Université de Mons
Faculté des Sciences
Département d'Informatique
Réseaux et Télécommunications

Conception et évaluation d'une architecture hybride de réseaux de capteurs reposant sur les technologies radio LoRa et IEEE 802.15.4

Directeur : Mr Bruno QUOITIN

Rapporteurs : Mr Prénom NOM
Mr Prénom NOM

Mémoire réalisé par
Arnaud PALGEN

en vue de l'obtention du grade de
Master en Sciences Informatiques

Année académique 2020-2021

Remerciements

Nous remercions ...

Table des matières

1 Etat de l'art **1**
1.1 Projets similaires 1
1.2 LoRa 2
1.3 IEEE 802.15.4e 4
1.4 RPL 8
1.5 RTOS 15

2 Architecture **19**
2.1 Topologie 19
2.2 Format d'adresse 21
2.3 Trames LoRaMAC 23
2.4 LoRaMAC 24
2.5 Plateformes de développement 29

3 Mise en oeuvre **34**
3.1 Racine LoRa 34
3.2 Racine RPL 38

4 Evaluation **46**

Chapitre 1

Etat de l'art

1.1 Projets similaires

Une grande partie du travail de ce mémoire consiste en l'élaboration d'un protocole MAC pour orchester les communications LoRa. Le protocole décrit au chapitre 2 s'inspire du protocole AloHa qui peut entraîner des collisions. Pour les minimiser, Yi Chu, Paul D. Mitchell et David Grace proposent dans leur article [**?**] un protocole MAC nommé ALOHA-QIR. Ce protocole combine Slotted ALOHA et le Q-Learning qui est une méthode d'apprentissage par renforcement qui permet aux noeuds d'utiliser les slots de temps de manière plus intelligente, ce qui diminue la probabilité d'avoir des collisions. Selon les auteurs de l'article, les simulations réalisées indiquent que leur protocole atteint plus du double du débit de slotted Aloha et que le débit de bout en bout ainsi que l'efficacité énergétique sont significativement améliorés.

Cependant, d'autres architectures sont possibles, comme les réseaux multi-sauts. Juan José López Escobar, Felipe Gil-Castiñeira et Rebeca P. Díaz Redondo présentent dans leur article [**?**] un protocole multi-sauts pour LoRa nommé JMAC. Ce protocole a été réalisé pour la récolte de données et utilise une topologie en arbre. Il supporte deux types de noeuds, la gateway et les capteurs qui collecte des données. Ces derniers ont leur radio active périodiquement ce qui leur permet d'économiser de l'énergie. Pour que les communications puissent s'effectuer, un noeud doit prédire le moment auquel la radio du prochain saut sera active. Pour cela, chaque capteur annonce la période durant laquelle sa radio est active.

Maite Bezunartea et ses co-auteurs proposent dans leur article [**?**], un protocole MAC pour LoRa qui est une adaptation de TSCH 1.3. Leur solution, implémentée dans Contiki-NG 1.5, permet d'utiliser le protocole de routage RPL 1.4. Elle fonctionne avec un *spreading factor* fixe (c.f. 1.2). Les

mécanismes de sauts de canaux et de division du temps en slots permettent d'obtenir des transmissions de données réussies tout en restant efficace en terme d'énergie.

Pour obtenir un *spreading factor* adapté à chaque lien LoRa, B.Satori et ses co-auteurs proposent dans leur article [**?**], un protocole MAC nommé RLMAC qui permet de sélectionner le *spreading factor* optimal pour chaque lien et également une adaptation de la fonction objectif OF0 1.4 permettant de sélectionner un chemin qui minimise le temps de transmission, ce qui permet de diminuer la consommation en énergie.

1.2 LoRa

LoRa (Long Range) [**?**][**?**] est une technologie radio fonctionnant sur une modulation propriétaire détenue par Semtech et basée sur *chirp spread spectrum (CSS). chirp spread spectrum* consiste en un signal d'une amplitude constante pour lequel la fréquence diminue ou augmente au cours du temps. Ces variations qui sont pour LoRa linéaires, sont respectivement appelées *downchirp* et *upchirp*. Cette modulation permet d'obtenir des communications longues portées et basses énergie.

Une communication LoRa dépend des paramètres suivants :

- **Spreading Factor (SF)** est une variable qui définit la vitesse de balayage du signal radio (Fig 1.1). Le spreading factor est compris entre 7 et 12. Plus sa valeur est grande, plus la vitesse de balayage est petite. Ceci implique un signal plus facile à décoder, mais avec un débit de données plus faible. Donc, inversement à une petite valeur de SF, une grande valeur de SF a un débit plus faible et une QoS (Quality of Service) plus élevée.
- **Bandwidth (BW)** détermine la largeur de la bande passante. Une bande passante plus large augmente la qualité du signal. Pour LoRa, en Europe, les valeurs de BW sont limitées à 125 KHz et 250KHz [**?**].
- **Coding rate (CR)** détermine la quantité de bits redondants ajoutés par le codage d'erreur cyclique utilisé par LoRa pour la détection et la correction d'erreurs. Cette méthode de correction d'erreur permet au signal de supporter de courtes interférences. Ainsi, il est possible de coder des données de 4 bits avec des redondances de 5, 6, 7 ou 8 bits. La valeur du CR se note donc 4/5, 4/6, 4/7 ou 4/8.

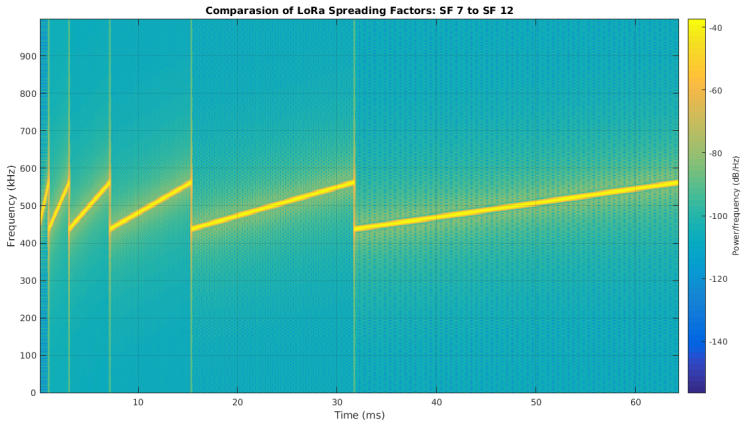


FIGURE 1.1 – Spectogramme des différents Spreading Factors [**?**].

Le format des paquets LoRa (Fig. 1.2) peut être explicite ou implicite. Dans le mode implicite, le header n'est pas inclus dans le paquet. Pour cela, la taille de la payload, le coding rate et la présence de la Payload CRC (Cyclic Redundancy Check) doivent être configurés. Ce mode permet de réduire la taille du paquet et donc du temps de transmission.

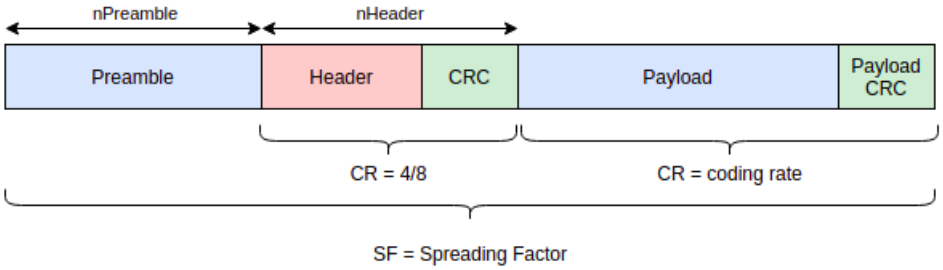


FIGURE 1.2 – Format d'un paquet LoRa.

La datasheet du SX1276 intégré dans le RN2383, utilisé comme transceiver LoRa pour ce projet (c.f. 2.5), définit le temps d'émission d'un paquet LoRa reprit par l'équation 1.1.

$$T_{frame} = T_{preamble} + T_{payload} \tag{1.1}$$

Où $T_{preamble}$ et $T_{payload}$ sont respectivement le temps d'émission du preamble et du payload définis aux équations 1.2 et 1.3.

$$T_{preamble} = (n_{preamble} + 4,25)T_{sym} \tag{1.2}$$

$$T_{payload} = n_{payload} * T_{sym} \tag{1.3}$$

Où $T_{sym} = \frac{1}{R_s}$, $n_{preamble}$ et $n_{payload}$ sont le nombre de symboles du preamble et du payload. $n_{preamble}$ est une valeur paramétrable du module radio et $n_{payload}$ est défini par l'équation 1.4.

$$n_{payload} = 8 + max\left(ceil\left[\frac{8PL - 4SF + 28 + 16CRC - 20IH}{4(SF - 2DE)}\right](CR + 4), 0\right) \tag{1.4}$$

où :

- PL est le nombre d'octets de la payload (1 à 255)
- SF est le spreading factor
- $CRC = 1$ si une CRC est présente $CRC = 0$ sinon
- $IH = 0$ si le header est présent, $IH = 1$ sinon
- $DE = 1$ si $LowDataRateOptimize = 1$, $DE = 0$ sinon. $LowDataRateOptimize$ permet d'optimiser la robustesse d'une transmission quand celle-ci est longue.
- CR prend la valeur 1 pour 4/5, 2 pour 4/6, etc

Cette dernière équation permet les déductions suivantes :

- Il y a toujours au minimum 8 symboles
- Le numérateur de la fraction quantifie un nombre de bits car PL est multiplié par 8
- La CRC a une taille de 16 bits
- Le header a une taille de 20 bits

1.3 IEEE 802.15.4e

802.15.4 est un protocole défini par IEEE en 2003. Il est destiné aux communications à faible débit réalisées par des dispositifs ayant une alimentation en énergie limitée. Ce protocole qui est un standard pour les réseaux PANs (Personal Area Networks) couvre la couche physique et MAC du modèle OSI. En 2012, IEEE 802.15.4e a été défini pour pallier certains problèmes de IEEE 802.15.4.

Types de noeuds et topologie

Cette norme définit deux types de noeuds : les Full Function Devices (FFD) qui peuvent être des coordinateurs de PAN, de simple coordinateurs ou de simple noeuds et les Reduced Function Devices (RFD) qui utilisent une implémentation réduite du protocole et ne peuvent être que de simples noeuds.

Ces noeuds peuvent former des réseaux suivant plusieurs topologies comme la topologie en étoile pour laquelle plusieurs RFD sont connectés à un FFD qui joue le rôle de coordinateur ou encore la topologie peer-to-peer pour laquelle les FFD sont connectés les uns aux autres.

Modes d'accès

La norme 802.15.4 définit en 2003 fournit deux modes d'accès : Beacon Enabled mode et Non-Beacon Enabled mode. Dans le premier, le réseau est synchronisé par des messages de contrôles (beacons) et une structure appelée Superframe.

Comme l'illustre la figure 1.3, cette Superframe est divisée en deux périodes : la période active et la période inactive. La période active est elle-même divisée en deux périodes : contention acces period (CAP) et contention free period (CFP). Dans la première, l'accès au canal se fait par l'algorithme slotted CSMA-CA (Carrier Sense Multiple Access with Collision Avoidance).

Dans la deuxième, l'accès au canal se fait par TDMA (Time Division Multiple Access). C'est à dire que les 7 slots de cette période sont attribués par le coordinateur aux noeuds ayant émis une requête durant la CAP pour l'utilisation d'un slot.

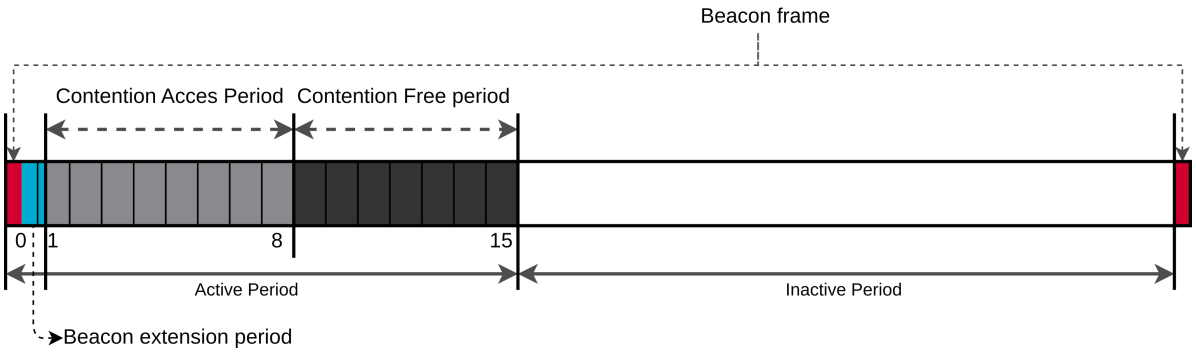


FIGURE 1.3 – 802.15.4 structure de la Superframe.

