Projet de fin d'étude :
Gestion des noeuds isolés en LoRa

Joffrey HÉRARD

Responsable : Olivier FLAUZAC

2017-2018

Table des matières

1 Introduction

La problémablique de notre étude portant sur la gestion des noeuds isolés en *LoRa* nous définirons, dans un premier temps, les technologies du *LoRaWAN* et du *LoRa*. Subsidiairement nous expliquerons les problèmes que peut présenter une architecture *LoRaWAN* et proposerons diverses solutions pour les résoudre. Avant de procéder à l'implémentation nous exécuterons des simulations sur le modèle de communication *LoRa* avec les algorithmes présentés auparavant. Par la suite, nous expliquerons comment programmer cette solution dans un capteur via l'utilisation de capteurs *LoPy*. Pour finir, nous proposerons une brève conclusion.

2 LoRaWAN

Comment connecter les milliards de capteurs qui seront potentiellement déployés dans le monde et qui participeront à la construction des villes intelligentes, de la mobilité et de l'industrie du futur ?
Il existe bon nombre de réponses à cette question mais nous pourrons en trouver assurément du côté des LPWAN. Par conséquent nous commenceront par développer ce modèle avant de décrire la spécification de version 1.0 de *LoRaWAN*, basé sur *LoRa*.

2.1 Convention de nommage

Afin de conserver une certaine cohérence vis à vis de la spécification des termes anglo-saxons, nous avons préféré les conserver dans ce document. Toute fois, vous en trouverez une brève définition ci-dessous :

- End-devices : Définissent les périphériques cibles comme les capteurs par exemple.
- Isolated-devices/Isolated Nodes : Définissent les périphériques cibles comme les capteurs par exemple mais cette fois ci. Isolé par rapport à une gateway *LoRaWAN*.
- Uplink : Correspond aux chemins réseaux des end-devices vers le serveur réseaux.
- Downlink : Correspond aux chemins réseaux du serveur vers le end-devices.
- Gateway : Correspond aux concentrateurs réseaux.
- LoRaGateway : Correspond aux concentrateurs réseaux qui sont des end-devices .

2.2 LPWAN

Pour certaines applications (villes intelligentes, maintenance prédictive, agriculture connectée, etc.), il s'agit de déployer des centaines de milliers de capteurs (monitoring énergétique, qualité de l'air, gestion des déchets) fonctionnant sur pile et communiquant quotidiennement de très faibles quantités de données, à faible débit vers des serveurs sur Internet (cloud).Les réseaux LPWAN, comme le laisse deviner l'acronyme, sont des réseaux sans fil basse consommation, bas débit et longue portée, optimisés pour les équipements aux ressources limitées pour lesquels une autonomie de plusieurs années est requise. Ces réseaux conviennent particulièrement aux applications qui n'exigent pas un débit élevé. Contrairement aux opérateurs mobiles les LPWAN utilisent des bandes de fréquences à usage libre, disponibles mondialement et sans licence : ISM (Industriel, Scientifique et Médical). Compte tenu des faibles débits et de la faible occupation spectrale des signaux, il faut en moyenne, pour un réseau LPWAN, 10 fois moins d'antennes pour couvrir la même surface qu'un réseau cellulaire traditionnel.

2.3 LoRaWAN

LoRaWAN (Long Range Radio Wide Area Network) est un réseau LPWAN basé sur la technologie radio *LoRa*. Cette technologie, développée par Cycleo en 2009 puis rachetée, 3 ans après, par l'américain Semtech, utilise une technique d'étalement de spectre pour la transmission des signaux radio (chirp spread spectrum). La technologie *LoRa*, à travers le réseau *LoRaWAN*, est poussée par un consortium d'industriels et d'opérateurs nommé *LoRa* Alliance qui regroupe notamment IBM, Cisco, Bouygues Télécom, etc...

2.3.1 LoRa Alliance

La *LoRa* Alliance est une association dont le but, non lucratif, est de standardiser le réseau *LoRaWAN* pour apporter à l'internet des objets (IoT) un moyen fiable pour se connecter à Internet. Cette association a été créée par Semtech et de nombreux acteurs industriels garantissent aujourd'hui l'interopérabilité et la standardisation de la technologie *LoRa*.

