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 Buyse
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 externe au réseau. Un inconvénient de ces réseaux est qu'ils ont une couverture limitée car chaque station doit se trouver à portée de l'AP.

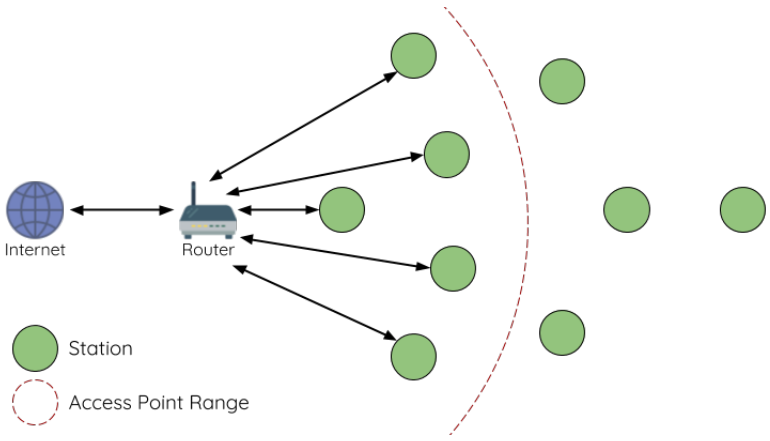

FIGURE 1 – Réseau Wi-Fi traditionnel [?]

L'objectif de ce projet est de concevoir 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 atteigent 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

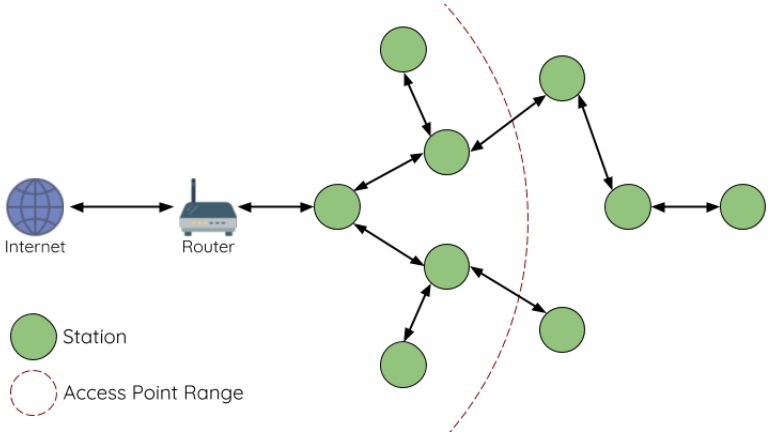

FIGURE 2 – Réseau MESH [?]

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 outils pemettant d'obtenir une consommation électrique ultra-faible.

Dans un premier temps, nous devrons choisir un protocole adapté à ce type de réseau et à l'ESP32. En effet, il existe une multitude de protocoles MESH dans la littérature scientifique.

Une fois choisi, nous implémenterons ce protocole pour créer un réseau fonctionnel. Le réseau ainsi créé sera testé pour en évaluer sa performance et ses fonctionnalités.

Enfin, nous nous attarderons sur l'économie d'énergie du microcontrôleur pour que notre réseau puisse être alimenté par batterie.

Table des matières

Chapitre 1

Etat de l'Art

1.1 Présentation de l'ESP

Aperçu

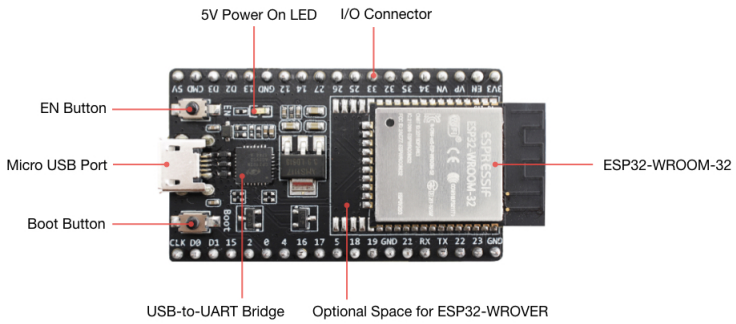

FIGURE 1.1 – ESP32-DevKitC V4 with ESP32-WROOM-32 module [**?**]

Comme dit plus haut, les noeuds de notre réseau seront des ESP32. Pour ce projet, nous utilisons un kit de développement (Fig. 1.1) équipé d'un ESP32-WROOM32, développé par Espressif. L'ESP32-WROOM32 un System-on-Chip (SoC), c'est à dire un circuit intégré rassemblant plusieurs composants comme des entrées/sorties, de la mémoire RAM, micorprocesseurs, microcontrôleurs, etc. Il a été choisis pour son faible coût et sa conception adaptée à l'Internet des Objets (IoT). En effet, en plus de supporter le Wi-Fi et le Bluetooth 2.4GHz, 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 micorprocesseurs 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 [?]