Dans le mode Non-Beacon Enabled, il n'y a pas de synchronisation. l'accès au canal se fait par l'algorithme Unslotted CSMA-CA.

Ces modes d'accès ont un certains nombres de limitations discutées dans l'article de Domenico De Guglielmo et ses co-auteurs [**?**] :

- Aucune garantie sur le délai maximal pour qu'un paquet puisse atteindre sa destination ne peut être fourni avec l'algorithme CSMA-CA.
- La fiabilité des communications est limitée par l'utilisation de l'algorithme slotted CSMA-CA qui offre un taux de transmission faible.
- Aucune protection contre les interférences en raison de l'utilisation d'un seul canal et en l'absence de mécanismes de sauts de fréquence (frequency hopping)

Ces limitations ont mené à la création de 802.15.4.e en 2012 qui redéfinit les protocoles MAC du standard. Ainsi, 5 modes de fonctionnement de la couche MAC sont introduits :

1. Time Slotted Channel Hopping (TSCH)
2. Deterministic and Synchronous Multi-channel Extension (DSME)
3. Low Latency Deterministic Network (LLDN)
4. Asynchronous multi-channel adaptation (AMCA)
5. Radio Frequency Identification Blink (BLINK)

D'après l'article de Domenico De Guglielmo et ses co-auteurs [**?**], le standard ne définit que brièvement AMCA et BLINK. LLDN est destiné aux réseaux à un seul saut et utilise un seul canal. Il n'est donc pas pertinent pour ce projet. DSME utilise le concept de multi-superframe semblable aux superframes de IEEE 802.15.4 mis à part la CFP qui divise chacun des 7 slots en plusieurs fréquences.

TSCH (Time Slotted Channel Hopping)

Ce mode de fonctionnement de la couche MAC, comme son nom l'indique, supporte à la fois les sauts en fréquence et des communications divisées en temps. Ces mécanismes réduisent efficacement les effets des interférences et les collisions ce qui améliore la fiabilité du réseau.

Slotframe

Dans ce mode, le concept de superframe de 802.15.4 est remplacé par le concept de slotframe. Une slotframe est un intervalle de temps qui est divisé en timeslots et se répète indéfiniment. Chaque timeslot permet à un noeud d'envoyer une trame et d'éventuellement recevoir son acquitement (ACK).

Chaque timeslot possède un identifiant appellé *Absolute Slot Number* (ASN) et un identifiant au sein de la slotframe appelé *Time Slot Number* (TSN).

TSCH permet l'utilisation de timeslots partagés et dédiés. Dans les timeslots partagés plusieurs noeuds peuvent communiquer. Dans ce cas, CSMA/CA est utilisé. Pour les timeslots dédiés, seuls deux noeuds peuvent communiquer. La figure 1.4 illustre une slotframe composée de trois timeslots. Dans cet exemple, on considère 3 noeuds : N, T et U. Chaque timeslot permet une communication entre deux noeuds. Par exemple le timeslot ayant comme TSN 0, permet à N de transmettre vers T.

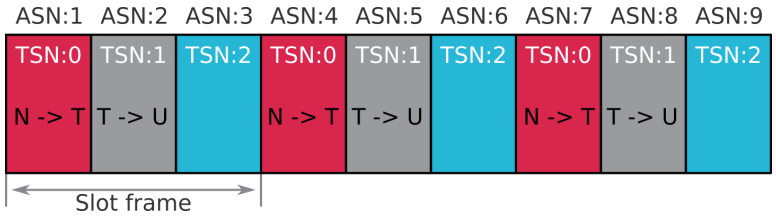


FIGURE 1.4 – Exemple de Slotframe.

Channel Hopping

TSCH peut utiliser 16 canaux différents. Un lien entre deux noeuds dans TSCH est alors défini par la paire (timeslot, canal). Ainsi, chaque slotframe est divisée par le nombre de canaux utilisés dans le réseau (Fig. 1.5). La fréquence f utilisée pour la communication entre deux noeuds est calculée comme suit :

$$f = F[(ASN + channelOffset)\%N_{channels}]$$

où $N_{channels}$ est le nombre de canaux utilisés pour le réseau et F peut être vue comme une table qui contient une permutation pseudo-aléatoire de canaux.

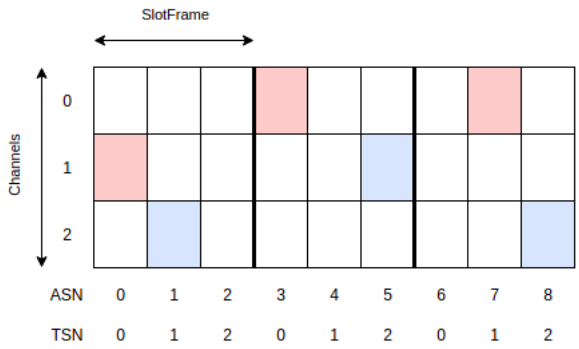


FIGURE 1.5 – Time Slotted Channel Hopping.

1.4 RPL

Routing Protocol for Low-Power and Lossy Networks [**?**] est un protocole de routage IPv6 destiné aux réseaux dont les noeuds sont contraints en énergie et dont les liens sont soumis à des pertes importantes de paquets (Low-power and Lossy Networks, LLNs).

D'après l'article de Tripathi, Oliveira et Vasseur [**?**], après avoir été étudiés par le groupe de travail IETF roll (Routing Over Low power and Lossy networks), il s'est avéré que les protocoles tel que OSPF, AODV ou OLSR ne satisfont pas toutes les exigences des LLNs. C'est pour cette raison que ce groupe de travail a introduit RPL. Il s'agit d'un protocole à vecteur de distance. De plus, RPL est un protocole proactif, c'est à dire que les routes sont établies avant qu'elles ne soient nécessaires.

RPL sépare le traitement et la transmission des paquets de contrôle de l'optimisation de l'objectif de routage. Cela permet de l'adapter à un large éventail d'applications des LLNs comme la domotique, l'automatisation industrielle ou encore la récolte de données dans divers environnements.

Cette section décrit le fonctionnement de RPL. Cette description est basée sur le RFC 6550 [**?**].

Topologie

RPL construit un DODAG (Destination Oriented Directed Acyclic Graph) qui est un graphe dirigé acyclique (DAG) ayant une seule racine (Fig. 1.6). De par cette architecture, RPL est adapté aux applications de collecte de données ce qui est le principal objectif de ce projet. En effet, la route d'un noeud à la racine est facilement établie car tous les noeuds du chemins envoient les données à leur parent.

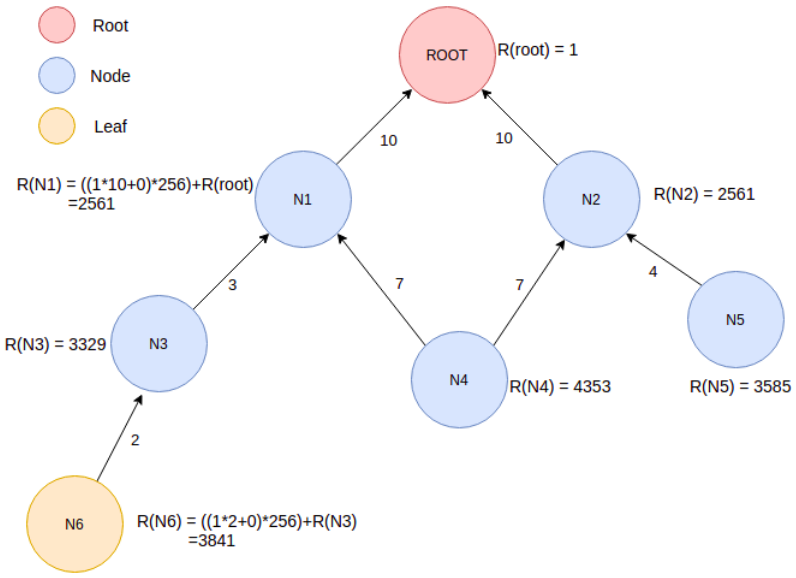


FIGURE 1.6 – DODAG.

Fonctions objectif

Une fonction objectif (OF) définit comment plusieurs métriques sont utilisées pour calculer le rang d'un noeud. Celui-ci détermine la position du noeud dans le DODAG par rapport aux autres noeuds. Le rang augmente strictement dans le sens descendant et diminue strictement dans le sens montant. Ainsi, pour un noeud n, la propriété $rang(n) > rang(parent(n))$ est toujours vérifiée. La figure 1.6 illustre un DODAG avec des valeurs de rang calculées avec la fonction objectif OF0, ses paramètres par défaut et des coûts de liens attribués administrativement. Sur cet exemple, la racine du DODAG a comme rang, la valeur par défaut ROOT_RANK = 1 définie dans le RFC 6550.

Les fonctions objectif OF0 et MRHOF, implémentées dans Contiki, sont décrites sur base du RFC 6552 [**?**] et du RFC 6719 [**?**].

OF0

Objective Function Zero est une fonction dont l'objectif est de choisir un parent qui permettra à un noeud d'être le plus proche possible la racine du DODAG. Le calcul du rang d'un noeud N, $R(N)$, est décrit par l'équation 1.5. Il est calculé pour chaque parent potentiel. Le parent préféré choisi sera alors le parent potentiel qui implique le plus petit $R(N)$.

Le calcul du rang commence par le calcul de la variable *step_of_rank*

(Sp) qui est basée sur les propriétés et métriques du lien avec le parent potentiel. Le RFC recommande de baser ce calcul sur des propriétés dynamiques comme la métrique *expected transmission count (ETX)* décrit dans l'article de Couto, Aguayo, Bicket et Morris [**?**], qui est une métrique qui permet de minimiser le nombre de transmissions requises pour transmettre un paquet. L'utilisation de propriétés dynamiques est recommandée car utiliser des propriétés statiques, comme un coût administratif, revient à avoir un rang analogue au nombre de sauts.

Sp est ensuite mutlipliée par la variable *rank_factor* (Rf) qui est utilisée pour multiplier son effet dans le calcul de *rank_increase*. Rf doit être un entier strictement positif et sa valeur par défaut est 1.

Sr est un terme plus petit ou égal à la variable *stretch_of_rank* qui est l'augmentation maximum de *step_of_rank* avec le parent.

La variable *rank_increase* est finalement multipliée par le facteur *MinHopRankIncrease* qui est défini par le RFC 6550 [**?**] comme l'augmentation minimum du rang entre un noeud et n'importe lequel de ses parents. Sa valeur par défaut est DEFAULT_MIN_HOP_RANK_INCREASE qui a une valeur de 256.

Le calcul du rang est la somme du rang du parent potentiel P, $R(P)$, et de *rank_increase*.

$$\begin{aligned} rank_increase &= (Rf * Sp + Sr) * MinHopRankIncrease \\ R(N) &= R(P) + rank_increase \end{aligned} \tag{1.5}$$

MRHOF

Minimum Rank with Hysteresis Objective Function est une fonction dont l'objectif est de sélectionner les routes qui minimisent une métrique additive en utilisant l'hystérésis pour réduire les changements de parents en réponse à de petites variations de la métrique.

Pour cela, un noeud va calculer le coût des chemins avec chaque candidat parent. Ce coût sera la somme de deux éléments :

- La valeur de la métrique utilisée contenue dans les DIOs d'un candidat parent
- La valeur de la métrique pour le lien entre ce noeud et le candidat parent

Si le coût d'un chemin est plus grand qu'une constante MAX_LINK_METRIC dont la valeur recommandée par le RFC 6719 [**?**] est 512, le noeud va exclure le candidat parent utilisant ce lien. Un noeud va choisir comme parent le candidat parent pour lequel le coût du chemin est le plus petit. Le changement

de parent ne s'effectue que si la condition suivante est satisfaite :

$$|cost - old_cost| > \text{PARENT_SWITCH_TRESHOLD}$$

Le RFC 6719 recommande 192 comme valeur pour *PARENT_SWITCH_TRESHOLD* si la métrique utilisée est ETX.

Messages RPL

RPL défini quatre types de mesages de contrôles : DIO, DIS, DAO et DAO-ACK. Ces messages de contôles sont des messages ICMPv6. Le RFC 6550 [**?**] indique que ces messages ont la portée d'un lien, à l'exception des DAO/DAO-ACK qui, dans le mode de fonctionnement *non-storing*, sont envoyés en unicast et donc utilisent des adresses unique-local ou global. Pour tous les autres messages, les adresses sources sont des adresses link-local et les adresses de destination sont soit link-local, soit l'adresse multicast pour tous les noeuds RPL qui est `ff02::1a`.