2.3.2 Les différentes couches d'un réseaux LoRaWAN

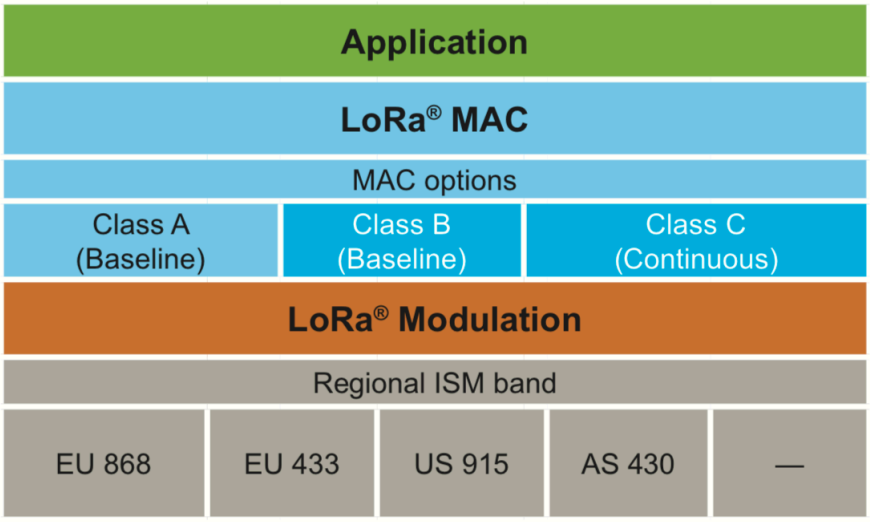

FIGURE 1 – Les différentes couches d'un réseau LoRaWAN

On peut alors constater que la couche physique fournie par la technologie *LoRa* n'est pas suffisante pour assurer la communication réseau. En définissant le protocole réseau (*LoRa*

MAC), pour une communication d'équipements *LoRa* à travers un réseau, le protocole *LoRaWAN* assure une communication bi-directionnelle et définit trois classes d'équipements différents. Nous définirons celles-ci dans la partie de notre étude consacrée à l'explication des différences sur les messages, les fenêtres de réception...

2.3.3 Différence entre le LoRa et LoRaWAN

D'un côté, *LoRaWAN* est un modèle d'architecture réseaux qui exploite la technologie radio *LoRa* pour faire communiquer les gateways avec les périphériques et les capteurs. En outre, nous détaillerons les particularités de cette architecture réseaux dans le point suivant. A noter, au sein de notre document, la présence de raccourcis d'écriture désignant *LoRa* comme la communication spécifiée en réseau *LoRaWAN* mais il s'agira bel et bien d'échanges effectués en *LoRa* avec une couche *LoRa* MAC.

