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

Mémoire réalisé par
Arnaud PALGEN

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

Année académique 2020-2021

Remerciements

Je te tiens à remercier les professeurs du département d'informatique pour leurs enseignements et de m'avoir permis de réaliser ce travail. Je remercie plus particulièrement mon directeur de mémoire, B.Quoitin, pour son accompagnement et ses conseils tout au long de ce travail. Finalement je tiens à remercier ma famille qui m'a soutenu pour ce travail et pour m'avoir aidé à la relecture.

Table des matières

1 Introduction **1**

2 État de l'art **3**
2.1 LoRa 3
2.2 IEEE 802.15.4e 6
2.3 RPL 10
2.4 RTOS 18
2.5 Projets similaires 21

3 Architecture **22**
3.1 Topologie 22
3.2 Format d'adresse 24
3.3 Trames LoRaMAC 27
3.4 LoRaMAC 28
3.5 Plateformes de développement 34

4 Mise en oeuvre **38**
4.1 Racine LoRa 38
4.2 Racine RPL 42

5 Validation et évaluation **51**
5.1 Validation 51
5.2 Evaluation 67

6 Conclusion **83**

Chapitre 1

Introduction

Le domaine de l'internet des objets connectés (Internet Of Things, IoT) est en constante augmentation. L'objectif de ce domaine est de connecter des actionneurs et des capteurs en réseau pour permettre de mettre en place la surveillance d'environnements ou d'objets et l'automatisation de certaines tâches. Ces réseaux d'objets connectés ont des applications dans un large éventail de domaines tels que la maintenance prédictive, l'agriculture, la santé, la domotique, etc.

Pour mettre en place de tels réseaux, il existe plusieurs standards illustrés par la Figure 1.1 en fonction du débit et de la portée. Ces standards peuvent être groupés en plusieurs catégories :

- **WLAN**(Wireless Local Area Network) : regroupe les réseaux locaux sans fil, comme le Wi-Fi, principalement utilisés comme réseau domestique
- **(L)WPAN**((Low-power) Wireless Personal Area Network) : rassemble les réseaux sans fil (à basse consommation énergétique) ayant une portée faible comme le Bluetooth
- **WWAN**(Wireless Wide Area Network) : regroupe les réseaux sans fil longue portée comme les réseaux cellulaires
- **LPWAN**(Low Power Wide Area Network) : est l'ensemble des réseaux sans fil longue portée et basse consommation comme les réseaux LoRaWAN

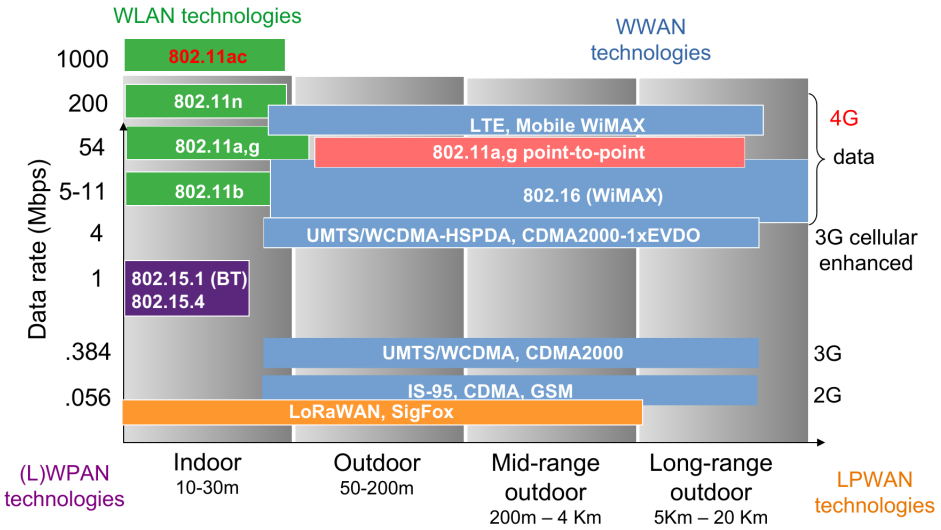


FIGURE 1.1 – Distance et débit de normes de liaison sans fils. [1]

Chaque technologie a donc des usages spécifiques qui dépendent de sa consommation d'énergie, des débits pouvant être atteints, de la portée et également des topologies réalisables.

Cependant, certains cas d'utilisation de réseaux de capteurs sans fils (Wireless Sensors Network, WSN) nécessitent l'usage d'un réseau hybride, utilisant plusieurs technologies radio, comme pour ce mémoire qui consiste au développement d'un réseau de capteurs se trouvant aux pieds de cultures dont la pousse de feuilles pourrait devenir un obstacle aux communications ou atténuer les signaux radio, et où les cultures se trouvent dans un terrain accidenté. Pour cela les deux technologies radio LoRa et IEEE 802.15.4 sont choisies.

Ces deux technologies possèdent un débit suffisant pour un réseau de capteurs et sont basse énergie.

IEEE 802.15.4 a une faible portée mais permet de créer des réseaux maillés (*mesh*). Tandis que LoRa possède une longue portée et utilise des réseaux points-à-points.

L'objectif de ce mémoire est donc de concevoir et d'évaluer un réseau hybride dans lequel LoRa permet de rendre des réseaux IEEE 802.15.4 accessibles à distance.

Chapitre 2

État de l'art

2.1 LoRa

LoRa (Long Range) [2][3] est une technologie radio fonctionnant sur une modulation propriétaire détenue par Semtech et basée sur une technologie d'étalement de spectre appelée *chirp spread spectrum (CSS)*. Celle-ci consiste en un signal d'une amplitude constante pour lequel la fréquence diminue ou augmente au cours du temps. Ces variations, linéaires pour LoRa, sont respectivement appelées *downchirp* et *upchirp*. Cette modulation permet d'obtenir un compromis entre la portée et la consommation en fonction d'un paramètre appelé le *spreading factor*.

2.1.1 Paramètres d'une communication LoRa

- **Spreading Factor (SF)** définit la vitesse de balayage du signal radio (Fig. 2.2). 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 [4].
- **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.

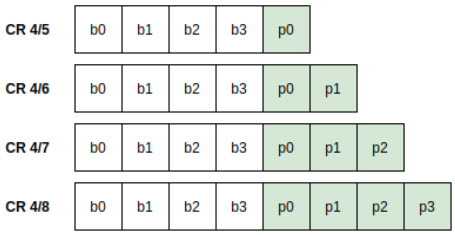


FIGURE 2.1 – Mots de 4 bits codés avec des redondances de 5, 6, 7 ou 8 bits.

La Figure 2.1 illustre l'ajout des bits redondants pour les différents coding rates.

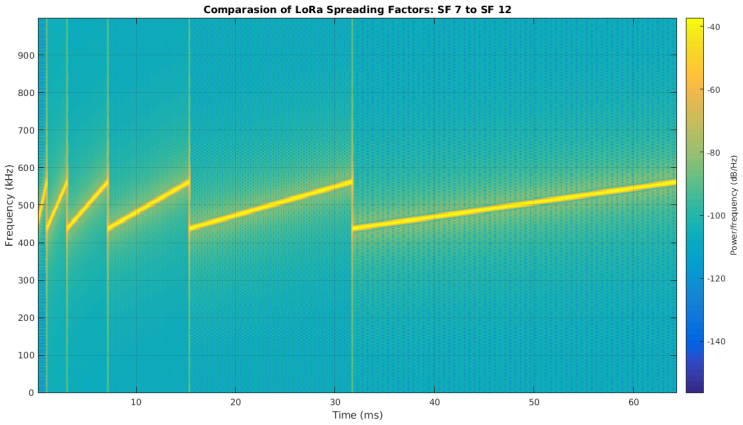


FIGURE 2.2 – Spectrogramme des différents Spreading Factors [5].

2.1.2 Format des paquets

La structure d'un paquet LoRa (Fig. 2.3) contient les champs suivants :

- **Preamble** : utilisé pour la synchronisation en fréquence et en temps
- **Header** : contient les informations sur la longueur du message, le CR et la présence d'un contrôle de redondance cyclique ou CRC (cyclic redundancy check) pour la payload.
- **Payload** : contient les données de la couche MAC.
- **CRC** : contient le code de contrôle de redondance cyclique (cyclic redundancy check) pour la payload.

Un paquet LoRa peut-être implicite ou explicite. Pour 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 doivent être configurés. Ce mode permet de réduire la taille du paquet et donc du temps de transmission. Dans le mode explicite, le header est présent.

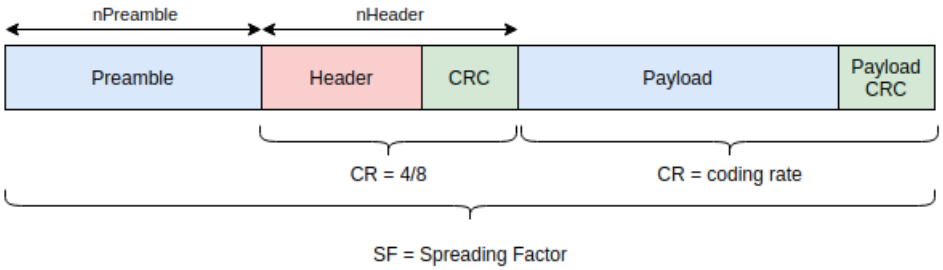


FIGURE 2.3 – Format d'un paquet LoRa.

2.1.3 Temps de transmission (Time-on-Air)

La datasheet du SX1276 [2], intégré dans le RN2383 [6], utilisé comme transceiver LoRa pour ce projet (cf. Chapitre 3 sous-section 3.5.1), définit le temps d'émission d'un paquet LoRa repris par l'Équation 2.1.

$$T_{frame} = T_{preamble} + T_{payload} \tag{2.1}$$

$T_{preamble}$ et $T_{payload}$ sont respectivement le temps d'émission du preamble et du payload définis aux Équations 2.2 et 2.3.

$$T_{preamble} = (n_{preamble} + 4,25)T_{sym} \tag{2.2}$$

$$T_{payload} = n_{payload} * T_{sym} \tag{2.3}$$

$T_{sym} = 1/R_s$ avec $R_s = BW/2^{SF}$. $n_{preamble}$ 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 2.4.

$$n_{payload} = 8 + max\left(\left\lceil \frac{8PL - 4SF + 28 + 16CRC - 20IH}{4(SF - 2DE)} \right\rceil (CR + 4), 0\right) \tag{2.4}$$

où :

- PL est le nombre d'octets de la payload (1 à 255)
- SF est le spreading factor (6 à 12)
- $CRC = 1$ si une CRC est présente, sinon $CRC = 0$
- $IH = 0$ si le header est présent, sinon $IH = 1$
- $DE = 1$ si $LowDataRateOptimize = 1$ et $DE = 0$ 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

2.2 IEEE 802.15.4e

IEEE 802.15.4 [7][8] 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 [9][7] a été défini pour pallier certains problèmes de IEEE 802.15.4 dans des environnements difficiles.

2.2.1 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, une topologie MESH pour laquelle les FFD sont connectés les uns aux autres et peuvent acheminer les paquets de leurs voisins vers le noeud suivant.

2.2.2 Modes d'accès

La norme 802.15.4 définie 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 2.4, 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.

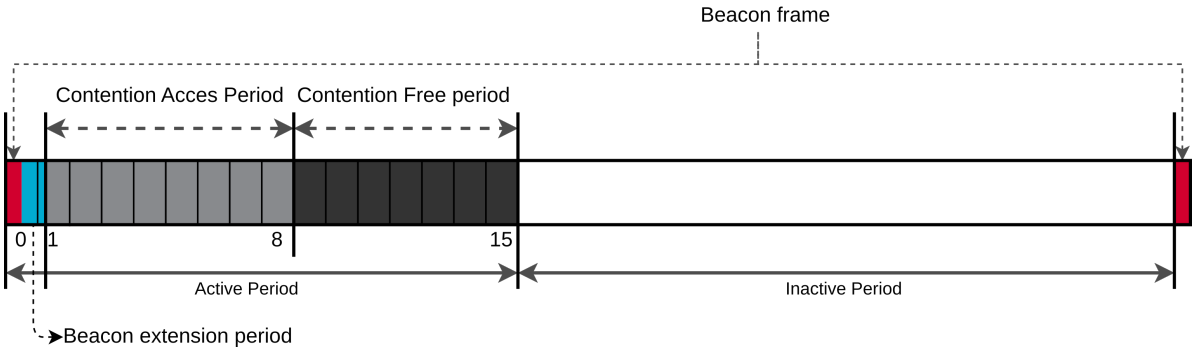


FIGURE 2.4 – 802.15.4 structure de la Superframe.

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 [7] :

- 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 [7], 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.

2.2.3 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 acquittement (ACK). Chaque timeslot possède un identifiant appelé *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 2.5 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.

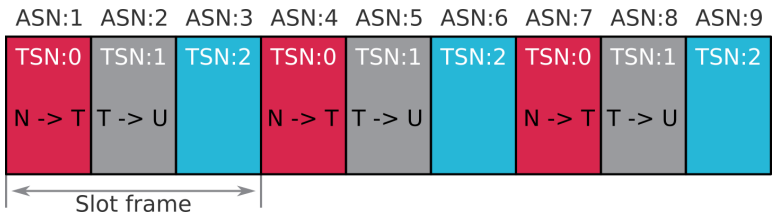


FIGURE 2.5 – Exemple de Slotframe.

Channel Hopping

TSCH peut utiliser 16 canaux différents dans la bande 2.4 GHz. 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. 2.6). La fréquence f utilisée pour la communication entre deux noeuds est calculée comme suit :

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

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.

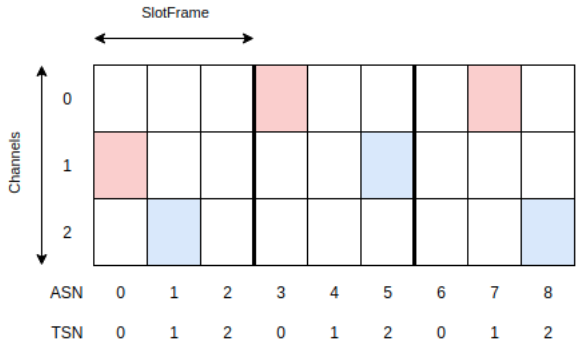


FIGURE 2.6 – Time Slotted Channel Hopping.

2.3 RPL

Routing Protocol for Low-Power and Lossy Networks [10] 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 [11], 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 [12], AODV [13] ou OLSR [14] 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 et 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 sur base du RFC 6550 [10].

2.3.1 Topologie

RPL construit un DODAG (Destination Oriented Directed Acyclic Graph) qui est un graphe dirigé acyclique (DAG) ayant une seule racine (Fig. 2.7). De par cette architecture, RPL est adapté aux applications de collecte de données. En effet, la route d'un noeud à la racine est facilement établie car tous les noeuds du chemin envoient les données à leur parent.

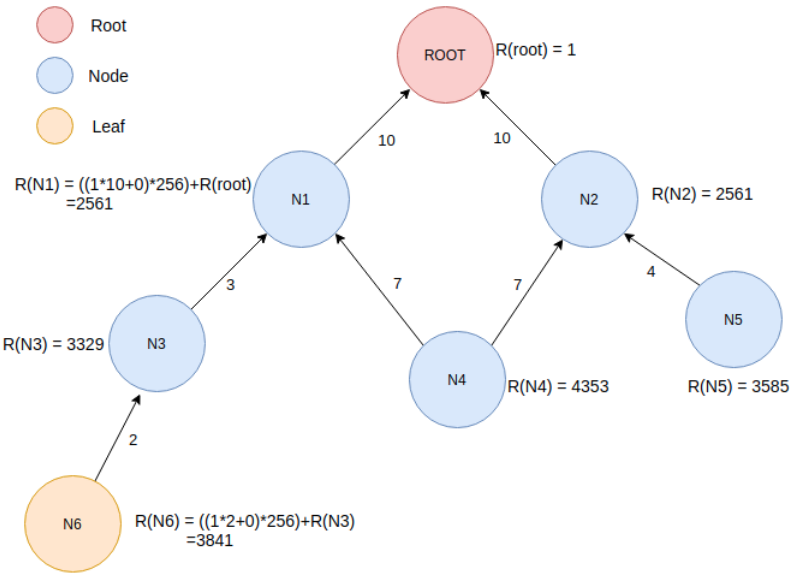


FIGURE 2.7 – DODAG.

2.3.2 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 et son parent $parent(n)$, la propriété $rang(n) > rang(parent(n))$ est toujours vérifiée. La Figure 2.7 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, sont décrites sur base du RFC 6552 [15] et du RFC 6719 [16].

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 de la racine du DODAG. Le calcul du rang d'un noeud N, $R(N)$, est décrit par l'Équation 2.5. Il est calculé pour chaque parent potentiel. Le parent préféré choisi sera alors le parent potentiel qui implique le plus petit $R(N)$.

$$\begin{aligned} rank_increase &= (Rf * Sp + Sr) * MinHopRankIncrease \\ R(N) &= R(P) + rank_increase \end{aligned} \tag{2.5}$$

Le calcul du rang commence par le calcul de la variable *step_of_rank* (Sp) qui est basé 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)* décrite dans l'article de Couto, Aguayo, Bicket et Morris [17], qui permet de minimiser l'espérance du nombre de transmissions requises pour transmettre un paquet. L'utilisation de propriétés dynamiques est recommandée car, utiliser des propriétés statiques, comme un coût administratif, aboutit à 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 l'augmentation maximum de *step_of_rank* avec le parent défini par le paramètre *stretch_of_rank*.

La variable *rank_increase* est finalement multipliée par le facteur *MinHopRankIncrease* qui est défini par le RFC 6550 [10] 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*.

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 messages annonçant des informations sur le DODAG (DIOs) provenant 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 [16] 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 :

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

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

2.3.3 Messages RPL

RPL défini quatre types de messages de contrôles qui sont des messages ICMPv6. Le RFC 6550 [10] indique que ces messages ont la portée d'un lien, à l'exception des DAO (cf.DAO) et des DAO-ACK (cf.DAO-ACK) qui, dans le mode de fonctionnement *non-storing* (cf. sous-section 2.3.5 Modes de fonctionnements), 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 2.8, inclut 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 (cf. sous-section 2.3.5 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

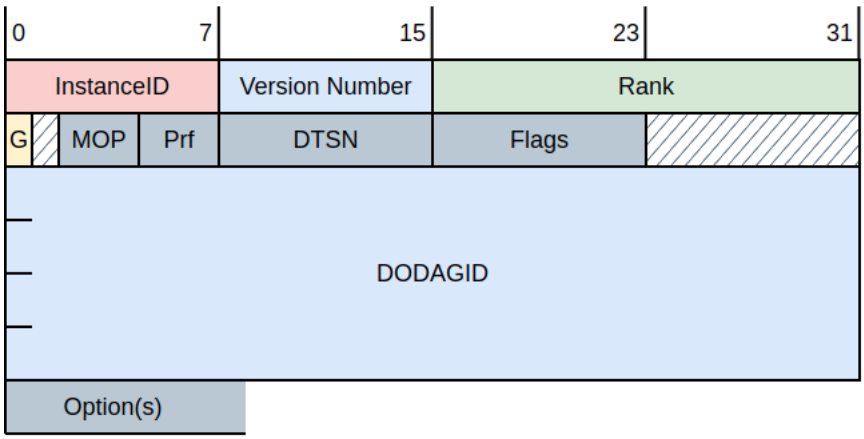


FIGURE 2.8 – DODAG Information Object.

Les DIOs sont transmis en utilisant l'algorithme Trickle [18]. Cet algorithme permet de réduire la quantité de messages envoyés en réduisant les transmissions quand les conditions sont stables et en les augmentant lorsqu'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 2.9. 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)

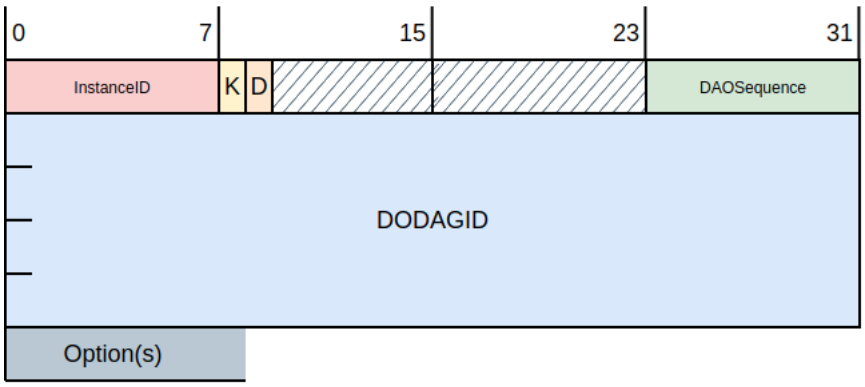


FIGURE 2.9 – DODAG Information Object.

DAO-ACK

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

2.3.4 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 2.10 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 (c'est-à-dire 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 sous-section 2.3.5 Modes de fonctionnements.

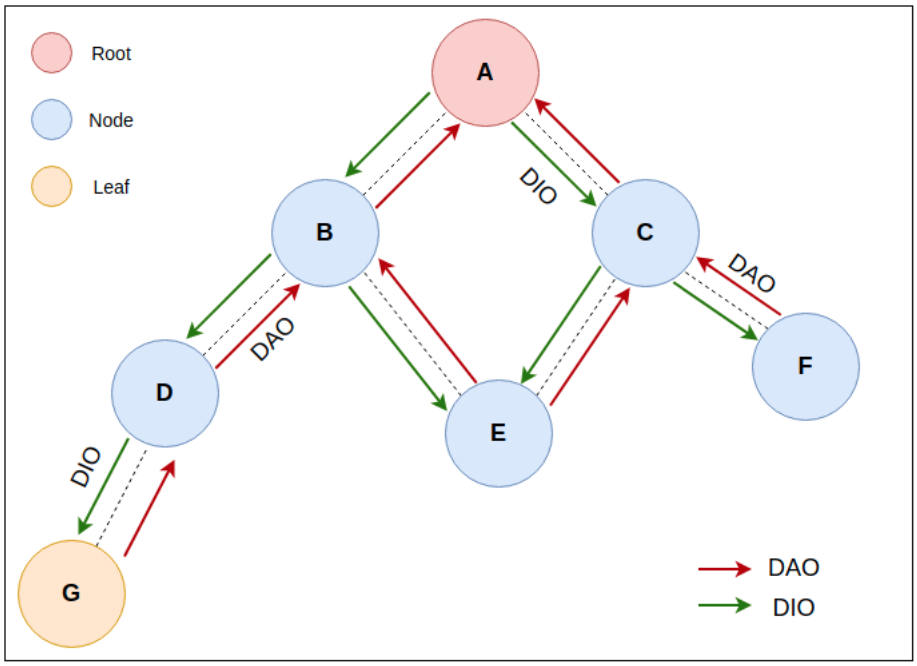


FIGURE 2.10 – Propagation des DIOs et DAOs.