DIO

Les DIOs (DODAG Information Object) annoncent des informations sur le DODAG qui permettent aux noeuds de découvrir une instance RPL, de sélectionner un parent ou encore de maintenir le DODAG. Un DIO, illustré à la figure 1.7, inclu notamment les champs suivants :

- InstanceID : Identifie l'instance RPL du DODAG
- Rank : Le rang du noeud qui émet le DIO
- MOP (Mode of Operation) : Le mode d'opération de l'instance RPL (c.f. section 1.4 Modes de fonctionnements).
  0. Pas de routes descendantes
  1. Non-storing mode
  2. Storing mode sans multicast
  3. Storing mode avec multicast
- DODAGID : Adresse IPv6 définie par la racine qui identifie le DODAG

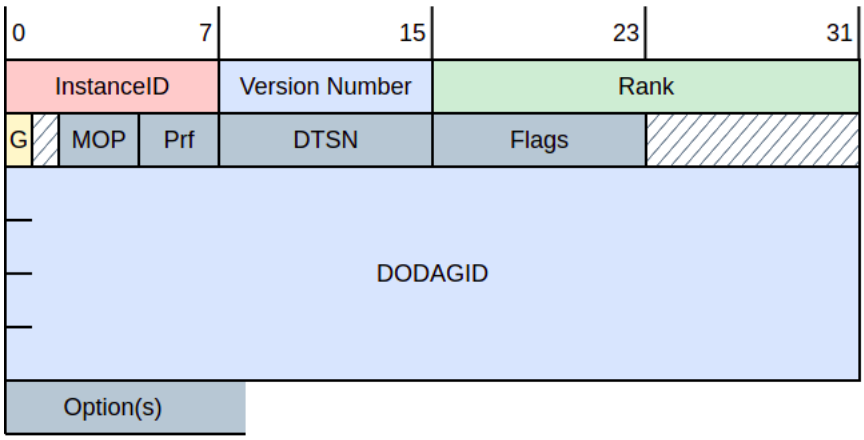


FIGURE 1.7 – DODAG Information Object.

Les DIOs sont transmis en utilisant l'algorithme Trickle [**?**]. Cet algorithme permet de réduire la quantité de messages envoyés en réduisant les transmissions quand les conditions sont stables et en les augmentant lorsque qu'un changement est observé dans le réseau.

DIS

Les DODAG Information Solicitation (DIS) sont utilisés pour solliciter un DIO d'un noeud RPL.

DAO

Les DAOs (Destination Advertisement Object) sont utilisés pour établir les routes descendantes. Ils sont donc envoyés vers le haut du DODAG. Le format du DAO est illustré à la figure 1.8. Un DAO est composé des champs suivants :

- InstanceID : identifie l'instance du DODAG
- K : indique que le destinataire doit répondre avec un DAO-ACK
- D : indique que le DODAGID est présent
- DAOSequence : Incrémenté à chaque DAO d'un noeud et répété dans le DAO-ACK
- DODAGID (optionnel)

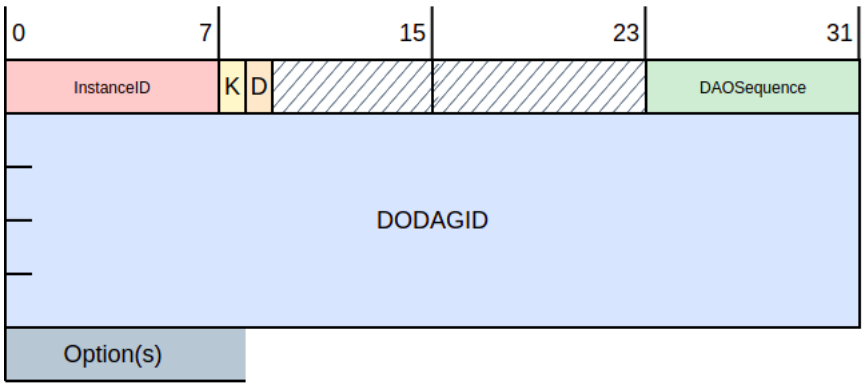


FIGURE 1.8 – DODAG Information Object.

DAO-ACK

Le format d'un DAO-ACK est similaire à celui d'un DAO. Il n'est donc pas utile de le décrire.

Construction du réseau

La construction d'un DODAG est réalisée via l'échange de messages DIOs pour les routes montantes et de DAOs pour les routes descendantes. La figure 1.9 illustre l'échange de ces messages.

Routes Montantes

Les routes montantes sont construites et maintenues avec les DIOs. L'algorithme de construction du DODAG est le suivant :

1. Les noeuds étant configurés comme racine d'un DODAG diffusent des DIOs en multicast à tous les noeuds RPL
2. Les noeuds voulant rejoindre un DODAG écoutent ces DIOs et utilisent leurs informations pour rejoindre le DODAG (i.e. sélectionner un parent) ou maintenir le DODAG existant en accord avec la fonction objectif
3. Les noeuds rajoutent des entrées dans leur table de routage pour les destinations spécifiées dans le DIO via leurs parents
4. Une fois qu'un noeud a rejoint le DODAG, il diffuse à son tour des DIOs pour que d'autres noeuds puissent rejoindre le DODAG.

Les noeuds qui opèrent comme feuille peuvent émettre des DIOs mais le rang contenu dans le DIO doit être l'infini. De cette manière, une feuille ne sera pas choisie comme parent.

Si une adresse de destination n'appartient pas au DODAG, la racine du DODAG peut transférer les paquets à l'extérieur de ce réseau, ou, si elle n'en est pas capable, les ignorer.

Routes Descendantes

Les routes descendantes sont construites et maintenues avec les DAOS. L'échange des DAOs diffère selon le mode de fonctionnement (MOP) du réseau. Ces différences sont détaillées en section 1.4 Modes de fonctionnements.

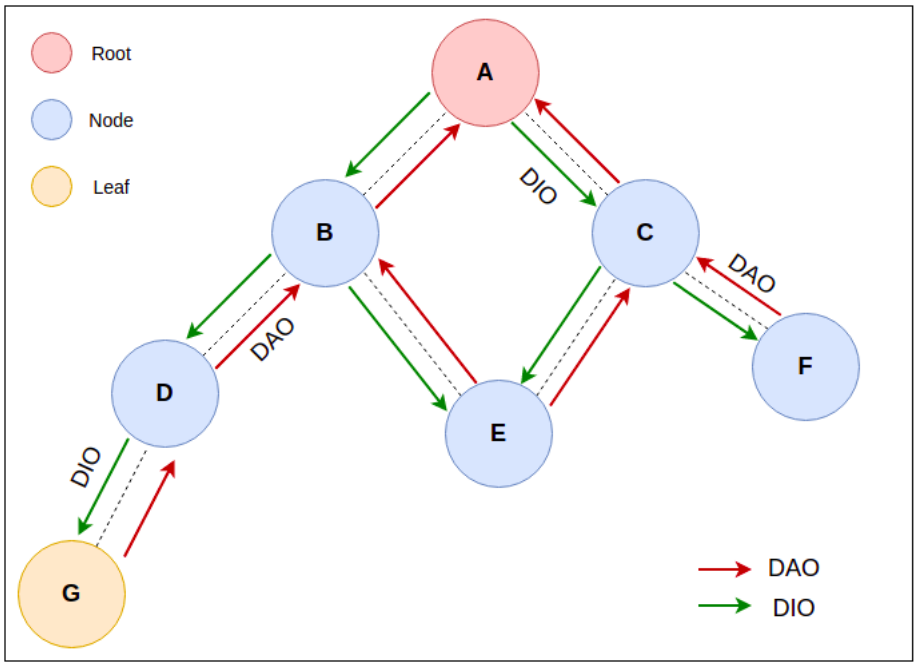


FIGURE 1.9 – Propagation des DIOs et DAOs.

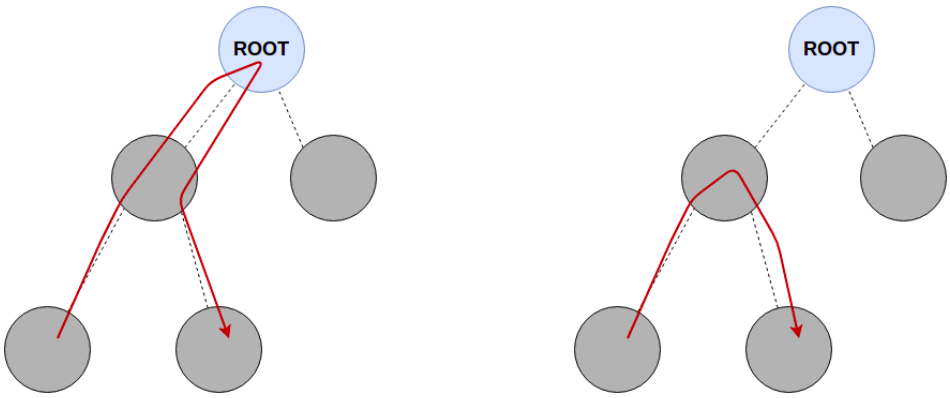
Modes de fonctionnements

Le mode de fonctionnement d'une instance RPL est défini administrativement et est annoncé par la racine. Les modes suivants sont disponibles :

- **Pas de routes descendantes**
  RPL ne maintient pas de routes descendantes. Dans ce MOP, les DAOs ne sont pas émis par les noeuds d'un DODAG, et les noeuds ignorent les DAOs.
- **Non-storing mode**
  Dans ce mode, les DAOs sont envoyés en unicast à la racine du DODAG. Les noeuds ne stockent pas de routes descendantes.

L'envoi des paquets le long de ces routes se fait donc par source-routing. Cela signifie qu'un en-tête de routage est ajouté aux datagrammes par la racine. Le RFC 6554 [**?**] défini les mécanismes liés à ce type d'en-tête ainsi que son format qui est similaire à celui de l'en-tête de routage IPv6 [**?**] (de Type 0). Toutefois, il introduit en plus un mécanisme qui permet de compresser les adresses quand elles ont toutes le même préfixe.
Dans ce mode, les paquets remontent donc jusqu'à la racine, qui ajoute l'en-tête de routage pour ensuite redescendre jusqu'au noeud de destination (Fig. 1.10a).

- **Storing mode**
  Le storing mode peut être utilisé avec ou sans multicast. Dans ce mode, les DAOs sont envoyés en unicast par les noeuds à leur parent. Les paquets remontent jusqu'à un ancêtre commun avec la destination avant de redescendre vers celle-ci (Fig. 1.10b). Les noeuds stockent les routes de leur sous DODAG. Ainsi, chaque saut dans un chemin examine sa table de routage pour choisir le saut suivant.



(a) Non-Storing. (b) Storing.

FIGURE 1.10 – Chemin d'un paquet en fonction du MOP.

Un noeud voulant rejoindre un DODAG doit être compatible avec le MOP du DODAG. Si ce n'est pas le cas, il doit rejoindre le DODAG comme feuille.

1.5 RTOS

Un RTOS (Real Time Operating System) est un sytème d'exploitation temps réel principalement destiné aux systèmes embarqués. Il permet

d'abstraire le matériel en fournissant une couche logicielle facilitant le développement d'une application. Un RTOS permet également de découper une application en tâches qui sont gérées par l'ordonnanceur.

Le RTOS choisi doit supporter la plateforme utilisée. Sa pile réseau doit comprendre les protocoles suivants : 802.15.4 pour la couche physique, TSCH avec un ou plusieurs algorithme d'ordonnancement pour la couche MAC, le support d'IPv6 et d'un algorithme de routage comme RPL pour la couche réseau et enfin, pour le transport, TCP et/ou UDP.

Une implémentation de LoRa n'est pas nécessaire car les communications LoRa sont réalisées via le RN2483 qui est controlé par UART.

Pour effectuer ce choix, les RTOS suivants ont été comparés : Contiki OS, FreeRTOS et RIOT OS.

Contiki OS

Le développement public de Contiki [**?**] a débuté en octobre 2012 [1] dans un premier repository : **contiki** [**?**]. Un fork de contiki a démarré en mai 2017 sous le nom de **contiki-ng** [**?**]. Comme le premier développement n'est plus maintenu, c'est ce dernier qui est utilisé pour la comparaison et qui est dénommé par la suite "Contiki".

