapes :

1. **Sélection de la racine**
   Le choix de la racine peut être réalisé par configuration. Dans ce cas, la racine se connecte au routeur et aucun processus d'élection n'a lieu. Dans le cas contraire, la racine va être sélectionnée automatiquement

via un processus d'élection. Pour cela, chaque noeud idle va transmettre son adresse MAC et la valeur de son RSSI (Received Signal Strength Indication) avec le routeur via des beacons. Dans le but de choisir comme racine, le noeud le plus proche de l'AP. Simultanément, chaque noeud scanne les beacons des autres noeuds. Si un noeud en détecte un autre avec un RSSI strictement plus fort, il va transmettre le contenu de ce beacon (càd voter pour ce noeud). Ce processus sera répété pendant un nombre minimum d'itérations (10 par défaut). Une itération pour un noeud, consiste à avoir reçu les beacons de tous les autres noeuds et avoir voté pour le noeud ayant le meilleur RSSI avec le routeur. Après toutes les itérations, chaque noeud va calculer le ratio suivant :

$$\frac{\textit{nombre de votes pour ce noeud}}{\textit{nombre de noeuds participants à l'élection}}$$

Ces deux informations sont connues par la réception des beacons. Si ce ratio est au-dessus d'un certain seuil (par défaut 90%), ce noeud deviendra la racine. [1]

2. **Formation de la deuxième couche**
   Une fois le processus d'élection d'une racine terminé, les noeuds idle à portée de la racine vont s'y connecter et devenir des noeuds intermédiaires

3. **Formation des autres couches**
   Chaque noeud du réseau ESP-MESH émet périodiquement des beacons contenant les informations suivantes :
   - Type du noeud (racine, intermédiaire, feuille, idle)
   - Couche sur laquelle se trouve le noeud
   - Nombre de couches maximum autorisées dans le réseau
   - Nombre de noeuds enfants
   - Nombre maximum d'enfants

   Sur base du contenu de ces beacons, les noeuds idle connaissent leurs potentiels parents. Si plusieurs parents sont possibles, un noeud choisira son parent selon deux critères :

   1. La couche sur laquelle se situe le candidat parent : le candidat se trouvant sur la couche la moins profonde sera choisi.

   2. Le nombre d'enfants du candidat parent : si plusieurs candidats se trouvent sur la couche la moins profonde, celui avec le moins d'enfants sera choisi.

1. Si plusieurs racines sont élues, deux réseaux ESP-MESH sont créés. Dans ce cas, ESP-MESH possède un mécanisme interne (dont le fonctionnement n'est pas décrit par Espressif) qui va fusionner les deux réseaux ssi les racines sont connectées au même routeur.

Un noeud peut également se connecter à un parent prédéfini. Une fois connectés, les noeuds deviennent des noeuds intermédiaires si le nombre maximal de couches n'est pas atteint. Sinon, les noeuds de la dernière couche deviennent automatiquement des feuilles, empêchant d'autres noeuds dans l'état idle de s'y connecter.

Pour éviter les boucles, un noeud ne va pas se connecter à un noeud dont l'adresse MAC se trouve dans sa table de routage.

La structure du réseau peut être affectée par l'ordre dans lequel les noeuds sont mis sous tension. Les noeuds ayant une mise en tension retardée suivront les deux règles suivantes :

1. Si le noeud détecte, par les beacons, qu'une racine existe déjà, il ne va pas essayer d'élire une nouvelle racine même si son RSSI avec le routeur est meilleur. Il va rejoindre le réseau comme un noeud idle.
   Si le noeud est la racine désignée, tous les autres noeuds vont rester idle jusqu'à ce que le noeud soit mis sous tension.
2. Si le noeud devient un noeud intermédiaire, il peut devenir le meilleur parent d'un autre noeud (cet autre noeud changera donc de parent).
3. Si un noeud idle a un parent prédéfini et que ce noeud n'est pas sous tension, il ne va pas essayer de se connecter à un autre parent.

3.3 Maintenance du réseau

Une fois l'arbre construit, il est nécessaire de le maintenir en réagissant aux événements faisant évoluer la topologie. En particulier, il est nécessaire de réagir à 3 types d'événements :

— **Défaillance de la racine**
  Si la racine tombe, les noeuds de la deuxième couche vont d'abord tenter de s'y reconnecter. Après plusieurs échecs, les noeuds de la deuxième couche vont entamer entre eux le processus d'élection d'une nouvelle racine. Si la racine ainsi que plusieurs couches tombent, le processus d'élection sera initialisé sur la couche la plus haute.

— **Défaillance d'un noeud intermédiaire**
  Si un noeud intermédiaire tombe, ses enfants vont d'abord tenter de s'y reconnecter. Après plusieurs échecs, ils se connecteront au meilleur parent disponible. S'il n'y a aucun parent possible, ils se mettront dans l'état idle. La figure 3.2 illustre ce processus.

— **Changement de racine**
  Ce changement ne peut être initié que par la racine elle-même via un appel à **esp_mesh_waive_root()**. Dans ce cas, un processus

d'élection de racine sera initialisé. La nouvelle racine élue enverra alors une *switch request* à la racine actuelle qui répondra par un acquittement. Ensuite la nouvelle racine se déconnectera de son parent et se connectera au routeur. L'ancienne racine se déconnectera du routeur et deviendra un noeud idle pour enfin se connecter à un nouveau parent.

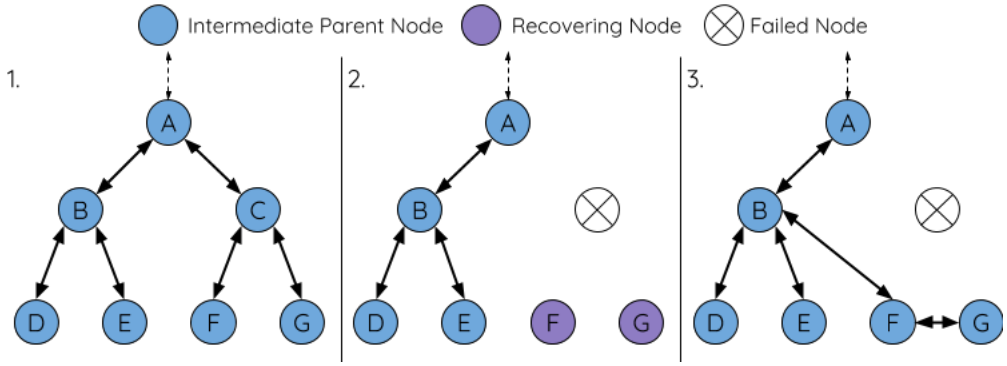

FIGURE 3.2 – Défaillance d'un noeud intermédiaire [1]

3.4 Paquets ESP-MESH

Les paquets ESP-MESH sont contenus dans une trame Wi-Fi. Une transmission multi-sauts utilisera un paquet ESP-MESH transporté entre chaque noeud par une trame Wi-Fi différente. La figure 3.3 montre la structure d'un paquet ESP-MESH.

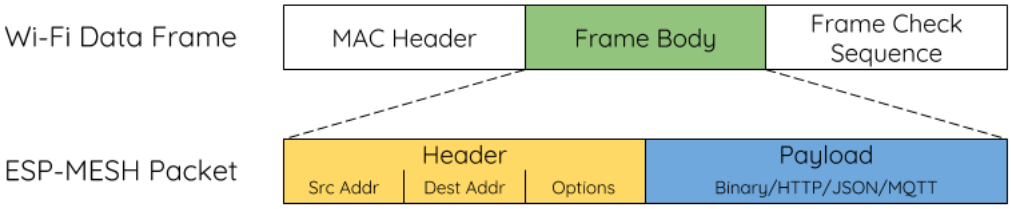

FIGURE 3.3 – Paquet ESP-MESH [1]

Le header d'un paquet ESP-MESH contient les adresses source et destination ainsi que diverses options. Ces adresses peuvent soit être une adresse MAC d'un noeud du réseau (au format EUI-48), soit être une adresse IPv4

encodée en base 16. Le payload d'un paquet ESP-MESH contient les données de l'application.