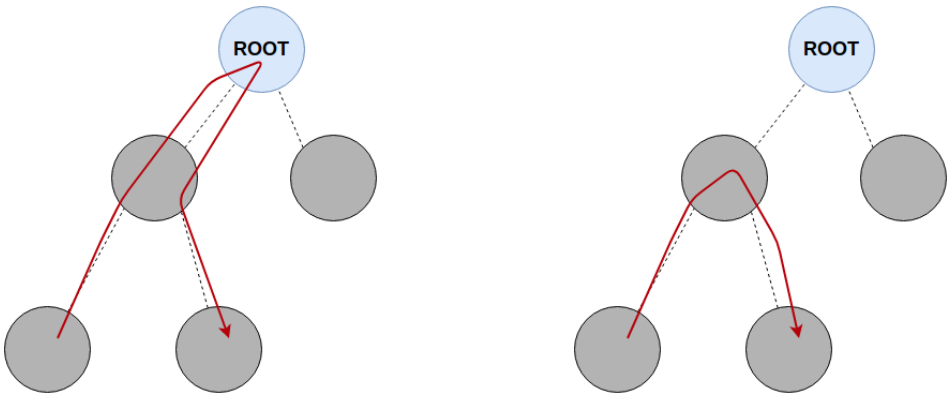
2.3.5 Modes de fonctionnements

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 [19] 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 [20] (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. 2.11a).

- **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. 2.11b). 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 2.11 – 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.

2.4 RTOS

Un RTOS (Real Time Operating System) est un système 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 un ordonnanceur.

Le RTOS choisi doit supporter la plateforme utilisée, le Zolertia RE-Mote Rev.B (cf. section 3.5 Plateformes de développement). Sa pile réseau doit comprendre les protocoles suivants : IEEE 802.15.4 pour la couche physique, TSCH avec un ou plusieurs algorithmes 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 contrôlé par UART.

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

2.4.1 Contiki OS

Le développement public de Contiki [21] a débuté en octobre 2012 [1] dans un premier repository : **contiki** [22]. Un fork de Contiki a démarré en mai 2017 sous le nom de **contiki-ng** [23]. 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 [24] qui abstrait la gestion de la programmation événementielle par des protothreads dont l'utilisation est similaire aux threads. L'ordonnanceur de cette librairie 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'exécution 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 [25], un ordonnanceur TSCH, Orchestra [26], est implémenté.

Contiki supporte plusieurs plateformes en fournissant pour chacune un BSP (Board Support Package). Pour le Zolertia RE-Mote, le BSP supporte

1. Date de création du repository Github.

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.

2.4.2 FreeRTOS

D'après le site officiel de FreeRTOS [27], ce RTOS est développé depuis 15 ans. 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+** contient notamment la librairie TCP/IP et les protocoles applicatifs MQTT et HTTP,
- **AWS IoT Libraries** fournit des librairies permettant la connectivité avec Amazon Web Services (AWS)
- **FreeRTOS Labs** contient des librairies complètement fonctionnelles 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.

2.4.3 RIOT OS

Le développement publique de RIOT [28] 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 [29] 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, 6LoW-

PAN, 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 cartes de développement et 64 CPU dont le Zolertia RE-Mote.

2.4.4 Conclusion

La Table 2.1 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	√	√	6Tisch, Orchestra	√√	RPL	√
FreeRTOS	×	×	×	√	×	×
RIOT OS	√	×	×	√√	RPL	√

Table 2.1 – 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.

La pile réseau utilisée comprend RPL pour le routage de paquets IPv6, le mode TSCH de IEEE 802.15.4e ordonnancé par Orchestra pour la couche MAC. Contiki implémente deux ordonnanceurs pour TSCH : 6Tisch et Orchestra. L'implémentation minimale de 6Tisch implique que la radio IEEE 802.15.4 est toujours allumée. Ce qui doit être évité pour économiser de l'énergie. C'est pourquoi, l'ordonnanceur Orchestra dont l'implémentation permet d'éteindre la radio lorsque c'est possible et donc d'économiser de l'énergie est utilisé.

Avec Orchestra [26], les noeuds établissent un ordonnancement des communications automatiquement sur base de l'évolution de la topologie RPL, ce qui est différent de 6Tisch qui utilise un processus de négociation entre noeuds voisins qui nécessite l'échange de messages entre ceux-ci.

Un ordonnancement Orchestra contient des slots qui sont chacun dédiés à un type de trafic particulier comme des beacons TSCH ou du trafic de donnée d'applications. Ensuite, les noeuds sélectionnent les slots à utiliser en fonction de règles d'ordonnancement.

2.5 Projets similaires

Une grande partie du travail de ce mémoire consiste en l'élaboration d'un protocole MAC pour orchestrer les communications LoRa. Le protocole décrit au chapitre 3 s'inspire du protocole ALOHA qui est un protocole où chaque noeud du réseau envoie ses paquets dès qu'il le souhaite et si un paquet est perdu, il est retransmis après un temps aléatoire. Ce protocole entraîne donc des collisions de paquets. Pour les minimiser, Yi Chu, Paul D. Mitchell et David Grace proposent dans leur article [30] un protocole MAC nommé ALOHA-QIR. Ce protocole combine Slotted ALOHA, une version d'ALOHA où le temps est divisé en slots et où les noeuds sont synchronisés pour connaître le temps de début de chaque slot, avec le Q-Learning qui est une méthode d'apprentissage par renforcement qui dans ce cas, a pour objectif de permettre aux noeuds d'utiliser les slots de manière plus intelligente. L'utilisation de cette méthode 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.

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 [31] 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 collectent 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 [32], un protocole MAC pour LoRa qui est une adaptation de TSCH. Leur solution, implémentée dans Contiki, permet d'utiliser le protocole de routage RPL. Elle fonctionne avec un *spreading factor* LoRa fixe. 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 [33], 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 permettant de sélectionner un chemin qui minimise le temps de transmission, ce qui permet de diminuer la consommation en énergie.

Chapitre 3

Architecture

3.1 Topologie

Cette section décrit et motive la topologie et les types de noeuds choisis pour élaborer un réseau hybride IEEE 802.15.4 - LoRa qui correspond au mieux au cas d'utilisation décrit dans l'introduction (chapitre 1 Introduction), c'est-à-dire, un réseau de capteurs établi aux pieds de cultures se trouvant sur un terrain accidenté.

Ce milieu de déploiement implique les exigences suivantes :

1. Les noeuds, alimentés par batterie sont fortement contraints en énergie
2. La croissance des cultures peut être un obstacle aux communications
3. L'utilisation du réseau doit pouvoir se faire à distance

La solution proposée permet d'obtenir un réseau répondant le mieux possible à ces exigences en utilisant TSCH avec Orchestra pour réduire le temps d'activité de la radio et donc d'économiser de l'énergie, et en combinant les deux technologies adaptées aux dispositifs contraints en énergie : LoRa et IEEE 802.15.4. La première permet d'établir des communications longue distance et la deuxième permet d'établir un réseau maillé. La combinaison de ces deux technologies permet donc de répondre aux mieux aux trois exigences.

3.1.1 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 de réseaux LWPANs. Elles possèdent une interface LoRa et une interface IEEE 802.15.4 afin de transférer les paquets entre la racine LoRa et les noeuds composant leur réseau
- Les **noeuds RPL** sont des noeuds équipés de capteurs/actionneurs dont le rôle est de transmettre les données des capteurs vers la racine LoRa, par le biais de leur racine RPL, et de changer l'état de leurs actionneurs en fonction des messages reçus depuis la racine LoRa

3.1.2 Topologie

Comme l'illustre la Figure 3.1, la topologie choisie est une topologie mixte, MESH-étoile, composée d'un ensemble de réseaux RPL, dans lesquels les communications se font par IEEE 802.15.4, et de liens LoRa.

Les réseaux RPL, formant des DODAGs, sont connectés à la racine LoRa via leur racine. Ils se voient attribuer un identifiant, appelé préfixe LoRa, par la racine LoRa. Par exemple $0x02$ ou $0x04$ sur la Figure 3.1. Les liens entre les racines RPL et la racine LoRa forment donc 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 réseaux RPL, la défaillance d'une racine RPL implique la déconnexion d'un seul réseau RPL, contrairement à une topologie composée d'un seul réseau RPL où la défaillance de la racine implique la déconnexion de l'ensemble des noeuds.

L'inconvénient de la topologie proposée 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 à établir pour les liens LoRa est plus complexe et nécessite un protocole de routage mais existe dans la littérature(cf. Chapitre 2 section 2.5). 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 sont assurés par RPL et IPv6. Les sections suivantes décrivent le protocole MAC mis en place pour les communications LoRa.

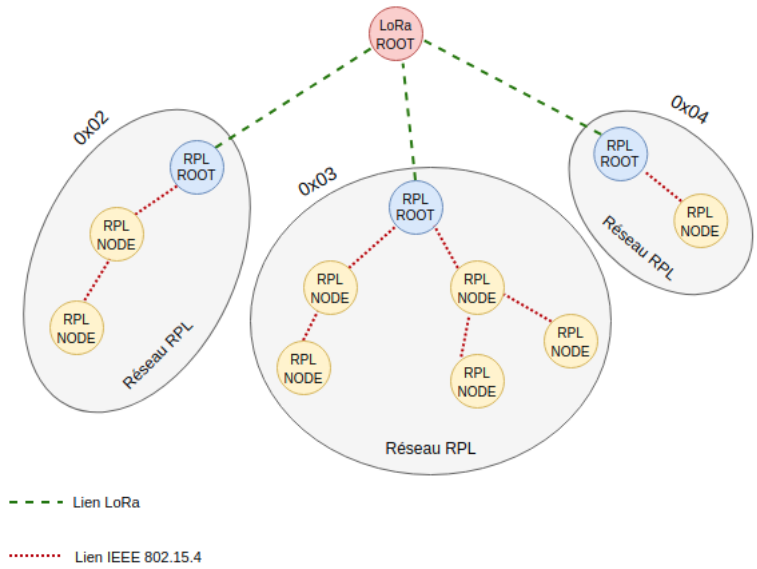


FIGURE 3.1 – Topologie du réseau hybride.

3.2 Format d'adresse

Cette section décrit les possibilités envisagées et motive le choix du format des adresses utilisées pour les communications LoRa entre les racines RPL et la racine LoRa.

L'objectif du format d'adresse choisi est d'être le plus court possible, afin de réduire le temps d'émission d'une trame, tout en ne nécessitant pas un processus de conversion vers une adresse IPv6 utilisant trop de mémoire ou trop d'opérations de calcul.

Dans chaque hypothèse, un réseau RPL possède un identifiant, le préfixe LoRa, qui lui est propre.

3.2.1 Adresse IPv6

La première solution envisagée est d'utiliser des adresses IPv6. Dans ce cas, chaque noeud est identifié par son adresse IPv6. Pour ce choix, le préfixe IPv6 est `fd00::<lora_prefix>` où *lora_prefix* est le préfixe LoRa du réseau RPL auquel appartient le noeud.

Ainsi, sur l'exemple de la Figure 3.1, un noeud RPL du réseau ayant le préfixe LoRa $0x02$ aurait par exemple, l'adresse suivante :
`fd00::2:eeff:212:4b00:60d:611b`.

Avec ce format d'adresse, un noeud LoRa peut définir si une trame lui est destiné (respectivement destiné à son réseau RPL pour une racine LoRa)

en comparant le préfixe LoRa se trouvant dans l'adresse avec le sien.

Cette solution est la plus simple à mettre en oeuvre car elle ne nécessite 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.

3.2.2 Préfixe LoRa et *node-id*

Une seconde approche est d'utiliser le préfixe LoRa 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 simple 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 3.1, un noeud RPL dont la valeur du *node-id* est 24859, appartenant au réseau ayant le préfixe LoRa $0x02$, aurait l'adresse suivante : `2:24859`.

3.2.3 Préfixe LoRa 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 3.1, un noeud RPL dont l'adresse MAC est `0012.4b00.060d.611b`, appartenant au réseau ayant le préfixe LoRa $0x02$, aurait l'adresse suivante : `2:0012.4b00.060d.611b`.

3.2.4 Solution retenue

La solution retenue est l'utilisation du préfixe LoRa 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*↔ adresse MAC, une meilleure solution est d'utiliser les adresses MAC des noeuds et de fixer les 6 premiers octets de cette dernière identiques pour tous les noeuds. Peu importe le réseau RPL auquel ils appartiennent. 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 précisément, les adresses IPv6 seront construites tel qu'illustré par la Figure 3.2.

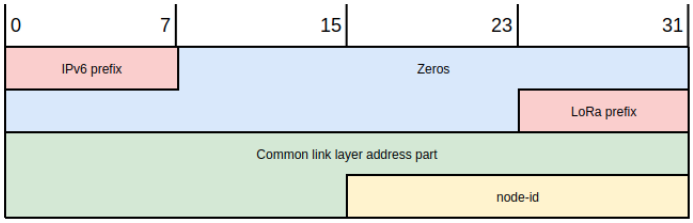


FIGURE 3.2 – Format d'adresse IPv6 à partir d'une adresse LoRaMAC.

Le préfixe IPv6 choisi est celui des adresses IPv6 unique local unicast. Ce type d'adresse est spécifié par le RFC 4193 [34]. 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.

3.3 Trames LoRaMAC

Le protocole MAC mis en place pour les communications LoRa utilise un seul format de trames illustré à la Figure 3.3.

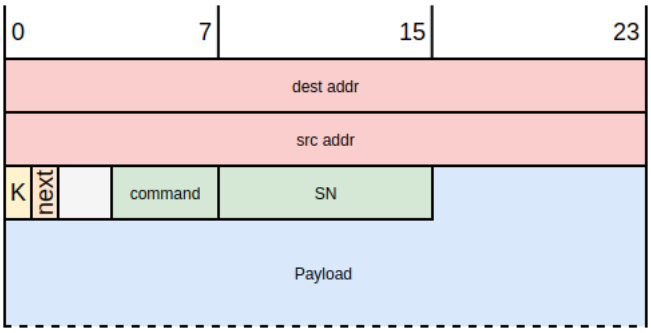


FIGURE 3.3 – Format de trame LoRaMAC.

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 3.1. 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 3.1 – Description des commandes LoRaMAC.

3.4 LoRaMAC

Cette section décrit le protocole MAC mis en place pour les communications LoRa. Ce protocole est 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 en compte cette contrainte.

3.4.1 États d'une Racine RPL

Les racines RPL utilisent 5 états illustrés dans le diagramme de la Figure 3.4. À l'initiation du réseau, une racine RPL se trouve dans l'état ALONE. Elle demande ensuite son préfixe à la racine LoRa (envoi d'une trame JOIN). 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 une réponse (une trame QUERY ou DATA nécessitant un acquittement), tant que celle-ci n'est pas reçue elle se trouvera dans l'état WAIT_RESPONSE. Une fois la réponse reçue, elle retournera dans l'état READY. Par contre, si la réponse n'est pas reçue après un délai d'attente fixé à 12 secondes (deux fois le temps le plus long possible d'une transmission d'une trame LoRa avec une petite marge, ce qui est suffisant pour recevoir une réponse de la racine LoRa), la trame est retransmise un maximum de trois fois. Après trois retransmissions échouées, la racine RPL retourne dans l'état READY.

Si une racine RPL envoie une trame DATA ne nécessitant pas d'acquittement, elle reste dans l'état READY car elle n'attend pas de réponse en retour.

Les trames qui doivent être envoyées le sont 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.

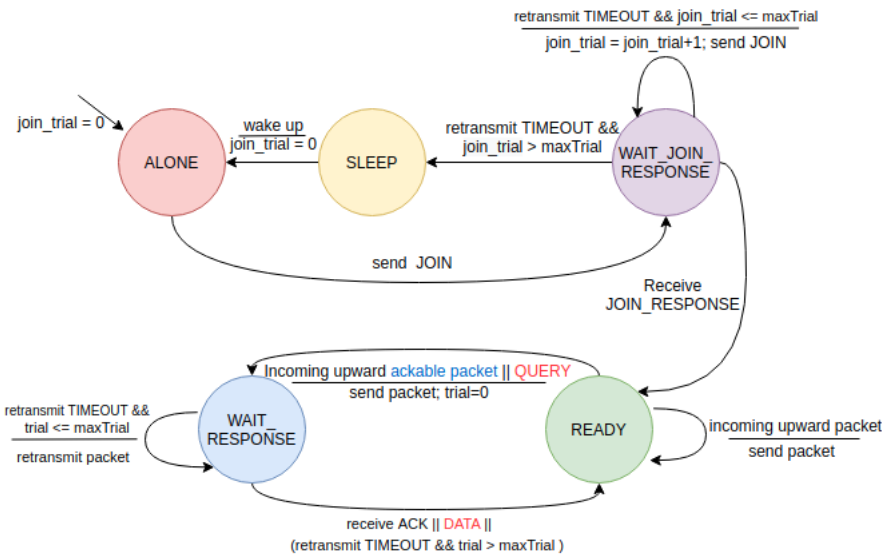


FIGURE 3.4 – Diagramme d'état des racines RPL.

3.4.2 États de la racine LoRa

Le diagramme d'état de la Figure 3.5 illustre les états de la racine LoRa pour chaque racine RPL.

À l'initialisation, la racine RPL se trouve dans l'état UNKNOWN, c'est-à-dire qu'elle ne connaît pas encore la racine RPL. Une fois une trame JOIN valide reçue, elle répond avec la JOIN_RESPONSE contenant le préfixe LoRa qui sera utilisé pour cette racine RPL et se trouve dans l'état NOT_JOINED. Dans cet état, si une trame JOIN est reçue elle sera retransmise. L'état n'est quitté qu'une fois la première trame reçue. À partir de cet état, la racine RPL a rejoint le réseau LoRa et peu donc envoyer des trames de type QUERY ou DATA et recevoir des trames de type DATA ou ACK.

L'état transitoire NOT_JOINED est ajouté entre les états UNKNOWN et READY car le préfixe LoRa utilisé dans l'adresse source de la trame JOIN (respectivement pour l'adresse de destination de la JOIN_RESPONSE) n'est pas son préfixe définitif. C'est en fait un préfixe LoRa provisoire qui est le premier octet de son *node-id*.

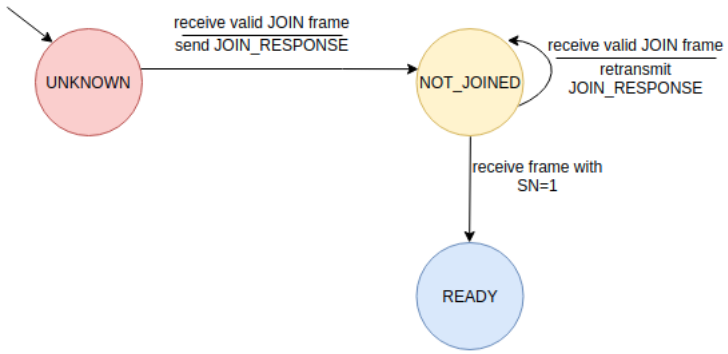


FIGURE 3.5 – Diagramme d'état de la racine LoRa.

3.4.3 Construction du réseau

À 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 LoRa du réseau de la racine RPL.

Si la trame JOIN ou la trame JOIN_RESPONSE ne sont pas reçues, la racine RPL retransmettra la trame JOIN à l'expiration du timer *retransmit_timer* déclenché à l'envoi de la trame JOIN. Cette retransmission se répète au maximum MAX_TRIAL fois. Si ce seuil est dépassé, la communication est considérée comme échouée. Dans ce cas, la racine RPL attend w secondes avant de répéter le processus. w est défini par l'Équation 3.1, dans laquelle r est un nombre aléatoire. Cette équation indique que w est compris entre 1 minute et 3 minutes. Pendant ce temps d'attente, la radio LoRa peut être désactivée pour économiser de l'énergie.

$$w = (60 + r) \mod 180 \tag{3.1}$$

Une fois le préfixe reçu, la racine RPL peut initialiser le réseau RPL et y diffuser le préfixe IPv6 défini dans la solution retenue de la Section 3.2.

Le diagramme de séquence de la Figure 3.6 illustre le processus de construction du réseau.

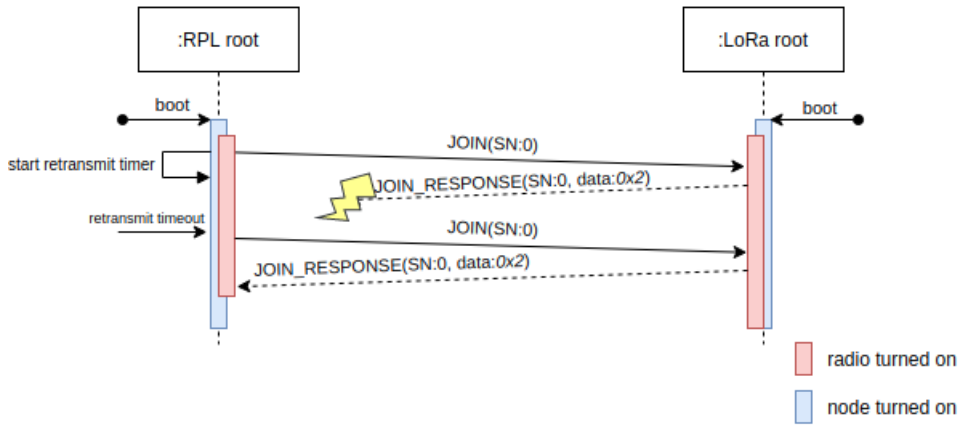


FIGURE 3.6 – Diagramme de séquence de la construction du réseau.