L'implémentation des processus est basée sur la librairie Protothreads [**?**] qui abstrait la gestion de la programmation événementielle par des protothreads dont l'utilisation est similaire aux threads. L'ordonnanceur de cette libraire est coopératif, c'est à dire que contrairement à un ordonnanceur préemptif, il ne va jamais forcer un changement de contexte (ensemble des ressources nécessaires à l'éxécution d'un processus) d'un processus à un autre. Le changement de contexte ne s'effectue que quand un processus rend volontairement le contrôle à l'ordonnanceur.

Ce RTOS open-source et multi-plateforme implémente toute une série de protocoles de communications basse énergie tels que IEEE 802.15.4, 6TiSCH, IPv6/6LoWPAN et RPL. En plus de 6TiSCH, un ordonnanceur TSCH, Orchestra, est implémenté.

Contiki supporte plusieurs plateformes en fournissant pour chacune un BSP (Board Support Package). Pour le Zolertia RE-Mote, le BSP supporte notamment la gestion de l'alimentation, des leds, des deux antennes ainsi que des interfaces UART, SPI et I2C.

Ce RTOS est également accompagné de Cooja, un simulateur réseau qui permet de simuler les communications entre plusieurs noeuds l'utilisant.

1. Date de création du repository Github.

FreeRTOS

D'après le site officiel de FreeRTOS [**?**], ce RTOS est développé depuis 15 ans. Le repository initial [**?**] a été créé en 2004.

L'ordonnanceur de tâches de FreeRTOS peut être configuré comme coopératif ou préemptif via le flag *configUSE_PREEMPTION* du fichier de configuration.

Ce RTOS, également open-source et multi-plateforme, offre des librairies divisées en trois catégories :

- **FreeRTOS+** qui contient notamment la librairie TCP/IP et les protocoles applicatifs MQTT et HTTP,
- **AWS IoT Libraries** qui fourni des librairies permettant la connectivité avec Amazon Web Services (AWS)
- **FreeRTOS Labs** qui contient des librairies complètement fonctionelles mais qui sont encore en cours d'amélioration comme le support d'IPv6, LoRaWan ou le support de plusieurs interfaces réseau

FreeRTOS n'a pour le moment pas été porté pour le Zolertia RE-Mote ou le CC2538. Fournir un tel support est un investissement conséquent et n'est pas l'objectif de ce mémoire.

RIOT OS

Le développement publique de RIOT [**?**] a débuté en décembre 2012[1]. RIOT est un RTOS qui utilise des standards comme la programmation en C++ ou une compatibilité POSIX pour faciliter le développement d'applications IoT.

L'ordonnanceur de RIOT est *tickless*, c'est à dire qu'il n'utilise pas un timer déclenché périodiquement pour effectuer les changements de contexte. L'ordonnanceur est préemptif et basé sur des priorités. Les changements de contexte sont initiés lors d'interruptions, volontairement ou quand une opération bloquante a lieu.

La pile réseau de RIOT comporte notamment les protocoles 6LoWPAN, IPv6, RPL, LoRaWan et 802.15.4. Ce RTOS intègre la pile réseau OpenWSN [**?**] mais cette intégration est pour le moment expérimentale.

OpenWSN est une implémentation open-source d'une pile réseau destinée à l'IoT. Cette pile réseau comprend les protocoles IEEE802.15.4e, 6LoWPAN, RPL, UDP et CoAP. L'objectif de cette implémentation est qu'elle puisse être utilisée sur une variété de RTOS et de plateformes.

D'après le site officiel, ce RTOS supporte 229 carte de développement et 64 CPU dont le Zolertia RE-Mote.

Conclusion

Le table 1.1 résume la comparaison de ces RTOS avec les critères les plus importants pour ce mémoire. Dans cette comparaison, l'utilisation d'OpenWSN n'a pas été prise en compte pour RIOT OS car son intégration reste expérimentale.

RTOS	802.15.4	TSCH	ord. TSCH	IPV6	Routage IP	comp. RE-Mote
Contiki OS	$\surd$	$\surd$	6Tisch, Orchestra	$\surd\surd$	RPL	$\surd$
FreeRTOS	$\times$	$\times$	$\times$	$\surd$	$\times$	$\times$
RIOT OS	$\surd$	$\times$	$\times$	$\surd\surd$	RPL	$\surd$

Table 1.1 – Comparatif de différents RTOS.

Le RTOS choisi est Contiki OS. Il est choisi pour sa maturité, la prise en charge du Zolertia RE-Mote et sa pile réseau complète et stable.

Chapitre 2

Architecture

2.1 Topologie

Cette section décrit et motive la topologie et les types de noeuds du réseau de capteurs hybride IEEE 802.15.4-LoRa. Cette topologie a été choisie pour correspondre au mieux au cas d'utilisation décrit dans l'introduction.**TODO: REF INTRO + à savoir ...**

Types de noeuds

Le protocole élaboré pour ce réseau hybride définit 3 types de noeuds :

- La **racine LoRa** est la passerelle entre le réseau et un réseau IP externe. Elle possède donc une interface LoRa et par exemple une interface Wi-Fi, Ethernet ou 3G/4G. La radio LoRa est toujours allumée car ce noeud n'est pas contraint en énergie.
- Les **racines RPL** sont les racines des réseaux RPL. Elles possèdent une interface LoRa pour communiquer avec la racine LoRa et une interface 802.15.4 pour communiquer avec les noeuds du réseau RPL.
- Les **noeuds RPL** sont des noeuds des réseaux RPL.

Topologie

Comme l'illustre la figure 2.1 la topologie choisie est une topologie mixte.

D'abord un ensemble de réseaux RPL possèdant chacun un préfixe de sous-réseau IP attribué par la racine LoRa (par exemple $0x02$ ou $0x04$ sur la figure 2.1), dans lesquels les communications se font par IEEE 802.15.4. Comme RPL est utilisé, chaque réseau forme un DODAG.

Ensuite, chaque réseau RPL est connecté à la racine LoRa via sa racine RPL. Les liens entre les racines RPL et la racine LoRa forment une topologie en étoile.

Ce choix de topologie a comme avantage de simplifier l'administration du réseau. En effet, un réseau RPL pourrait, par exemple, correspondre à une parcelle de terrain. Comme cette topologie est composée d'un ensemble de réseau RPL, la défaillance d'une racine RPL pourrait impliquer la déconnexion d'un réseau RPL, contrairement à une topologie qui utilise un seul réseau RPL où la défaillance de la racine pourrait impliquer la déconnexion de l'ensemble des noeuds.

L'inconvéninent de cette topologie est que la racine LoRa constitue un seul point de défaillance. Une solution plus robuste qui a été envisagée, est un réseau maillé pour les communications LoRa. Cependant, une telle architecture est difficile à mettre en oeuvre car le protocole MAC a établir pour les liens LoRa est plus complexe et nécessite un protocole de routage.

L'essentiel du travail consiste donc à définir et mettre en place un protocole MAC utilisant LoRa pour les communications entre les racines RPL et la racine LoRa ainsi que les mécanismes permettant de passer d'un réseau à un autre. Le routage et le transfert des paquets dans les réseaux RPL est quand à lui assuré par RPL et IPv6. Les sections suivantes décrivent le protocole MAC mis en place pour les communications LoRa.

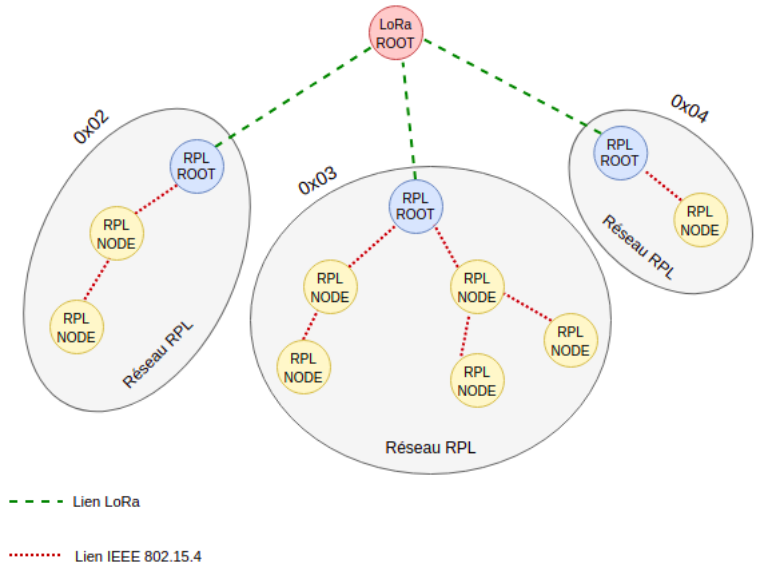


FIGURE 2.1 – Topologie du réseau hybride.

2.2 Format d'adresse

Cette section décrit les possibilités envisagées pour le choix du format des adresses utilisées pour les communications LoRa et motive le choix d'un format. Dans chaque hypothèse, un réseau RPL possède un préfixe de sous-réseau qui lui est propre.

Adresse IPv6

La première solution envisagée est d'utiliser les adresses IPv6 des noeuds. Une racine RPL peut ainsi vérifier si un paquet est destiné à son réseau et la racine LoRa à elle même, via le préfixe de sous réseau de l'adresse de destination. Cette solution est la plus simple à mettre en oeuvre car elle ne nécéssite pas de table de routage ou de processus de conversion d'adresse. Cette approche a néanmoins le désavantage que les adresses IP de 128 bits sont transportées sur un lien LoRa ce qui représente un temps de transmission important.

Ainsi, sur l'exemple de la figure 2.1, un noeud RPL du réseau ayant le préfixe $0x02$ aurait par exemple l'adresse suivante :
`fd00::2:eeff:212:4b00:60d:611b`.

Préfixe et *node-id*

Une seconde approche est d'utiliser un préfixe et le *node-id*. Ce dernier est un identifiant de noeud sur 16 bits utilisé dans Contiki. Son calcul, basé sur les deux derniers octets de l'adresse MAC du noeud (*link-layer address*) est le suivant :

$$node_id = link_addr[s-1] + (link_addr[size-2] << 8)$$

où s est la taille de la *link-layer address*.

Le format d'adresse utilisé serait alors un préfixe sur 8 bits et le *node-id* sur 16 bits ce qui fait un total de 24 bits.

L'avantage de cette solution est que la taille des adresses est réduite d'un facteur de 18,75%. Néanmoins, un processus de conversion est nécessaire ainsi qu'une table de conversion d'un *node-id* vers une adresse MAC pour chaque noeud RPL. En effet, le calcul du *node-id* ne permet pas de reconstituer l'adresse MAC de 64 bits à partir de celui-ci. Un autre inconvénient est que l'utilisation du préfixe de 8 bits limite le nombre de réseaux RPL à 256, ce qui reste néanmoins acceptable.

Ainsi, sur l'exemple de la figure 2.1, un noeud RPL dont la valeur du *node-id* est 24859, appartenant au réseau ayant le préfixe $0x02$, aurait par exemple l'adresse suivante : `2:24859`.

Préfixe et adresse MAC

La troisième approche envisagée a pour objectif de résoudre le problème de la conversion *node-id* $\leftrightarrow$ adresse MAC, en remplaçant le *node-id* par l'adresse MAC d'un noeud.

Cette solution permet une conversion d'adresse simple et la taille des adresses est de 72 bits (64 bits de l'adresse MAC et 8 bits du préfixe).

Ainsi, sur l'exemple de la figure 2.1, un noeud RPL dont l'adresse MAC est `0012.4b00.060d.611b`, appartenant au réseau ayant le préfixe $0x02$, aurait par exemple l'adresse suivante : `2:0012.4b00.060d.611b`.

Solution retenue

La solution retenue est l'utilisation du préfixe de 8 bits et du *node-id* car c'est la solution qui offre les adresses les plus courtes. Pour éviter l'utilisation d'une table de conversion *node-id*$\leftrightarrow$ adresse MAC, une meilleure solution est d'utiliser les adresses MAC des noeuds et de modifier les 6 premiers octets de cette dernière. Cela permet de reconstituer une adresse IPv6 simplement, mais cela réduit le nombre d'adresses possibles qui reste cependant largement suffisant pour ce projet.

Plus précisement, les adresses IPv6 seront construites tel qu'illustré par la figure 2.2.

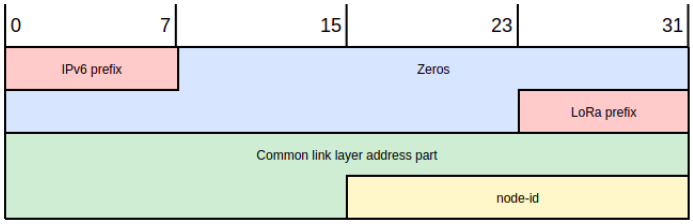


FIGURE 2.2 – Format d'adresse IPv6 à partir d'une adresse LoRaMAC.