Dans le cas où l'adresse de destination est une adresse IP, le paquet sera envoyé à la racine du réseau ESP-MESH qui transmet le payload du paquet (par exemple en initiant une connexion tcp à l'aide d'un socket).

3.5 Contrôle de flux

Pour éviter que les parents soient submergés de flux venant de leurs enfants, chaque parent va assigner une fenêtre de réception à chaque enfant. Chaque noeud enfant doit demander une fenêtre de réception avant chaque transmission. La taille de la fenêtre peut être ajustée dynamiquement. Une transmission d'un enfant vers un parent se déroule en plusieurs étapes :

1. Le noeud enfant envoie à son parent une requête de fenêtre. Cette requête contient le numéro de séquence du paquet en attente d'envoi.
2. Le parent reçoit la requête et compare le numéro de séquence avec celui du précédent paquet envoyé par l'enfant. La comparaison est utilisée pour calculer la taille de la fenêtre qui est transmise à l'enfant [2].
3. L'enfant transmet le paquet conformément à la taille de fenêtre spécifiée par le parent. Une fois la fenêtre de réception utilisée, l'enfant doit renvoyer une demande de fenêtre.

3.6 Performances

Espressif fournit une estimation des performances d'ESP-MESH pour un réseau de maximum 100 noeuds, 6 couches et un nombre d'enfants par noeud de 6 (voir table 3.1).

Temps de construction du réseau	$<$ 60 secondes
Latence par saut	10 à 30 millisecondes
Temps de réparation du réseau	Si la racine tombe : $<$ 10 secondes Si un noeud enfant tombe : $<$ 5 secondes

TABLE 3.1 – Performances d'ESP-MESH [1]

2. Ce calcul n'est pas précisé par Espressif.

3.7 Discussion

A première vue, une topologie en arbre n'est pas robuste car si la racine tombe, tout le reste du réseau est déconnecté. De plus, le processus d'élection d'une nouvelle racine semble long selon les résultats fournis par Espressif. Un point négatif du protocole est que pour un noeud donné, sa table de routage contient tous les noeuds de son sous-arbre. On imagine donc difficilement utiliser ce protocole pour un nombre élevé de noeuds.

Chapitre 4

Mise en oeuvre

Ce chapitre décrit l'étude du protocole ESP-MESH. A chaque étape un extrait de code est expliqué et les communications sont analysées.

Ce chapitre décrit également la mise en oeuvre d'un proxy réalisé avec ESP-MESH ainsi que la réalisation d'un dissecteur Wireshark et de la découverte du protocole ESP-NOW.

La figure 4.1 illustre l'installation utilisée pour la réalisation de l'étude. La version 3.3.1 d'IDF est utilisée car c'est une version stable supportée jusqu'en février 2022.

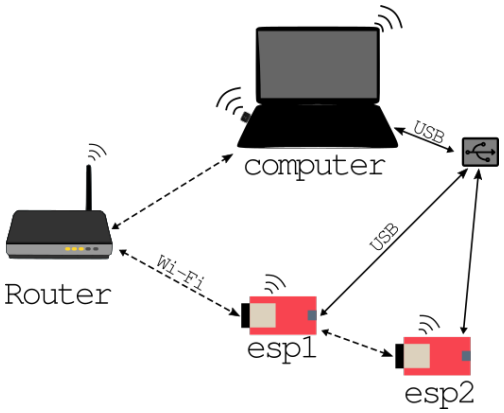

FIGURE 4.1 – Installation utilisée.

4.1 Construction du réseau ESP-MESH

La première étape est d'établir un réseau ESP-MESH composé de deux noeuds (une racine et son enfant). Pour cela, le serveur DHCP est d'abord désactivé, en effet, la racine étant la passerelle entre le réseau ESP-MESH et l'extérieur, elle est la seule à avoir besoin d'une adresse IP. Le serveur DHCP sera activé sur la racine une fois celle-ci élue. Ensuite le Wi-Fi est initialisé ainsi que le réseau ESP-MESH, pour enfin démarrer ce dernier.
Voici un extrait du code permettant de construire ce réseau :

```
1  void app_main(void){
2      /* stop DHCP server for softAP and station interfaces */
3      ESP_ERROR_CHECK(tcpip_adapter_dhcps_stop(TCPIP_ADAPTER_IF_AP));
4      ESP_ERROR_CHECK(tcpip_adapter_dhcpc_stop(TCPIP_ADAPTER_IF_STA));
5      /* wifi initialisation */
6      wifi_init_config_t config = WIFI_INIT_CONFIG_DEFAULT();
7      ESP_ERROR_CHECK(esp_wifi_init(&config));
8      ESP_ERROR_CHECK(esp_wifi_start());
9      /*  mesh initialisation*/
10     ESP_ERROR_CHECK(esp_mesh_init());
11     mesh_cfg_t cfg = MESH_INIT_CONFIG_DEFAULT();
12     /* event handler */
13     cfg.event_cb = &mesh_event_handler;
14     /* ... */
15     ESP_ERROR_CHECK(esp_mesh_set_config(&cfg)); //set mesh configuration
16     ESP_ERROR_CHECK(esp_mesh_start()); //start mesh network
17 }
```

L'analyse de la construction du réseau avec deux noeuds est réalisée avec les logs d'un ESP32 et une capture des trames Wi-Fi réalisée avec Wireshark. [1]

```
I (1479) mesh: <MESH_NWK_LOOK_FOR_NETWORK>need_scan:0x1,
    need_scan_router:0x0, look_for_nwk_count:1
I (1779) mesh: [FIND][ch:7]AP:0, otherID:0, MAP:0, idle:0,
    candidate:0, root:0[00:00:00:00:00:00]
I (1779) mesh: [FIND:1] fail to find a network , channel:0,
    cfg<channel:7, router:MyRouter, 00:00:00:00:00:00>

I (1789) mesh: <MESH_NWK_LOOK_FOR_NETWORK>need_scan:0x3,
    need_scan_router:0x1, look_for_nwk_count:2
I (1919) mesh: [S1]MyRouter, 1a:b2:c3:d4:e5:f6, channel:7, rssi:-43
I (1919) mesh: find router:[ssid_len:13]MyRouter, rssi:-43,
```

1. Les logs et captures Wireshark ont été réduit pour ne garder que les données utiles à la compréhension du fonctionnement du réseau ESP-MESH.

```
    1a:b2:c3:d4:e5:f6(encrypted), new channel:7, old channel:0
I (1929) mesh: [FIND][ch:7]AP:1, otherID:0, MAP:0, idle:0, candidate:0,
    root:0[1a:b2:c3:d4:e5:f6]router found<scan router>
I (1939) mesh: [FIND:2]find a network, channel:7, cfg<channel:7,
    router:MyRouter, 00:00:00:00:00:00>

I (1949) wifi: mode : sta (3c:71:bf:0d:83:08) + softAP (3c:71:bf:0d:83:09)
I (2269) mesh: [SCAN][ch:7]AP:2, other(ID:0, RD:0), MAP:1, idle:1,
   candidate:1,root:0, topMAP:0[c:1,i:1][1a:b2:c3:d4:e5:f6]router found<>
I (2269) mesh: 1022<pre>my_vote_num:0, voter_num/max_connection:4,
    2nd_layer_count:0
I (2279) mesh: 6104[SCAN]init rc[ttl:127/votes:1][3c:71:bf:0d:7e:1d,-120]
I (2279) mesh: 6104[SCAN]init rc[ttl:127/votes:1][3c:71:bf:0d:7e:1d,-120]
I (2289) mesh: 1250, vote myself, router rssi:-45 > voted rc_rssi:-120
I (2299) mesh: [SCAN:1/10]rc[128][3c:71:bf:0d:83:09,-45],
    self[3c:71:bf:0d:83:08, -45,reason:0,votes:1,idle]
        [mine:1,voter:1(1.00)percent:0.90][128,1,3c:71:bf:0d:83:09]

                                ....
I (8049) mesh: [SCAN:10/10]rc[128][3c:71:bf:0d:83:09,-45],
    self[3c:71:bf:0d:83:08,-39,reason:0,votes:2,idle][mine:2,voter:2(1.00)percent:0.90][128,2,3

I (8069) mesh: [DONE]connect to router:MyRouter, channel:7,
    rssi:-39, 1a:b2:c3:d4:e5:f6[layer:0, assoc:0], my_vote_num:2/voter_num:2, rc[3c:71:bf:0d:83
```

Les logs montrent d'abord que le noeud cherche un réseau ESP-MESH. Comme il n'en trouve pas, (ligne en rouge) il cherche un point d'accès. Il trouve le point d'accès *"MyRouter"* (ligne en vert) qui est sur le canal 7 et avec lequel il a un RSSI de -43. En écoutant les beacons des autres noeuds, il détecte également un autre noeud idle (ligne en jaune).