3.4.4 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 (flag K à *true*), la racine RPL n'enverra pas d'autres trames tant que l'acquittement n'est pas reçu (état WAIT_RESPONSE). À 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. Lorsque la racine LoRa reçoit une trame DATA avec un numéro de séquence déja reçu, c'est-à-dire plus petit que celui attendu, cette trame est considérée comme une demande de retransmission, et donc la dernière trame envoyée est retransmise. Après 3 retransmissions 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 (flag K à *false*), 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 3.7 illustre le déroulement des communications montantes. Le diagramme ne l'illustre pas, mais lorsque la radio n'émet pas ou qu'elle n'est pas en écoute (c'est-à-dire qu'aucune trame n'est attendue) elle est donc au repos. Si c'est possible la radio peut être mise en mode sommeil quand elle ne transmet pas et qu'aucune trame n'est attendue.

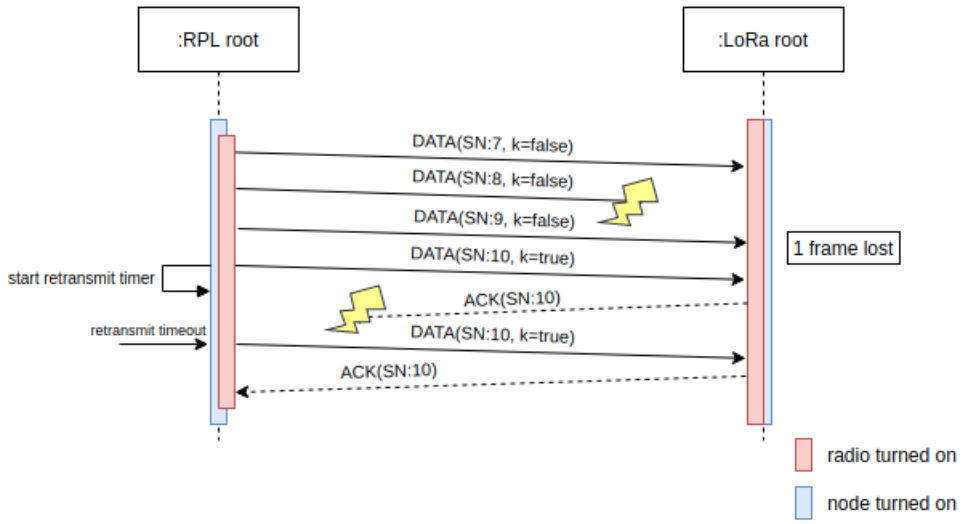


FIGURE 3.7 – Diagramme de séquence du trafique montant.

3.4.5 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 *true* pour toutes les trames envoyées à l'exception de la dernière.

Si la racine RPL reçoit une trame DATA avec le flag *next* à *true*, elle envoie une QUERY pour recevoir la prochaine trame DATA et elle ne réactive pas le timer *query_timer*.

Comme pour les communications montantes, les retransmissions sont initiées par les racines RPL. Le timer *retransmit_timer* est activé lorsque la QUERY est envoyée. Si les trois retransmissions échouent, le timer *query_timer* est redémarré.

Lorsque la racine RPL reçoit une trame DATA avec le flag *next* à *false* ou un acquittement, elle redémarre le timer *query_timer*

Le diagramme de séquence de la Figure 3.8 illustre le déroulement des communications descendantes. Comme pour les communications montantes, le diagramme n'illustre pas quand la radio est au repos. Quand c'est possible elle peut être mise en mode sommeil.

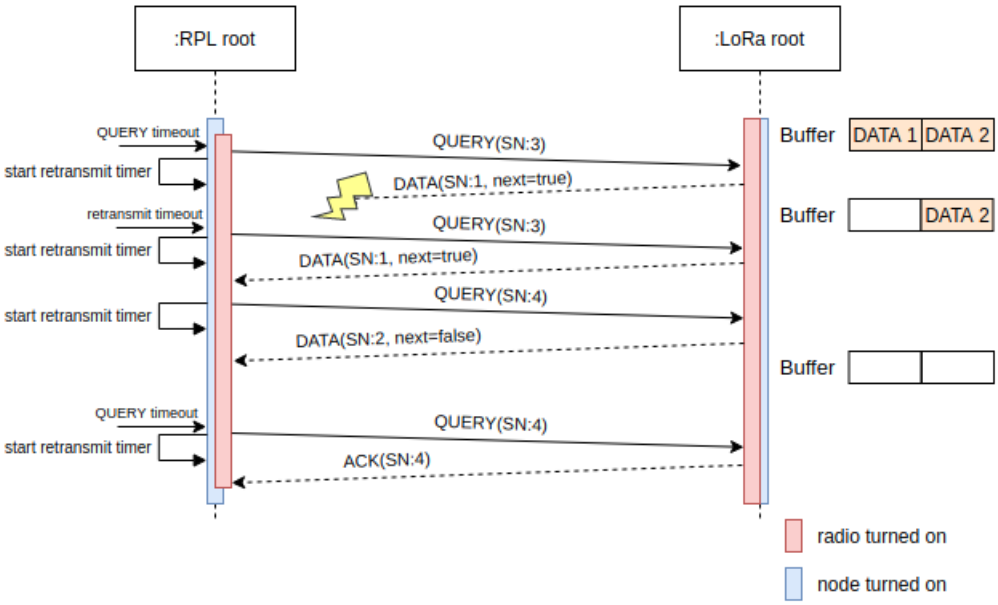


FIGURE 3.8 – Diagramme de séquence du trafic descendant.

3.4.6 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. Une trame reçue avec un numéro de séquence inférieur au compteur *expected_sn* est considérée comme une demande de retransmission.

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. Les trames JOIN et JOIN_RESPONSE ont donc un numéro de séquence égale à zéro.

3.5 Plateformes de développement

Cette section décrit le matériel utilisé pour mettre en oeuvre la topologie décrite en section 3.1. 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. Le Zolertia RE-Mote Rev.B est également utilisé pour les noeuds RPL.

3.5.1 RN2483

Le RN2483 (Fig. 3.9) est un modem LoRa compatible LoRaWAN^{TM} basse énergie. D'après Nestor Ayuso [35], il intègre un transceiver Semtech SX1276 [2] 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 [6], sa portée maximale est de 15km en agglomération et 5km en zone urbaine. Comme l'illustre la Figure 3.9, pour ce mémoire, le RN2483 a été monté sur une carte d'interface réalisée par B.Quoitin qui comporte deux LEDs, une petite antenne hélicoïdale ainsi que les connecteurs permettant d'utiliser des câbles de prototypages.

FIGURE 3.9 – RN2483.

La Figure 3.10 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.

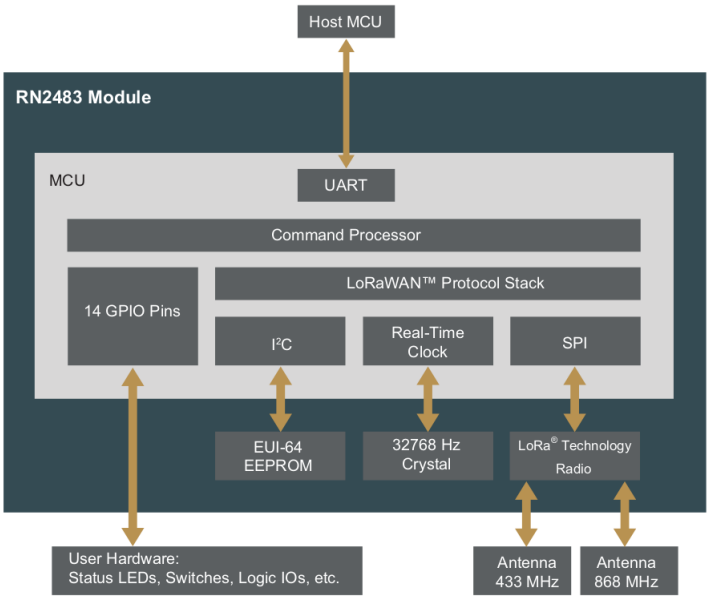


FIGURE 3.10 – Schéma-bloc du RN2483 [6].

La Table 3.2 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 3.2 – Consommation de courant (à 25 °C) [6].

3.5.2 Raspberry Pi

Le Raspberry 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, elles 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. 3.11). 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 3.3 reprend les principales caractéristiques de ce modèle.

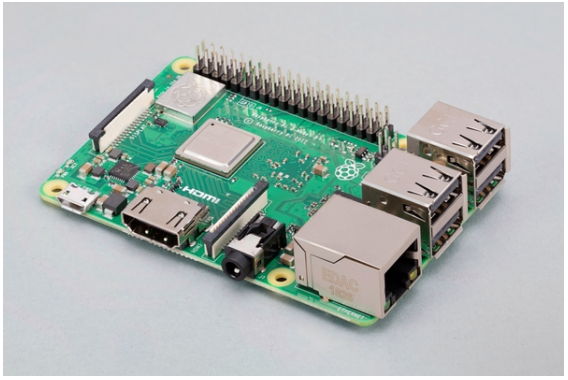


FIGURE 3.11 – Raspberry Pi 3B+ [36].

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 3.3 – Spécifications du Raspberry Pi 3B+ [36].

3.5.3 Zolertia RE-Mote

Pour ce mémoire, la plateforme Zolertia RE-Mote Rev.B (Fig. 3.12) 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€ [37].

FIGURE 3.12 – Zolertia RE-Mote révision B [37].

La Table 3.4 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 3.4 – Spécifications du Zolertia RE-Mote rev.b [38].

Chapitre 4

Mise en oeuvre

Ce chapitre décrit la mise en oeuvre de l'architecture définie dans le chapitre 3.

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 son 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.

4.1 Racine LoRa

Dans la topologie choisie à la section 3.1, 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` faisant référence à une instance de la classe `NetworkStack`. Cette classe donne accès 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'enregistrer 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.

4.1.1 Couche physique

La couche physique définit le format des adresses et trames LoRaMAC, ainsi 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 leurs définitions établies aux sections 3.2 et 3.3. Ces deux classes possèdent les fonctions nécessaires pour sérialiser les trames LoRaMAC (resp. désérialiser) en (resp. depuis) une chaîne de caractères hexadécimale 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 est envoyée, il est bloqué par une variable conditionnelle jusqu'au moment où le Thread de réception lui signale qu'une réponse attendue en retour de la trame envoyée est 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 du driver pour la couche supérieure 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

Mis à part la fonction `getFrame()`, toutes les autres fonctions alimentent le buffer d'envoi avec la trame UART appropriée. Les trames UART (en caratères ASCII) construites et les réponses UART attendues correspondent au format définit dans la référence des commandes du module RN2483 [39].

4.1.2 Couche MAC

L'implémentation du protocole LoRaMAC n'utilise pas explictement les états définis dans la Section 3.4. Cependant, son fonctionnement est identique à celui décrit par la machine à 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 qu'elle 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 et 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 envoie la prochaine trame disponible dans le buffer de cet enfant.

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 nécessite un acquittement, celui-ci est envoyé.

Fonctions publiques de la couche MAC

La couche MAC possède trois fonctions accessibles par une couche supérieure.

- **`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-à-dire, initialise la couche physique, désactive le watchdog timer, met le RN2483 en mode écoute et démarre le Thread de réception.

4.1.3 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 convertie en adresse LoRaMAC. Le reste du paquet est la payload de la trame LoRaMAC.

Les fonctions publiques de la couche d'interface 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 lorsqu'un paquet est disponible
- **`send(ip_packet: IPv6)`**
  Permet d'envoyer un paquet IPv6

4.2 Racine RPL

Dans la topologie choisie à la Section 3.1, une racine RPL est la plateforme de développement Zolertia RE-Mote Rev.B à laquelle la carte d'interface du RN2483 est raccordée par des câbles de prototypage. La Figure 4.1 illustre le raccordement entre le RN2483 et le Zolertia.

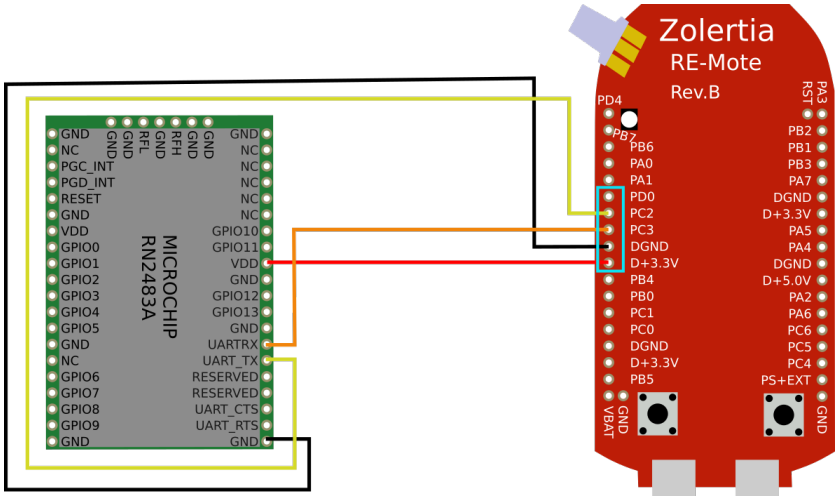


FIGURE 4.1 – Schéma du raccordement du RN2483 au Re-Mote.

Comme l'indique la Table 4.1 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 car ils sont accessibles à l'extérieur 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 pla-

teforme sont nécessaires. 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/loramacv2`. Dans l'implémentation réalisée, le *framer*, qui encode et décode les trames LoRaMAC ainsi que la gestion du buffer sont implémentés dans des fichiers distincts.

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 4.1 – Extrait de la table de descrition des pins du Zolertia [38].

4.2.1 Buffer

La gestion du buffer de la pile réseau LoRa est implémentée dans le fichier `lorabuf.c`. Cette implémentation est similaire à celle utilisée dans les différents buffers de Contiki, comme, le buffer IP implémenté dans `os/net/ipv6/uipbuf.c`. Ainsi, le buffer est divisé en 3 tableaux déclarés comme suit :

```
lorabuf_attr_t lorabuf_attrs[LORABUF_NUM_ATTRS];
lora_addr_t lorabuf_addrs[LORABUF_NUM_ADDRS];
uint8_t lorabuf[LORA_PAYLOAD_BYTE_MAX_SIZE];
```

Le premier tableau contient les attributs d'un paquet. Les 7 attributs disponibles sont décrit dans la Table 4.2.

Attribut	Description
LORABUF_ATTR_UART_CMD	Commande UART
LORABUF_ATTR_UART_EXP_RESP1	Première réponse UART possible
LORABUF_ATTR_UART_EXP_RESP2	Deuxième réponse UART possible
LORABUF_ATTR_MAC_CONFIRMED	Flag K d'une trame LoRaMAC
LORABUF_ATTR_MAC_SEQNO	Numéro de séquence d'une trame LoRaMAC
LORABUF_ATTR_MAC_NEXT	Flag next d'une trame LoRaMAC
LORABUF_ATTR_MAC_CMD	Commande MAC d'une trame LoRaMAC

Table 4.2 – Attribut d'un paquet.

Le deuxième tableau contient les adresses LoRaMAC source et destination d'un paquet et le dernier tableau contient la payload du paquet.

En plus de ces trois tableaux, un tableau de caractère, `char lorabuf_c[LORA_FRAME_CHAR_MAX_SIZE+LORA_UART_CHAR_SIZE]`, est utilisé pour stocker les commandes ASCII qui sont reçues ou à envoyer en UART pour le RN2483.

Pour manipuler ces tableaux, une série de fonctions permettent d'obtenir la longueur des données, de changer ou d'obtenir la valeur d'un paramètre ou d'une adresse ou encore d'écrire ou de lire des données.

4.2.2 Couche physique

La couche dite physique est utilisée pour les communications avec le RN2483. Elle est implémentée dans le fichier `loraphy.c`. Pour l'envoi d'un message UART, deux étapes sont nécessaires :

1. La préparation de message UART qui est réalisée par la fonction

   ```
   loraphy_prepare_data(loraphy_command_t command,
           loraphy_param_t parameter, char* value, int16_t len,
           loraphy_cmd_response_t exp1, loraphy_cmd_response_t exp2)
   ```

   Cette fonction prépare le buffer pour l'envoi en y définissant les réponses attendues, la chaîne de caractère qu'elle construit, ainsi que la longueur de cette chaîne.

2. L'écriture du buffer de caractères sur la connexion UART qui est possible si et seulement si une réponse valide au dernier message a été reçue.

Lors la réception d'un message UART, chaque caractère est reçu par la fonction `int uart_rx(unsigned char c)`. Une fois le caractère de fin reçu, la fonction `void loraphy_input()` vérifie si la réponse reçue est celle attendue. Si c'est le cas, elle appelle une fonction callback enregistrée par la couche supérieure qui prend en paramètre un enum `loraphy_sent_status_t`. La valeur passée à cette fonction est `LORAPHY_INPUT_DATA` si la réponse UART reçue contient des données, sinon la valeur passée est `LORAPHY_SENT_DONE`. C'est donc à la couche supérieure d'attendre l'appel à la fonction callback de cette couche pour envoyer un prochain message UART.

Pour faciliter l'utilisation de la couche physique, des macros ont été ajoutées dans le fichier d'entête. Par exemple pour l'envoi de données, la macro suivante est définie comme suit :

```
1 define LORAPHY_TX(data, len){\
2     loraphy_prepare_data(LORAPHY_CMD_RADIO_TX, LORAPHY_PARAM_NONE, data, len,\
3         LORAPHY_CMD_RESPONSE_RADIO_TX_OK, LORAPHY_CMD_RESPONSE_RADIO_ERR);\
4     loraphy_send();\
5 }
```

Cette macro permet donc d'appeller la fonction qui prépare le message UART avec les bons paramètres et d'appeler la fonction d'envoi. Les autres macros `LORAPHY_SET_PARAM(param, value)`, `LORAPHY_RX()`, `LORAPHY_SLEEP(duration)` sont identiques, mis à part les paramètres de la fonction `loraphy_prepare_data()` qui sont adaptés.

4.2.3 Couche MAC

La couche MAC implémente le protocole décrit à la Section 3.4. À l'initialisation, tous les paramètres de la radio sont initialisés selon la configuration se trouvant dans le fichier `loramac-conf.h`. La Table 4.3 décrit tous ses paramètres ainsi que les valeurs possibles pour ceux-ci [1].

Paramètre	Description	valeurs possibles	valeur par défaut
LORA_RADIO_BW	la bande passante (KHz)	125, 250, 500	125
LORA_RADIO_CR	le coding rate	4/5, 4/6, 4/7, 4/8	4/5
LORA_RADIO_FREQ	la fréquence (Hz)	de 433050000 à 434790000 ou de 863000000 à 870000000	868100000
LORA_RADIO_MODE	méthode de modulation	lora, fsk	lora
LORA_RADIO_PWR	puissance de la radio	-3 à 15	1
LORA_RADIO_SF	le spreading factor	sf7, sf8, sf9, sf10, sf11, sf12	sf10

TABLE 4.3 – Paramètres de la radio LoRa.

Une fois la configuration de la radio terminée, le processus de la couche MAC est démarré et une trame JOIN est envoyée.

Dans Contiki, un processus est déclaré comme suit :

```
1 PROCESS(the_process, "Process name");
```

Le premier argument de cette macro est la variable faisant référence au processus et le second, le nom du processus. Une fois déclaré, le processus peut être créé comme suit :

```
1 PROCESS_THREAD(the_process, ev, data)
2 {
3     PROCESS_BEGIN();
4
5         /*process code*/
6
7     PROCESS_END();
8 }
```

Les variables `ev` et `data` contiennent respectivement la valeur d'un évènement entrant et un pointeur optionnel vers un argument de l'évènement. Le code qui se trouve entre les instructions des lignes 3 et 7 dépend du processus.

1. Toutes les valeurs doivent être sous forme de chaîne de caractères.

Le processus `loramac_process` gère les appels à la couche physique pour l'envoi des trames. Ainsi, quand une trame est prête à être envoyée, il la prépare, l'envoie via la couche physique et, si une réponse est attendue, met la radio en mode écoute et n'enverra pas d'autres trames tant qu'une réponse n'est pas reçue.

La préparation d'une trame consiste à indiquer le bon numéro de séquence dans le buffer, si ce n'est pas une retransmission, à copier les informations de la trame dans les variables `last_sent_frame` et `last_sent_data` qui contiennent respectivement l'entête et les données de la trame. Cette copie permet de ne plus dépendre du buffer après l'appel à la fonction d'envoi, ce qui permet par exemple à la couche supérieure d'écrire la prochaine trame à envoyer dans le buffer directement après l'appel à la fonction d'envoi et facilite également une retransmission si nécessaire.

De plus, lorsqu'une trame de données avec le flag *has_next* à *true* est reçue, il met la radio en mode écoute. Toutes ces actions sont réalisées en réactions à des évènements qui sont créés à l'initialisation de la couche MAC.

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

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

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

Pour la couche MAC, les évènements suivants sont utilisés :

- `loramac_event_output` : indique au processus MAC qu'une trame est prête à être envoyée
- `loramac_phy_done` : indique au processus MAC que l'envoi de la commande UART est terminé

La couche MAC utilise deux timers : le `retransmit_timer` et le `query_timer`. Le premier est activé quand la radio est mise en mode réception (c'est-à-dire en attente d'une réponse) et stoppe si la réponse est reçue. S'il expire, la dernière trame envoyée est renvoyée pour un maximum de 3 fois. Le deuxième est utilisé pour la transmission de trames avec la commande QUERY à intervalle régulier. Il est donc activé une fois que le réseau est rejoint et à chaque transmission d'une QUERY. Il est stoppé lors de la réception d'une trame DATA ou d'une trame ACK si la dernière trame envoyée est une QUERY.

Contiki possède plusieurs types de timers mais seul deux donnent la possibilité de passer en paramètre une fonction callback qui est appelée lors de l'expiration du timer : les `ctimer` et les `rtimer`. Le deuxième à l'avantage

d'éviter les interruptions. Cependant la documentation indique que Contiki supporte au maximum qu'un seul `rtimer`. Comme ce type de timer est utilisé par TSCH dans Contiki, le `ctimer` est donc utilisé.