2.3.4 Architecture d'un réseaux LoRaWAN

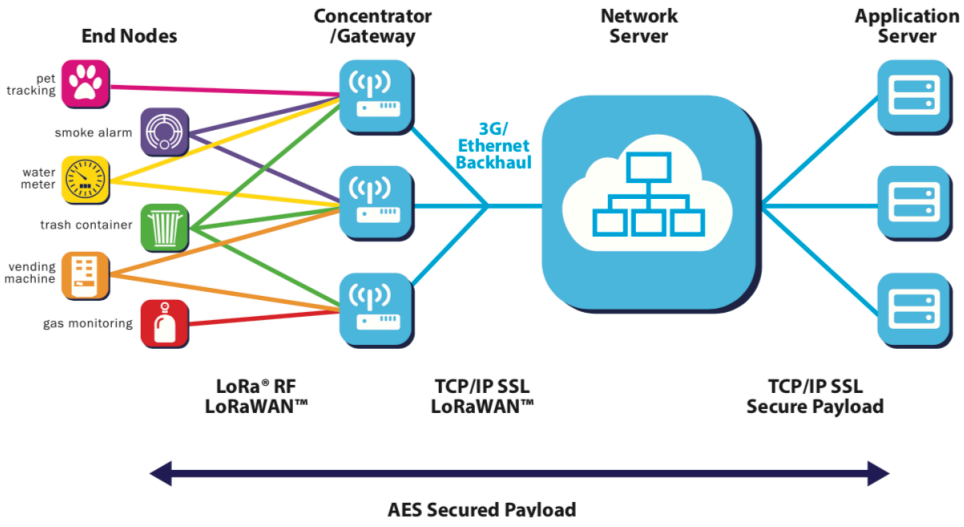

FIGURE 2 – Architecture d'un réseaux LoRaWAN

Cette figure représente une architecture classique d'un réseau *LoRaWAN*. Typiquement un réseau *LoRaWAN* est en topologie "star of star" étoile en étoile, le centre de cette topologie est le serveur de réseau qui assure la gestion du débit adaptatif, de la sécurité des données ou encore de la redondance des données. Celui ci est entouré d'un côté par les gateways connectés en ethernet/4G au serveur de réseau, auquel les end-devices seront connectés, eux en *LoRa*. D'autre part du serveur réseaux, nous avons les serveur d'applications lié par ethernet, ces mêmes serveur d'application lié elle mêmes à une interface web (échange HTTP, MQTT). Une des particularités d'un réseau *LoRaWAN*, est qu'un équipement ne communique pas exclusivement à travers un concentrateur. Tous les concentrateurs couvrant l'équipement peuvent recevoir les données transmises par ce dernier. Cela

facilite grandement la communication avec les équipements en mobilité en dispensant le réseau de mécanismes de hand-over (passage d'un concentrateur à un autre) qui auraient pour effet de complexifier sa gestion et très probablement de réduire ses performances. Par contre, lorsque le serveur envoie un message à destination d'un équipement, c'est par le biais d'un seul concentrateur. C'est le cas en particulier des messages requérant un acquittement par le serveur. assure l'échange entre le serveur de réseaux et les serveurs d'applications : AppsKey. La sécurité après ces serveurs n'est plus géré par la spécification *LoRaWAN*.

2.3.5 Sécurité d'un réseaux LoRaWAN

Une question importante est aussi celle de la sécurité a travers le réseaux, on parle de capteurs voir d'un ensemble d'informations qui vont communiqué, donc c'est une question importante, et surtout ça concerne l'Internet des Objets dans son ensemble . La sécurité est assuré de bout en bout par un chiffrement AES-128 bits, elles sont aux nombre de deux, la première qui couvre les capteurs jusqu'au serveur réseaux nommé NwkSKey.

— Network Session Key (NwkSKey), assure l'authenticité des équipements sur le réseau .

— Application Session Key (AppSKey), la sécurité et la confidentialité des données transmises à travers le réseau.

En d'autre termes, la clé réseau permet à l'opérateur de sécuriser son réseau alors que la clé applicative permet au fournisseur de l'application de sécuriser les données qui transitent à travers le réseau.

Les données utiles que souhaite transmettre sont tout d'abord chiffrées via la AppSKey. Un en-tête, contenant entre autres l'adresse de l'équipement, est ensuite ajouté aux données chiffrées. À partir de cela, le MIC – Message Integrity Code – est calculé via NwkSKey. Le MIC permet au réseau de vérifier l'intégrité des données et de l'équipement sur le réseau. Enfin, le MIC est ajouté au message contenant l'en-tête et les données chiffrées avant transmission.

À réception du message par le serveur de gestion du réseau, ce dernier pourra vérifier l'intégrité des données grâce au MIC tout en préservant la confidentialité des données (chiffrées par AppSKey). L'ensemble des trames d'échange vont être décrit dans la suite du document.

2.3.6 Classe A : All end-devices

Méssage couche physique

Messages uplink Envoyé par les appareils, reçu par le serveur réseaux et ceux-ci traversant une ou plusieurs stations de base .

— Preambule
— PHDR : *LoRa* en-tête physique
— PHDR_CRC : en tête affublé d'une en tête CRC
— PHYPayload : Le payload
— le champ CRC

Messages downlink
— Preambule
— PHDR : *LoRa* en-tête physique
— PHDR_CRC : en tête affublé d'une en tête CRC
— PHYPayload : Le payload

Fenêtre de réception Après chaque transmission en uplink, le device ouvre deux fenêtres de réception courtes. Les heures de début de la fenêtre de réception sont des périodes configurées à la fin de la transmission du message au niveau du dernier octet du message .