Ensuite le vote commence à la ligne surlignée en jaune. À chaque itération du vote, le noeud écoute les beacons des autres noeuds, ce qui lui permet de détecter un autre noeud idle qui a un RSSI avec le routeur de -120. Comme $-45 > -120$, il va voter pour lui-même, c'est à dire, émettre un beacon avec ses propres informations. Si son RSSI était moins fort que celui de l'autre noeud, il aurait émis un beacon avec les informations de l'autre noeud. Ce qui est décrit ici correspond à une itération du vote. Dans ce cas, il y aura 10 itérations. Ce nombre est un paramètre du réseau ESP-MESH. A la fin des 10 itérations, le noeud compare son ratio (défini à la section 3.2) à un seuil prédéfini (ici de 0.90). Comme ce ratio (ici de 1) est plus grand que le seuil, le noeud devient la racine du réseau ESP-MESH et se connecte au routeur. Ces observations correspondent bien à la description du protocole faite au chapitre 3. Analyse des trames capturées avec Wireshark :

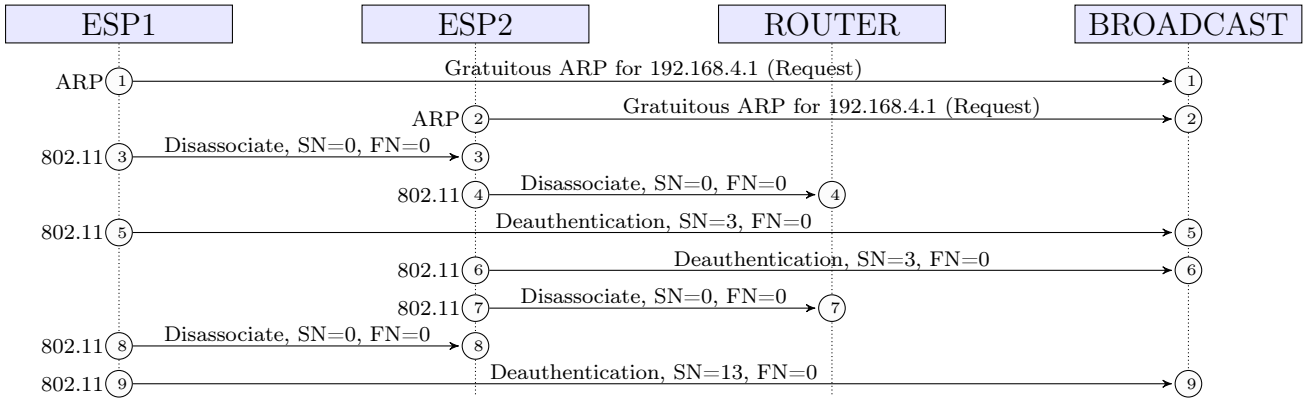

FIGURE 4.2 – Diagramme de séquence des paquets précédant le vote

Sur la figure 4.2, montre d'abord deux requêtes ARP de la part de chaque ESP32 pour l'adresse 192.168.4.1. La raison de cette requête reste inconnue. Cette adresse IP été associée à un point d'accès qui servait également de serveur pour les premières expérimentations réalisées pour la communication avec une adresse externe au réseau ESP-MESH. Ensuite, une série de trames de désassociation et désauthentification sont échangées. Après cette étape, le vote d'élection de la racine débute.

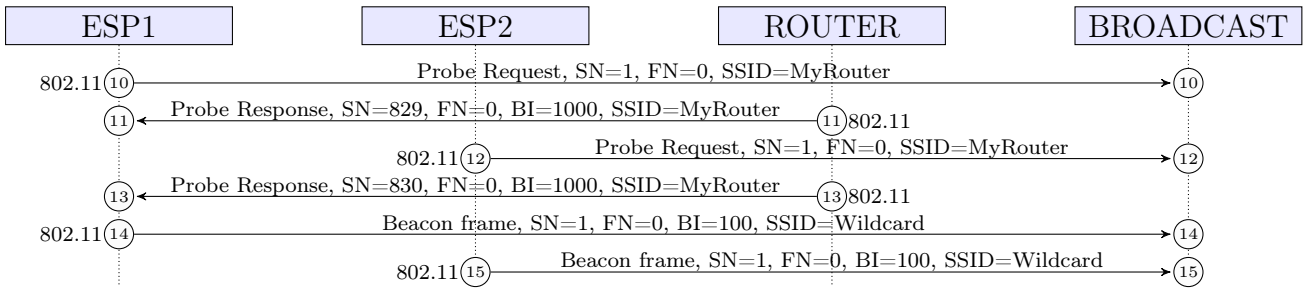

FIGURE 4.3 – Diagramme de séquence d'une itération du vote

La figure 4.3 représente une itération de processus de vote. Chaque noeud émet d'abord une *Probe Request* pour le point d'accès configuré. Il reçoit ensuite une *Probe Request* si le point d'accès est à sa portée. Avec la réception de cette trame, le noeud possède les informations dont il a besoin pour émettre son beacon contenant notamment son RSSI avec le point d'accès et l'ID du réseau ESP-MESH. Les autres noeuds à portée de ce noeud vont recevoir ce beacon et pourront donc voter. Voici un exemple d'un beacon émis lors du processus de vote.

```
IEEE 802.11 Beacon frame, Flags: ........C
    Type/Subtype: Beacon frame (0x0008)
    Receiver address: Broadcast (ff:ff:ff:ff:ff:ff)
    Destination address: Broadcast (ff:ff:ff:ff:ff:ff)
    Transmitter address: Espressi_0d:83:09 (3c:71:bf:0d:83:09)
    Source address: Espressi_0d:83:09 (3c:71:bf:0d:83:09)
    BSS Id: Espressi_0d:83:09 (3c:71:bf:0d:83:09)
IEEE 802.11 wireless LAN
    Tagged parameters (235 bytes)
        Tag: Vendor Specific: Espressif Inc.
            Tag Number: Vendor Specific (221)
            Tag length: 69
            OUI: 18:fe:34 (Espressif Inc.)
            Vendor Specific OUI Type: 1
            Vendor Specific Data: 0102007777777777771900040000000000000
                000000000088880000000000000880000000000008800000000000
                0000000000000000000a0000000f00005f8ad240
        Tag: Vendor Specific: Espressif Inc.
            Tag Number: Vendor Specific (221)
            Tag length: 22
            OUI: 18:fe:34 (Espressif Inc.)
            Vendor Specific OUI Type: 6
            Vendor Specific Data: 0602000b4553504d5f304438333038cd20f923
        Tag: Vendor Specific: Espressif Inc.
            Tag Number: Vendor Specific (221)
            Tag length: 21
            OUI: 18:fe:34 (Espressif Inc.)
            Vendor Specific OUI Type: 12
            Vendor Specific Data: 0c020000000000006defa84d995c80b0d2d3
```

Une fois que toutes les itérations ont été réalisées, la racine se connecte au routeur et le deuxième noeud se connecte à la racine. Ceci est illustré sur la figure 4.4.