Lors de l'expiration d'un timer, la trame à envoyer est écrite dans le buffer. Donc malgré que c'est une interruption, cela ne pause aucun problème au niveau de la gestion du buffer car la trame en cours d'envoi a été copiée par la couche MAC.

L'utilisation d'un `ctimer` est illustrée par le code suivant :

```
1  static struct ctimer timer;
2
3  void
4  on_timeout(void *ptr)
5  {
6      ctimer_reset(&timer);
7      printf("timeout !\n");
8  }
9
10 void
11 fun(void)
12 {
13     ctimer_set(&timer, 3*CLOCK_SECOND, on_query_timeout, NULL);
14 }
```

A la première ligne le timer est déclaré. À la ligne 13, le timer est démarré. Le deuxième paramètre est le temps d'expiration du timer. Ici, le temps fixé est de 3 secondes. Comme le temps est exprimé en ticks de l'horloge, la variable `CLOCK_SECOND` qui est le nombre de ticks par secondes est utilisée. Le paramètre suivant est la fonction callback appelée lorsque le timer expire.

4.2.4 Couche IP

Cette couche joue le rôle d'interface entre le protocole LoRaMAC et la pile réseau de Contiki. Elle est implémentée dans le fichier `lorabridge.c`.

Les paquets destinés à une adresse pour lesquels aucune route n'est trouvée sont récupérés 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`. 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 est choisie, car il n'est actuellement pas possible d'utiliser deux interfaces réseaux dans Contiki. C'est donc le meilleur moyen trouvé pour récupérer les paquets destinés à l'interface LoRa. L'autre solution envisagée est de considérer la pile réseau LoRa comme une application Contiki ce qui rend le développement d'une application plus complexe.

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 facilitant l'accès au buffer uIP comme la macro `UIP_IP_BUF` qui permet d'accéder directement au header IPv6 qui 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 };
```

La fonction `output()` se charge de copier le buffer `uip_buf` vers le buffer LoRaMAC sauf, l'adresse source et l'adresse destination. Seul l'adresse source est convertie en adresse LoRaMAC par la fonction `void ipv62lora(uip_ip6addr_t *src_addr, lora_addr_t *dest_addr)` implémentée dans le fichier `loraaddr.c` dans lequel la gestion des adresses LoRaMAC est implémentée. L'adresse de destination utilisée sera celle de la racine.

Enfin, la fonction `void bridge_input()` se charge d'écrire dans l'`uip_buf` une trame LoRaMAC reçue à destination d'un noeud du réseau RPL (racine incluse). Pour cela, elle convertit l'adresse source et l'adresse destination via la fonction `void lora2ipv6(lora_addr_t *src_addr, uip_ip6addr_t *dest_addr)` puis, les intercale entre le header IPv6 et la payload.

Comme la structure `uip_fallback_interface` l'indique, une fonction `void init()` est également implémentée. Cette fonction désactive la radio, initialise la couche MAC. Une fois le réseau LoRaMAC est rejoint, la couche MAC appelle la fonction `lora_network_joined()` qui défini le préfixe IPv6 qui sera diffusé dans le réseau RPL, allume la radio et démarre le protocole de routage.

Chapitre 5

Validation et évaluation

5.1 Validation

Cette section décrit les différents tests réalisés pour valider le prototype développé. Ceux-ci sont réalisés avec l'exemple `udp-rpl-lora` qui permet de tester le trafic montant, descendant et bidirectionnel de messages UDP entre la racine LoRa et un noeud/racine RPL. Cet exemple envoie ou reçoit des paquets en fonction du mode de fonctionnement paramétré. Les noeuds du réseau (racine LoRa, racine RPL ou noeud rpl) peuvent fonctionner selon les quatre modes de fonctionnement suivant :

- **pingpong** : dans ce mode, le premier message "PING" est envoyé par un noeud RPL à la racine LoRa. Une fois la QUERY reçue, celle-ci répond avec un message "PONG". Lorsque ce message est reçu par le noeud RPL, il répond avec un "PING" et ainsi de suite
- **receiver** : le noeud dans ce mode est destiné à recevoir des messages. Il n'envoie donc aucun message à l'exception d'un noeud RPL qui doit envoyer un premier message pour que l'application de la racine LoRa puisse connaître son adresse IP.
- **sender** : le noeud dans ce mode envoie un nombre déterminé de messages. Si la racine LoRa est dans ce mode, elle doit attendre un premier message provenant d'un noeud RPL pour démarrer l'envoi des messages
- **sleep** : dans ce mode, destiné uniquement aux racines RPL, le noeud n'envoie pas de message et n'est pas destiné à en recevoir

Les tests sont réalisés avec différentes topologies, des simulations de perte de paquets et, contrairement à la phase de développement, la racine LoRa n'est plus un ordinateur portable mais le Raspberry Pi.

5.1.1 Une Racine RPL

La première topologie utilisée pour la validation du prototype (Fig.5.1) est la plus simple possible : la racine LoRa et une racine RPL.

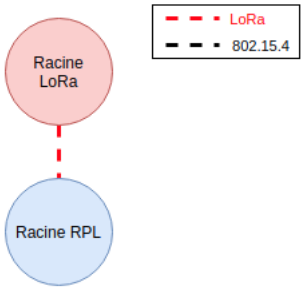


FIGURE 5.1 – Racine LoRa et racine RPL.

Construction du réseau

Le premier élément vérifié est la construction du réseau qui est illustré par les logs suivants issus de la racine RPL.

```
1  [MAC:loramac_root_start] Initial node address: 27:24859
2  [PHY:loraphy_init      ] Init LoRaPHY
3  [PHY:write_uart        ] Write to UART {mac pause} len=9
4  [PHY:loraphy_input     ] Receive UART data {4294967245} (len=10)
5  [PHY:write_uart        ] Write to UART {radio set bw 125} len=16
6  [PHY:loraphy_input     ] Receive UART data {okdio set bw 125} (len=2)
7  [PHY:write_uart        ] Write to UART {radio set cr 4/5} len=16
8  [PHY:loraphy_input     ] Receive UART data {okdio set cr 4/5} (len=2)
9  [PHY:write_uart        ] Write to UART {radio set freq 868100000} len=24
10 [PHY:loraphy_input     ] Receive UART data {okdio set freq 868100000} (len=2)
11 [PHY:write_uart        ] Write to UART {radio set mod lora100000} len=18
12 [PHY:loraphy_input     ] Receive UART data {okdio set mod lora100000} (len=2)
13 [PHY:write_uart        ] Write to UART {radio set pwr 1ora100000} len=15
14 [PHY:loraphy_input     ] Receive UART data {okdio set pwr 1ora100000} (len=2)
15 [PHY:write_uart        ] Write to UART {radio set sf sf10a100000} len=17
16 [PHY:loraphy_input     ] Receive UART data {okdio set sf sf10a100000} (len=2)
17 [MAC:set_conf          ] LoRaMAC configuration done
```

Les lignes 3 et 4 correspondent à l'initialisation de la couche physique. Celle-ci envoie la commande ASCII `mac pause` au RN2483 pour qu'il désactive sa couche MAC (LoRaWAN). La réponse reçue est le temps en millisecondes durant lequel LoRaWAN sera désactivé (cela correspond à 49 jours). D'après la documentaion du RN2483 [39] il n'existe aucun moyen de mettre LoRaWAN en pause indéfiniment. Cela se justifie par le fait que cette commande a pour but de désactiver la couche MAC pour configurer la radio et

non l'utiliser sans la couche MAC. Ensuite, entre les lignes 5 à 17, a lieu l'initialisation de la couche MAC où celle-ci paramètre le RN2483 avec les valeurs indiquées dans le fichier `loramac-conf.h` du service LoRaMAC.

Une fois l'initialisation terminée, la trame JOIN peut être envoyée à la racine LoRa. Les logs ci-dessous illustrent cet envoi ainsi que la mise en écoute de la radio LoRa après l'envoi.

```
1  [MAC:loramac_process] BEGIN LORAMAC process
2  [PHY:write_uart     ] Write to UART {radio set wdt 1450000000} len=19
3  [MAC:send_join_reque] Send JOIN request
4  [PHY:loraphy_input  ] Receive UART data {okdio set wdt 1450000000} (len=2)
5  [MAC:loramac_process] Receive packet to send
6  [MAC:prepare_last_sent_frame] Preparation of a frame
7  [MAC:loramac_process] Send packet
8  [PHY:write_uart     ] Write to UART {radio tx 1B611B0100000000} len=25
9  [PHY:loraphy_input  ] Receive UART data {okdio tx 1B611B0100000000} (len=2)
10 [PHY:loraphy_input  ] Response is not the expected.
11 [PHY:loraphy_input  ] Receive UART data {radio_tx_ok611B0100000000} (len=11)
12 [MAC:loramac_process] Frame need a response
13 [PHY:write_uart     ] Write to UART {radio rx 0k611B0100000000} len=10
14 [MAC:loramac_process] set retrans |
   ↪   [PHY:loraphy_input] Receive UART data {okdio rx 0k611B0100000000} (len=2)
15 [PHY:loraphy_input] Response is not the expected.
16 it timer
17 [MAC:loramac_process] Waiting ...
```

La ligne 2 n'est exécutée qu'au lancement du processus MAC. Elle permet de définir le délai, en millisecondes du radio watchdog timer du RN2483, ici à 14500 ms.

Pour tous les logs, lorsque le buffer de commandes UART est affiché, comme à la ligne 2, son contenu complet est affiché. Il faut donc utiliser la longueur affichée pour déterminer les données qui sont envoyées ou lues. Par exemple à la ligne 2, les données écrites sont `radio set wdt 14500` car la longueur indiquée est 19.

Ensuite, la trame est préparée et envoyée par la couche MAC aux lignes 6 et 7. A la ligne 10, la couche physique indique que la réponse reçue (`ok`) n'est pas celle attendue. En effet, pour l'envoi d'une trame la réponse attendue est soit `radio_err`, soit `radio_tx_ok`. Cette dernière, indiquant que l'envoi est effectué, est reçue à la ligne 11.

Comme ce type de trame nécessite une réponse, la radio est mise en mode écoute à la ligne 13 et le *retransmit_timer* est activé à la ligne 14. L'impression du log "set retransmit timer" de la ligne 14 est coupée par une interruption et se termine à la ligne 16. Cette interruption provient de la réception du message UART "ok" confirmant la mise en écoute du RN2483.

Finalement la dernière ligne indique que le processus MAC attend la prochaine trame à envoyer.

Une fois la trame JOIN envoyée, elle est reçue par la racine LoRa. Les logs suivants illustrent le traitement de cette trame. Il est à noté qu'avant ces logs, la racine LoRa est initialisée de la même manière qu'une racine RPL mis à part le watchdog timer du RN2483 qui est désactivé. Une fois l'initialisation réalisée, la racine LoRa se met en mode écoute à la première ligne des logs suivants.

```
1  lora_phy:_uart_tx         : PHY TX:radio rx 0
2  lora_phy:_uart_rx: PHY RX : {ok}
3  lora_phy:_process_response: UNEXPECTED UART RESPONSE
4  lora_phy:_uart_rx: PHY RX : {radio_rx  1B611B0100000000}
5  lora_mac:_rx_process: MAC RX: LoraFrame(src_addr=LoraAddr(prefix=27,
   ↪  node_id=24859), dest_addr=LoraAddr(prefix=1, node_id=0),
   ↪  command=<MacCommand.JOIN: 0>, payload='', seq=0, k=False,
   ↪  has_next=False)
6  lora_mac:_rx_process      : look child with prefix: 27
7  lora_mac:_rx_process      : child is: None
8  lora_mac:_on_join         : new child Child(2:24859) created
9  lora_mac:_on_join         : send join response
10 lora_phy:phy_send: MAC TX:{LoraFrame(src_addr=LoraAddr(prefix=1,
   ↪  node_id=0), dest_addr=LoraAddr(prefix=27, node_id=24859),
   ↪  command=<MacCommand.JOIN_RESPONSE: 1>, payload='02', seq=0, k=False,
   ↪  has_next=False)}
11 lora_phy:_uart_tx         : PHY TX:radio tx 0100001B611B010002
12 lora_phy:_uart_rx         : PHY RX: {ok}
13 lora_phy:_process_response: UNEXPECTED UART RESPONSE
14 lora_phy:_uart_rx         : PHY RX: {radio_tx_ok}
15 lora_phy:_uart_tx         : PHY TX:radio rx 0
```

La trame JOIN est reçue à la ligne 4 et le résultat du décodage de ces octets se trouve à la ligne 5. Ensuite, cette trame est traitée par la fonction `_on_join`. Comme le réseau RPL de préfixe 27 n'existe pas, un nouveau réseau avec le préfixe disponible, ici 2, est créé. Ensuite, la JOIN_RESPONSE est envoyée à la racine RPL avec comme payload son préfixe et la racine LoRa se remet en mode écoute.

Enfin, les logs suivants illustrent la réception et le traitement de la JOIN_RESPONSE.

```
1  [PHY : loraphy_input     ] Receive UART data {radio_rx  0100001B611B010002} (len=28)
2  [ADDR: loraaddr_is_in_dag] ADDR 27:24859 is in dag
3  [MAC : on_join_response  ] Receive JOIN RESPONSE (sn: 0, len: 1)
4  [ADDR: loraaddr_compare  ] Compare 27:24859 and 27:24859. addresses are equal
5  [MAC : on_join_response  ] Valid JOIN RESPONSE. Stop retransmit timer
6  [MAC : on_join_response  ] New node address: 2 :24859
```

```
7  [MAC : on_join_response  ] Start query timer
8  [MAC : set_state         ] STATE ALONE -> READY
9  [BRIDGE: lora_network_joined] LoRa network joined
10 [APP : app_process       ] Network joined
11 [APP : app_process       ] Sending PING 0 to 5f:756d:6f6e:735f:6c6f:7261:6d61:6300
```

La JOIN_RESPONSE est reçue à la première ligne. La racine RPL vérifie d'abord que la trame LoRa reçue est destinée à un noeud de son réseau RPL ou à elle-même à la ligne 2. Ensuite, les valeurs de différents champs de la trame sont vérifiés aux lignes 3, 4 et 5. Comme la trame reçue est valide, le timer servant à la retransmission peut être stoppé. L'adresse du noeud est actualisée à la ligne 6 avec le préfixe reçu. L'état de la couche MAC est changé à READY et finalement, la couche MAC notifie à l'interface avec la couche IP (bridge) que le réseau LoRa est rejoint. De cette manière, la couche MAC de contiki ainsi que RPL peuvent être activés par le bridge. Enfin, le premier message UDP est bien envoyé par l'application une fois que le réseau RPL est atteignable à la dernière ligne. L'adresse IPv6 de destination peut être n'importe qu'elle adresse tant qu'elle n'appartient pas à ce réseau RPL. Ainsi, le paquet UDP est transmis à la racine LoRa.

Les analyses des précédents logs montrent donc que la procédure de construction du réseau avec la topologie choisie est fonctionnelle lorsque aucune retransmission ne doit avoir lieu. Les logs de la racine RPL ci-dessous illustrent le déroulement de la construction du réseau si la première trame JOIN n'est pas reçue par la racine LoRa ou que la JOIN_RESPONSE n'est pas reçue par la racine RPL. Cette simulation est réalisée en modifiant le code de la racine LoRa pour qu'elle ne réponde pas à la première trame JOIN.

```
1  [MAC  :send_join_request    ] Send JOIN request
2  [MAC  :loramac_process      ] Receive packet to send
3  [MAC  :loramac_process      ] Frame need a response
4  [MAC  :loramac_process      ] Set retrans
5  [MAC  :loramac_process      ] Waiting ...
6  [PHY  :loraphy_input        ] Receive UART data {radio_err0k611B0100000000} (len=9)
7  [MAC  :on_retransmit_timeout] Retransmit timeout
8  [MAC  :on_retransmit_timeout] Copy last sent frame to lorabuf
9  [MAC  :on_retransmit_timeout] Attempts: 1
10 [MAC  :loramac_process      ] Receive packet to send
11 [MAC  :loramac_process      ] Frame need a response
12 [MAC  :loramac_process      ] Waiting ...
13 [MAC  :on_join_response     ] Receive JOIN RESPONSE (sn: 0, len: 1)
14 [MAC  :on_join_response     ] Valid JOIN RESPONSE. Stop retransmit timer
15 [BRIDGE: lora_network_joined] LoRa network joined
```

Aux lignes 1 à 5, comme illustré précédemment, la trame JOIN est envoyée. La ligne 6 est le message UART reçu lors de l'expiration du watchdog

timer du RN2483. À partir de cette ligne, la radio n'est donc plus en écoute. À la ligne suivante, le timer *retransmit timer* expire et donc la trame est retransmise aux lignes 8 à 11. Ensuite, la réponse à cette trame est reçue et le réseau LoRa rejoint, comme expliqué précédemment.

Il est à noter que certains logs comme ceux de la couche physique ou de l'initialisation du noeud ne sont plus présents et ne le seront plus dans les logs suivants car leur fonctionnement est déja démontré.

L'analyse des logs ci-dessus, montre donc que la procédure de construction du réseau avec cette topologie est validée également lors de la perte d'une trame.

Trafic montant

Le trafic montant est constitué de paquets UDP envoyés d'un noeud RPL vers la racine LoRa. Les logs ci-dessous illustrent la réception d'un paquet UDP par la racine LoRa.

```
1  pyloramac.lora_phy: phy_rx      : MAC RX
2  pyloramac.lora_mac: _on_data    : RECEIVE DATA frame LoraFrame(^
   ↪  src_addr=LoraAddr(prefix=2, node_id=24859),^
   ↪  dest_addr=LoraAddr(prefix=1, node_id=0),^
   ↪  command=<MacCommand.DATA: 2>,^
   ↪  payload='600000000011114022 3D162E00114E5A6D65737361676520 37',^
   ↪  seq=8, k=False, has_next=False)
3  pyloramac.lora_mac: _on_data    : expected sn: 8
4  pyloramac.lora_ip : lora_to_ipv6: LoraAddr: 2:24859 converted to^
   ↪  IPv6: fd00:0000:0000:0002:0212:4b00:060d:611b
5  pyloramac.lora_ip : lora_to_ipv6: LoraAddr: 1:0 converted to^
   ↪  IPv6: fd00:0000:0000:0001:0212:4b00:060d:0000
6  __main__          :on_packet    : Receive packet:
7  ###[ IPv6 ]###
8    version   = 6
9    plen      = 17
10   nh        = UDP
11   src       = fd00::2:212:4b00:60d:611b
12   dst       = fd00::1:212:4b00:60d:0
13 ###[ UDP ]###
14      sport     = 8765
15      dport     = 5678
16      len       = 17
17      chksum    = 0x4e5a
18 ###[ Raw ]###
19         load      = 'message 7'
20 pyloramac.lora_phy: phy_rx      : MAC RX
```

À la deuxième ligne, la trame LoRaMAC reçue est affichée avec les valeurs de ces différents champs. Ensuite, aux lignes 4 et 5, les adresses IPv6 sont

correctement converties et enfin, le paquet IPv6 est construit par la couche IP et transmis à l'application.

Le flag k de cette trame LoRaMAC est à `False`. Cela signifie qu'elle n'attend pas d'acquittement en retour. Donc, si un paquet est perdu, il n'est pas retransmis.

Les logs suivants illustrent le test avec des trames nécessitant un acquittement et la simulation d'un acquittement perdu. Cette simulation est réalisée en modifiant le code de la racine LoRa pour qu'elle ne réponde pas à la première réception d'une trame DATA.

```
1  [APP: app_process            ] Sending message 0 to^
   ↪  5f:756d:6f6e:735f:6c6f:7261:6d61:6300
2  [MAC: loramac_process        ] Send packet
3  [MAC: loramac_process        ] Frame need a response
4  [MAC: loramac_process        ] Set retrans
5  [MAC: loramac_process        ] Waiting ...
6  [MAC: on_retransmit_timeout  ] Retransmit timeout
7  [MAC: on_retransmit_timeout  ] Copy last sent frame to lorabuf
8  [MAC: on_retransmit_timeout  ] Attempts: 1
9  [MAC: loramac_process        ] Receive packet to send
10 [MAC: prepare_last_sent_frame] Preparation of a frame that is retransmitted
11 [MAC: loramac_process        ] Send packet
12 [MAC: loramac_process        ] Frame need a response
13 [MAC: loramac_process        ] Set retrans
14 [MAC: loramac_process        ] Waiting ...
15 [MAC: on_ack                 ] Receive ACK (sn: 1, len: 0)
16 [MAC: on_ack                 ] Stop retransmit timer
17 [APP: app_process            ] Sending message 17 to::8d68:2000
```

Comme l'acquittement n'est pas reçu, le *retransmit_timer* expire à la ligne 6. Ensuite, la trame est retransmise et l'acquittement retransmis par la racine LoRa est reçu à la ligne 15. L'envoi de messages peut ensuite reprendre. Entre la première ligne et la dernière, on remarque que l'application a envoyé 16 messages (non illustrés). Dans l'exemple utilisé, un message est envoyé toutes les secondes. Le délai entre le message 0 et le message 17 provient des temps d'envoi et du temps d'expiration du *retransmit_timer* fixé pour ce test à 15 secondes. Lorsque le processus MAC ne fait rien, il rend la main à l'ordonnanceur de Contiki et donc l'application reprend la main et envoie des paquets UDP. Les 16 messages envoyés par l'application entre le message 0 et le message 17 ne sont pas envoyés par LoRaMAC car la réponse au message 0 n'est pas reçue. Une fois l'acquittement reçu, LoraMAC peut envoyer le message 17.

Les résultats de ces différents tests valident donc le fonctionnement du trafic montant.

Trafic descendant

Comme l'illustre le diagramme de la Figure 5.2, pour que l'envoi de messages descendants soit possible, après la procédure d'initialisation, la racine RPL doit envoyer le premier message UDP, contenant la payload "SYN" pour l'exemple, afin que l'application de la racine LoRa puisse connaître l'adresse IP de destination. Ensuite, après l'envoi de la première QUERY, l'envoi des messages peut commencer.

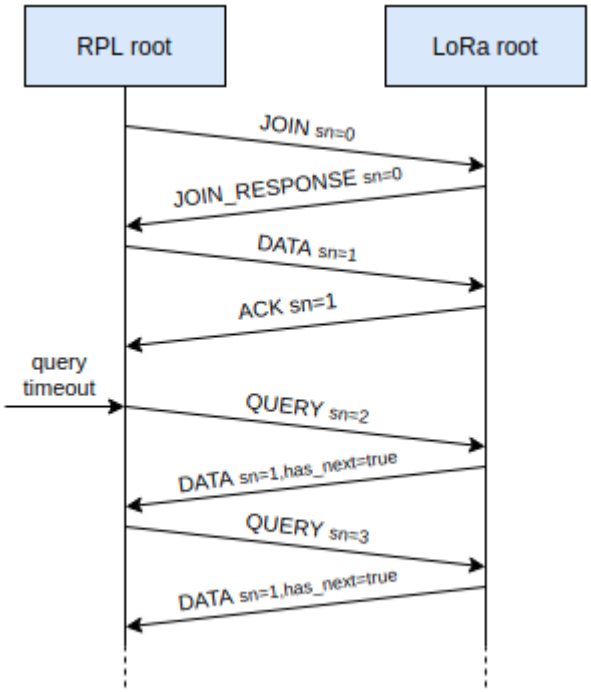


FIGURE 5.2 – Diagramme de séquence du trafic descendant.

Les logs suivants illustrent la réception de ces messages, ainsi que l'envoi d'une QUERY avant la réception d'un message descendant.