Schéma-bloc

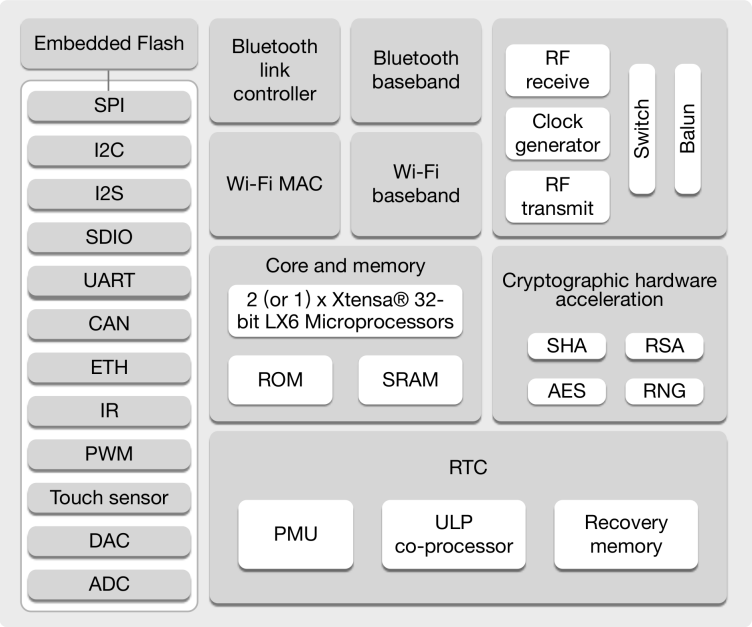

FIGURE 1.2 – Schéma-bloc [?]

Mémoire [?]
La mémoire interne inclut :
— 448 KB de ROM pour le démarrage et les fonctionsd de base
— 520 KB de SRAM pour les données et les instructions
L'ESP32 prend aussi en charge de la mémoire externe.

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 [**?**]

1.2 Environnement de développement

Trois environnements s'offrent à nous :

1. **MicroPython**
   Selon le site officiel de MicroPython [**?**], MicroPython est une implémentation simple et efficace de Python 3 incluant un petit sous-ensemble de la bibliothèque standard Python et est optimisé pour fonctionner sur des microcontrôleurs.
   MicroPython est open source et facile à utiliser. Cependant, il n'est pas assez bas niveau pour ce projet.
   Par exemple il serait impossible d'envoyer des paquets au niveau de la couche lisaison de données ou encore, avoir accès aux tables de routages IP.

2. **IoT Development Framework (IDF)**
   IDF est l'environnement du constucteur de l'ESP32 (Espressif). La documentation est complète mais le code source n'est pas entièrement disponible. Pour certaines parties du framework, nous n'avons accès qu'aux fichiers d'entête.
   Ce framework est natif et nous apportera donc une plus grande fidélité à l'ESP32.
   Cet environnement nous donne aussi accès à des fonctionnalités de FreeRTOS (free real-time operating system).
   FreeRTOS est un système d'exploitation temps réel open source pour

microcontrôleurs. Ses fonctionnalités pourront nous être utiles pour le projet.

3. **Arduino**
   L'environnement Arduino se base sur IDF. Il est donc possible que certaines fonctionnalités d'IDF ne soient pas disponible. La documentation est moins complète qu'IDF mais tout le code source est disponible. Cet environnement semble assez bas niveau car nous avons accès au driver Ethernet et à la couche IP.

Il est évident que nous ne choisirons pas MicroPython car certaines fonctionnalités utiles pour ce projet ne sont pas accessibles.
Nous choisirons IDF pour sa documentation complète, sa nativité et l'accès aux fonctionnalités de FreeRTOS.

1.3 Protocoles de routage

Dans cette section, nous discutons des différents protocoles de routage envisageables.
Nous allons d'abord établir un classement des protocoles de routage MESH.
Ensuite nous allons décrire brièvement les protocoles les plus cités dans la littérature pour les classer en fonction de leur appartenance à une catégorie établie dans notre classement. Enfin, nous pourrons choisir un protocole à implémenter pour ce projet.

Classification

Les protocoles de routages 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.

Nous écarterons les protocoles proactifs pour ce projet car ils 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.

Description

Il existe une multitude de protocols de routage MESH. Ci-dessous, en voici quelques-uns souvent cités dans la littérature :

— **AODV** [?] Ad-hoc On-demand Distance Vector
Protocole réactif à vecteur de distance que nous décrirons en détail par la suite.

— **DSR** [?] Dynamic Source Routing
Similaire à AODV mais ici, les paquets servant à la découverte d'un chemin ($RREQ$) contiennent tous les sauts de ce chemin.

— **OLSR** [?] Optimized Link State Routing
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.

— **B.A.T.M.A.N** [?] Better Approach to Mobile Adhoc Networking
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.