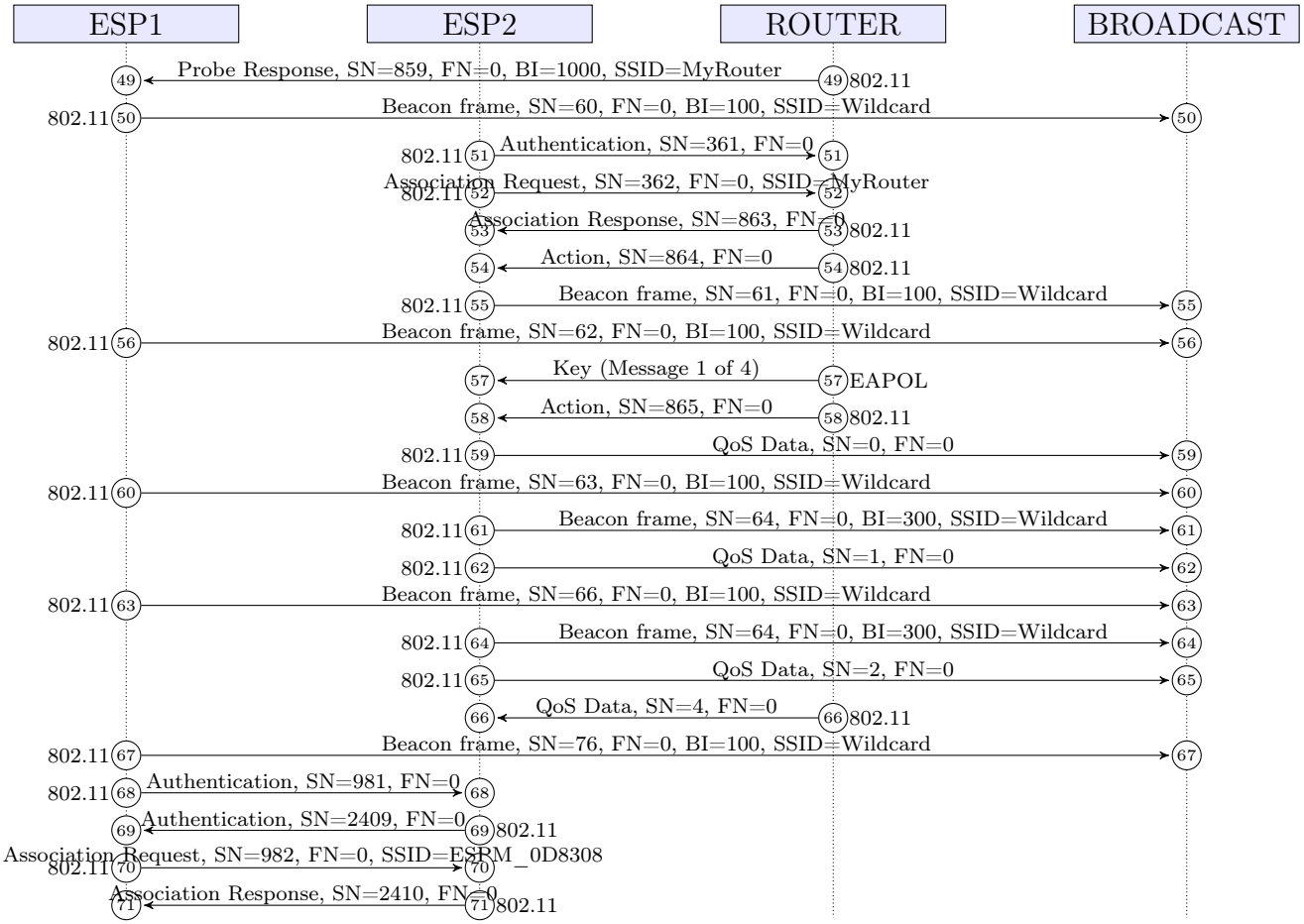

FIGURE 4.4 – Diagramme de séquence des connexions

Entre les messages 49 et 67, la racine élue se connecte au routeur, et des beacons continuent à être émis. Ensuite, du message 68 au message 71, le deuxième noeud se connecte à la racine.

4.2 Communications internes

La deuxième étape est de faire communiquer ces deux noeuds. Des messages ESP-MESH sont envoyés de la feuille vers la racine. Pour repérer plus facilement les paquets ESP-MESH dans Wireshark, ces derniers contiennent un tableau de 20 occurrences de EE. Une fois l'évènement **MESH_EVENT_PARENT_CONNECTED** détecté, les communications sont initialisées en fonction du type de noeud (racine ou autre). Cette information est obtenue via **esp_mesh_is_root()**

— Pour la racine : le serveur DHCP et une tâche FreeRTOS (crée via **xTaskCreate()**) sont démarrés. Cette tâche écoute en permanence les paquets ESP-MESH destinés à ce noeud via la méthode **esp_mesh_recv()**.
— Pour les autres types de noeuds du réseau, une tâche FreeRTOS est également démarrée. Cette tâche envoie continuellement des paquets ESP-MESH contenant un tableau de 20 occurrences de EE vers la racine. Cet envoi est fait via la méthode **esp_mesh_send()**.

Voici l'extrait du code réalisant ce qui vient d'être d'expliqué :

```
1  void esp_mesh_rx(void *arg){
2      uint8_t rx_buf[RX_BUF_SIZE]={0,}; //receive buffer
3      mesh_addr_t from; //src addr
4      /* mesh data */
5      mesh_data_t data;
6      data.data = rx_buf;
7      data.size = sizeof(rx_buf);
8      while(){
9          /* from addr, data, timeout in ms (0:no wait,
10          * portMAX_DELAY:wait forever), flag, options, number of options
11          */
12         err = esp_mesh_recv(&from, &data, 5000, &flag, NULL, 0);
13         if(err == ESP_OK){
14             printArray(rx_buf, RX_BUF_SIZE);
15         }
16     }
17     vTaskDelete(NULL); //delete the task
18 }
19 void esp_mesh_tx(void *arg){
20     static uint8_t tx_buf[TX_BUF_SIZE]= {238, 238, 238, 238, 238/*...*/};
21     /* mesh data */
22     mesh_data_t mesh_data;
23     mesh_data.data = tx_buf;
24     mesh_data.size = sizeof(tx_buf);
25     mesh_data.proto = MESH_PROTO_BIN;
26     while(){
27         /* dest addr, data: NULL for the root, flag, options,
28          * number of options
29          */
30         err = esp_mesh_send(NULL, &mesh_data, 0, NULL, 0);
31     }
```

```
32      vTaskDelete(NULL); //delete the task
33  }
34  void esp_mesh_comm_p2p_start(){
35      static bool p2p_started = false;
36      if(!p2p_started){
37          if(esp_mesh_is_root()){// root node
38              xTaskCreate(esp_mesh_rx, "MPRX", 3072, NULL, 5, NULL);
39          }else{// intermediate or leaf node
40              xTaskCreate(esp_mesh_tx, "MPTX", 3072, NULL, 5, NULL);
41          }
42      }
43  }
44  void mesh_event_handler(mesh_event_t event){
45      if (event.id == MESH_EVENT_PARENT_CONNECTED){
46          if (esp_mesh_is_root()) {
47              /* start DHCP server for the root node */
48              tcpip_adapter_dhcpc_start(TCPIP_ADAPTER_IF_STA);//
49          }
50          esp_mesh_comm_p2p_start();
51      }
52  }
```

Les logs d'un noeud n'apportant aucune information utile, seul les trames capturées par Wireshark sont analysées.