Le préfixe IPv6 choisi est celui des adresses IPv6 unique local unicast. Ce type d'adresse est spécifié par le RFC 4193 [**?**]. Car, comme indiqué dans le RFC, ces adresses sont destinées à des communications locales et sont destinées à être routées dans un réseau limité, ce qui correspond à notre réseau. Comme défini par le RFC, le préfixe IPv6 est donc `FC00::/7` suivi

d'un bit à 1 qui est destiné à une utilisation future. Les adresses commencent donc par `FD`.

Ce préfixe est suivi de 6 octets de zéros, du préfixe LoRa sur 1 octet, de la partie commune de l'adresse MAC sur 6 octets et finalement du *node-id* du noeud.

L'adresse de l'interface LoRa de la racine LoRa est quand à elle configurée administrativement.

2.3 Trames LoRaMAC

Le protocole MAC mis en place pour les communications LoRa utilise un seul format de trames illustré à la figure 2.3.

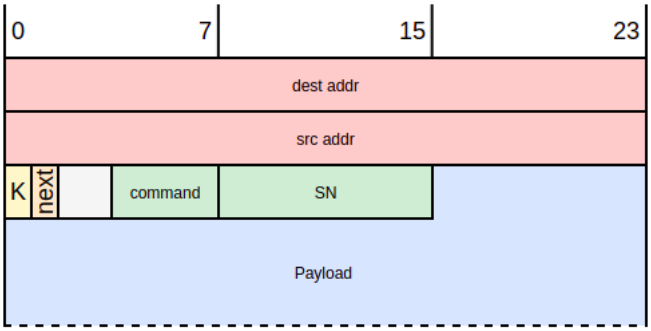


FIGURE 2.3 – Format de trame LoRaMAC.

Les trames LoRaMAC sont composées des champs suivants :

- **dest_addr** : l'adresse de destination (24 bits)
- **src_addr** : l'adresse source (24 bits)
- **K** : un flag qui indique si la trame nécessite un acquittement (1 bit)
- **next** : un flag qui indique si une autre trame suit (1 bit)
- **reserved** : 2 bits réservés pour évolution future
- **command** : la commande MAC (4 bits).
  Les 5 commandes disponibles ainsi que leurs valeurs sont reprises dans la table 2.1. Il est possible d'ajouter 10 commandes supplémentaires pour des usages futurs.
- **SN** : le numéro de séquence de la trame (8 bits)
- **Payload** : la payload ayant une taille maximale de 247 octets. Cette limitation provient de la radio LoRa utilisée, le RN2483, qui limite les transmissions à 255 octets, valeur de laquelle est soustrait les 8 octets des champs précédents.

Valeur	Commande	description	payload
0	JOIN	requête pour rejoindre le réseau	×
1	JOIN_RESPONSE	réponse à la commande JOIN	préfixe du sous réseau
2	DATA	transport de données	données
3	ACK	acquittement	×
4	QUERY	demande de trafic descendant	×

TABLE 2.1 – Description des commandes LoRaMAC.

2.4 LoRaMAC

Cette section décris le protocole MAC mis en place pour les communications LoRa. Ce protocole est donc utilisé pour les communications entre la racine LoRa et les racines RPL. Ces dernières étant contraintes en énergie, leur radio LoRa n'est pas tout le temps allumée. Le protocole prend donc en compte cette contrainte.

Etats d'une Racine RPL

Les racines RPL utilisent 4 états illustrés dans le diagramme d'état de la figure 2.4. A l'initiation du réseau, une racine RPL se trouve dans l'état *alone*. Elle demande ensuite son préfixe à la racine LoRa. Si la transmission échoue trop de fois, elle entre dans l'état *sleep* dans lequel ses radios sont éteintes, jusqu'à ce que l'évènement *wake up* soit déclenché pour retourner dans l'état *alone*. Si la transmission est réussie, et donc que le préfixe est reçu, la racine RPL se trouve dans l'état *ready*. Quand une racine RPL envoie une trame qui nécessite un acquittement, tant que celui-ci n'est pas reçu elle se trouvera dans l'état *wait_response*. Une fois l'acquittement reçu, elle retournera dans l'état *ready*.

Les trames qui doivent être envoyées le seront en fonction de l'état dans lequel se trouve une racine RPL. Dans l'état *ready*, les trames de données peuvent être envoyées, mais ce n'est pas le cas dans les états *alone* et *wait_response*. En effet, dans le premier état, la racine RPL ne peut pas envoyer de données car elle n'a pas encore rejoint le réseau et dans le second, elle attend une réponse.

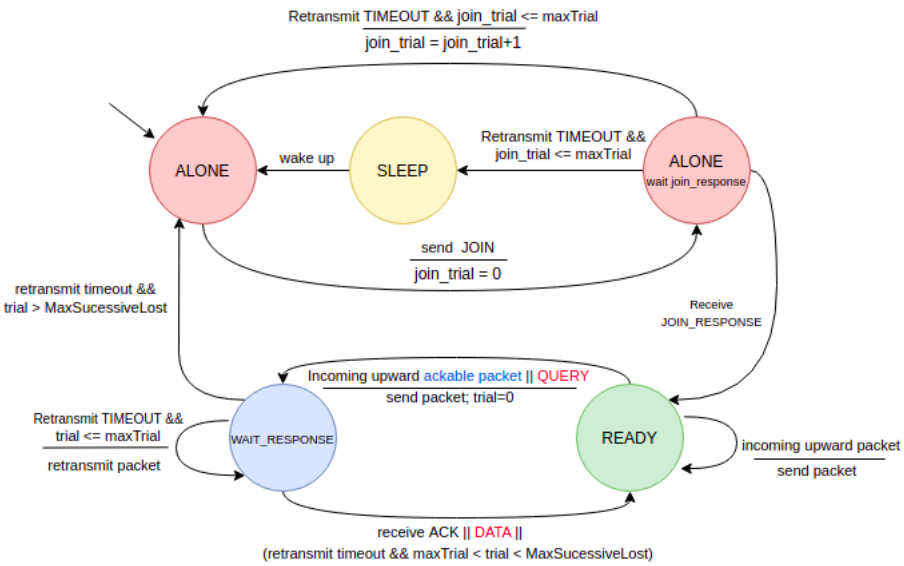


FIGURE 2.4 – Diagramme d'état des racines RPL.

Etats de la racine LoRa

Le diagramme d'état de la racine LoRa pour chaque racine RPL qui a rejoint le réseau est illustré à la figure 2.5. Une fois que la racine RPL (enfant de la racine LoRa) a rejoint le réseau, l'état est READY. Cela signifie que la racine LoRa est prête à recevoir des données de cet enfant. Si elle reçoit une trame DATA, les données sont transmises à la couche supérieure. Elle peut donc rester dans le même état. Elle reste aussi dans l'état READY si elle reçoit une QUERY et qu'il n'y a aucune données disponible pour cet enfant. En revanche, si des données sont disponibles, la racine LoRa entre dans l'état TRANSMIT dans lequel elle va envoyer tous les paquets disponibles pour cet enfant. Une fois dans l'état TRANSMIT la racine LoRa envoie un paquet et bascule dans l'état WAIT TIMER dans lequel elle attend une possible demande de retransmission de son enfant. Si une retransmission est demandée, le timer est réinitialisé. Une fois le timer expiré, elle retourne dans l'état TRANSMIT. S'il n'y a plus de données pour cet enfant, elle retourne dans l'état READY.

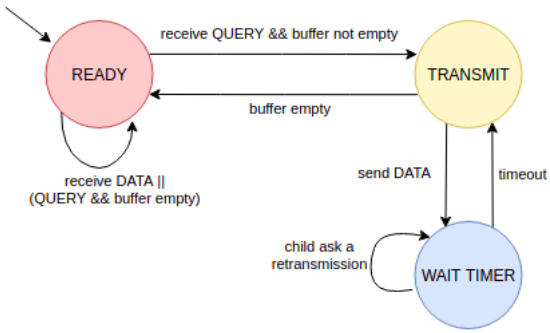


FIGURE 2.5 – Diagramme d'état de la racine LoRa.

Construction du réseau

A l'initialisation du réseau, une racine RPL doit envoyer une trame avec la commande JOIN. Une fois reçue par la racine LoRa, celle-ci va répondre avec une trame JOIN_RESPONSE qui fait office d'acquittement de la trame JOIN et dont la payload est le préfixe du réseau de la racine RPL qu'elle doit diffuser dans son réseau.

Si la trame JOIN ou la trame JOIN_RESPONSE ne sont pas reçues, la racine RPL retransmettra la trame JOIN à l'expiration d'un timer *retransmit_timer* déclenché à l'envoi de la trame JOIN. Cette retransmission se répète au maximum MAX_ATTEMPT fois. Si ce seuil est dépassé, la communication est considérée comme échouée. Dans ce cas, la racine attend w secondes avant de répéter le processus. w est défini par l'équation 2.1, dans laquelle, r est le temps ajouté au *retransmit_timeout* qui est le temps après lequel le timer *retransmit_timer* expire. r est le modulo *retransmit_timeout* d'un nombre *rand* qui doit être choisi aléatoirement.

$$\begin{aligned} r &= rand\%retransmit_timeout \\ w &= (retransmit_timeout + r)\%60 \end{aligned} \tag{2.1}$$

Une fois le préfixe reçu, la racine RPL peut initialiser le réseau RPL et y diffuser le préfixe.

Le diagramme de séquence de la figure 2.6 illustre le processus de construction du réseau.

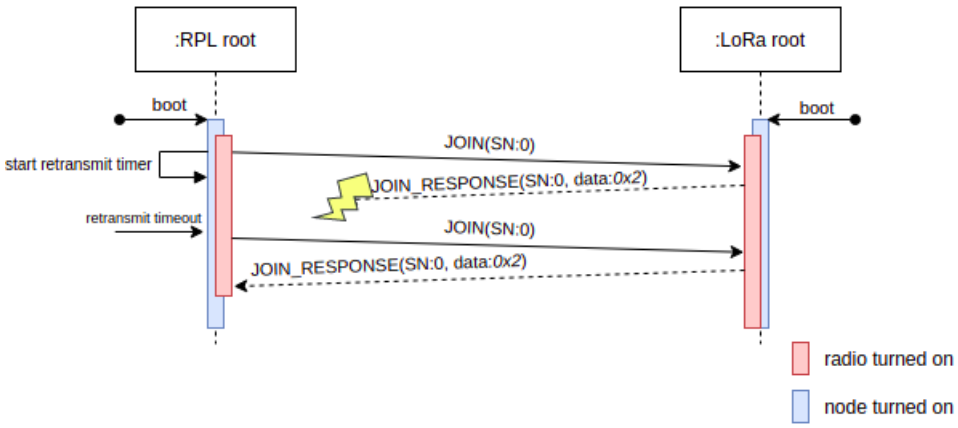


FIGURE 2.6 – Diagramme de séquence de la construction du réseau.

Communications montantes

Lorsqu'un paquet est destiné à une adresse IPv6 externe au réseau d'une racine RPL, il va remonter jusqu'à celle-ci, qui va ensuite l'envoyer à la racine LoRa.

Si la trame envoyée à la racine LoRa nécessite un acquittement, la racine RPL n'enverra pas d'autres trames tant que l'acquittement n'a pas été reçu (état *wait_response*). A l'envoi de cette trame, le timer *retransmit_timer* est déclenché. S'il expire avant que l'acquittement ne soit reçu, la trame sera retransmise pour un maximum de 3 fois. Arpès 3 transmissions qui n'ont pas réussi, la racine RPL retourne dans l'état READY et le paquet est perdu.

Si la trame envoyée à la racine LoRa ne nécessite pas d'acquittement, la réception de cette trame n'est pas garantie et la racine RPL peut directement transmettre les trames suivantes s'il y en a.

Le diagramme de séquence de la figure 2.7 illustre le déroulement des communications montantes.