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

Nous pouvons donc classer ces protocoles de la manière suivante :

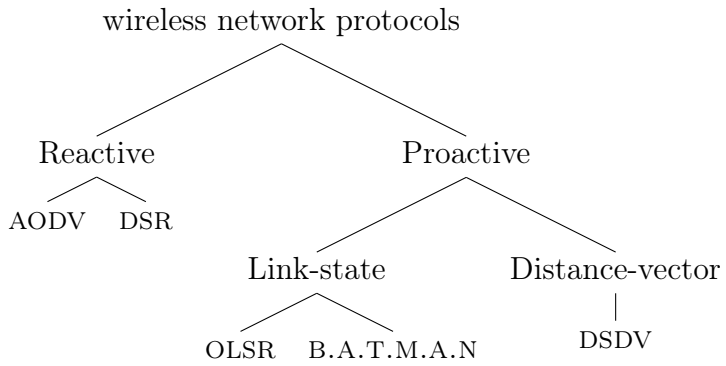

Diagram 1.1 – Classifications des protocols de routage

Choix d'un protocole

Comme dit plus haut, nous écartons les protocoles proactifs. Il nous reste donc le choix entre AODV et DSR. Nous allons retenir AODV pour la taille fixe de ses paquets. En effet, DSR utilise plus de données quand les routes contiennent un grand nombre de sauts.

Chapitre 2

ESP MESH

ESP-MESH est le protocole du constructeur Espressif permettant d'établir un réseau mesh avec des ESP32. Cette section explique le fonctionnement de ce protocole. ESP-MESH a pour objectif la création d'un arbre recouvrant. Il existe plusieurs 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.

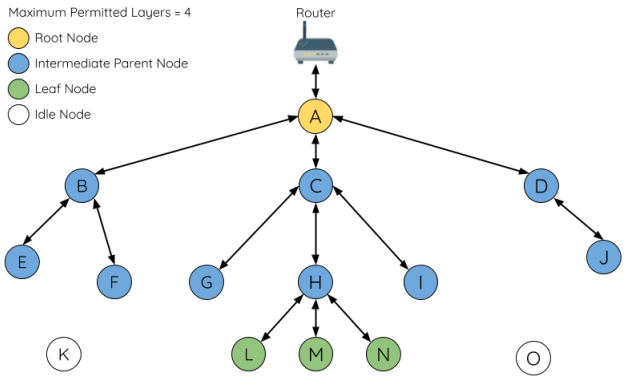

FIGURE 2.1 – Topologie d'un réseau ESP-MESH [?]

Routage

— Table de 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 qui correspondent aux sous-arbres des enfants de p.
— Acheminement de paquets
Quand un paquet est 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.

Construction d'un réseau

1. Élection de la racine
   — **Sélection automatique**
   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. Après toutes les itérations, chaque noeud va calculer le ratio

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

   Si ce ratio est au-dessus d'un certain seuil (par défaut 90%), ce noeud deviendra la racine. [1]
   — **Sélection par l'utilisateur**
   La racine se connecte au routeur et elle, ainsi que les autres noeuds, oublient le processus d'élection.

1. Si plusieurs racines sont élues, deux réseaux ESP-MESH seront créés. Dans ce cas, ESP-MESH possède un mécanisme interne qui va fusionner les deux réseaux ssi les racines sont connectées au même routeur.

2. Formation de la deuxième couche
   Une fois le processus d'élection d'une racine terminé, chaque noeud va émettre des beacons pour permettre aux autres noeuds de détecter sa présence et de connaître son statut. Ces beacons contiennent 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

   Les noeuds idle à portée de la racine vont s'y connecter et devenir des noeuds intermédiaires.
3. Formation des autres couches
   Les noeuds idle à portée de noeuds intermédiaires vont s'y connecter. Si plusieurs parents sont possibles, un noeud choisira son parent selon deux critères connus par les beacons des noeuds intermédiaires.

   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.

   Un noeud peut aussi 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.

Mise sous tension asynchrone

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 une racine existe déjà, le noeud ne vas 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 intermédiraire, 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.

Défaillance d'un noeud

— 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.

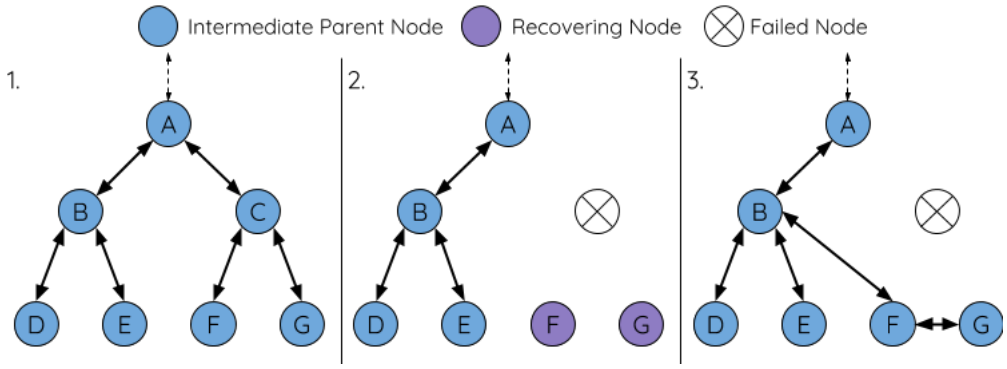

FIGURE 2.2 – Défaillance d'un noeud intermédiaire [?]

Changement de racine

Un changement de racine n'est possible que dans deux situations :

1. La racine tombe. (voir point précédent)
2. La racine le demande. 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.