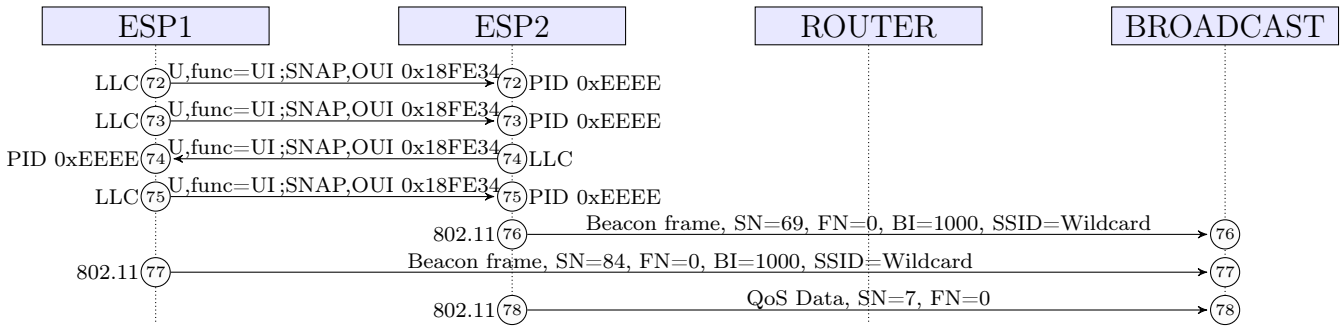

FIGURE 4.5 – Diagramme de séquence d'échange de données

Ce qui est illustré à la figure 4.5, est l'échange se déroulant après le construction du réseau. Dans ce cas, l'ESP1 envoie le payload choisi ($20 * \text{EE}$) à l'ESP2 (la racine). La première observation est que le protocole utilisé est LLC (*Logical Link Control*). Dans le modèle OSI, LLC est situé au niveau de la couche liaison de données, ce protocole sert de lien entre MAC et la couche réseau. Avant d'envoyer le payload, qui est le message 75, l'échange entre les

deux noeuds peut correspondre au calcul de la taille de la fenêtre comme évoqué dans la section 3.5. Les paquets 72 et 73 n'ont pas la même structure. Ci-dessous, un exemple d'un paquet de la même structure que le paquet 72.

```
Source                Destination           Protocol Length
Espressi_0d:7e:1c     Espressi_0d:83:09     LLC      252

Frame 380: 252 bytes on wire (2016 bits), 252 bytes captured (2016 bits)
Radiotap Header v0, Length 26
802.11 radio information
IEEE 802.11 QoS Data, Flags: ....R..TC
    Type/Subtype: QoS Data (0x0028)
    Frame Control Field: 0x8809
    Receiver address: Espressi_0d:83:09 (3c:71:bf:0d:83:09)
    Transmitter address: Espressi_0d:7e:1c (3c:71:bf:0d:7e:1c)
    Destination address: Espressi_0d:83:09 (3c:71:bf:0d:83:09)
    Source address: Espressi_0d:7e:1c (3c:71:bf:0d:7e:1c)
    BSS Id: Espressi_0d:83:09 (3c:71:bf:0d:83:09)
    STA address: Espressi_0d:7e:1c (3c:71:bf:0d:7e:1c)
    Frame check sequence: 0x17b820b5 [correct]
    Qos Control: 0x0000
Logical-Link Control
    DSAP: SNAP (0xaa)
        1010 101. = SAP: SNAP
        .... ...0 = IG Bit: Individual
    SSAP: SNAP (0xaa)
    Control field: U, func=UI (0x03)
    Organization Code: 18:fe:34 (Espressif Inc.)
    Protocol ID: 0xeeee
Data (188 bytes)
0000  21 2f bc 00 31 80 c0 00 00 00 00 00 00 00 3c 71
0010  bf 0d 7e 1c 00 00 00 00 ff ff ff 0f 03 9a 00 00
0020  3c 71 bf 0d 7e 1c 00 00 00 00 00 00 00 00 00 00
...
00b0  00 00 00 00 00 00 01 00 00 00 00 00
```

En vert, l'adresse de destination (cette adresse spécifique est utilisée pour la racine du réseau ESP-MESH) et en rouge, l'adresse source qui apparait deux fois. Voici maintenant un paquet de la même structure que le paquet 73.

```
Source                Destination           Protocol Length
Espressi_0d:7e:1c     Espressi_0d:83:09     LLC      92

Frame 382: 92 bytes on wire (736 bits), 92 bytes captured (736 bits)
Radiotap Header v0, Length 26
802.11 radio information
IEEE 802.11 QoS Data, Flags: .......TC
    Type/Subtype: QoS Data (0x0028)
    Frame Control Field: 0x8801
```

```
    Receiver address: Espressi_0d:83:09 (3c:71:bf:0d:83:09)
    Transmitter address: Espressi_0d:7e:1c (3c:71:bf:0d:7e:1c)
    Destination address: Espressi_0d:83:09 (3c:71:bf:0d:83:09)
    Source address: Espressi_0d:7e:1c (3c:71:bf:0d:7e:1c)
    BSS Id: Espressi_0d:83:09 (3c:71:bf:0d:83:09)
    STA address: Espressi_0d:7e:1c (3c:71:bf:0d:7e:1c)
    Frame check sequence: 0x6aaf2b8b [correct]
    Qos Control: 0x0000
Logical-Link Control
    DSAP: SNAP (0xaa)
    SSAP: SNAP (0xaa)
    Control field: U, func=UI (0x03)
    Organization Code: 18:fe:34 (Espressif Inc.)
    Protocol ID: 0xeeee
Data (28 bytes)
0000  01 07 1c 00 31 81 00 00 3c 71 bf 0d 83 09 3c 71
0010  bf 0d 7e 1c 01 00 00 00 01 00 00 00
```

Ici l'adresse de destination (en vert) n'est pas l'adresse spéciale utilisée pour la racine mais son adresse MAC. Elle est suivie de l'adresse MAC du noeud source (en rouge).

Analyse d'un paquet contenant nos données :

```
Source                Destination           Protocol Length
Espressi_0d:7e:1c     Espressi_0d:83:09     LLC      112

Frame 386: 112 bytes on wire (896 bits), 112 bytes captured (896 bits)
Radiotap Header v0, Length 26
802.11 radio information
IEEE 802.11 QoS Data, Flags: .......TC
    Type/Subtype: QoS Data (0x0028)
    Frame Control Field: 0x8801
    Receiver address: Espressi_0d:83:09 (3c:71:bf:0d:83:09)
    Transmitter address: Espressi_0d:7e:1c (3c:71:bf:0d:7e:1c)
    Destination address: Espressi_0d:83:09 (3c:71:bf:0d:83:09)
    Source address: Espressi_0d:7e:1c (3c:71:bf:0d:7e:1c)
    BSS Id: Espressi_0d:83:09 (3c:71:bf:0d:83:09)
    STA address: Espressi_0d:7e:1c (3c:71:bf:0d:7e:1c)
    .... .... .... 0000 = Fragment number: 0
    0000 0000 0010 .... = Sequence number: 2
    Frame check sequence: 0x3593d4d2 [correct]
    [FCS Status: Good]
    Qos Control: 0x0000
Logical-Link Control
    DSAP: SNAP (0xaa)
    SSAP: SNAP (0xaa)
    Control field: U, func=UI (0x03)
    Organization Code: 18:fe:34 (Espressif Inc.)
```

```
    Protocol ID: 0xeeee
Data (48 bytes)
0000  21 07 30 00 31 06 40 01 00 00 00 00 00 00 3c 71
0010  bf 0d 7e 1c 01 00 00 00 01 00 00 00 ee ee ee ee
0020  ee ee ee ee ee ee ee ee ee ee ee ee ee ee ee ee
```

Le paquet ESP-MESH contient les champs suivants :

- 8 octets supposés être utilisés pour des options.
- 6 octets utilisés pour l'adresse de destination (en vert). Cette adresse est celle utilisée pour la racine du réseau.
- 6 octets utilisés pour l'adresse source (en rouge).
- 8 octets supposés être utilisés pour des options.
- 20 octets utilisés pour le payload (en bleu).