```
[APP: app_process    ] Network joined
[APP: app_process    ] Sending SYN to 5f:756d:6f6e:735f:6c6f:7261:6d61:6300
[MAC: loramac_process] Send packet
[MAC: loramac_process] Frame need a response
[MAC: on_ack         ] Receive ACK (sn: 1, len: 0)
[MAC: on_query_timeou] Query timeout
[MAC: loramac_process] Send packet
[MAC: loramac_process] Frame need a response
[MAC: on_data        ] Receive DATA (sn: 1, len: 25)
[APP: udp_rx         ] Received 'message 0' from fd00::1:212:4b00:60d:0
[MAC: on_data        ] Waiting for another frame. Send QUERY
```

```
12 [MAC: loramac_process] Send packet
13 [MAC: loramac_process] Frame need a response
14 [MAC: on_data        ] Receive DATA (sn: 2, len: 25)
15 [APP: udp_rx         ] Received 'message 1' from fd00::1:212:4b00:60d:0
16 [MAC: on_data        ] Waiting for another frame. Send QUERY
```

Dans le cas où une QUERY n'est pas reçue par la racine LoRa ou que sa réponse n'est pas reçue par la racine RPL, celle-ci doit demander une retransmission. Les logs suivants illustrent la retransmission de la QUERY envoyée à la ligne 2.

```
1  [APP:udp_rx                 ] Received 'message 2' from fd00::1:212:4b00:60d:0
2  [MAC:on_data                ] Waiting for another frame. Send QUERY
3  [MAC:loramac_process        ] Frame need a response
4  [MAC:loramac_process        ] Waiting ...
5  [MAC:on_retransmit_timeout  ] Retransmit timeout
6  [MAC:on_retransmit_timeout  ] Attempts: 1
7  [MAC:prepare_last_sent_frame] Preparation of a frame that is retransmitted
8  [MAC:loramac_process        ] Send packet
9  [MAC:loramac_process        ] Frame need a response
10 [MAC:loramac_process        ] Waiting ...
11 [MAC:on_data                ] Receive DATA (sn: 4, len: 25)
12 [APP:udp_rx                 ] Received 'message 3' from fd00::1:212:4b00:60d:0
```

Ces différents tests permettent donc de valider le fonctionnement du trafic descendant.

Trafic bidirectionnel

Finalement, le test du trafic bidirectionnel permet de vérifier qu'il ne reste pas un défaut dans l'implémentation qui rendrait ce type de trafic impossible. Les logs suivants de l'application de la racine RPL, illustrent le bon fonctionnement de ce type de trafic.