Paquets ESP-MESH

Les paquets ESP-MESH sont contenus dans une trame WiFi. Une transmission multi-sauts utilisera un paquet ESP-MESH transporté entre chaque noeud par un paquet wifi différent.
La figure 2.3 montre la structure d'un paquet ESP-MESH :

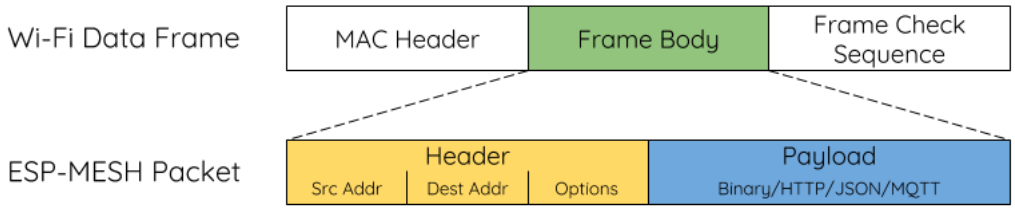

FIGURE 2.3 – Paquet ESP-MESH [?]

Le header d'un paquet ESP-MESH contient les adresses MAC source et destination ainsi que diverses options.
Le payload d'un paquet ESP-MESH contient les données de l'application.

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 [2] 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 :

2. Dans un réseauESP-MESH, chaque groupe a un ID unique

1. Quand un noeud intermédiare 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
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)

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 envoit 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.
3. L'enfant transmet le paquet en accord avec la taille de fenêtre. Une fois la fenêtre de réception utilisée, l'enfant doit renvoyer une demande de fenêtre.

Performances

Espressif fournit les performances d'ESP-MESH pour un réseau de 100 noeuds

avec un nombre maximum de couches de 6 et un nombre d'enfants maximum par noeuds de 6.(voir table 2.1)

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 2.1 – Performances d'ESP-MESH [**?**]

Discussion

A première vue, une topologie en arbre n'est pas robuste car si la racine tombe, tout le reste du réseau est déconnecté. Cependant le processus d'élection d'une nouvelle racine semble efficace selon les résulats 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 3

AODV

Ad-hoc On-demand Distance Vector (AODV) est un protocole réactif à vecteur de distance.
Ce protocole définit 3 types de messages :

1. Route request (*RREQs*)
2. Route Remplies (*RREPs*)
3. Route Errors (*RERRs*)

Format des paquets
RREQ

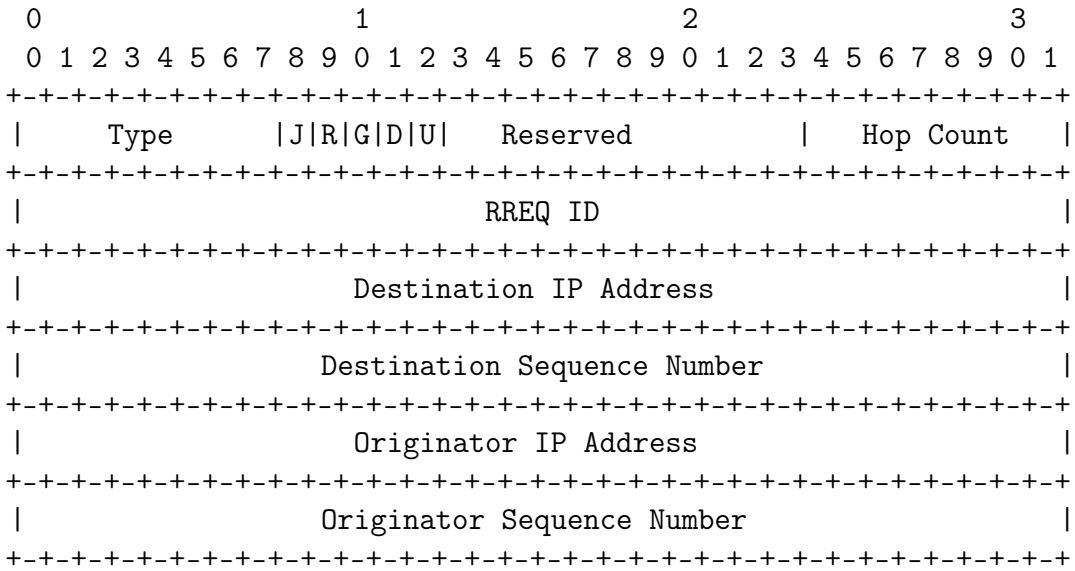

FIGURE 3.1 – format d'un paquet RREQ [**?**]

Où J,R,G,D,U sont des flags.