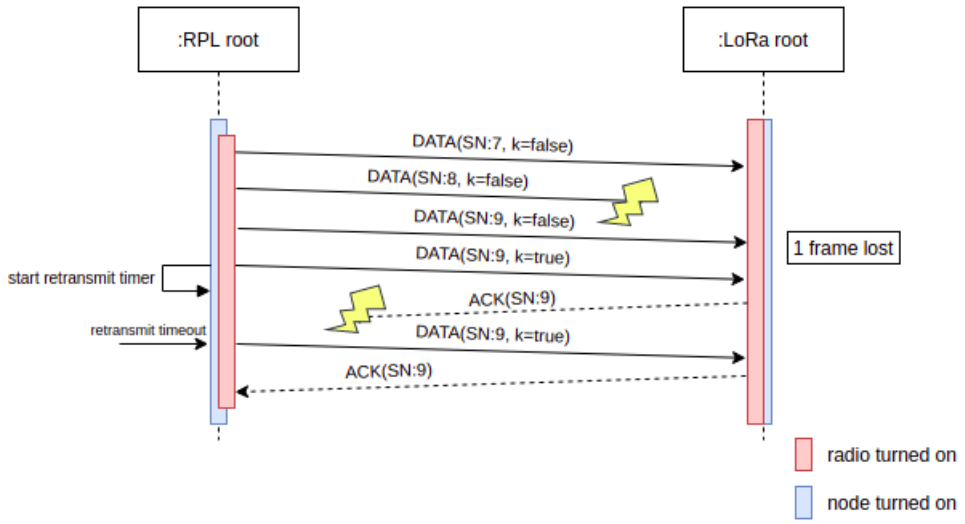


FIGURE 2.7 – Diagramme de séquence du trafique montant.

Communications descendantes

La racine LoRa possède un buffer pour chaque réseau RPL dans lequel elle accumule les paquets destinés à ce réseau. Pour chaque racine RPL, lors de l'expiration, à intervalle régulier, du timer *query_timer*, une trame avec la commande QUERY est transmise à la racine LoRa.

La réponse à cette trame doit être un acquittement, si la racine LoRa n'a pas de paquets pour le réseau de cette racine RPL, ou une trame avec la commande DATA si un ou plusieurs paquets sont disponibles.

Si la racine LoRa possède plusieurs paquets à destination du même réseau RPL, la valeur du flag *next* doit être à 1 pour toutes les trames envoyées à l'exception de la dernière.

Comme pour les communications montantes, les retransmissions sont initiées par les racines RPL. Le timer *retransmit_timer* est activé lorsque la QUERY est envoyée ou lorsqu'une trame DATA avec le flag *next* à 1 est reçue.

Les acquittements ne sont pas nécessaires pour les communications descendantes, car, si la racine LoRa envoie une trame, cela signifie qu'une racine RPL attend cette trame. Si elle n'est pas reçue, la racine RPL demandera une retransmission via l'envoi de la dernière trame envoyée, c'est à dire la trame avec la commande QUERY.

Après 3 retransmissions, le paquet est perdu et le prochain sera transmis lors de la réception d'une QUERY.

Le diagramme de séquence de la figure 2.8 illustre le déroulement des

communications descendantes.

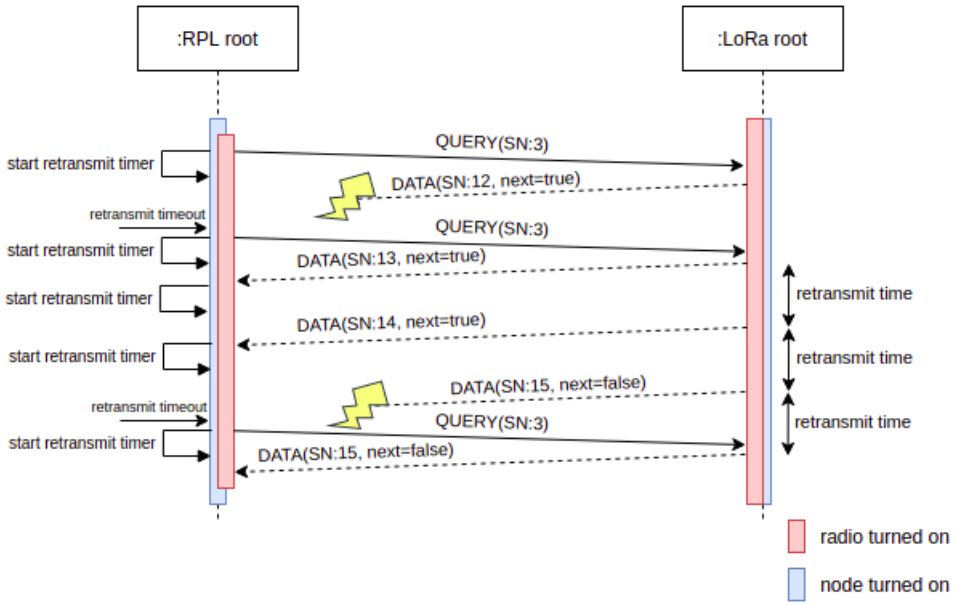


FIGURE 2.8 – Diagramme de séquence du trafic descendant.

Gestion des numéros de séquence

Les racines RPL et la racine LoRa possèdent deux compteurs : *expected_sn* et *next_sn*.

La valeur du compteur *expected_sn* est le numéro de séquence attendu de la prochaine trame reçue. Ce compteur permet de savoir si une trame a été perdue. Sa valeur est le numéro de séquence de la dernière trame reçue incrémenté de 1.

La valeur du compteur *next_sn* est le numéro de séquence de la prochaine trame. Il est incrémenté de 1 lorsqu'une trame a été envoyée.

Les deux compteurs sont initialisés à zéro.

2.5 Plateformes de développement

Cette section décrit le matériel utilisé pour mettre en oeuvre la topologie décrite en section 2.1. Pour les racines RPL, la plateforme de développement Zolertia RE-Mote Rev.B est utilisée, et, pour la racine LoRa, un Raspberry pi 3B+ est utilisé. Leur interface LoRa est un RN2483. Les noeuds des réseaux RPL sont également la plateforme de développement Zolertia RE-Mote Rev.B.

RN2483

Le RN2483 (Fig. 2.9) est un modem LoRa compatible LoRaWAN^{TM} basse énergie. D'après Nestor Ayuso [**?**], il intègre un transceiver Semtech SX1276 [**?**] et un microcontrôleur Microchip PIC18LF46K22. La communication avec ce modem est effetuée par des commandes ASCII envoyées via une interface UART. Il prend en charge les modulations FSK, GFSK et LoRa. Il possède également 14 GPIOs (General Purpose Input/Output). Ses fréquences opérationnelles sont situées dans les bandes 433 MHz et 868 MHz. D'après la datasheet [**?**], sa portée maximale est de 15km en agglomération et 5km en zone urbaine. Comme l'illustre la figure. 2.9, pour ce mémoire, le RN2483 a été monté sur un carte d'interface réalisée par B.Quoitin qui comporte deux LEDs, une petite hélicoïdale ainsi que les connecteurs permettant d'utiliser des câbles de prototypages.

FIGURE 2.9 – RN2483.

La figure 2.10 reprend le schéma-bloc du RN2483. Il contient notamment l'interface UART, les antennes 433 MHz et 868Mhz ainsi que les GPIOs et la stack LoRaWan.

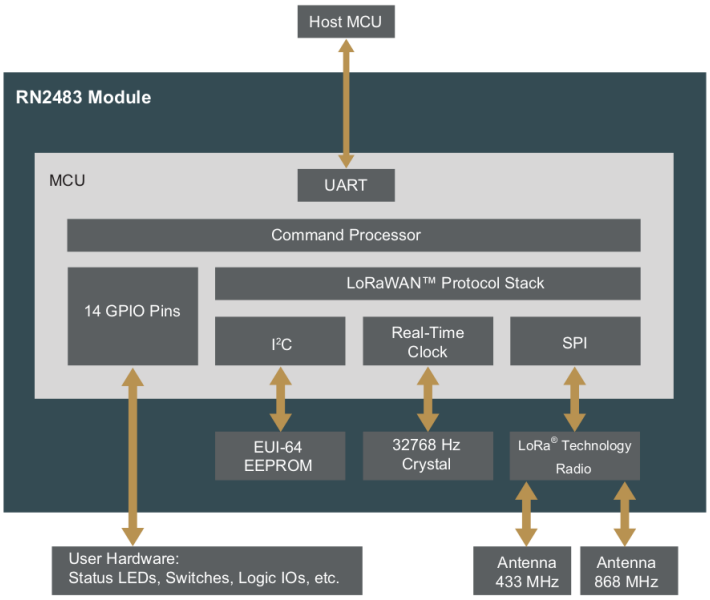


FIGURE 2.10 – Schéma-bloc du RN2483 [**?**].

La table 2.2 reprend la consommation électrique du RN2483 en fonction de son mode de fonctionnement pour alimentation de 3.3V qui est celle fournie par le RE-Mote et le Raspberry Pi.

Mode	Courant(mA) VDD=3.3V
Idle	2.8
Transmit	38.9
Sleep	0.0016
Receive	14.22

TABLE 2.2 – Consommation de courant (à 25 °C) [**?**].

Raspberry Pi

Le Raspberri Pi est un ordinateur monocarte. Il possède notamment 40 pins GPIO, une interface Wi-Fi, Bluetooth, 4 ports USB 2.0 et un connecteur pour une caméra. Toutes ces caractéristiques en font une passerelle idéale pour ce projet. En effet, ils permettent d'utiliser plusieurs interfaces réseaux et d'y connecter des capteurs et actuateurs.

Le modèle utilisé pour ce projet est un Raspberry Pi 3 modèle B+ (Fig. 2.11). Ce modèle est équipé d'un processeur quadri-coeur Broadcom

BCM2837B0, Cortex-A53 64-bit SoC cadencé à 1.4GHz et embarque 1GB de mémoire RAM.

La table 2.3 reprend les principales caractéristiques de ce modèle.

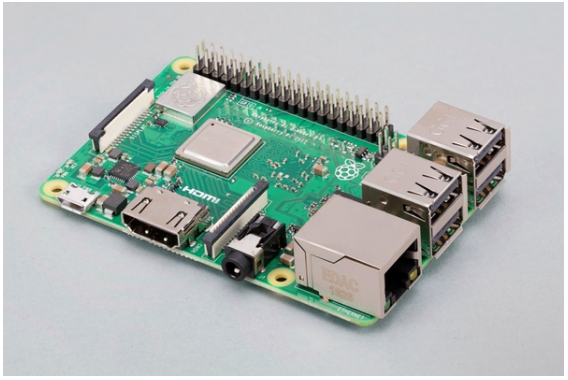


FIGURE 2.11 – Raspberry Pi 3B+ [**?**].

Element	Spécification
CPU	Broadcom BCM2837B0, Cortex-A53 64-bit SoC à 1.4GHz
Mémoire	1GB LPDDR2 SDRAM
Connectivité	• IEEE 802.11.b/g/n/ac, Bluetooth 4.2, BLE • Gigabit Ethernet over USB 2.0 • 4 × USB 2.0 ports
Alimentation	5V/2.5A DC

TABLE 2.3 – Spécifications du Raspberry Pi 3B+ [**?**].

Zolertia RE-Mote

Pour ce mémoire, la plateforme Zolertia RE-Mote Rev.B (Fig. 2.12) est utilisée. Cette plateforme, basée sur un system on chip (SoC) CC2538 ARM Cortex-M3, a été conçue par des universités et des industriels dans le but de permettre aux chercheurs et makers de développer des applications IoT et des objets connectés.

Le Zolertia RE-Mote a été choisi car il est équipé de deux radios compatibles IEEE 802.15.4, permet une consommation électrique faible et possède de nombreux pins de connexion qui peuvent être utilisés pour y connecter des capteurs, actuateurs, radios, etc.

Le prix du constructeur pour cette plateforme est de 93,95€ [**?**].

FIGURE 2.12 – Zolertia RE-Mote révision B [**?**].

La table 2.4 reprend les principales spécifications du Zolertia RE-Mote rev.b.

Element	Spécification
Radio	- IEEE 802.15.4 2.4 GHz - IEEE 802.15.4 863-950 MHz
CPU	ARM® Cortex® -M3 jusqu'à 32 MHz
RAM	32 KB (16 KB pour tous les Power Modes)
Flash programmable	512KB
I/O	- RGB led, boutton user et reset - Slot Micro-SD - USB 2.0 à 12Mbps - Real-Time Clock

TABLE 2.4 – Spécifications du Zolertia RE-Mote rev.b [**?**].

Chapitre 3

Mise en oeuvre

Ce chapitre décrit la mise en oeuvre de l'architecture définie dans le chapitre 2.

Deux implémentations sont réalisées. Celle de la racine LoRa en Python 3.8.10 et celle de la racine RPL avec la version `Contiki-NG-develop/v4.7-18-ga3f6b04c5` de Contiki.