La taille maximale du payload (MPS) est de 1472 octets. Le total donne donc 1500 octets ce qui correspond bien au MTU défini pour ESP-MESH. [2]

4.3 Communications externes

La racine joue le rôle de passerelle entre le réseau ESP-MESH et l'extérieur. Lorsqu'elle reçoit des données pour l'extérieur, elle initie une communication avec l'adresse de destination via un socket. L'implémentation de la couche IP avec IDF est *lwIP* (lightweight IP). Cette implémentation de la couche IP est légère et adaptée aux systèmes embarqués. Pour se familiariser à l'utilisation de sockets avec *lwIP*, une communication TCP entre un ESP32 et un serveur TCP est d'abord réalisée. Extrait de code :

```
1  void mySend(){
2      /* set dest addr */
3      struct sockaddr_in destAddr;
4      destAddr.sin_addr.s_addr = inet_addr(DEST_ADDR);
5      destAddr.sin_port = htons(DEST_PORT);
6      destAddr.sin_family = AF_INET;
7      /* set src addr */
8      struct sockaddr_in srcAddr;
9      srcAddr.sin_port = htons(SRC_PORT);
10     srcAddr.sin_family = AF_INET;
11     //get station info
12     tcpip_adapter_ip_info_t ipInfo;
```

2. Les constantes MTU et MPS sont définies dans le fichier **esp_mesh.h**

```
13      esp_err_t r = tcpip_adapter_get_ip_info(TCPIP_ADAPTER_IF_STA, &ipInfo);
14      //set srcAddr IP to station IP
15      memcpy((u32_t *) &srcAddr.sin_addr, &ipInfo.ip.addr,
16          sizeof(ipInfo.ip.addr));
17      /* create TCP socket */
18      int sock = socket(AF_INET, SOCK_STREAM, IPPROTO_IP);
19      /* bind socket to srcAddr */
20      bind(sock, (struct sockaddr *)&srcAddr, sizeof(srcAddr))
21      /* connect socket to destAddr */
22      connect(sock, (struct sockaddr *) &destAddr, sizeof(destAddr))
23      /* send data */
24      send(sock, payload, sizeof(payload), 0)
25  }
```

Tout d'abord, les adresses sources et destinations sont créées. Comme la source est l'ESP32, son adresse IP est récupérée via **tcpip_adapter_get_ip_info()** auquel il est demandé les informations de l'interface *station*. Ensuite, le socket peut être créé et lié à l'adresse source pour être finalement connecté à la destination afin de pouvoir envoyer des données.

Pour que les noeuds du réseau ESP-MESH puissent envoyer des paquets vers l'extérieur, ce code doit être adapté. Lorsqu'un message destiné à une adresse IP externe est reçu par la racine, il est récupéré via la fonction **esp_mesh_recv_toDS()**. Un socket TCP est ensuite créé avec comme adresse source, celle de la racine et comme adresse destination, celle spécifiée dans le paquet ESP-MESH reçu par la racine. Voici l'extrait de code correspondant :

```
1  mesh_addr_t mesh_to_addr;
2  struct sockaddr_in ip_to_addr;
3
4  err = esp_mesh_recv_toDS(&mesh_from_addr, &mesh_to_addr, &mesh_data,
5      timeout, &flag, NULL, 0);
6
7  memcpy((u32_t *) &ip_to_addr.sin_addr, &mesh_to_addr.mip.ip4.addr,
8      sizeof(mesh_to_addr.mip.ip4.addr));
9  /*create, bind and connect socket*/
10 sock_error = send(sock, mesh_data.data, mesh_data.size, 0);
```

Une difficulté a été rencontré à la ligne 7 car il a fallu connaître le type de l'adresse IP du paquet ESP-MESH ainsi que celui de l'adresse de *sockaddr_in*.

Dans l'analyse des échanges de trames, seul les paquets llc contenant le payload sont intéressant. En voici un extrait :

```
Source                Destination           Protocol Length
Espressi_0d:7e:18     Espressi_0d:7e:35     LLC      112

Frame 413: 112 bytes on wire (896 bits), 112 bytes captured (896 bits)
Radiotap Header v0, Length 26
802.11 radio information
IEEE 802.11 QoS Data, Flags: .......TC
    Type/Subtype: QoS Data (0x0028)
    Frame Control Field: 0x8801
    Receiver address: Espressi_0d:7e:35 (3c:71:bf:0d:7e:35)
    Transmitter address: Espressi_0d:7e:18 (3c:71:bf:0d:7e:18)
    Destination address: Espressi_0d:7e:35 (3c:71:bf:0d:7e:35)
    Source address: Espressi_0d:7e:18 (3c:71:bf:0d:7e:18)
    BSS Id: Espressi_0d:7e:35 (3c:71:bf:0d:7e:35)
    STA address: Espressi_0d:7e:18 (3c:71:bf:0d:7e:18)
    .... .... .... 0000 = Fragment number: 0
    0000 0000 0001 .... = Sequence number: 1
    Frame check sequence: 0x66bc9fd7 [correct]
    Qos Control: 0x0000
Logical-Link Control
    DSAP: SNAP (0xaa)
    SSAP: SNAP (0xaa)
    Control field: U, func=UI (0x03)
    Organization Code: 18:fe:34 (Espressif Inc.)
    Protocol ID: 0xeeee
Data (48 bytes)
0000  21 07 30 00 31 04 00 01 c0 a8 00 69 89 13 3c 71
0010  bf 0d 7e 18 01 00 00 00 01 00 00 00 ee ee ee ee
0020  ee ee ee ee ee ee ee ee ee ee ee ee ee ee ee ee
```

Comme attendu, le paquet ESP-MESH contient l'adresse source en rouge et le payload en bleu. Pour les paquets destinés à un noeud du réseau ESP-MESH, les octets de l'adresse de destination sont ceux en vert. Si les 4 premiers octets des 6 en vert sont convertis en base 10, ils correspondent bien à l'adresse IP de destination que nous utilisée pour cet exemple (192.168.0.105).

4.4 Réalisation d'un proxy

Le proxy implémenté (extrait de code : voir annexe B), permet d'établir une communication entre un noeud interne du réseau ESP-MESH et une IP externe. Celui-ci est réalisé avec la racine car elle est l'unique lien vers l'extérieur du réseau ESP-MESH. Lorsqu'un message destiné à une nouvelle adresse IP est reçu, elle va initier une connexion TCP via un nouveau socket lié à un nouveau port. Les sockets restent ouverts de tel façon que si l'hôte

externe répond, le paquet sera retransmis au noeud concerné du réseau ESP-MESH. La fonction **select()** est utilisée pour écouter les sockets ouverts. Les communications bidirectionnelles sont donc possibles entre un hôte extérieur au réseau et un noeud du réseau ESP-MESH, à partir du moment où le noeud a initié la connexion. La création des sockets et l'envoi vers une adresse IP externe se déroule comme expliqué à la section 4.3. Mis à part qu'après chaque nouveau socket, le numéro de port est incrémenté et un nouveau record (défini ci-dessous) est créé avec l'entier représentant le socket venant d'être créé et l'adresse IP de destination.

```
1 struct record{
2     int sock;
3     uint8_t addr[6];
4 };
5
6 static struct record* matching_table[MATCHING_TABLE_SIZE];
```

4.5 Extension à Wireshark

Avec les informations connues sur la structure d'un paquet ESP-MESH [1], l'implémentation d'un plugin pour Wireshark permettant de décoder ce protocole a été initiée. L'implémentation est basée sur les réponses trouvées sur le forum de Wireshark [7]. L'analyse de trames avec Wireshark se réalise avec des dissecteurs. Comme l'indique la documentation de Wireshark [9], chaque dissection de trame passe d'abord par le dissecteur de trame qui dissèque les détails du fichier de capture (comme par exemple le timestamp). Il passe ensuite les données au dissecteur de plus bas niveau et ainsi de suite jusqu'au moment où toutes les données de la trame ont été décodée.