RREP

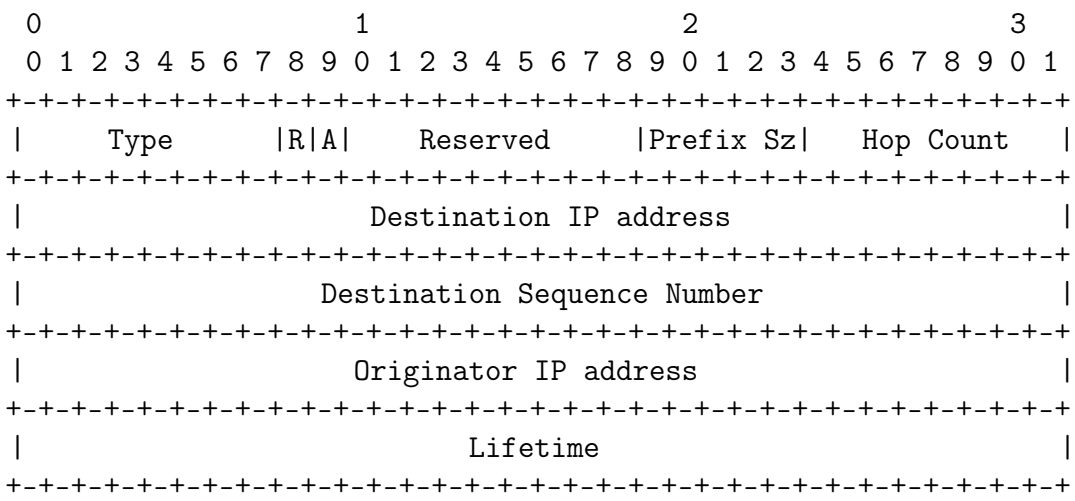

FIGURE 3.2 – format d'un paquet RREP [**?**]

Où R et A sont des flags.

Découverte d'un chemin

La découverte d'un chemin est intiée par un noeud voulant envoyer des paquets à une destination pour laquelle il n'a aucune information.
Chaque noeud possède deux compteurs : ***sequence_number*** et ***rreq_id***.

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.

Propagation du *RREQ*

- Noeud intermédiaire
  A la réception d'un *RREQ*, un noeud intermédiaire va pouvoir rajouter ou mettre à jour les routes vers le noeud source du *RREQ* et vers son prédécesseur.
  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. La condition en 1. n'est pas satisfaite.
   Dans ce cas, 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 le noeud source du *RREQ* et vers son prédécesseur.
  Si le ***Destination Sequence Number*** du *RREQ* est égale à son ***sequence_ number***, il va incrémenter ce dernier et envoyer un *RREP* en unicast vers la source du *RREQ*.

Propagation du *RREP*

A 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*.

Table de routage

Chaque entrée d'une table de routage contient les informations suivantes :

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 3.1 – entrée d'une table de routage AODV [?]

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 millisecondes).

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 : $currenttime + active\_route\_timeout$

Evitement de boucles

A priori les numéros de séquences suffisent pour éviter les boucles. Cependant, d'après [**?**], il y a des ambiguités dans le RFC [**?**]. Dû à ces ambiguités, l'implémentation pourrait introduire des boucles. Nous approfondirons la lecture de cet article si nous choisissons ce protocole afin d'éviter les boucles dans notre implémentation.

Défaillance d'un lien

Un noeud faisant partie d'une route active 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 informé ses voisins impactés par un RERR.

Chapitre 4

Mise en oeuvre

TODO: INTRO

Nous avons utilisé la version 3.3.1 d'IDF car c'est une version stable suportée jusqu'en février 2022.

4.1 ESP-MESH

1. **Construction du réseau**
   Notre première étape a été d'établir un réseau ESP-MESH composé de deux noeuds (une racine et son enfant). Voici un extrait du code permettant de construire ce réseau :

```
void app_main(void){
    /* stop DHCP server for softAP and station interfaces */
    ESP_ERROR_CHECK(tcpip_adapter_dhcps_stop(TCPIP_ADAPTER_IF_AP));
    ESP_ERROR_CHECK(tcpip_adapter_dhcpc_stop(TCPIP_ADAPTER_IF_STA));

    /* wifi initialisation */
    wifi_init_config_t config = WIFI_INIT_CONFIG_DEFAULT();
    ESP_ERROR_CHECK(esp_wifi_init(&config));
    ESP_ERROR_CHECK(esp_wifi_start());
    /*  mesh initialisation*/
    ESP_ERROR_CHECK(esp_mesh_init());
    mesh_cfg_t cfg = MESH_INIT_CONFIG_DEFAULT();
    /* event handler */
    cfg.event_cb = &mesh_event_handler;

            /* ... */

    /* set mesh configuration */
    ESP_ERROR_CHECK(esp_mesh_set_config(&cfg));
    /* mesh start */
    ESP_ERROR_CHECK(esp_mesh_start());
}
```