Chaque implémentation est divisée en trois parties analogues aux couches physique, liaison et réseau du modèle OSI. La chouche physique contient notamment un driver permettant d'abstraire les communications UART avec le RN2483, rendant sont utilisation plus facile. La couche liaison implémente le protocole LoRaMAC, et la couche réseau sert d'interface permettant d'envoyer ou de recevoir des paquets IPv6.

3.1 Racine LoRa

Dans la topologie choisie à la section 2.1, une racine LoRa est le RN2483 interfacé à un Raspberry Pi. Pour le développement, le RN2483 est interfacé à un ordinateur portable. Que ce soit à l'ordinateur portable ou au raspberry pi, la carte d'interface du RN2483 est raccordée en USB via un bridge USB-UART.

L'implémentation se présente sous forme d'un package python qui donne accès à une variable `NETWORK_STACK` qui fait référence à une instance de la classe `NetworkStack` et permet d'accèder aux trois différentes couches, l'adresse LoRaMAC du noeud, son adresse IPv6 issue de la conversion de son adresse LoRaMAC ainsi que trois fonctions : `init`, `send` et `register_listener` qui sont des fonctions de l'interface IP et permettent respectivement d'initialiser la pile réseau, d'envoyer des paquets IPv6 et d'enregister une fonction callback qui sera appelée lorsqu'un paquet IPv6 reçu est disponible. L'usage

et l'implémentation de ses fonctions sont détaillées dans la description de la Couche IP.

Couche physique

La couche physique définit le format des adresses et trames LoRaMAC, ainsi que que le driver permettant de simplifier l'utilisation du RN2483 par l'abstraction des communications UART.

Les classes `LoraAddr` et `LoraFrame` définissent respectivement les adresses LoRaMAC et les trames LoRaMAC conformément à leur définition établie dans le chapitre 2. Ces deux classes possèdent les fonctions nécessaires pour sérialiser les trames LoRamAC (resp.désérialiser) en (resp. depuis) une chaîne de caractères hexadecimal qui est le format de donnée demandé par le RN2483.

Le driver du RN2483 est implémenté par la classe `LoraPhy`. Elle possède deux Threads servant à la réception et à l'envoi des trames UART. Le Thread d'envoi récupère les trames dans un buffer qui est une `Queue` FIFO de la librairie python `queue`. Une fois qu'une trame UART a été envoyée, il est bloqué par une variable conditionelle jusqu'au moment où le Thread de réception lui signale qu'une réponse attendue en retour de la trame envoyée a été reçue.

Quand le Thread de réception reçoit une trame UART contenant une trame LoRaMAC, celui-ci l'ajoute dans un buffer de réception qui sera lu par la couche MAC.

Les fonctions publiques du driver sont les suivantes :

- `init()` : Initialise la connexion série en utilisant la librairie python *pyserial* (version 3.4), configure le RN2483 avec les paramètres de la radio (par remple, le *SF*, la fréquence ou la bande passante) et démarre les Threads.
- `phy_send(loraFrame: LoraFrame)` : Permet d'envoyer une trame LoRaMAC
- `phy_timeout(timeout: int)` : Permet de définir le temps d'expiration du watchdog timer qui est activé pour chaque transmission ou réception. Pour le désactiver, la valeur qui doit être utilisée est zéro.
- `phy_rx()` : Permet de mettre le RN2483 en mode réception
- `getFrame()` : Permet d'obtenir la prochaine trame du buffer de réception. Si le buffer est vide, la fonction est bloquante.

Mise à part la fonction `getFrame()`, toutes les autres fonctions publiques alimentent le buffer d'envoi avec la trame UART appropriée.

Couche MAC

L'implémentation du protocole LoRaMAC n'utilise pas explictement les états définis dans la section 2.4. Cependant son fonctionnement est identique à celui décrit par la machien a état de la racine LoRa.

L'implémentation utilise deux classes : LoraChild et LoraMac. La première sert à conserver toutes les informations utiles d'une racine RPL (enfant) comme son adresse, ses compteurs de numéros de séquence ou encore la dernière trame envoyée. La deuxième est l'implémentation du protocole LoRaMAC.

Les trames LoRaMAC sont récupérées du buffer de la couche physique par un Thread. Une fois qu'une trame est récupérée, une première fonction vérifie que la trame LoRaMAC est bien destinée à la racine du réseau et récupère l'enfant s'il existe dans la liste des enfants. Si le numéro de séquence de la trame est 1, l'enfant est marqué comme ayant rejoint le réseau. Cette fonction envoi ensuite la trame vers la fonction de traitement appropriée selon sa commande MAC.

Pour la suite du mémoire, sf est défini comme le numéro de séquence d'une trame reçue et se la valeur d'un compteur `expected_sn` qui représente le prochain numéro de séquence attendu.

Réception d'une trame JOIN

Lorsque la trame est reçue, si le numéro de séquence est égal à zéro, une trame JOIN_RESPONSE est envoyée avec le préfixe comme payload. Sinon, la trame est ignorée est aucune action n'est réalisée. Une instance de LoraMac possède un compteur qui sert à attribuer les préfixes et qui est incrémenté à chaque JOIN_RESPONSE envoyée. Si tous les préfixes sont pris, la racine LoRa ne répond pas à la trame JOIN.

Réception d'une trame QUERY

Lorsqu'une trame QUERY est reçue, si $sf < se$ la dernière trame envoyée est retransmise. Si $sf > se$ cela signifie que $sf - se$ trames ont été perdues. Enfin si $sf \geq se$, la racine LoRa va instancie un nouveau Thread pour l'enfant, s'il n'en possède pas déja un actif. Ce Thread a pour rôle d'envoyer toutes les trames LoRaMAC à destination de cet enfant. Tant que le buffer d'envoi de l'enfant n'est pas vide, ce Thread procède comme suit :

1. Il récupère une trame dans le buffer et l'envoie
2. Il attend maximum MIN_WAIT_TIME secondes

3. Si aucune retransmission n'a eu lieu pendant l'attente, il envoie la prochaine trame
4. Si une retransmission a eu lieu, il stoppe cette attente et retourne au point 2

Si le nombre de retransmissions a atteint MAX_RETRANSMIT, le trame est perdue et la prochaine sera envoyée lors de la réception de la prochaine QUERY.

Réception d'une trame DATA

Comme lors de la réception d'une QUERY, si $sf < se$ la dernière trame envoyée est retransmise. Si $sf > se$ cela signifie que $sf - se$ trames ont été perdues. Si $sf \geq se$, l'adresse source et la payload sont envoyées à la couche supérieure. Si la trame reçue nécéssite un acquittement, celui-ci est envoyé.

Fonctions publiques de la couche MAC

La couche MAC possède trois fonctions publiques pour qu'une couche supérieure puisse l'utiliser.

- **`register_listener(listener: Callable[[LoraAddr, str], None])`**
  Permet à une couche supérieure d'enregistrer une fonction callback qui sera appelée lorsque un paquet est disponible.
- **`mac_send(dest:LoraAddr, payload:str)`**
  Permet d'envoyer une trame LoRaMAC.
- **`init()`**
  Initialise la couche MAC. C'est a dire, initialise la couche physique, désactive le watchdog timer, met le RN2483 en mode écoute et démarre le Thread de réception.

Couche IP

La couche IP est une interface permettant d'envoyer et de recevoir des paquets IPv6 via le protocole LoRaMAC. Pour la représentation et la manipulation des adresses IPv6, la librairie *ipaddress* (version 1.0) est utilisée. Pour la manipulation des paquets IPv6, la librairie *scapy* (version 2.4.5) est utilisée.

Lorsqu'une trame de la couche MAC est reçue, le paquet IPv6 est reconstruit comme suit :

1. Les 8 octets de poids fort de la payload sont extraits
2. Les adresses source et destination sont converties en adresses IPv6 et ajoutées à la suite de ces 8 premiers octets

3. Le reste de la trame IPv6 est ajoutée à la suite des adresses

Une fois le paquet IPv6 construit, il est envoyé à la couche supérieure.

Pour les paquets IPv6 devant être envoyés, les adresses IPv6 sont extraites du paquet et l'adresse de destination est conveertie en adresse LoRaMAC. Le reste du paquet est la payload de la trame LoRaMAC.

Les fonctions publiques de la couche d'inteface IP sont les suivantes :

- **`init()`**
  Initialise la couche d'interface IP. C'est à dire, initialise la couche MAC et enregistre sa fonction callback de réception de trame LoRaMAC auprès de celle-ci.
- **`register_listener(listener: Callable[[IPv6], None])`**
  Permet à une couche supérieure d'enregistrer une fonction callback qui sera appelée lorsque un paquet est disponible.
- **`send(ip_packet: IPv6)`**
  Permet d'envoyer un paquet IPv6.

3.2 Racine RPL

Dans la topologie choisie à la section 2.1, une racine RPL est la plateforme de développement Zolertia RE-Mote Rev.B à laquelle la carte d'interface du RN2483 a été raccordée par des câbles de prototypage. La figure 3.1 illustre le raccordement entre le RN2483 et le Zolertia.

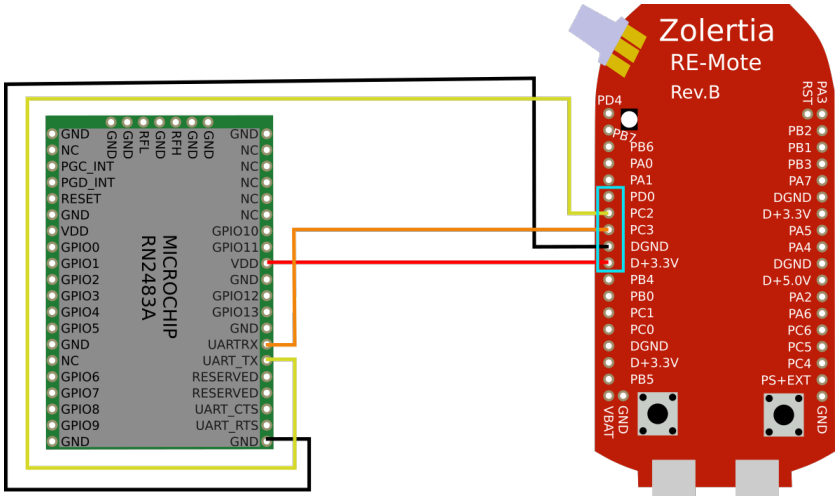


FIGURE 3.1 – Schéma du raccordement du RN2483 au Re-Mote.

Comme l'indique la table 3.1 de description des pins, les pins PC2 et PC3 utilisés pour la connexion ne sont pas ceux défini pour l'usage d'une

connexion UART. Ces pins sont choisis pour la connexion UART car ils sont accessibles à l'exterieur du boitier de la plateforme par un connecteur.

Pour que la connexion UART avec le RN2483 soit possible via les pins choisis, des modifications dans Contiki du fichier de configuration de la plateforme sont nécéssaires. Ainsi, les valeurs des macros suivantes ont été modifiées dans le fichier `arch/cpu/cc2538/cc2538-conf.h` :

- `UART1_CONF_BAUD_RATE` est définie à 57600 qui est le baudrate par défaut du RN2483
- `UART1_RX_PIN` est définie à 2
- `UART1_TX_PIN` est définie à 3

Comme pour l'implémentation de la racine LoRa, celle de la racine RPL est réalisée en 3 couches. qui sont intégrées dans Contiki comme un service. C'est à dire que le code des 3 couches se trouve dans le répertoire suivant : `os/services/loramac`.

Pin	Default name	MC	Description
4	UART0.RX	PA0	Connected to the CP2104 USB-to-serial converter
5	UART0.TX	PA1	Connected to the CP2104 USB-to-serial converter
6	GPD0/I2C.Interrupt	PD0	Generic pin, may be used as auxiliary pin for I2C/SPI
7	I2C.SDA	PC2	GPIO 20 mA output capability, pull-up. Shared with the RTCC and on-board low-power PIC
8	I2C.SCL	PC3	GPIO 20 mA output capability, pull-up. Shared with the RTCC and on-board low-power PIC
9	DGND	N/A	Digital Ground
10	D+3.3	N/A	3.3VDC output pin
11	CC1200.GPIO0	PB4	CC1200 GPIO0 pin. If RF switch disables sub-GHz is available for other purposes
12	CC1200.GPIO2	PB0	CC1200 GPIO2 pin. If RF switch disables sub-GHz is available for other purposes
13	UART1.RX	PC1	GPIO 20 mA output capability, no pull-up or pull-down.
14	UART1.TX	PC0	GPIO 20 mA output capability, no pull-up or pull-down.

Table 3.1 – Extrait de la table de descrition des pins [**?**].