Première fenêtre de réception, débit de données, démarrage La première fenêtre de réception RX1 utilise le même canal de fréquence que celle du message uplink et un débit de données qui est en fonction du débit de données utilisé pour l'uplink. RX1 ouvre RECEIVE_DELAY1 secondes (+/- 20 microsecondes) après la fin de la modulation du message uplink. La relation entre le débit de données de downlink et de l'uplink de RX1 sont spécifique à la région. Par défaut, le premier datarate de la fenêtre de réception est identique à la donnée du dernier message uplink.

Seconde fenêtre de réception, débit de données, démarrage La seconde fenêtre de réception RX2 utilise une fréquence et un débit de données configurables fixes et ouvre RECEIVE_DELAY2 secondes (+/- 20 microsecondes) après la fin de la modulation d'un message uplink. La fréquence et le débit de données utilisés peuvent être modifiés au moyen des commandes MAC. La fréquence par défaut et le débit de données à utiliser sont spécifiques à la région .

Durée de la fenêtre La durée d'une fenêtre de réception doit être au moins le temps requis par l'émetteur-récepteur radio du device pour détecter efficacement un préambule de message downlink.

Activité du receveur durant la fenêtre de réception Si un préambule est détecté pendant l'une des fenêtres de réception, le récepteur radio reste actif jusqu'à ce que une trame downlink soit démodulée. Si une trame a été détectée et ensuite démodulée pendant la première fenêtre de réception et que la trame était destinée à cet appareil après vérification de l'adresse et du code MIC (message integrity code), le terminal n'ouvre pas la seconde fenêtre de réception.

Envoi d'un message à travers le réseaux vers un appareil Si le serveur réseau à l'intention de transmettre un message uplink , il initiera toujours la transmission précisément au début de l'une de ces deux fenêtres de réception.

A noter Un terminal ne doit pas transmettre un autre message uplink avant qu'il ait reçu un message downlink dans la première ou la deuxième fenêtre de réception de la transmission précédente, ou que la deuxième fenêtre de réception de la transmission précédente ait expiré.

Message couche MAC Tout les messages de type uplink et downlink contienne un PHY payload(Payload) cela commence avec un octet d'en tête MAC (MHDR) suivi d'un Payload MAC (MACPayload), et ce termine par 4 octet d'intégrité du code (MIC)

Couche MAC (PHYPayload) La taille maximum M d'un MACPayload est spécifique à la région

Size (bytes)	1	1..*M*	4
PHYPayload	MHDR	MACPayload	MIC

FIGURE 3 – Format couche MAC

En tête MAC. (MHDR field) L'en tête MAC spécifie le type de message (MType) et est encodé par la couche *LoRaWAN*.

Bit#	7..5	4..2	1..0
MHDR bits	MType	RFU	Major

FIGURE 4 – En tête MAC

Type de messages (MType bit field) Il y a six messages MAC différents :
— join request décrit pour l'utilisation de la procédure "over-the-air activation"
— join accept décrit pour l'utilisation de la procédure "over-the-air activation"
— unconfirmed data up/down
— confirmed data up/down

Données des messages Les données des messages sont utilisé pour transféré à la fois les commandes MAC et les données des applications, celles ci peuvent être combiné en un seul message. Un message confirmed-data message doit être accusé de réception par le destinataire cependant un unconfirmed-data message n'en as pas le besoin. Des messages propriétaires peuvent être utilisé pour implémenté des formats de messages non standards qui eux ne sont par inter-opérable avec les messages standards. Ces messages doivent être uniquement utilisé parmi les devices qui ont la même connaissances des ajouts propriétaires.

Le payload d'un message sur la couche MAC (MACPayload) Le MAC Payload = données du message, sont aussi appellée *data frame*, qui contient une partie d'en tête (FHDR). suivi par un champ optionnel de désignation de port (FPort) mais aussi une partie optionnelle pour un payload (FRMPayload).