```
1  [APP:app_process] Network joined
2  [APP:app_process] Sending PING 0 to 5f:756d:6f6e:735f:6c6f:7261:6d61:6300
3  [APP:udp_rx     ] Received 'PONG 0' from fd00::1:212:4b00:60d:0
4  [APP:udp_rx     ] Sending PING 1 to fd00::1:212:4b00:60d:0
5  [APP:udp_rx     ] Received 'PONG 1' from fd00::1:212:4b00:60d:0
6  [APP:udp_rx     ] Sending PING 2 to fd00::1:212:4b00:60d:0
7  [APP:udp_rx     ] Received 'PONG 2' from fd00::1:212:4b00:60d:0
8  [APP:udp_rx     ] Sending PING 3 to fd00::1:212:4b00:60d:0
9  [APP:udp_rx     ] Received 'PONG 3' from fd00::1:212:4b00:60d:0
10 [APP:udp_rx     ] Sending PING 4 to fd00::1:212:4b00:60d:0
```

5.1.2 Une racine RPL avec deux enfants

La deuxième topologie utilisée pour la validation du prototype est composée de la racine LoRa et d'une racine RPL possédant deux enfants (Fig 5.3). Comme l'envoi de messages depuis la racine RPL est déja testé, son mode de fonctionnement sera `SLEEP` pour tous les tests suivants. C'est-à-dire qu'elle n'enverra pas de messages UDP et qu'elle n'est pas destinée à en recevoir.

La construction du réseau LoRaMAC est identique à celle réalisée pour la topologie précédente. Elle n'est donc pas détaillée. De plus, comme cette topologie ainsi que les suivantes sont plus complexes, par souci de clarté, les logs sont remplacés par des diagrammes de séquence des messages envoyés.

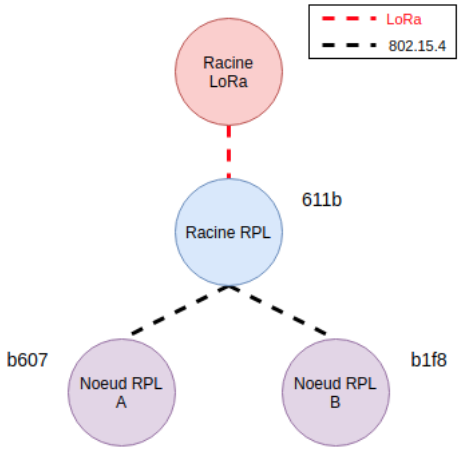


FIGURE 5.3 – Racine LoRa, racine RPL avec deux enfants.

Trafic montant

Le diagramme de la Figure 5.4, issu des logs des différents noeuds, illustre les messages envoyés entre ceux-ci.

Les messages illustrés entre les noeuds RPL A et B et la racine RPL sont les paquets UDP envoyés avec 802.15.4 et les messages entre la racine RPL et la racine LoRa sont des trames LoRaMAC.

Comme indiqué sur le diagramme, le flag K des trames DATA est à *true* pour cet exemple, ce qui signifie que ces trames nécessitent un acquittement en retour.

Pour cet exemple, les paquets UDP sont envoyés à intervalle d'une seconde. Le diagramme montre que certains de ces paquets sont perdus car le débit de données 802.15.4 est plus élevé que celui de LoRaMAC.

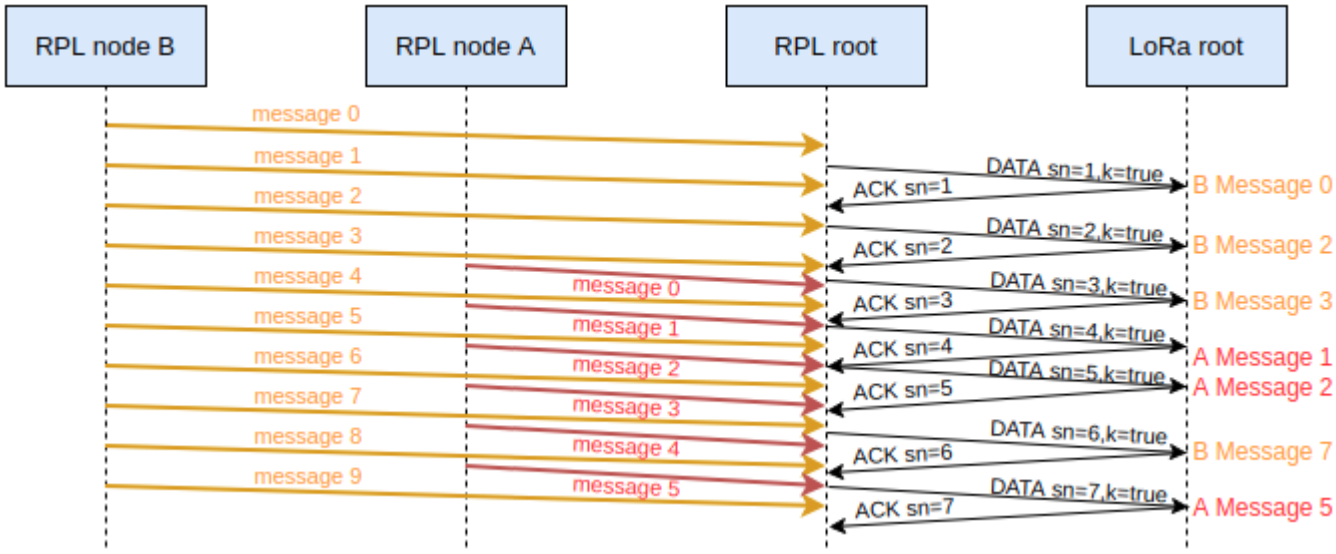


FIGURE 5.4 – Diagramme de séquence du trafic montant de la deuxième topologie.

Ces résultats montrent donc que le trafic montant pour cette topologie est fonctionnel mais que certains choix d'implémentation au niveau de la racine RPL réduisent les performances du réseau.

Trafic descendant

Le diagramme de la Figure 5.5, issu des logs des différents noeuds, illustre les messages envoyés entre ceux-ci.

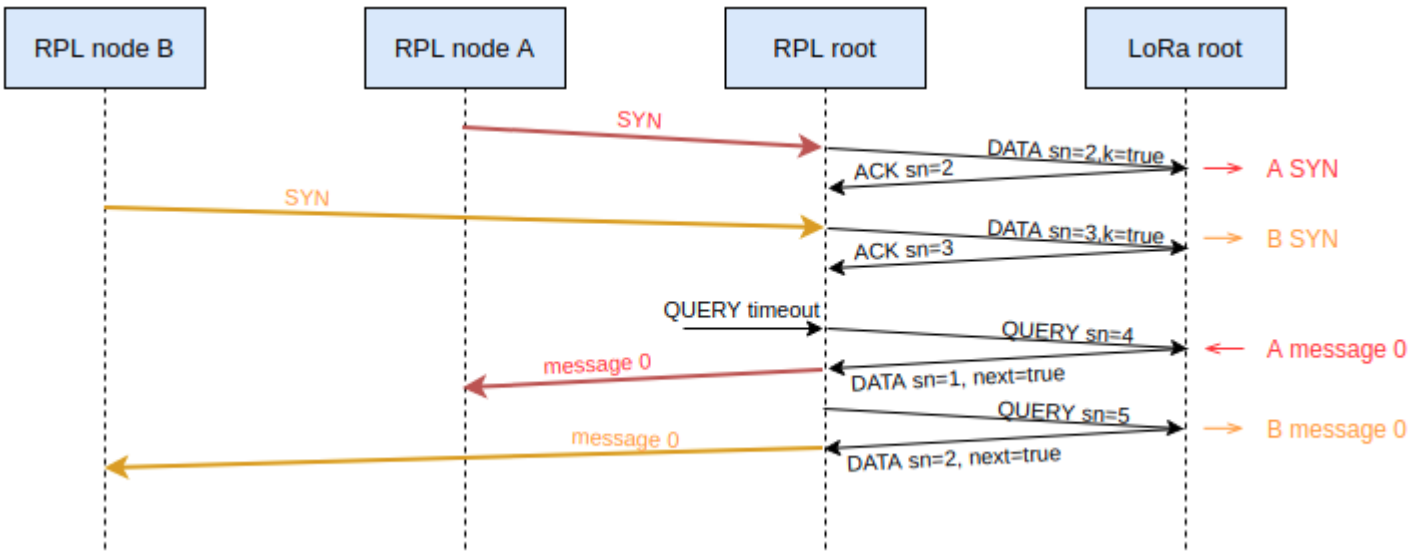


FIGURE 5.5 – Diagramme de séquence du trafic descendant de la deuxième topologie.

Une fois le réseau construit, chaque noeud RPL envoie un premier paquet UDP qui permet à l'application de la racine LoRa de connaître son adresse IP. Sur le diagramme, le numéro de séquence de la trame LoRaMAC transportant le premier message UDP est 2 car avant l'envoi de celui-ci, une QUERY qui n'est pas illustrée a été envoyée.

Une fois ce premier message envoyé, il faut attendre qu'une QUERY soit envoyée par la racine RPL pour que le premier message descendant lui soit envoyé. Une fois celui-ci reçu, la racine RPL le transmet via RPL au bon noeud. Comme la trame LoRaMAC a le flag *next* à *true*, la racine RPL répond directement avec une autre QUERY.

Le diagramme n'illustre que l'envoi des deux premiers messages. Cependant, le test réalisé envoie une dizaine de messages. Les logs correspondant à cet essai montrent que l'envoi des messages UDP se fait par vague de 3 à 6 messages. Cela vient du délai d'envoi de messages indiqué dans l'application de la racine LoRa. Cela implique qu'à un certain moment, le buffer contenant les messages pour ce réseau RPL peut être vide. Dans ce cas, le flag *next* de la trame LoraMAC de type DATA est à *false*. De ce fait, la racine RPL n'envoie pas directement une QUERY en retour. Pour cela, il faut attendre l'expiration du *query_timer*.

Trafic bidirectionnel

Le diagramme de séquence de la Figure 5.6 illustre les messsages échangés entre les différents noeuds de la topologie pour un trafic bidirectionnel.

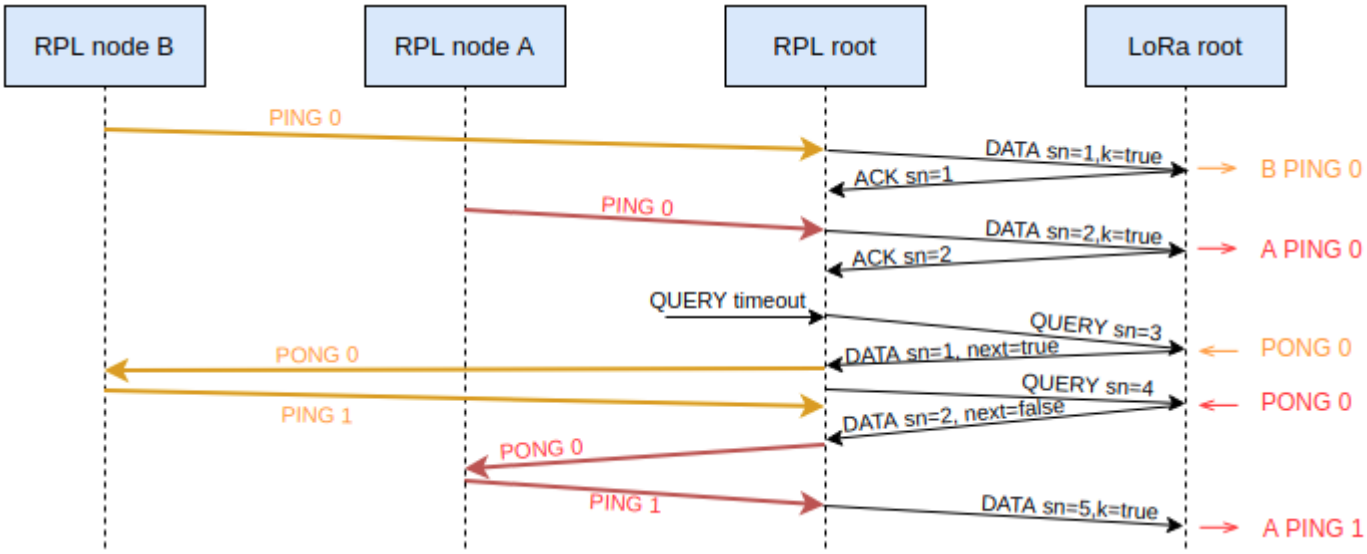


FIGURE 5.6 – Diagramme de séquence du trafic bidirectionnel de la deuxième topologie.

Dans cet exemple, le premier message "PING X" est envoyé par un noeud RPL. En réaction à la réception de ce message, l'application de la racine LoRa envoie un message "PONG X" et en réaction à la réception de ce message, le noeud RPL envoie de nouveau un message "PING X" et ainsi de suite.

Une fois les messages "PONG X" reçus des deux noeuds RPL, conformément à la définition du protocole LoRaMAC, il faut attendre l'envoi d'une QUERY déclenché par l'expiration du timer dédié.

Le diagramme illustre le fait que le message "PING 1" envoyé par le noeud RPL B n'est pas transféré par la racine RPL. En effet, à ce moment, la racine RPL attend la réponse à une QUERY. Elle ne peut donc pas envoyer un message simultanément.

Du fait que le message "PING 1" du noeud RPL B n'est jamais reçu par l'application de la racine LoRa, celle-ci n'enverra jamais le "PONG X" correspondant, et donc, la communication bidirectionnelle ne continue pas. En revanche, elle continue pour le noeud RPL A.

Ce dernier test montre que l'implémentation est fonctionnelle et confirme que les performances de celle-ci ont besoin d'être améliorées.

5.1.3 Deux racines RPL et un noeud RPL

La dernière topologie utilisée pour la validation du prototype (Fig 5.7) est composée de la racine LoRa, de deux racines RPL dont une qui possède un enfant. L'utilisation de cette topologie pour la validation permet de vérifier que la racine RPL supporte bien les communications avec plusieurs racines RPL.

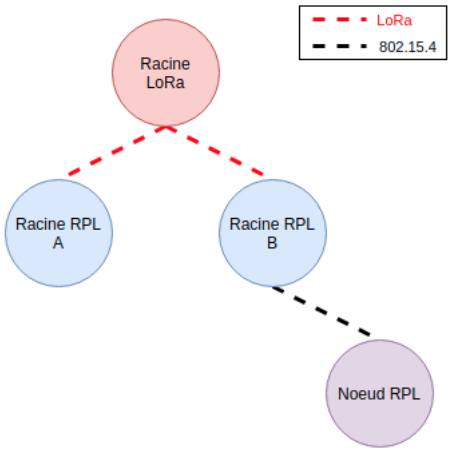


FIGURE 5.7 – Racine LoRa, deux racines RPL et un noeud RPL.

Construction du réseau

Le diagramme de séquence de la Figure 5.8 illustre l'échange de messages, issus des logs, pour la construction du réseau.

Tout d'abord, la première racine (RPL root A) est démarrée et elle rejoint le réseau LoRa. De cette façon, son enfant (RPL node) peut envoyer un premier message qu'elle transmet à la racine LoRa. Ensuite, la deuxième racine RPL est démarrée. Elle rejoint le réseau LoRa comme la première racine et reçoit le préfixe 3. Une fois le réseau rejoint, elle peut envoyer son premier message. Si les deux racines sont démarrées simultanément, le scénario attendu est une collision des deux trames JOIN qui sont ensuite retransmises par les racines RPL, car la racine LoRa ne peut pas les recevoir à cause de la collision.

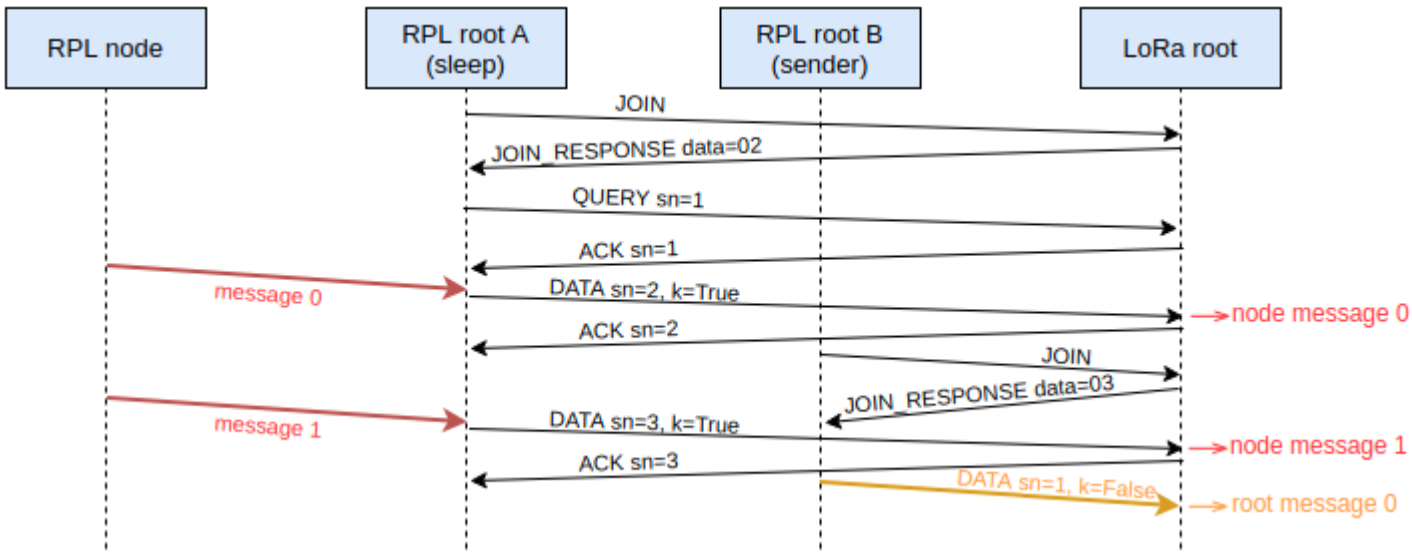


FIGURE 5.8 – Construction du réseau avec deux racines RPL.

Avec cette analyse, on peut donc conclure que la construction du réseau est fonctionnelle avec plusieurs racines RPL.

Trafic montant

Le diagramme de séquence de la Figure 5.9 illustre l'échange de trames, issu des logs, pour l'envoi de messages montants.

Pour cet exemple, le flag k des trames DATA est activé pour les trames provenant du noeud RPL et désactivé pour les trames provenant de la racine RPL B. Le diagramme illustre bien que les acquittements sont envoyés quand ils sont nécessaires.

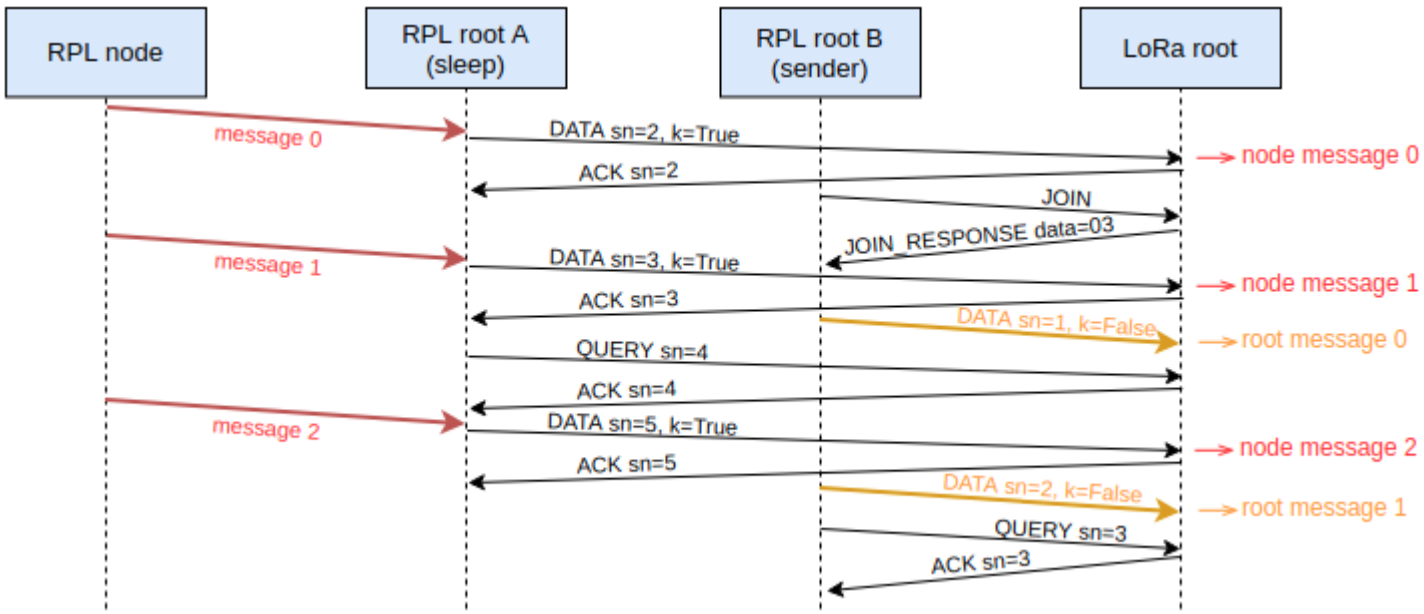


FIGURE 5.9 – Trafic montant avec deux racines RPL.

Ce test valide donc le fonctionnement du prototype malgré la perte de paquets constatée dans les logs des différents noeuds causée par des collisions de trames LoRa et les choix d'implémentation pour la racine RPL.

Le trafic descendant et bidirectionnel n'est pas illustré car son fonctionnement est identique à ce qui a déja été expliqué et les problèmes rencontrés sont égalements identiques.

5.1.4 Discussion

Les différents tests réalisés valident le fonctionnement du prototype. Cependant, des pertes de paquets sont constatées. Ces pertes proviennent de l'implémentation et de la collision de trames LoRa quand plusieurs racines RPL sont présentes dans le réseau.

Pour diminuer ces pertes plusieurs solutions peuvent être mises en place. Au niveau de l'implémentation, le buffer LoRaMAC qui ne peut actuellement contenir qu'un seul paquet peut être remplacé par un buffer pouvant contenir plusieurs paquets. Cette solution implique une utilisation de la mémoire plus importante. L'agrégation de plusieurs trames peut également être envisagée. Le flag k des trames DATA peut également être désactivé. Ainsi, la racine RPL ne doit pas attendre l'acquittement et peut envoyer directement la prochaine trame ce qui augmente donc le débit possible. Cependant, si ce flag est désactivé, l'envoi des trames n'est pas garanti.

Au niveau des collisions de trames LoRa, le protocole mis en place peut être amélioré vers Slotted ALOHA ou un protocole plus complexe comme une adaptation de TSCH comme la solution proposée par Maite Bezunartea et ses co-auteurs dans leur article [32].

5.2 Evaluation

L'objectif de cette section est d'évaluer le prototype réalisé. Pour ce faire, la consommation électrique d'une racine RPL est analysée ainsi que les débits pouvant être atteints.

5.2.1 Consommation électrique

L'instrument utilisé pour mesurer la consommation électrique est le Joulescope [40]. Cet instrument de mesure possède les caractéristiques suivantes :

- Plage de tension : -1V à 15V
- Plage de courant : -1A à 3A
- Une résolution de 1.4 nA
- Une Bande passante de 250 kHz avec 2 millions d'échantillons par seconde
- Il possède deux pins GPI (*General Purpose Input*) et deux pins GPO(*General Purpose Output*) qui permettent d'envoyer ou de recevoir des signaux vers ou en provenance d'un équipement.

RN2483

La consommation du RN2483 est analysée lorsque la radio est en émission ainsi qu'en réception. La Figure 5.10 illustre la tension et l'intensité du courant en fonction du temps lors de l'envoi de 6 trames LoRa de 100 octets chacunes. La première trame est envoyé avec $SF = 1$, la deuxième avec $SF = 2$ et ainsi de suite jusqu'à la dernière trame envoyée avec $SF = 12$.

Comme attendu, plus le SF est grand plus le temps d'émission l'est également. Les temps d'émission (TOA, Time On Air) estimés et calculés par le logiciel *LoRa Modem Calculator Tool*, fourni par Semtech, sont repris dans la Table 5.1. Ce calculateur utilise l'Équation 2.1 pour fournir ces valeurs.

Pour ces mesures, les paramètres de la radio sont les suivants :

Bande passante	125 KHz
Coding rate	4/5
Taille du préambule	8 symboles
Header explicite	Oui
Présence de la CRC	Oui
Fréquence	868100000 Hz

Un dernier paramètre important, le *Low Datarate Optimiser* n'est pas mentionné dans la configuration ci-dessus. Pour faire correspondre les valeurs de mesures avec les valeurs calculées de la Table 5.1, cet optimiseur est désactivé pour les SF7 à SF10 et activé pour les SF11 et SF12 dans le calculateur.

Ceci correspond aux informations de la datasheet du SX1276 [2] qui indique d'activer le bit *LowDataRateOptimize*, qui augmente la robustesse d'un lien LoRa, lorsque la durée d'un symbole excède 16ms. C'est le cas pour cette configuration avec les SF11 et SF12 où la durée d'un symbole donnée par le calculateur est respectivement de 16.38 ms et 32.77 ms. L'hypothèse est donc que le RN2483 active automatiquement ce paramètre si nécessaire.

Spreading Factor	TOA mesuré (ms)	TOA calculé (ms)
7	174.425	174.34
8	307.176	307.71
9	551.508	553.98
10	1020.82	1026.05
11	2214.19	2215.94
12	3941.51	3940.35

Table 5.1 – Temps d'émission de 100 octets en fonction du SF.

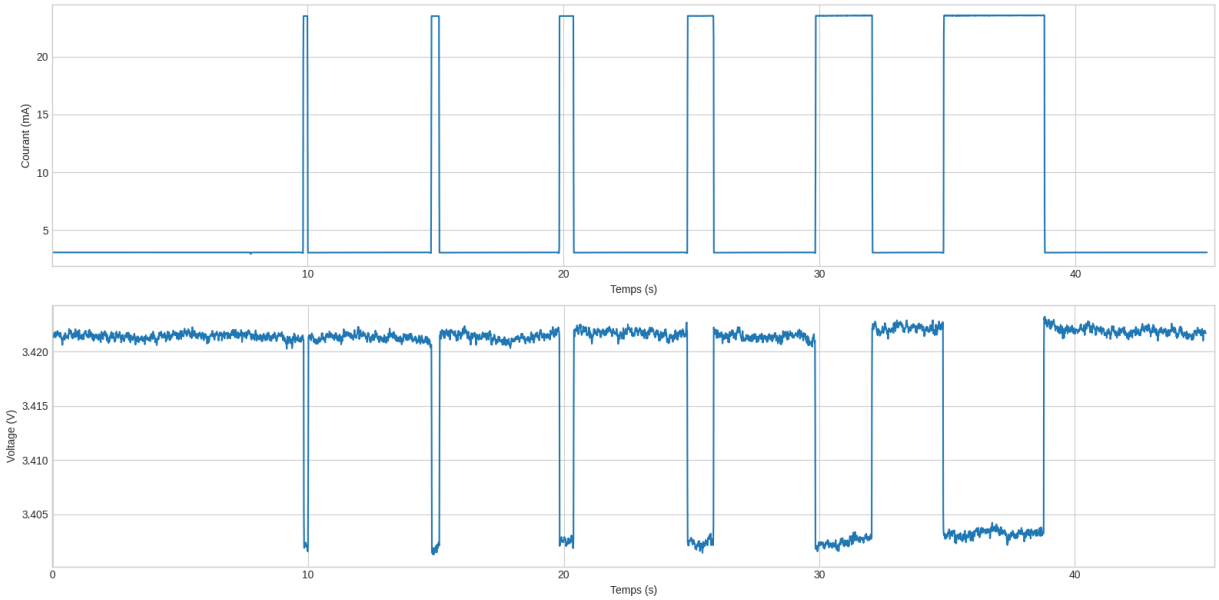


FIGURE 5.10 – Courant et tension du RN2483 en émission.

Pour le courant, les mesures relevées par le Joulescope sont de 23.5mA en moyenne avec un écart type de 0.5mA lors de l'envoi et de 3.07mA en moyenne avec écart type de 0.3mA lorsque le RN2483 est au repos.

Pour la tension, les valeurs relevées sont de 3.40V en émission et 3.42V au repos en moyenne, toutes les deux avec un écart type de 0.0014V.

Ces mesures diffèrent de celles données par la datasheet du RN2483 reprises dans la Table 3.2, dans laquelle la valeur attendue en émission est de 38.9 mA et 2.8mA au repos.

Pour la réception, la Figure 5.11 illustre la tension et l'intensité du courant en fonction du temps lorsque le RN2483 est en écoute et qu'il reçoit des données. La durée des paliers varie car les paquets ne sont pas envoyés à intervalle régulier.

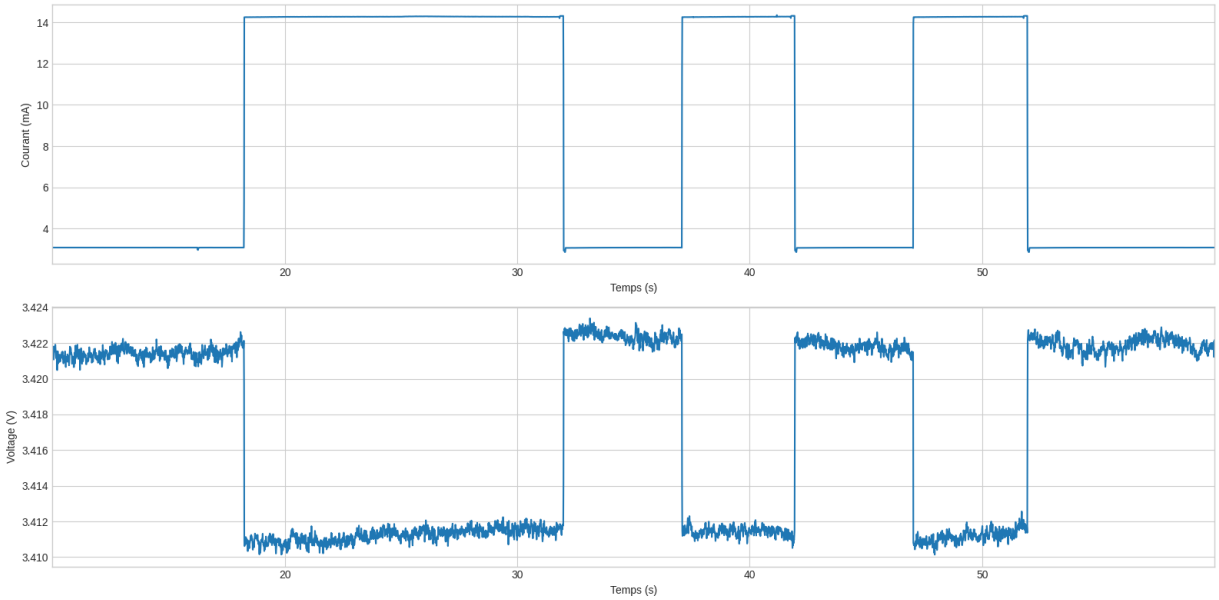


FIGURE 5.11 – Courant et tension du RN2483 en réception.

Pour le courant, les mesures relevées sont de 14.27mA en moyenne avec un écart type de 0.19mA en écoute et 3.07mA en moyenne avec un écart type de 0.17mA au repos.

Pour la tension, les mesures relevées sont de 3.41V en moyenne lorsque le RN2483 est en écoute et 3.42V en moyenne lorsqu'il est au repos. L'écart type pour les mesures de tension est de 0.001V.

La valeur mesurée du courant en réception est proche de celle indiquée dans la Table 3.2 qui est de 14.22mA.

Une autre mesure, qui n'est pas illustrée, montre que la tension ou l'intensité du courant ne varient pas si le RN2483 est en écoute et qu'il ne reçoit aucune donnée ou s'il est en écoute et qu'il reçoit des données.

Racine RPL

Les mesures suivantes sont effectuées sur une racine RPL composée du Zolertia auquel est raccordé le RN2483. La Figure 5.12 illustre la tension et l'intensité du courant en fonction du temps pour une racine RPL fonctionnant avec l'exemple `ping-pong`.

Tout d'abord, les graphiques mettent en évidence 4 pics qui correspondent à l'expiration du QUERY timer. Lors de l'expiration de ce timer, une QUERY est envoyée à la racine LoRa, qui répond ensuite avec un paquet UDP contenant un message "PONG".

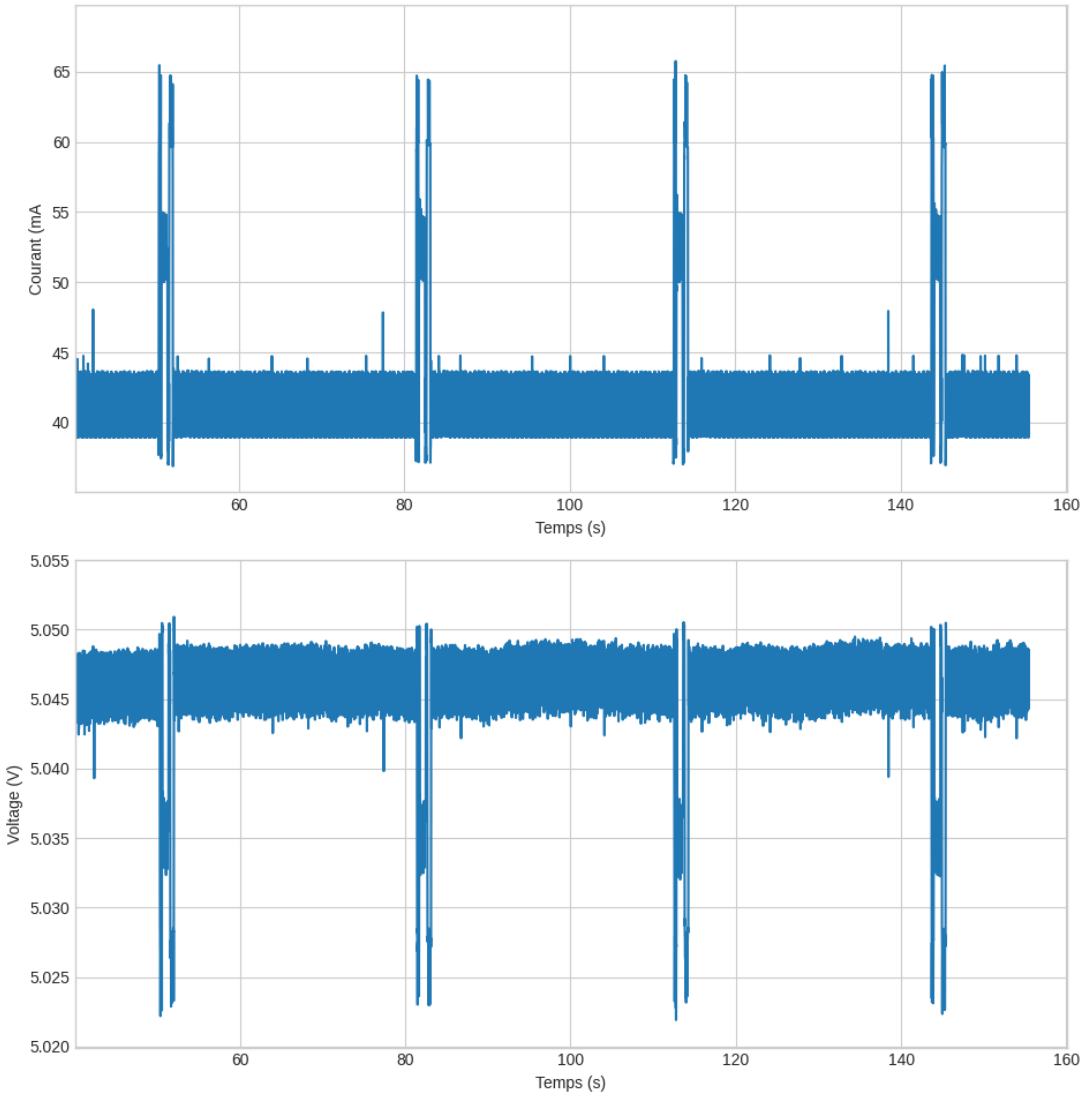


FIGURE 5.12 – Courant et tension d'une racine RPL avec l'exemple ping-pong.

Ensuite, entre les pics se trouve beaucoup de bruit. La cause de ce bruit n'est pas identifiée. Ce bruit laisse également penser que la radio 802.15.4 n'est jamais éteinte. Cela peut provenir soit d'Orchestra qui n'éteint pas la radio comme attendu ou de la désactivation de la gestion des modes de basse consommation d'énergie de Contiki. Cette désactivation est une solution proposée par l'article de blog de M.Charlier [41] pour résoudre l'apparition de l'erreur de compilation `.stack' will not fit in region FRSRAM memory error` qui intervient lors de l'activation de TSCH et de la pile IPv6. D'après

cet article, la mémoire RAM du Zolertia est divisée en deux parties de 16KB. La première est entièrement conservée dans tous les modes d'alimentation tandis que l'autre non. Ainsi, lorsque le CPU passe en mode veille, la partie non conservée sera corrompue pour économiser de l'énergie. Pour utiliser TSCH et IPv6, il est nécessaire de pouvoir utiliser plus que 16KB, c'est-à-dire, de désactiver le mode basse consommation de Contiki en ajoutant la macro suivante dans le fichier de configuration de l'application :