Couche physique

Comme pour la racine LoRa, la couche physique définit les trame et les adresses LoRaMAC et sert de driver pour le RN2483. La réception et l'envoi de trames UART est réalisée par deux processus.

Le processus de récéption permet d'éviter que le reste du code soit interrompu par une trame UART entrante. Le code suivant illustre la déclaration de processus dans contiki.

```
1 PROCESS_THREAD(ph_rx, ev, data){
2     PROCESS_BEGIN();
3     /* UART configuration */
4     uart_init(UART);
5     uart_set_input(UART, &uart_rx);
6     PROCESS_END();
7 }
```

Tout ce qui se trouve entre les instructions des lignes 2 et 6 dépend du processus. Ainsi, la fonction, `uart_rx` qui traite les trames UART entrantes dépend bien de ce processus. Le rôle de cette fonction est d'accumuler tous les caractères d'une trame UART pour ensuite appeler une fonction qui traîte cette trame. Elle supprime également les caractères ayant comme valeur entière 254, 248, 192 et 240. Ces caractèrent apparaissent lors de certaines réponses UART et perturbe la reconstruction de la trame. La cause de l'apparition de ces caractères n'est pas identifiée.

Une fois la trame UART assemblée, une fonction suivante vérifie si cette réponse est une réponse attendue pour la trame précédemment envoyée. Si c'est le cas, elle envoie un évènement qu'attend le processus d'envoi pour envoyer la prochaine trame UART. Si la trame UART reçue contient la commande *"radio_rx"*, la fonction construit une trame LoRaMAC et l'envoie à la couche supérieure.

Le processus d'envoi apour rôle d'envoyer toutes les trames UART présentes dans le buffer d'envoi. Il peut être bloqué si le buffer d'envoi est vide ou s'il attend une réponse à la dernière trame envoyée. Si le buffer d'envoi est vide, le processus sera débloqué par la réception de l'évènement `new_tx_frame_event`, et s'il attend une réponse, il sera débloqué par la réception de l'évènement `can_send_event`.

Dans Contiki, les évènements sont créés et envoyé de la manière suivante :

```
1         process_event_t new_tx_frame_event = process_alloc_event();
2         process_post(&ph_tx, new_tx_frame_event, NULL);
```

Le type type `process_event_t` est en réalité un `unsigned char` qui est incrémenté à chaque appel de `process_alloc_event()`.

Les fonctions publiques de la couche physique sont les suivantes :

- **`void phy_init()`**
  Initialise la couche physique. C'est à dire, alloue les évènements, démarre les processus et envoie les trames UART permettant de mettre en pause LoRaWAN et de définir les paramètres de la radio.
- **`void phy_register_listener(int (* listener)(lora_frame_t frame))`**
  Enregistre la fonction callback qui sera appelée lorsqu'une trame LoRaMAC est reçue.
- **`int phy_tx(lora_frame_t frame)`**
  Envoie une trame LoRaMAC.
- **`int phy_timeout(int timeout)`**
  Définis le temps d'expiration du watchdog timer.
- **`int phy_sleep(int duration)`**
  Met le RN2483 en mode sommeil.
- **`int phy_rx()`**
  Met le RN2483 en mode réception.

Toues ces fonctions créent une trame UART qu'elle ajoute au buffer d'envoi.

Couche MAC

La couche MAC définit 3 états : ALONE, READY et WAIT_RESPONSE. Ces états correspondent à ceux défini dans la section 2.4, mis à part l'état SLEEP qui n'a pas besoin d'être défini explicitement, car dans cet état, le noeud attend, et donc rien d'autre ne peut se passer. En effet, comme le noeud ne peut basculer dans cet état que si le transmission d'une trame JOIN a échouée (i.e. après trois retransmissions), et qu'aucune trame LoRaMAC ne peut être reçue il impossible qu'il bascule dans cet état. De plus, aucune trame ne peut être envoyée, car tant que le noeud n'a pas rejoint le réseau LoRaMAC, la radio 802.15.4 et le protocole de routage RPL ne sont pas actifs. Le premier état est l'état initiale, le second est utilisé quand la racine RPL est prête à envoyer une trame et le dernier est utilisé quand elle attend une réponse à une trame envoyée.

Cette implémentation utilise un processus qui envoi les trames disponibles dans un buffer. Ce processus peut être bloqué si le buffer est vide ou si l'état est différent de READY. Il peut être débloqué respectivement par les évènements `new_tx_frame_event` et `state_change_event`. Ce processus démarre également un timer, `retransmit_timer` lorsqu'une trame attend une

réponse en retour. C'est à dire, lorsqu'il envoie une trame QUERY ou une trame avec le flag k à *true*.

Comme pour l'implémentation de la racine LoRa, une fois que la fonction qui réceptionne une trame a vérifié par l'adresse de destination, que la trame est destinée à un noeud de son réseau RPL ou à elle même, elle va envoyer cette trame vers la fonction appropriée.

Réception d'une trame JOIN_RESPONSE

La trame est considérée comme valide si les conditions suivantes sont vérifiées :

- L'état MAC est ALONE
- L'adresse de destination est celle de la racine RPL
- La taille du payload est 2 (taille de la chaîne de caractères hexadecimal du préfixe d'un octet)
- le numéro de séquence de la trame est 0

Si la trame n'est pas valide elle est ignorée. En revanche, si elle est valide, l'état est changé à READY, la processus d'envoi est démarré et la racine RPL met à jour son adresse. En effet, pour l'envoi de la trame JOIN la racine RPL n'ayant pas encore de préfixe, son préfixe était les huit bits de poids fort de son *node-id*. En plus de ces actions, un timer `query_timer` servant à envoyer periodiquement une trame QUERY est démarré, et l'évènement `loramac_network_joined` est envoyé à tous les processus.

Réception d'une trame ACK

Si le numéro de séquence de la trame est différent à celui de la dernière trame envoyée, elle est ignorée. Sinon, les actions suivantes sont éffectuées :

- le timer `retransmit_timer` est stoppé
- si la dernière trame envoyée était une QUERY, le timer `query_timer` est redémarré
- l'état est changé à READY et le processus d'envoi est prévenu du changement d'état

Réception d'une trame DATA

Si $sf < se$, la trame est ignorée. Sinon elle est traitée. Si $sf > se$, cela signifie que $sf - se$ trames ont été perdues. Ensuite la trame est envoyée à la couche supérieure.

Si la flag *next* est à *true*, cela signifie que d'autres trames vont être reçues. Donc, la racine RPL se met en mode écoute et redémarre le timer

`retransmit_timer`. Sinon, le timer `query_timer` est redémarré, l'état est changé à READY et le processus d'envoi est prévenu de ce changement.

Expiration du timer `retransmit_timer`

A l'expiration de ce timer, si le nombre de retransmissions n'a pas atteint son maximum défini par la macro `MAX_RETRANSMIT`, la dernière trame envoyée est retransmissions. Sinon l'envoi de la dernière trame est considéré comme échoué et l'état est changé à READY sauf si la trame qui n'a pas pu être envoyée est trame JOIN. Dans ce cas, le temps d'expiration du timer est allongé comme décrit par l'équation 2.1.

Fonctions publiques

Les fonctions publiques utilisable par les autres couches sont les suivantes :

- **`void mac_root_start()`**
  Initialise le protocole LoRaMAC. C'est à dire, défini l'adresse du noeud, l'état initiale, crée les évènements, initialise la couche physique, envoie une trame JOIN, met la radio en mode réception et active le `retransmit_timer`.
- **`int mac_send_packet(lora_addr_t src_addr, bool need_ack, void* data)`**
  Permet d'envoyer un paquet à l'adresse de destination. Si le paramètre `need_ack` est à *true*, un aquitement sera demandé pour la trame envoyée.
- **`void loramac_set_input_callback(void (* listener)(lora_addr_t *src, lora_addr_t *dest, char* data))`**
  Permet d'enregistrer une fonction callback qui sera appelée quand un paquet est disponible.

Couche IP

Ce qui est appelé la couche IP est une interface entre le protocole LoRaMAC et la stack TCP/IP de Contiki. Les paquets destinés à une adresse pour lesquels aucune route n'est trouvée sont récupérer par une interface définie comme suit dans le fichier `os/net/ipv6/uip.h`.

```
1 struct uip_fallback_interface {
2   void (*init)(void);
3   /**
4    * \retval >=0
```

```
5     *          in case of success
6     * \retval <0
7     *          in case of failure
8     */
9   int (*output)(void);
10 };
```

Cette interface doit être la valeur d'une macro `UIP_FALLBACK_INTERFACE` qui pour ce projet est définie dans `module-macros.h` présent dans le service `loramac`. Ainsi, quand un paquet IPv6 est destiné à une adresse pour laquelle aucune route n'est trouvée, la fonction `output()` de l'interface est appelée.

L'utilisation de cette interface a été choisie car il n'estactuellement pas possible d'utiliser deux interfaces réseaux dans Contiki. C'est donc le seul moyen trouvé pour récupérer les paquets destinés à une adresses externe au réseau RPL.

Contiki possède un buffer, le buffer uIP, qui est utilisé pour stocker les paquets entrants et sortants. L'accès à ce buffer se fait via la macro `uip_buf` définie dans le même fichier que la structure `uip_fallback_interface`. Une autre macro de Contiki utilisée dans l'interface est la macro `uip_len` qui est la longueur d'un paquet disponible dans le buffer. Enfin, Contiki fourni des macros faciliant l'accès au buffer uIP comme la macro `UIP_IP_BUF` qui permet d'accèder directement au header IPv6. Le header Ipv6 est défini comme suit :

```
1 struct uip_ip_hdr {
2   /* IPV6 header */
3   uint8_t vtc;
4   uint8_t tcflow;
5   uint16_t flow;
6   uint8_t len[2];
7   uint8_t proto, ttl;
8   uip_ip6addr_t srcipaddr, destipaddr;
9 };
```

Le code suivant est celui de la fonction `output` de l'interface.

```
1 static int output(void){
2   static char data[(UIP_CONF_BUFFER_SIZE-32)*2];
3   int uip_index = 0;
4   int data_index = 0;
5
6   while(uip_index<uip_len){
7     if(uip_index==8){
```

```
8         // skip src and dest ipv6 addr
9         uip_index= UIP_IPH_LEN; // UIP_IPH_LEN is the ipv6 header size
10      }
11
12      sprintf(data+data_index, "%02X", uip_buf[uip_index]);
13      data_index+=2;
14      uip_index++;
15    }
16
17    lora_addr_t src_addr;
18    ipv62lora(&(UIP_IP_BUF->srcipaddr), &src_addr);//convert src address
19    mac_send_packet(src_addr, true, &data);
20    return 0;
21  }
```

A la ligne 1, le tableau de caractère contenant la payload de la trame LoRaMAC est défini. Sa taille est `(UIP_CONF_BUFFER_SIZE-32)*2`. La macro `UIP_CONF_BUFFER` défini la taille du buffer uIP. Cette macro est définie dans `contiki-default-conf.h`. Pour ce projet, sa valeur initiale de 1280 a été modifiée à 279. Cette taille est 247, la taille de la payload d'une trame LoRaMAC à laquelle on a ajouté 32 octets qui est la taille des deux adresses Ipv6 (source et destination) car celles-ci ne se retrouvent pas dans une trame LoRaMAC. A la taille du buffer uIP, la taille des deux adresses IPv6 est soustraite car elles ne se trouvent pas dans la payload de la trame LoRaMAC. Ensuite cette taille est mutlipliée par deux car il faut deux caractères hexadecimal pour représenter un octet.

Ensuite la boucle de la ligne 6 parcourt le buffer uIP et convertit chaque octet en hexadecimal. A la fin de cette boucle l'adresse source est convertie en adresse LoRaMAC comme défini dans la solution retenue de la section 2.2. Enfin, l'adresse source et la payload sont envoyées à la couche LoRaMAC.

Lorsqu'une trame LoRaMAC est reçue la fonction callback enregistré auprès de la couche LoRaMAC convertit les adresses LoRaMAC source et destination en adresses IPv6, écrit les 8 octets de poids fort du header dans le buffer uIP et écrit les adresses converties dans ce buffer avant d'écrire la payload de la trame LoRaMAC.

La fonction `init` de l'interface démarre un processus. Ce processus désactive la radio, initialise la couche MAC et attend l'évènement `loramac_network_joined`. Une fois cet évènement reçu, ce process défini le préfixe IPv6 qui sera diffusé dans le réseau RPL, allume la radio et démarre le protocole de routage.

Chapitre 4

Evaluation

Bonjour