Comme toutes les informations concernant la structure d'un paquet ESP-MESH ne sont pas connues, un *post dissector* est réalisé. Ce type de dissecteur est appelé après que tous les dissecteurs classiques ont terminé leur dissection. De ce fait, tous les champs dans l'arbre de dissection sont conservés et un champ ESP-MESH est ajouté. Le code complet se trouve dans l'annexe C. La figure 4.6 illustre le résultat de notre dissecteur dans Wireshark.

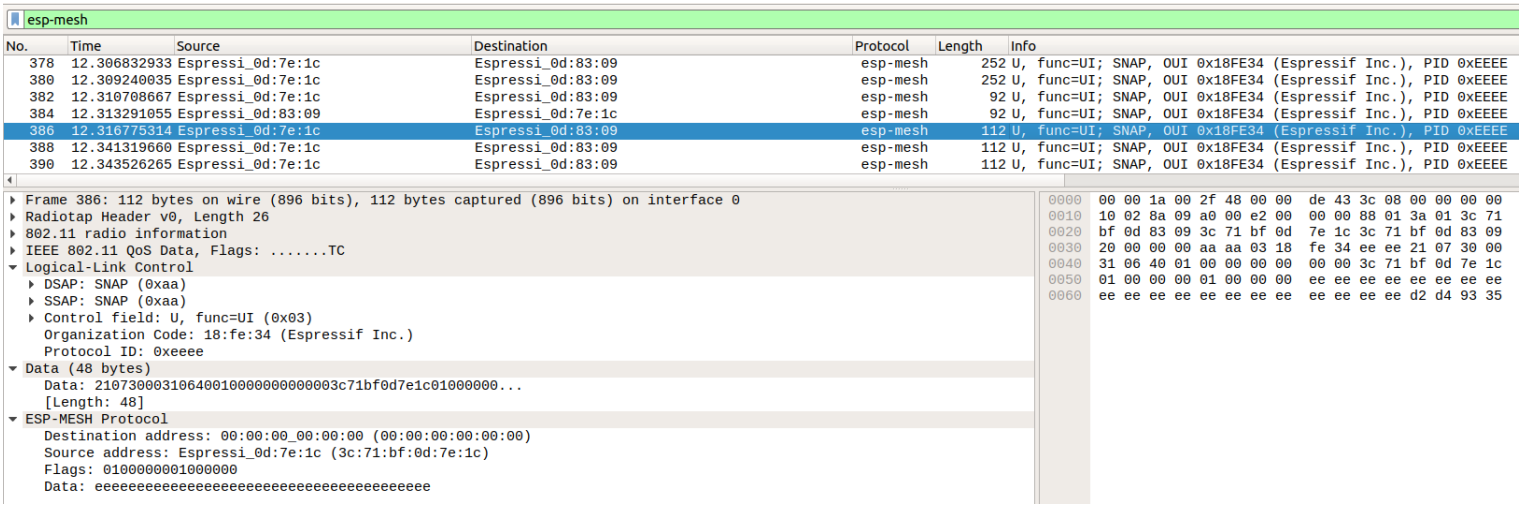

FIGURE 4.6 – Aperçu de Wireshark utilisant le dissecteur

4.6 ESP-NOW

Le driver Wi-Fi d'*IDF* ne permet pas d'avoir une connexion avec plusieurs noeuds simultanément. C'est peut-être la raison pour laquelle ESP-MESH utilise une structure d'arbre et non de graphe. ESP-NOW est une solution qui pallie à ce problème. En effet, avec ESP-NOW, un noeud peut avoir au maximum 20 voisins, ce qui rend possible l'implémentation d'un protocole comme AODV. Par contre, comparé à ESP-MESH (voir section 4.2), le MTU (maximum transmission unit) d'un paquet ESP-NOW est plus petit. En effet il est de 250 octets. [3]

Pour l'expérimentation, 3 noeuds sont utilisés. Un des noeuds envoie des données en broadcast aux autres. Voici l'extrait du code permettant de réaliser ce qui vient d'être décrit :

```
#define ESPNOW_WIFI_MODE WIFI_MODE_STA // Wi-Fi mode: sta, ap or sta+ap
#define ESPNOW_WIFI_IF ESP_IF_WIFI_STA // Wi-Fi interface sta or ap
static const uint8_t broadcast_addr[ESP_NOW_ETH_ALEN] =
    {0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF};
void espnow_recv_cb(const uint8_t *mac_addr, const uint8_t *data,
    int data_len){
    ESP_LOGI(TAG, "receive %d bytes:", data_len);
    printArray(data, data_len);
}
```

3. le MTU et le nombre de voisins maximum sont définis dans le fichier **esp_now.h**

```
10 void esp_now_tx(void *arg){
11     /* ... */
12     esp_now_peer_info_t peer;//create peer broadcast
13     peer.channel = CHANNEL;
14     peer.ifidx = ESPNOW_WIFI_IF;
15     peer.encrypt = false;
16     memcpy(peer.peer_addr, broadcast_addr, ESP_NOW_ETH_ALEN);
17     ESP_ERROR_CHECK(esp_now_add_peer(&peer)); // add peer
18     while(is_running){
19         esp_now_send(&broadcast_addr, &data, sizeof(data));
20         vTaskDelay( 1000 / portTICK_PERIOD_MS ); // delay the task
21     }
22     vTaskDelete(NULL);
23 }
24 void app_main(void){
25     /* ... */
26     /* Wi-Fi initialization */
27     wifi_init_config_t config = WIFI_INIT_CONFIG_DEFAULT();
28     ESP_ERROR_CHECK(esp_wifi_init(&config));
29     ESP_ERROR_CHECK(esp_wifi_set_mode(ESPNOW_WIFI_MODE));
30     ESP_ERROR_CHECK(esp_wifi_start());
31     /* ESP-NOW initialization */
32     ESP_ERROR_CHECK(esp_now_init());
33     ESP_ERROR_CHECK(esp_now_register_recv_cb(espnow_recv_cb));
34     /* ... */
35 }
```

Tout d'abord, le driver Wi-Fi est initialisé et son mode Wi-Fi est défini à *station*. Cette interface sera utilisée par ESP-NOW. Il est également possible d'utiliser l'interface *access point*. Ensuite, la fonction qui sera appelée lorsqu'un paquet ESP-NOW est reçu est définie. Après, pour le noeud source, la tâche **esp_now_tx()** est créée. Elle se chargera d'envoyer les données en broadcast. Pour cela, il faut d'abord créer un "voisin" broadcast, pour ensuite envoyer les données vers ce voisin. Pour envoyer des données vers un noeud précis, il faut créer un voisin avec l'adresse MAC du noeud.

Pour construire un réseau MESH, il faudrait, par exemple que chaque nouveau noeud émette en broadcast un paquet ESP-NOW contenant au moins son adresse MAC. De cette façon, les noeuds voisins pourraient l'ajouter à leurs liste de voisins.

Conclusion

L'objectif de ce projet était d'étudier et mettre en oeuvre le protocole de routage ESP-MESH d'Espressif permettant d'établir un réseau MESH avec des ESP32. Le protocole AODV a été également décrit à titre de comparaison avec ESP-MESH. L'étude d'ESP-MESH a permis d'en apprendre un maximum sur ce protocole malgré l'absence du code source. La mise en oeuvre de ce protocole a été réalisée par étape. Elle a permis d'acquérir une maîtrise avancée des sockets en C. Il a été intéressant de découvrir le fonctionnement des dissecteurs de Wireshark, outil utilisé régulièrement. Par manque de temps, des mesures du réseau ESP-MESH n'ont pas pu être réalisées. Le développement du dissecteur Wireshark pourrait être amélioré par une compréhension plus poussée du contenu des paquets ESP-MESH. Il serait également intéressant d'implémenter un autre protocole de routage sur ESP32 afin de le comparer avec ESP-MESH. Une étude du fonctionnement d'ESP-NOW serait aussi intéressante à réaliser.