```
#undef LPM_CONF_ENABLE
#define LPM_CONF_ENABLE 0
```

Sur la durée de la mesure, le courant le plus haut est à 154.917 mA et le plus bas à 32.8142 mA. Entre les pics, la moyenne du courant est estimée à 39.48 mA avec un écart type maximal de 3.29 mA. À titre d'exemple, l'intensité moyenne du premier pic est estimée à 52.96 mA avec un écart type de 8.36 mA.

Sur la durée de la mesure, la tension la plus haute est à 5.0519V et la plus basse à 4.93211V. La moyenne de la tension entre les pics est estimée à 5.04 V avec un écart type de 0.003V. À titre d'exemple, la tension moyenne du premier pic est estimée à 5.03V avec un écart type de 0.008V.

Ensuite, un zoom est effectué sur un pic d'intensité afin de l'analyser. La Figure 5.13 illustre ce zoom sur l'intensité en fonction du temps suivi par l'état du GPI 0 en fonction du temps.

Ce graphique montre 3 paliers. Pour identifier la provenance de ces paliers, des signaux placés judicieusement dans l'implémentation LoRaMAC sont envoyés du Zolertia au GPI 0 du Joulescope.

Le premier palier débute à l'entré du protothread utilisé pour l'envoi de trames. Dans le cas de l'exemple `ping-pong`, cela correspond à l'envoi d'une QUERY. Ce premier palier se termine une fois l'envoi de la QUERY terminé. Sa durée est de 299.5 ms. Ensuite, le GPI 0 est désactivé pendant une période de 818.4ms, durant laquelle la racine RPL est en écoute.

Le petit pic qui suit cette période démarre lorsque la réponse à la QUERY (c'est à dire, une trame DATA contenant un paquet UDP avec comme payload la chaîne de caractère "PONG x") est reçue et se termine lorsque la réponse est décodée en une trame LoRaMAC. La durée du décodage est estimée à 7.35 ms. Après ce pic, la durée du traitement de la trame reçue est de 104.9 ms.

Le deuxième palier, comme pour le premier commence à l'entrée du protothread utilisé pour l'envoi de trames et se termine une fois la trame envoyée. Dans ce cas, ce palier correspond à l'envoi de la réponse au "PONG", c'est-à-dire, une trame DATA contenant un paquet UDP avec comme payload la chaîne de caractère "PING x". Sa durée est de 515.5 ms. Comme attendu,

le deuxième palier est plus long que le premier car la trame envoyée contient plus de données que la première.

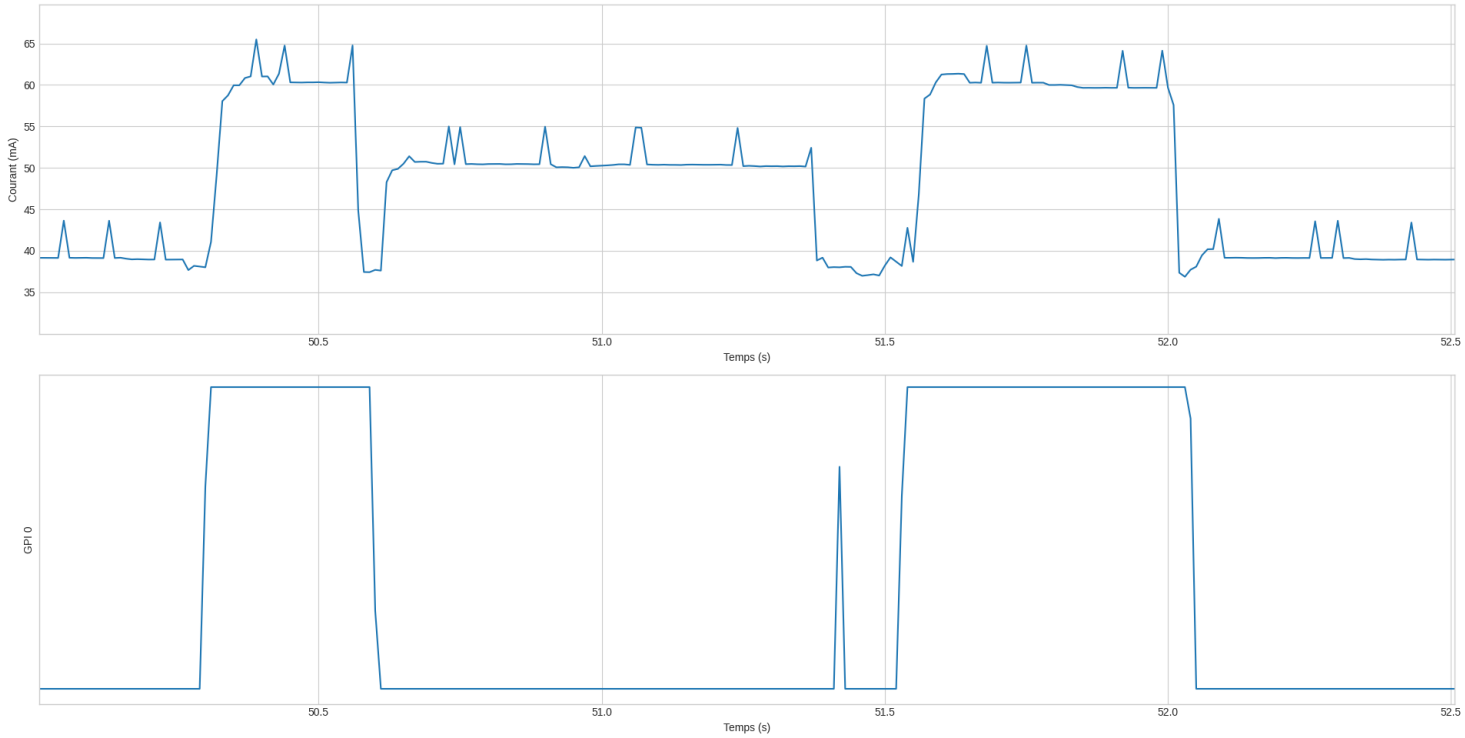


FIGURE 5.13 – Zoom sur une augmentation de courant de la figure 5.12.

5.2.2 Débit

L'objectif de cette section est d'évaluer les débits pouvant être atteints sur un lien LoRa du réseau. Comme la validation du prototype l'a illustré, ce débit est plus faible que celui du réseau RPL. C'est donc sur celui-ci que les mesures sont réalisées.

Pour effectuer cette analyse, des mesures sont réalisées pour le trafic montant sans acquittement et descendant entre une racine RPL et la racine LoRa. Le SF7 et SF12 sont utilisés pour l'envoi de paquets UDP de 10 et 200 octets. Les paquets sont envoyés à un intervalle plus petit que ce que le réseau peut supporter afin d'avoir la certitude d'atteindre le débit maximal du réseau.

Trafic montant

La Figure 5.14 illustre le débit en octets par seconde en fonction du temps. Cette mesure est réalisée en utilisant le SF7 et des paquets UDP ayant une payload de 10 octets. La ligne verte est l'instant où une QUERY est reçue.

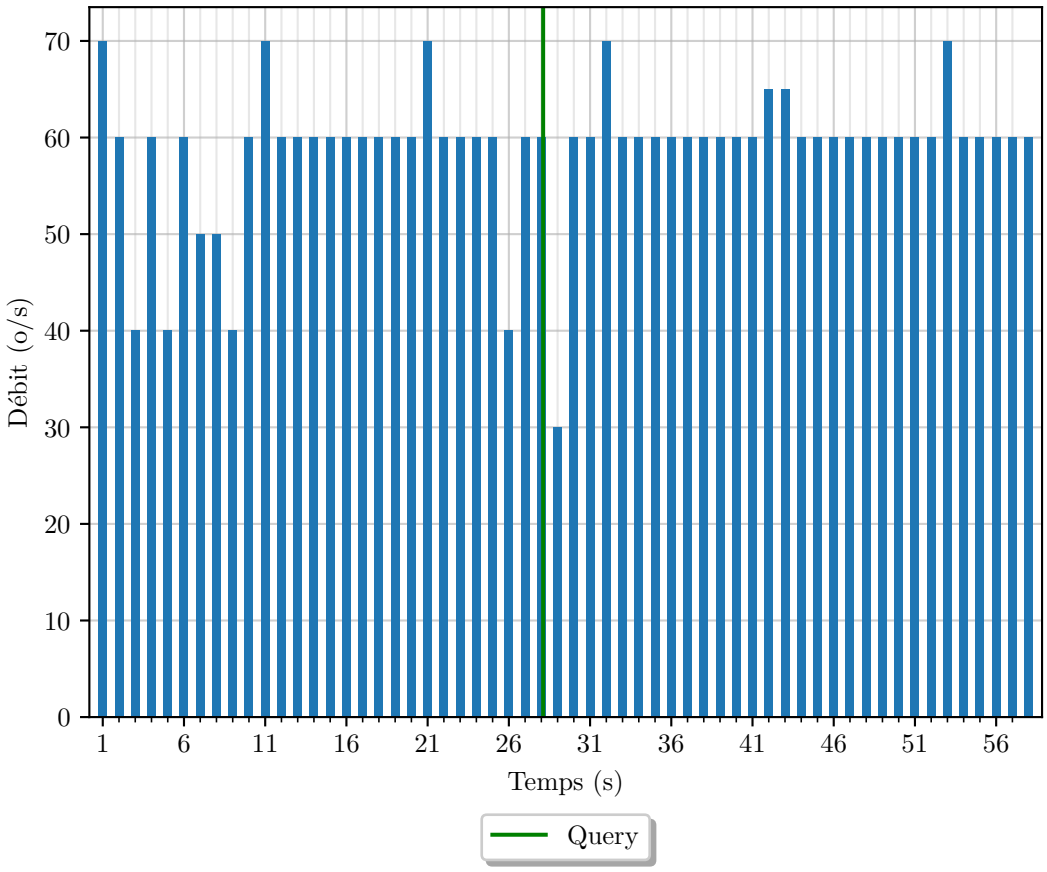


FIGURE 5.14 – Débit montant par seconde en fonction du temps utilisant le SF7 et des paquets de 10 octets.

Les chutes de débit aux secondes 3,5, 9 et 26 proviennent d'une mauvaise gestion des numéros de séquence corrigée pour les mesures suivantes. Dans ces cas, la racine LoRa reçoit des trames ayant un numéro de séquence plus petit que celui attendu. Ce dernier est incorrectement calculé. Comme le numéro de séquence est incorrect, les données ne sont pas délivrées à l'application.

Après la réception de la QUERY, le débit chute également car tant

que la racine RPL ne reçoit pas la réponse à cette QUERY, elle ne peut pas envoyer de trames de données.

Le débit moyen sur cette mesure est de 58.79 o/s avec un écart type de 7.38 o/s, un maximum de 70 o/s et un minimum de 30 o/s.

La mesure réalisée avec des paquets UDP ayant une payload de 200 octets est illustré par la Figure 5.15.

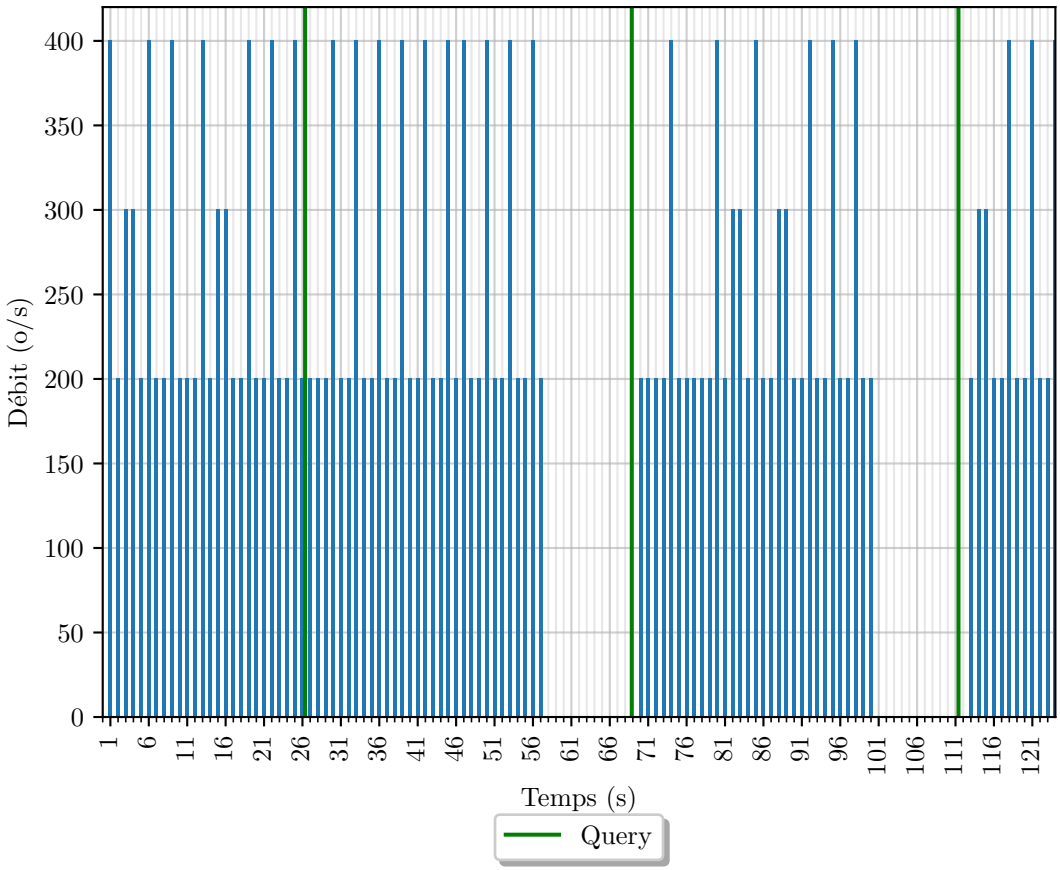


FIGURE 5.15 – Débit montant par seconde en fonction du temps utilisant le SF7 et des paquets de 200 octets.

Pour cette mesure, le débit oscille entre 400 o/s et 200 o/s. Comme pour la mesure précédente, l'envoi d'une QUERY diminue le débit.

Il arrive deux fois qu'aucun paquet de données ne soit reçu. Ces deux périodes se terminent après la réception d'une QUERY, ce qui indique que la racine RPL a dû les retransmettre. Cette hypothèse est vérifiée par l'analyse des logs. Pour la troisième QUERY, les logs mettent en évidence une erreur lors la réception caractérisée par la réponse `radio_err` du RN2483 re-

çue après la mise en écoute. En revanche, pour les deux premières QUERY, aucune erreur n'est reportée dans les logs. La cause la plus probable est que quand l'un des noeuds émet un paquet, l'autre n'est pas en écoute.

Le débit moyen obtenu est de 211.29 o/s avec un écart type de 129.64 o/s, un maximum de 400 o/s et un minimum de 0 o/s.

Ensuite, les mêmes mesures sont réalisées avec le SF 12. D'abord avec des payload UDP de 10 octets dont le débit au cours du temps est illustré par la Figure 5.16.

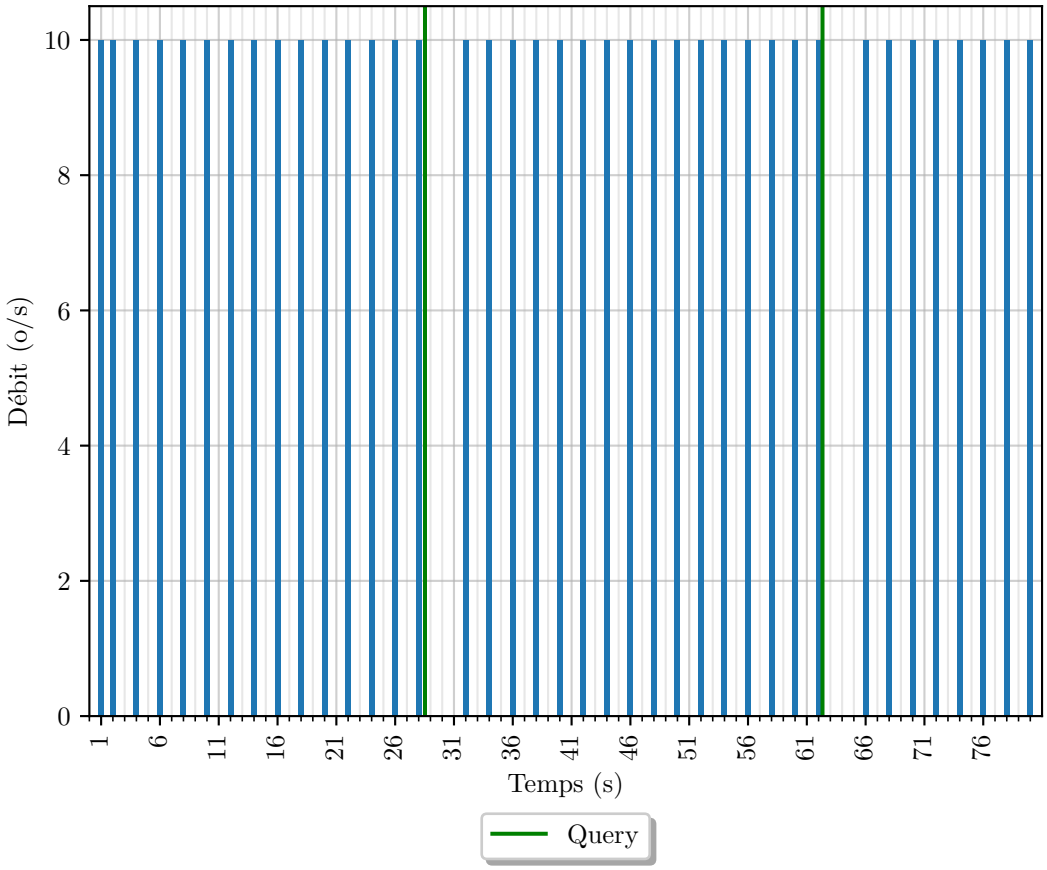


FIGURE 5.16 – Débit montant par seconde en fonction du temps utilisant le SF12 et des paquets de 10 octets.

Ce graphique montre que sauf pour les deux premières secondes, une trame de données est reçue toutes les deux secondes sauf lors de la réception d'une QUERY.

Le débit moyen obtenu est de 4.81 o/s avec un écart type de 4.99 o/s, un maximum de 10 et un minimum de 0.

La mesure réalisée avec des paquets UDP ayant une payload de 200 octets est illustré par la Figure 5.17.

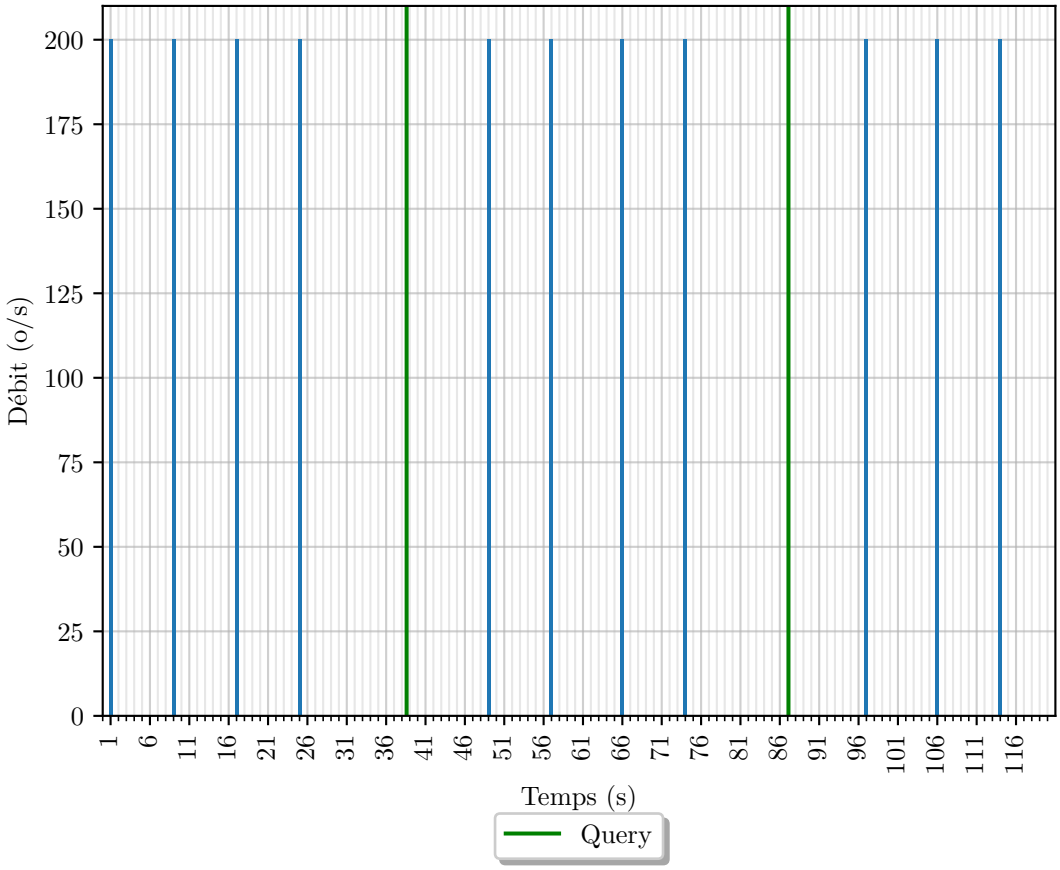


FIGURE 5.17 – Débit montant par seconde en fonction du temps utilisant le SF12 et des paquets de 200 octets.

Les paquets arrivent à un plus grand intervalle que pour les mesures précédentes et les QUERYs allongent cet intervalle. L'analyse des logs indique que les deux QUERYs sont chacune retransmises une fois par la racine RPL.

Pour cette mesure, le débit moyen est de 18.2 o/s avec un écart type de 57.49, un minimum de 0 et un maximum de 200.

Trafic descendant

Comme pour le trafic montant, les débits sont évalués avec des paquets UDP ayant une payload de 10 et de 200 octets et les spreading factors 7 et 12.

La Figure 5.18 illustre le débit par seconde mesuré avec le SF7 et une taille de payload des paquets UDP de 10 octets.

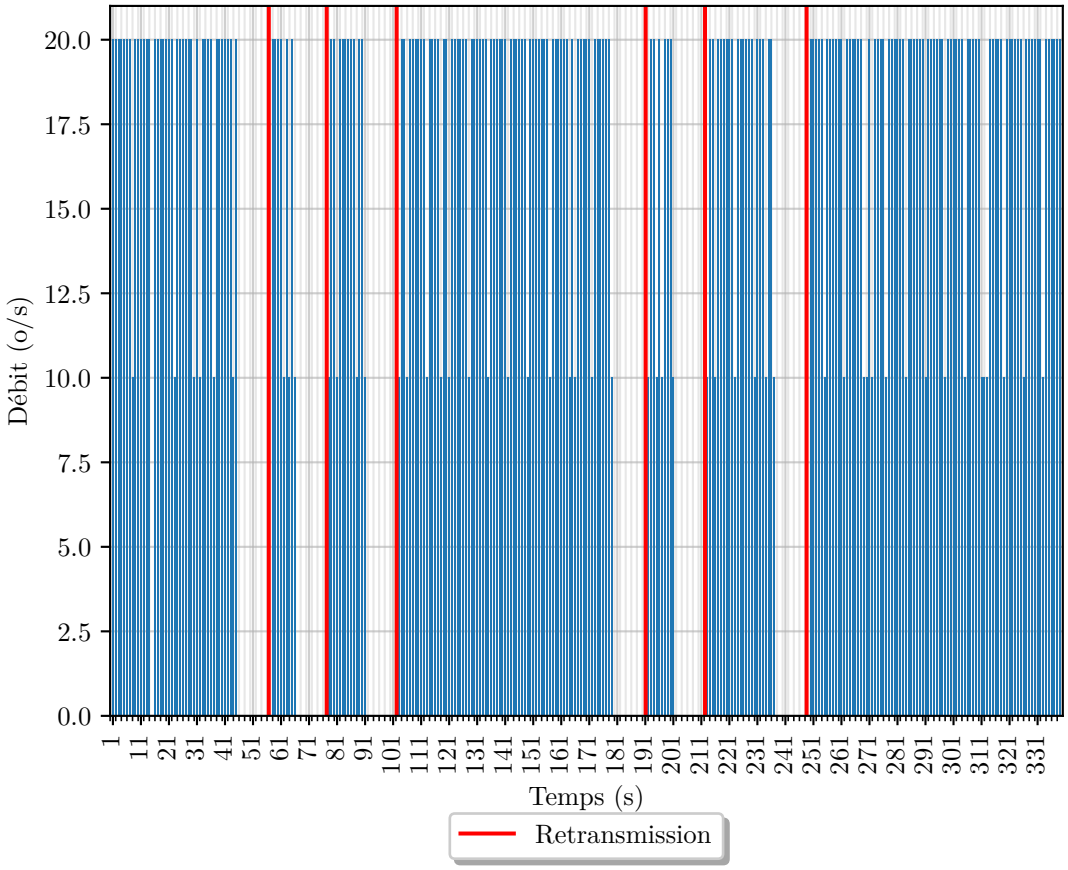


FIGURE 5.18 – Débit descendant par seconde en fonction du temps utilisant le SF7 et des paquets de 10 octets.

Durant ce test, 6 retransmissions ont lieu. Elles sont caractérisées sur le graphique par une attente de réponse durant laquelle aucune donnée n'est reçue suivie d'une retransmission. La première et la dernière sont dues à une commande MAC inconnue. La troisième et la cinquième sont dues à une adresse de destination qui n'est pas celle de la racine RPL et qui n'appartient à aucun noeud. Enfin, la deuxième et la quatrième n'ont pas de cause établie, mis à part que lorsqu'un noeud envoi une trame l'autre n'est pas en écoute.

Les retransmissions causées par une adresse de destination incorrecte ont toutes les deux la même adresse. Ceci fait penser que ces trames ainsi que les trames ayant une commande MAC inconnue sont soit corrompues pendant l'envoi, soit elles sont correctes et les erreurs proviennent d'un problème dans

l'implémentation.

Pour cette mesure, le débit moyen est de 14.32 o/s avec un écart type de 8.1 o/s.

Ensuite, le même test est réalisé avec des paquets UDP ayant une payload de 200 octets. Les débits de ce test sont illustrés par la Figure 5.19.

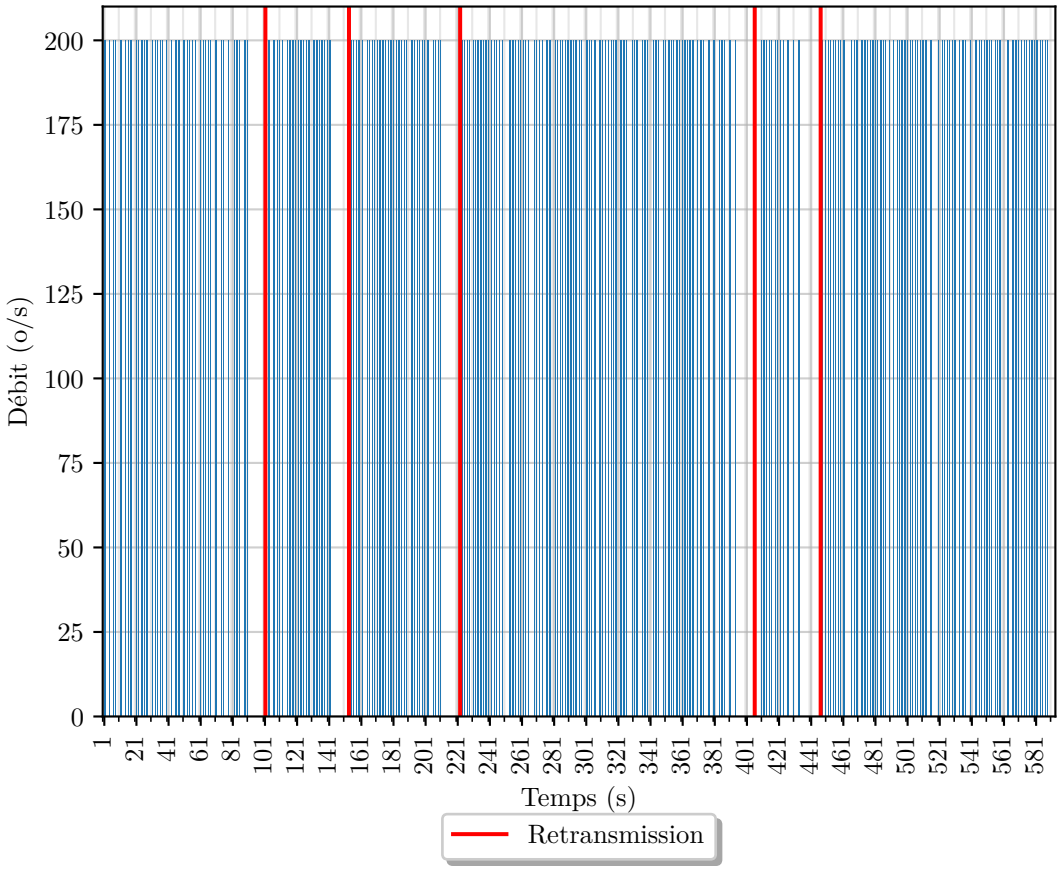


FIGURE 5.19 – Débit descendant par seconde en fonction du temps utilisant le SF7 et des paquets de 200 octets.

Durant cette transmission, 5 retransmissions ont lieu. La première et la dernière sont dues à une commande MAC incorrecte, la deuxième une adresse de destination qui ne correspond pas à celle de la racine RPL et qui est égale à celle de la mesure précédente et enfin, la troisième et la quatrième retransmission n'ont pas de cause établie.

Pour cette mesure, le débit moyen est de 117.03 o/s avec un écart type de 98.53 o/s.

Ensuite, les mêmes mesures sont réalisées avec le spreading factor égal

à 12. D'abord avec des payload UDP de 10 octets dont le débit au cours du temps est illustré par la Figure 5.20.

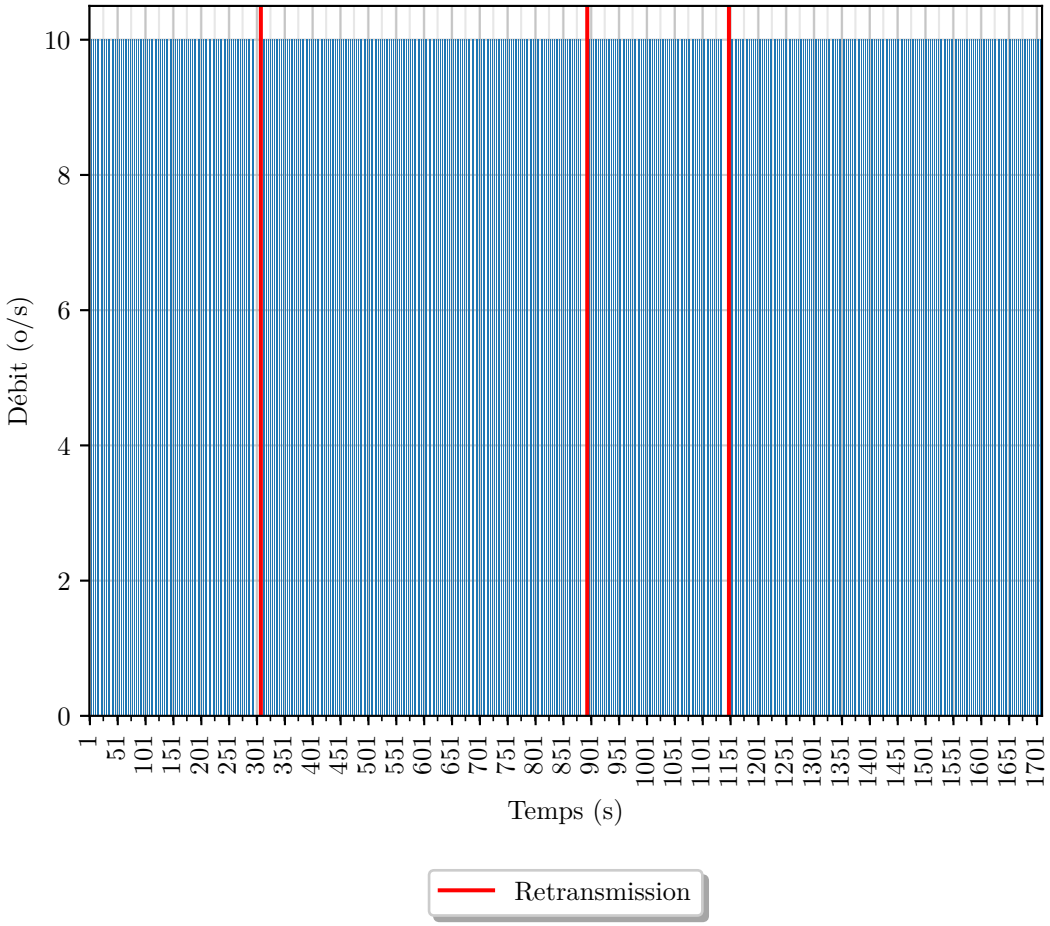


FIGURE 5.20 – Débit descendant par seconde en fonction du temps utilisant le SF12 et des paquets de 10 octets.

Le graphique illustre que trois retransmissions ont lieu durant ce test. Seul la dernière provient d'une commande MAC non reconnue. Le débit moyen est de 2.91 o/s avec un écart type de 4.55 o/s.

Enfin, la Figure 5.21 illustre le débit au cours du temps utilisant le SF12 avec des paquets de 200 octets.

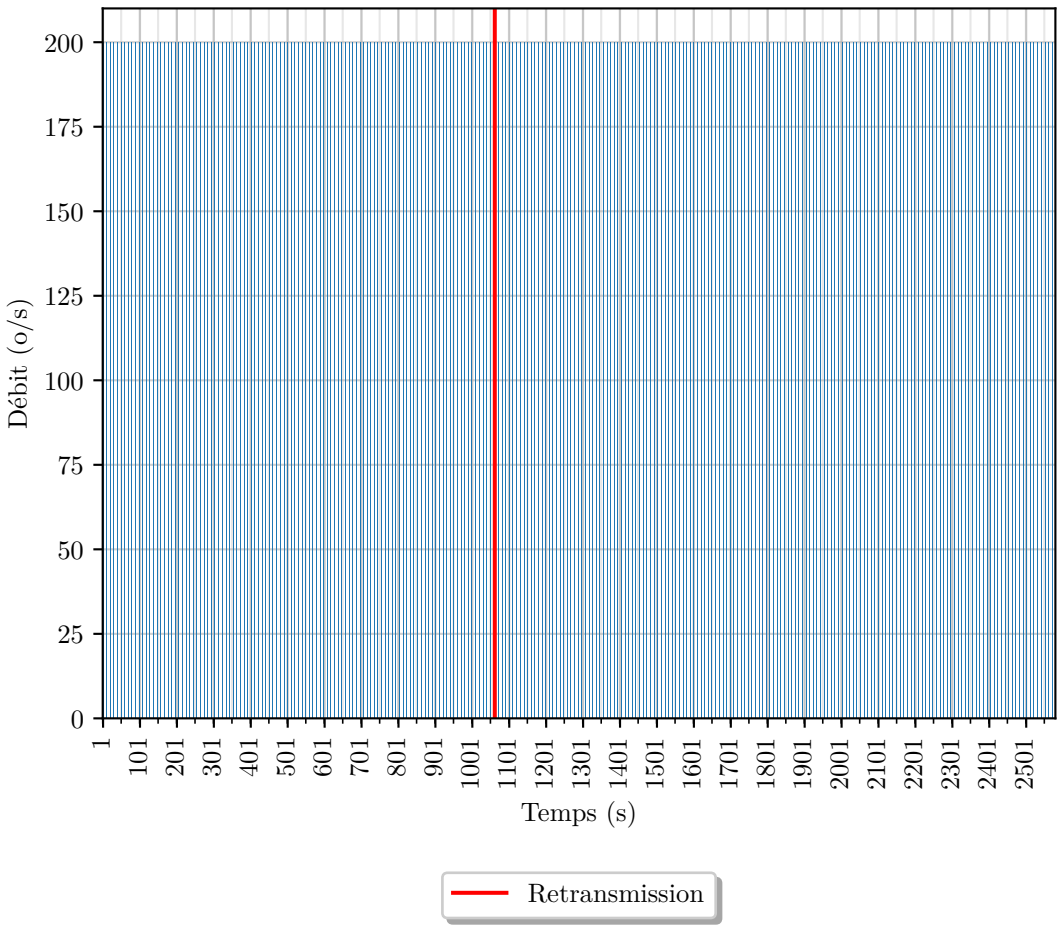


FIGURE 5.21 – Débit descendant par seconde en fonction du temps utilisant le SF12 et des paquets de 200 octets.