On remarque que nous désactivons le serveur DHCP. 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.
Analysons la construction du réseau avec 2 noeuds :

```
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,
```

```
12      1a:b2:c3:d4:e5:f6(encrypted), new channel:7, old channel:0
13  I (1929) mesh: [FIND][ch:7]AP:1, otherID:0, MAP:0, idle:0, candidate:0,
14      root:0[1a:b2:c3:d4:e5:f6]router found<scan router>
15  I (1939) mesh: [FIND:2]find a network, channel:7, cfg<channel:7,
16      router:MyRouter, 00:00:00:00:00:00>
17
18  I (1949) wifi: mode : sta (3c:71:bf:0d:83:08) + softAP (3c:71:bf:0d:83:09)
19  I (2269) mesh: [SCAN][ch:7]AP:2, other(ID:0, RD:0), MAP:1, idle:1,
20      candidate:1,root:0, topMAP:0[c:1,i:1][1a:b2:c3:d4:e5:f6]router found<>
21  I (2269) mesh: 1022<pre>my_vote_num:0, voter_num/max_connection:4,
22      2nd_layer_count:0
23  I (2279) mesh: 6104[SCAN]init rc[ttl:127/votes:1][3c:71:bf:0d:7e:1d,-120]
24  I (2279) mesh: 6104[SCAN]init rc[ttl:127/votes:1][3c:71:bf:0d:7e:1d,-120]
25  I (2289) mesh: 1250, vote myself, router rssi:-45 > voted rc_rssi:-120
26  I (2299) mesh: [SCAN:1/10]rc[128][3c:71:bf:0d:83:09,-45],
27      self[3c:71:bf:0d:83:08, -45,reason:0,votes:1,idle]
28          [mine:1,voter:1(1.00)percent:0.90][128,1,3c:71:bf:0d:83:09]
29
30                                  ....
31  I (8049) mesh: [SCAN:10/10]rc[128][3c:71:bf:0d:83:09,-45],
32      self[3c:71:bf:0d:83:08,-39,reason:0,votes:2,idle][mine:2,voter:2(1.00)p
33
34  I (8069) mesh: [DONE]connect to router:Sitecom0BD453, channel:7,
35      rssi:-39, 64:d1:a3:0b:d4:53[layer:0, assoc:0], my_vote_num:2/voter_num
```

On remarque d'abord que le noeud cherche un réseau ESP-MESH. Comme il n'en trouve pas, il va chercher un point d'accès. Il trouve le point d'accès *"MyRouter"* qui est sur le canal 7 et avec lequelil a un RSSI de -43. Ensuite il écoute les beacons des autres noeuds. On remarque bien qu'il détecte un autre noeud idle. Ensuite le vote commence à la ligne 19. À chaque itération du vote, le noeud écoute les beacons des autres noeuds. Ici il en détecte un autre noeud idle qui a un RSSIavec le routeur de -120. Comme $-45 > -120$, il va voter pour lui-même. Ce qui veut dire émettre un beacon avec ses 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 que nous avons décrit ici correspond à une itération du vote. Dans notre cas il y aura 10 itérations. Ce nombre est un paramètre tu réseau ESP-MESH. A la fin des 10 itérations, le noeud compare son ratio au seuil (ici de 0.90). Comme ce ratio (ici 1) est plus grand que le seuil, le noeud devient la racine du réseau ESP-MESH et se connecte au routeur. Ses observations

correspondent bien à la description du protocole faite plus haut.

Nous avons également observer la construction du réseau avec Wireshark. On remarque bien cet échange de beacons. Voici un exemple du contenu d'un beacon :

```
0000 00 00 1a 00 2f 48 00 00 27 b3 bb 07 00 00 00 00
0010 10 02 8a 09 a0 00 de 00 00 00 80 00 00 00 ff ff
0020 ff ff ff ff 3c 71 bf 0d 83 09 3c 71 bf 0d 83 09
0030 10 00 db 9c 01 00 00 00 00 00 64 00 21 04 00 00
0040 01 08 8b 96 82 84 0c 18 30 60 03 01 07 05 05 01
0050 02 00 00 00 07 06 43 4e 00 01 0d 14 2a 01 00 32
0060 04 6c 12 24 48 2d 1a 6e 11 00 ff 00 00 00 00 00
0070 00 00 00 00 00 00 01 00 00 00 00 00 00 00 00 00
0080 00 3d 16 07 05 00 00 00 00 00 00 00 00 00 00 00
0090 00 00 00 00 00 00 00 00 00 dd 18 00 50 f2 02 01
00a0 01 04 00 03 a4 00 00 27 a4 00 00 42 43 5e 00 62
00b0 32 2f 00 dd 45 18 fe 34 01 02 00 77 77 77 77 77
00c0 77 19 00 04 00 00 00 00 00 00 00 00 00 00 00 88
00d0 88 00 00 00 00 00 00 00 88 00 00 00 00 00 00 88
00e0 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 0a
00f0 00 00 00 0f 00 00 5f 8a d2 40 dd 16 18 fe 34 06
0100 02 00 0b 45 53 50 4d 5f 30 44 38 33 30 38 cd 20
0110 f9 23 dd 15 18 fe 34 0c 02 00 00 00 00 00 00 6d
0120 ef a8 4d 99 5c 80 b0 d2 d3 a1 d2 b2 10
```

En rouge, nous avons l'entête 802.11 et en bleu les paramètres supplémentaires d'Espressif pour le réseau ESP-MESH. Dans l'entête 802.11, on remarque l'adresse de destination qui est l'adresse Broadcast ainsi que l'adresse source qui est celle du noeud qui a émis ce beacon. Dans les paramètres supplémentaires, nous avons seulement distinguer l'id du réseau ESP-MESH qui est 77 77 77 77 77 77.