Annexe A

Multicasting et Broadcasting avec ESP-MESH

Multicasting

Le multicasting permet d'envoyer simultanément un paquet ESP-MESH à plusieurs noeuds du réseau. Le multicasting peut être réalisé en spécifiant

- Soit un ensemble d'adresses MAC
  Dans ce cas, l'adresse de destination doit être `01:00:5E:xx:xx:xx`
  Cela signifie que le paquet est un paquet multicast et que la liste des adresses peut être obtenue dans les options du header.
- Soit un groupe préconfiguré de noeuds
  Dans ce cas, l'adresse de destination du paquet doit être l'ID[1] du groupe et un flag MESH_DATA_GROUP doit être ajouté.

Broadcasting

Le broadcasting permet de transmettre un paquet ESP-MESH à tous les noeuds du réseau. Pour éviter de gaspiller de la bande passante, ESP-MESH utilise les règles suivantes :

1. Quand un noeud intermédiaire reçoit un paquet broadcast de son parent, il va le transmettre à tous ses enfants et en stocker une copie.
2. Quand un noeud intermédiaire est la source d'un paquet broadcast, il va le transmettre à son parent et à ses enfants.
3. Quand un noeud intermédiaire reçoit un paquet d'un de ses enfants, il va le transmettre à ses autres enfants, son parent et en stocker une copie.

1. Dans un réseau ESP-MESH, chaque groupe a un ID unique ayant la même structure qu'une adresse mac (par exemple 7 7 : 7 7 : 7 7 : 7 7 : 7 7 : 7 7)

4. Quand une feuille est la source d'un paquet broadcast, elle va le transmettre à son parent.
5. Quand la racine est la source d'un paquet broadcast, elle va le transmettre à ses enfants.
6. Quand la racine reçoit un paquet broadcast de l'un de ses enfants, elle va le transmettre à ses autres enfants et en stocker une copie.
7. Quand un noeud reçoit un paquet broadcast avec son adresse MAC comme adresse source, il l'ignore.
8. Quand un noeud intermédiaire reçoit un paquet broadcast de son parent, s'il possède une copie de ce paquet (càd que ce paquet a été à l'origine transmis par l'un de ses enfants), il va l'ignorer pour éviter les cycles (protocole d'inondation).

Annexe B

Extrait de code du proxy

```
1  void esp_mesh_external_rx(void *arg){
2     struct sockaddr_in src;
3     socklen_t sockLen;
4
5     mesh_addr_t mesh_dest_addr;
6     mesh_data_t mesh_data;
7     mesh_data.proto = MESH_PROTO_BIN;
8
9     uint8_t *mac_addr;
10    static fd_set readSet;//set of file descriptors
11    struct timeval timeout = {.
12     tv_usec = 500000/*in microseconds*/
13     };
14    while(is_running){
15        FD_ZERO(&readSet);//clear the set
16        /*add sockets from the matching table to the set*/
17        for(int i=0; i<MATCHING_TABLE_SIZE; i++){
18            /*...*/
19            currentSock=matching_table[i]->sock;
20            FD_SET(currentSock,&readSet);
21            /*...*/
22        }
23        if(select(maxSock+1, &readSet, NULL, NULL, &timeout) < 0){
24            ESP_LOGE(TAG, "select error");
25            continue;
26        }
27        /*search for the record that corresponds to the socket that
28         *received data
```

```
29          */
30         for(int i=0; i<MATCHING_TABLE_SIZE; i++){
31             /*...*/
32             sock = matching_table[i]->sock;
33             mac_addr = matching_table[i]->addr;
34             if(FD_ISSET(sock, &readSet)){//we found the record
35                 recv_value = recvfrom(sock, rx_buf, sizeof(rx_buf), 0,
36                     &src, &sockLen);
37                 /*...*/
38                 mesh_data.size = sizeof(rx_buf);//set mesh data
39                 mesh_data.data = rx_buf;
40                 memcpy(mesh_dest_addr.addr, mac_addr, 6);//set mesh addr
41                 /*send data*/
42                 err=esp_mesh_send(&mesh_dest_addr,&mesh_data,
43                     MESH_DATA_P2P,NULL,0);
44                 /*...*/
```

Annexe C

Code du dissecteur

```
esp_mesh = Proto("ESP-MESH", "ESP-MESH Protocol")

dest_addr = ProtoField.ether("espmesh.dest", "Destination address", base.HEX)
src_addr = ProtoField.ether("espmesh.src", "Source address", base.HEX)
data = ProtoField.bytes("espmesh.data", "Data", base.NONE)
flags = ProtoField.bytes("espmesh.flags", "Flags", base.NONE)


esp_mesh.fields = {data, dest_addr, src_addr, flags}

local espmesh_PID = 0xeeee

local data_data = Field.new("data.data")
local llc_pid = Field.new("llc.pid")

function esp_mesh.dissector(tvbuf, pinfo, tree)
    local llc_pid_ex = llc_pid()

    if llc_pid_ex == nil or data_data() == nil or
        llc_pid_ex.value ~= espmesh_PID then
        return
    end
    pinfo.cols.protocol:set("esp-mesh")

    local data_tvb = data_data().range()
    local esp_mesh_tree = tree:add(esp_mesh, data_tvb(0, data_tvb:len()))

    esp_mesh_tree:add(dest_addr, data_tvb(8, 6))
```

```
29      esp_mesh_tree:add(src_addr, data_tvb(14, 6))
30      esp_mesh_tree:add(flags, data_tvb(20, 8))
31      if data_tvb:len() > 28 then
32          esp_mesh_tree:add(data, data_tvb(28))
33      end
34  end
35
36  register_postdissector(esp_mesh)
```

Bibliographie

[1] ESP-MESH api guide. https://docs.espressif.com/projects/esp-idf/en/v3.3.1/api-guides/mesh.html. Accessed : 04-06-2020.

[2] ESP-IDF Programming Guide. https://docs.espressif.com/projects/esp-idf/en/v3.3.1/. Accessed : 04-06-2020.

[3] Espressif Systems. *ESP32-WROOM-32 Datasheet*, 2019. Rev : 2.9.

[4] Espressif Systems. *ESP32 Series Datasheet*, 2020. Rev : 3.4.

[5] Protocole B.A.T.M.A.N. https://fr.wikipedia.org/wiki/BATMAN_(protocole). Accessed : 28-02-2020.

[6] E. Belding-Royer C. Perkins and S. Das. Ad hoc on-demand distance vector (aodv) routing. RFC 3561, RFC Editor, July 2003.

[7] 802.11 lua dissector. https://ask.wireshark.org/question/16067/80211-lua-dissector/. Accessed : 09-06-2020.

[8] MicroPython. https://micropython.org/. Accessed : 04-06-2020.

[9] Chapter 9. Packet Dissection. https://www.wireshark.org/docs/wsdg_html_chunked/ChapterDissection.html. Accessed : 8-03-2020].

[10] ESP32-DevKitC V4 Getting Started Guide. https://docs.espressif.com/projects/esp-idf/en/latest/hw-reference/get-started-devkitc.html. Accessed : 04-06-2020.

[11] T. Clausen and P. Jacquet. Optimized Link State Routing Protocol (OLSR). RFC 3626, RFC Editor, October 2003.

[12] Protocole DSDV. https://en.wikipedia.org/wiki/Destination-Sequenced_Distance_Vector_routing. Accessed : 8-03-2020].

[13] Freertos. https://www.freertos.org/. Accessed : 08-06-2020.

[14] Rob Van Glabbeek, Peter Höfner, Wee Lum Tan, and Marius Portmann. Sequence numbers do not guarantee loop freedom : AODV can yield routing loops, 2013.

[15] Arduino core for the esp32. https://github.com/espressif/arduino-esp32. Accessed : 04-06-2020.

[16] Y. Hu D. Johnson and D. Maltz. The Dynamic Source Routing Protocol (DSR) for Mobile Ad Hoc Networks for IPv4. RFC 4728, RFC Editor, February 2007.