Le graphique illustre qu'une seule retransmission a lieu durant ce test. Elle provient d'une commande MAC non reconnue.

Le débit moyen est de 20.38 o/s avec un écart type de 60.41 o/s.

Discussion

La Table 5.2 et la table 5.3 résument les débits mesurés.

	10 octets	200 octets
SF7	58.79	211.29
SF12	4.81	18.2

TABLE 5.2 – Débit moyen en o/s du trafic montant en fonction du spreading factor et de la taille de la payload UDP.

	10 octets	200 octets
SF7	14.32	117.03
SF12	2.91	20.38

TABLE 5.3 – Débit moyen en o/s du trafic descendant en fonction du spreading factor et de la taille de la payload UDP.

Elles permettent de mettre en évidence que le débit du trafic montant sans acquittement est plus élevé que le débit moyen du trafic descendant. Ce résultat est attendu car moins de messages sont échangés pour du trafic montant sans acquittement.

La différence entre SF7 et SF12 est aussi clairement mise en évidence. Enfin le débit moyen est plus élevé lorsque les trames sont plus grandes. Ce qui renforce l'idée que la mise en place d'une agrégation de trames peut améliorer le réseau.

Lors des mesures du débit descendant, un problème est apparu sur la racine LoRa. Certains paquets ne sont pas délivrés à l'application malgré qu'ils soient correctement livrés à la pile réseau de Contiki. Les temps de réception des paquets sont donc pris au moment où ils sont délivrés à la pile réseau de Contiki. La cause exacte de ce problème n'est pas identifiée. L'hypothèse est que l'interruption générée par la réception d'une trame UART qui a lieu lors de la réception d'une trame LoRaMAC ne laisse pas le temps à la pile réseau et à l'application de traiter correctement la trame LoRaMAC précédente.

Chapitre 6

Conclusion

Ce mémoire fournit une base de connaissance sur le fonctionnement de la modulation LoRa, de IEEE 802.15.4 avec TSCH et RPL, ainsi qu'un aperçu des RTOS existants. Ceci a permis de concevoir un réseau hybride LoRa - IEEE 802.15.4 qui possède les caractéristiques idéales pour un réseau de capteurs devant se trouver aux pieds de cultures et accessible à distance. Le protocole MAC, LoRaMAC, élaboré permet d'établir des communications LoRa point à point sans la nécéssité d'utiliser un réseau LoRaWAN plus complexe à mettre en place et plus couteux.

La validation et l'évaluation de ce protocole implémenté avec Contiki montrent que le réseau élaboré est fonctionnel et pourrait être amélioré afin d'augmenter les débits possibles et de diminuer la perte de paquets. L'utilisation du SX1276 à la place du RN2483 permettrait également une utilisation plus flexible et plus maîtrisable de la radio LoRa. De plus, le fait de ne pas devoir utiliser de commandes ASCII pour contrôler la radio pourrait également améliorer la rapidité du réseau.

L'évaluation et la validation du prototype pourraient également être plus exhaustives en testant d'autres configurations, en utilisant plus de noeuds ou en testant le prototype en condition réelle.

Finalement, ce prototype est une bonne base qui peut être utilisée pour l'élaboration d'un réseau hybride LWPAN-LPWAN. D'autres technologies radio ainsi que d'autres plateformes de développement pourrait également être envisagée pour ce type de réseau.

Bibliographie

[1] Bruno Quoitin. Computer networks ii, ch.4 wireless networks, 2021.

[2] Semtech Corporation. *SX1276/77/78/79 - 137 MHz to 1020 MHz Low Power Long Range Transceiver*, 5 2020.

[3] Tapparel Joachim. Complete reverse engineering of lora phy.

[4] LoRa Alliance Technical Committee Regional Parameters Workgroup. *RP002-1.0.2 LoRaWAN® Regional 40 Parameters*, 10 2020.

[5] Sakshama Ghoslya. Lora : Symbol generation. [Accès en ligne le 11 juillet 2021].

[6] Microchip. *Low-Power Long Range LoRa® Technology Transceiver Module*, 6 2020. Rev. E.

[7] Domenico De Guglielmo, Simone Brienza, and Giuseppe Anastasi. Ieee 802.15.4e : A survey. *Computer Communications*, 88 :1–24, 2016.

[8] IEEE P802.15 Working Group for Wireless Personal Area Networks (WPANs). Ieee 802.15.4 mac overview, 2004.

[9] Harrison Kurunathan, Ricardo Severino, Anis Koubaa, and Eduardo Tovar. Ieee 802.15.4e in a nutshell : Survey and performance evaluation. *IEEE Communications Surveys Tutorials*, 20(3) :1989–2010, 2018.

[10] Roger Alexander, Anders Brandt, JP Vasseur, Jonathan Huii, Kris Pister, Pascal Thubert, P Levis, Rene Struik, Richard Kelsey, and Tim Winter. RPL : IPv6 Routing Protocol for Low-Power and Lossy Networks. RFC 6550, March 2012.

[11] Joydeep Tripathi, Jose Oliveira, and J.-P Vasseur. A performance evaluation study of rpl : Routing protocol for low power and lossy networks. pages 1–6, 03 2010.

[12] John Moy. OSPF Version 2. RFC 2328, April 1998.

[13] Samir R. Das, Charles E. Perkins, and Elizabeth M. Belding-Royer. Ad hoc On-Demand Distance Vector (AODV) Routing. RFC 3561, July 2003.

[14] Thomas H. Clausen and Philippe Jacquet. Optimized Link State Routing Protocol (OLSR). RFC 3626, October 2003.

[15] Pascal Thubert. Objective Function Zero for the Routing Protocol for Low-Power and Lossy Networks (RPL). RFC 6552, March 2012.

[16] Omprakash Gnawali and P Levis. The Minimum Rank with Hysteresis Objective Function. RFC 6719, September 2012.

[17] Douglas S. J. De Couto, Daniel Aguayo, John Bicket, and Robert Morris. A high-throughput path metric for multi-hop wireless routing. 2003.

[18] P Levis, Thomas H. Clausen, Omprakash Gnawali, Jonathan Huii, and JeongGil Ko. The Trickle Algorithm. RFC 6206, March 2011.

[19] David Culler, Jonathan Huii, JP Vasseur, and Vishwas Manral. An IPv6 Routing Header for Source Routes with the Routing Protocol for Low-Power and Lossy Networks (RPL). RFC 6554, March 2012.

[20] Dr. Steve E. Deering and Bob Hinden. Internet Protocol, Version 6 (IPv6) Specification. RFC 8200, July 2017.

[21] Adam Dunkels, Björn Grönvall, and Thiemo Voigt. Contiki - a lightweight and flexible operating system for tiny networked sensors. In *Proceedings of the First IEEE Workshop on Embedded Networked Sensors (Emnets-I)*, Tampa, Florida, USA, November 2004.

[22] The Contiki Operating System. `https://github.com/contiki-os/contiki`. [Accès en ligne le 15 juin 2021].

[23] Contiki-NG : The OS for Next Generation IoT Devices. `https://github.com/contiki-ng/contiki-ng`. [Accès en ligne le 15 juin 2021].

[24] Adam Dunkels, Oliver Schmidt, Thiemo Voigt, and Muneeb Ali. Protothreads : Simplifying event-driven programming of memory-constrained embedded systems. In *Proceedings of the Fourth ACM Conference on Embedded Networked Sensor Systems (SenSys 2006)*, Boulder, Colorado, USA, November 2006.

[25] Pascal Thubert. An Architecture for IPv6 over the Time-Slotted Channel Hopping Mode of IEEE 802.15.4 (6TiSCH). RFC 9030, May 2021.

[26] Simon Duquennoy, Beshr Al Nahas, Olaf Landsiedel, and Thomas Watteyne. Orchestra : Robust mesh networks through autonomously scheduled tsch. 11 2015.

[27] FreeRTOS Real-time operating system for microcontrollers. `https://freertos.org/index.html`. [Accès en ligne le 15 juin 2021].

[28] Emmanuel Baccelli, Cenk Gündoğan, Oliver Hahm, Peter Kietzmann, Martine S. Lenders, Hauke Petersen, Kaspar Schleiser, Thomas C.

Schmidt, and Matthias Wählisch. Riot : An open source operating system for low-end embedded devices in the iot. *IEEE Internet of Things Journal*, 5(6) :4428–4440, 2018.

[29] OpenWSN. `https://openwsn.atlassian.net/wiki/spaces/OW/overview?mode=global`. [Accès en ligne le 22 juin 2021].

[30] Yi Chu, Paul D. Mitchell, and David Grace. Aloha and q-learning based medium access control for wireless sensor networks. In *2012 International Symposium on Wireless Communication Systems (ISWCS)*, pages 511–515, 2012.

[31] Juan José López Escobar, Felipe Gil-Castiñeira, and Rebeca Redondo. Jmac protocol : A cross-layer multi-hop protocol for lora. *Sensors*, 20 :6893, 12 2020.

[32] Maite Bezunartea, Roald Van Glabbeek, An Braeken, Jacques Tiberghien, and Kris Steenhaut. Towards energy efficient lora multihop networks. In *2019 IEEE International Symposium on Local and Metropolitan Area Networks (LANMAN)*, pages 1–3, 2019.

[33] Benjamin Sartori, Steffen Thielemans, Maite Bezunartea, An Braeken, and Kris Steenhaut. Enabling rpl multihop communications based on lora. In *2017 IEEE 13th International Conference on Wireless and Mobile Computing, Networking and Communications (WiMob)*, pages 1–8, 2017.

[34] Brian Haberman and Bob Hinden. Unique Local IPv6 Unicast Addresses. RFC 4193, October 2005.

[35] Nestor Ayuso. Microchip rn2483 teardown : The french (wireless) connection, Jul 2015. [Accès en ligne le 11 juillet 2021].

[36] Raspberry Pi 3 Model B+. `https://www.raspberrypi.org/products/raspberry-pi-3-model-b-plus/`. [Accès en ligne le 15 juin 2021].

[37] Zolertia RE-Mote. `https://zolertia.io/product/re-mote/`. [Accès en ligne le 14 juin 2021].

[38] Zolertia. *Zolertia RE-Mote Revision B Internet of Things hardware development platform, for 2.4-GHz and 863-950MHz IEEE 802.15.4, 6LoWPAN and ZigBee® Applications*, 9 2016. v.1.0.0.

[39] Microchip. *RN2483 LoRa ® Technology Module Command Reference User's Guide*, 8 2018.

[40] Jetperch. *Joulescope™ JS110 User's Guide. Precision DC Energy Analyzer*, 6 2020. Rev. 1.0.

[41] Maximilien Charlier. Get rid of "frsram" error on zoul platform [contiki], Mars 2020. [Accès en ligne le 11 décembre 2021].