2. **Communications internes**

   La deuxième étape a été de faire communiquer ses deux noeuds. Nous avons donc envoyer des messages ESP-MESH de la feuille vers la racine. Pour repérer plus facilement les paquets ESP-MESH dans Wireshark, nous avons envoyé 20 fois 238 ce qui vaut EE en hexadécimal. Une fois l'évènement MESH_EVENT_PARENT_CONNECTED détecté, les communications sont initialisées en fonction que le noeud soit la racine ou non. Nous obtenons cette infrmation via **esp_mesh_is_root()**

— Pour la racine : Le serveur DHCP et une tâche FreeRTOS (créer via **xTaskCreate()**) sont démarrés. Cette tâche écoute en permanance les paquets ESP-MESH qui sont destinés au noeud via la méthode **esp_mesh_recv()**.
— Pour les autres types de noeuds du réseau, une tâche FreeRTOS est démarrée. Cette tâche envoie continuellement des paquets ESP-MESH contenant 20 fois EE comme données à la racine. Cet envoi ce fait via la méthode **esp_mesh_send()**.

Voici un extrait du code réalisant ce que nous venons d'expliquer :

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

```
32      static uint8_t tx_buf[TX_BUF_SIZE]= {238, 238, 238, 238, 238, 238, 238,
33
34      /* mesh data */
35      mesh_data_t mesh_data;
36      mesh_data.data = tx_buf;
37      mesh_data.size = sizeof(tx_buf);
38      mesh_data.proto = MESH_PROTO_BIN;
39
40      while(){
41          /* send:
42           *  - dest addr
43           *  - data: NULL for the root
44           *  - flag
45           *  - options
46           *  - number of options
47           **/
48          err = esp_mesh_send(NULL, &mesh_data, 0, NULL, 0);
49      }
50      /* delete the task */
51      vTaskDelete(NULL);
52  }
53  void esp_mesh_comm_p2p_start(){
54      static bool p2p_started = false;
55      if(!p2p_started){
56          if(esp_mesh_is_root()){// root node
57              xTaskCreate(esp_mesh_rx, "MPRX", 3072, NULL, 5, NULL);
58          }else{// intermediate or leaf node
59              xTaskCreate(esp_mesh_tx, "MPTX", 3072, NULL, 5, NULL);
60          }
61      }
62  }
63
64  void mesh_event_handler(mesh_event_t event){
65      if (event.id == MESH_EVENT_PARENT_CONNECTED){
66          if (esp_mesh_is_root()) {
67              /* start DHCP server for the root node */
68              tcpip_adapter_dhcpc_start(TCPIP_ADAPTER_IF_STA);
69          }
70          esp_mesh_comm_p2p_start();
71      }
72  }
```

Analysons les communications :
Grâce aux données que nous avons choisies d'envoyer, nous avons repérer facilement les paquets avec lequels ces données sont envoyées. Pour cet exemple de paquet, nous avons utilisé 3 noeuds : La racine, un noeud intermédiaire et une feuille. La feuille envoie un paquet à la racine.

```
0000    00 00 1a 00 2f 48 00 00 b0 23 e1 04 00 00 00 00
0010    10 02 8a 09 a0 00 e2 00 00 00 88 01 3a 01 3c 71
0020    bf 0d 83 09 3c 71 bf 0d 7e 18 3c 71 bf 0d 83 09
0030    50 00 00 00 aa aa 03 18 fe 34 ee ee 21 07 30 00
0040    31 04 40 01 00 00 00 00 00 00 3c 71 bf 0d 7e 18
0050    02 00 00 00 02 00 00 00 ee ee ee ee ee ee ee ee
0060    ee ee ee ee ee ee ee ee ee ee ee ee f8 5e 65 af
```

En rouge nous avons l'entête 802.11 où nous pouvons distinguer les adresses MAC du noeud source et du noeud de destination. En vert, Nous trouvons la couche **TODO: bof** LLC (logical link control) qui agit comme interface entre la couche MAC et la couche réseau. Enfin en bleu, le paquet ESP-MESH contenant :

— 8 octets que nous supposons être utilisés pour des options
— 6 octets utilisés pour l'adresse de destination (cette adresse est celle utilisée pour la racine du réseauenumerate).
— 6 octets utilisés pour l'adresse source
— 8 octets que nous supposons être utilisés pour des options
— 20 octets utilisés pour notre payload

La taille maximale du payload (MPS) est de 1472 octets. Le total nous donne donc 1500 octets ce qui correspond bien au MTU défini pour ESP-MESH. [1]

3. **Communications externes**
TODO: ~ extérieur
Comme nous l'avons déja dit plus haut, la racine joue le rôle de passerelle entre le réseau ESP-MESH et l'extérieur. Lorsqu'elle reçoit des données pour l'extérieur, elle va initier une communication avec l'adresse de destination via des socket. L'implémentation de la couche IP avec IDF est lwIP (lightweight IP). lwIP est une implémentation légère de la couche IP adaptée aux systèmes embarqués. Pour nous familiariser à l'utilisation de socket avec lwIP, nous avons d'abord

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

réalisé une communication TCP entre un ESP32 et un Raspberry pi utilisé comme serveur. Voici un extrait de code :

```
void mySend(){

    /* set dest addr */
    struct sockaddr_in destAddr;
    destAddr.sin_addr.s_addr = inet_addr(DEST_ADDR);
    destAddr.sin_port = htons(DEST_PORT);
    destAddr.sin_family = AF_INET;

    /* set src addr */
    struct sockaddr_in srcAddr;
    srcAddr.sin_port = htons(SRC_PORT);
    srcAddr.sin_family = AF_INET;
    //get station info
    tcpip_adapter_ip_info_t ipInfo;
    esp_err_t r = tcpip_adapter_get_ip_info(TCPIP_ADAPTER_IF_STA, &ipInfo);
    //set srcAddr IP to station IP
    memcpy((u32_t *) &srcAddr.sin_addr, &ipInfo.ip.addr,
        sizeof(ipInfo.ip.addr));

    /* create TCP socket */
    int sock = socket(AF_INET, SOCK_STREAM, IPPROTO_IP);

    /* bind socket to srcAddr */
    bind(sock, (struct sockaddr *)&srcAddr, sizeof(srcAddr))

    /* connect socket to destAddr */
    connect(sock, (struct sockaddr *) &destAddr, sizeof(destAddr))

    /* send data */
    send(sock, payload, sizeof(payload), 0)
}
```

Tout d'abord, nous créons les adresses sources et destinations. Comme la source sera notre ESP32, nous récupérons son adresse IP via **tcpip_adapter_get_ip_info()** auquel nous demandons les informations de l'interface *station*. Ensuite, nous pouvons créer un socket, le lier à l'adresse source et enfin le connecter à la destination pour pouvoir envoyer des données.

Ensuite nous avons rajouter la possibilité pour les noeuds du réseau

ESP-MESH d'envoyer des paquets vers l'extérieur. Lorsqu'un message destiné à une adresse IP externe est reçu par la racine, il est récupéré via la fonction **esp_mesh_recv_toDS()**. Nous devons ensuite créer un socket TCP avec comme adresse source, celle de la racine et comme adresse destination, celle spécifié dans le paquet ESP-MESH reçu par la racine. Voici l'extrait de code qui nous intéresse :

```
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 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*.

TODO: SOCKET SELECT

4. **Extension à Wireshark**

4.2 ESP-NOW

Le driver Wi-Fi d'*IDF* ne nous permet pas d'avoir une connexion avec plusieurs noeuds simultanément. Nous supposons que c'est pour cette raison qu'ESP-MESH utilise une structure d'arbre et non de graphe. ESP-NOW est une solution qui palie à ce problème.
En effet un noeud peut avoir maximum 20 voisins. Ce qui est suffisant pour établir un réseau MESH tel que nous l'envisagons. Par contre comparé à ESP-MESH, le MTU est plus petit. En effet il est de 250 octets. [2]

Utilisons ESP-NOW :
Nous avons utilisés 3 noeuds. Un des noeuds envoie des données en broadcast aux autres. Voici un exrait du code permettant de réalisé ce que nous venons de décrire :

```
1  #define ESPNOW_WIFI_MODE WIFI_MODE_STA // Wi-Fi mode: sta, ap or sta+ap
2  #define ESPNOW_WIFI_IF ESP_IF_WIFI_STA // Wi-Fi interface sta or ap
```

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

```
3
4  static const uint8_t broadcast_addr[ESP_NOW_ETH_ALEN] =
5      {0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF};
6
7  void espnow_recv_cb(const uint8_t *mac_addr, const uint8_t *data,
8      int data_len){
9      ESP_LOGI(TAG, "receive %d bytes:", data_len);
10     printArray(data, data_len);
11 }
12
13 void esp_now_tx(void *arg){
14             /* ... */
15     /* create peer broadcast */
16     esp_now_peer_info_t peer;
17     peer.channel = CHANNEL;
18     peer.ifidx = ESPNOW_WIFI_IF;
19     peer.encrypt = false;
20     memcpy(peer.peer_addr, broadcast_addr, ESP_NOW_ETH_ALEN);
21     ESP_ERROR_CHECK(esp_now_add_peer(&peer)); // add peer
22
23     ESP_LOGI(TAG, "send to all nodes");
24     while(is_running){
25         esp_now_send(&broadcast_addr, &data, sizeof(data));
26         vTaskDelay( 1000 / portTICK_PERIOD_MS ); // delay the task
27     }
28     vTaskDelete(NULL);
29 }
30
31 void app_main(void){
32             /* ... */
33     /* Wi-Fi initialization */
34     wifi_init_config_t config = WIFI_INIT_CONFIG_DEFAULT();
35     ESP_ERROR_CHECK(esp_wifi_init(&config));
36     ESP_ERROR_CHECK(esp_wifi_set_mode(ESPNOW_WIFI_MODE));
37     ESP_ERROR_CHECK(esp_wifi_start());
38
39     /* ESP-NOW initialization */
40     ESP_ERROR_CHECK(esp_now_init());
41     ESP_ERROR_CHECK(esp_now_register_recv_cb(espnow_recv_cb));
42             /* ... */
43 }
```

Tout d'abord nous initialisons le driver Wi-Fi. Nous définissions le mode du driver Wi-Fi à *station*. Cette interface sera utilisée par ESP-NOW. Il est également possible d'utiliser l'interface *acces point*. Ensuite nous définissions la fonction qui sera appellée lorsqu'un paquet ESP-NOW est reçu. Nous créons ensuite, pour le noeud source, la tâche **esp_now_tx()** qui envoie les données en broadcast. Pour cela, nous devons 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 façons, les noeuds voisins pourraient le rajouter à leurs liste de voisins.