En-tête (FHDR) Le FHDR contient l'adresse de périphérique (DevAddr), un octet de contrôle de trame (FCtrl), un compteur de trames de 2 octets (FCnt) et jusqu'à 15 octets d'options de trame (FOpts) utilisés pour transporter des commandes MAC .

Champ du port (Fport) Si le champ payload de la trame n'est pas vide, le champ de port doit être présent. S'il est présent, une valeur FPort de 0 indique que FRMPayload contient uniquement des commandes MAC. Les valeurs FPort 1..223 (0x01..0xDF) sont spécifiques à l'application. Les valeurs FPort 224..255 (0xE0..0xFF) sont réservées aux futures extensions d'application normalisées.

Chiffrement du Payload sur une trame MAC (FRMPayload) Si une trame de données transporte un payload, FRMPayload doit être chiffrée avant l'envoi du message. Le code d'intégrité de chiffrement du payload de trame MAC (FRMPayload et MIC) est calculé. Le schéma de chiffrement utilisé est basé sur l'algorithme générique utilisant AES avec une longueur de clé de 128 bits. Par défaut, le chiffrement / déchiffrement est effectué par la couche *LoRaWAN* pour tous les FPort. Le chiffrement / déchiffrement peut être effectué au-dessus de la couche *LoRaWAN* pour des FPorts spécifiques sauf 0, si cela est plus pratique pour l'application. Les informations à partir d'un noeud concernant le protocole de chiffrement / déchiffrement de FPort à l'extérieur de la couche *LoRaWAN* doivent être communiquées au serveur à l'aide d'un canal hors bande .

calcul du MIC msg = MHDR | FHDR | FPort | FRMPayload

le MIC est calculé comme ceci :

cmac = aes128_cmac(NwkSKey, B0 | msg) MIC = cmac[0..3]

Commandes MAC Pour l'administration de réseau, un ensemble de commandes MAC peut être échangé exclusivement entre le serveur de réseau et la couche MAC d'un périphérique . Les commandes de couche MAC ne sont jamais visibles par l'application, le serveur d'applications ou l'application s'exécutant sur le périphérique . Une trame de données unique peut contenir n'importe quelle séquence de commandes MAC, greffées dans le champ FOpts ou, lorsqu'elles sont envoyées en tant que trame de données séparée, dans le champ FRMPayload avec le champ FPort défini sur 0. Les commandes MAC superposées sont toujours envoyées sans chiffrement et ne doit pas dépasser 15 octets. Les commandes MAC envoyées en tant que FRMPayload sont toujours chiffrées et ne doivent pas dépasser la longueur FRMPayload maximale.

Activation d'un end-device Pour participer dans un réseaux *LoRaWAN* chaque end-device doit être personalisé et activé.

Adresse des device (DevAddr) 32 bits identifient le end-devices dans le réseaux courant. 31-25 = Network ID, 24-0 = NetworkAdress . les 7 bits de poids forts sont utilisé comme le NwkID . les 25 derniers sont utiisées pour l'adresse du device qui peut être arbitrairement assigné par l'administrateur réseau .

Identification de l'application (AppEUI) Il identifie le provider de l'application.

Clé de session réseaux (NwkSkey) Elle est spécifique au device utilisé par le serveur réseaux et le device pour calculer l'intégrité de tout les messages.

Clé de session d'application (AppSKey) Elle est spécifique au device utilisé par le serveur réseaux et le device pour chiffré/déchiffré le champ Payload de tout les messages. Mais aussi pour calculer l'intégrité de tout les messages.

Activation avec Over the Air (OTA) Les devices doivent forcément utilisé la procédure *join* pour participer aux échanges de données avec le serveur de réseaux. Un device doit forcement utilisé une nouvelle fois le fonction *join* à chaque fois que il a perdu les informations contextuelles lié à la session .

Identifiant des device (DevEUI) l'iddentifiant est au format IEEE EUI64 .

Clé de l'application(AppKey) C'est la clé AES 128 bits.

Procédure Join Consiste en 2 messages MAC entre un device et un serveur de réseau. Ceux-ci sont appelés join request et join accept.

Message Join-request Initialisé par le device en envoyant un message join-request 8 bits AppEUI, 8bits DevEUI, 2bits DevNonce DevNonce est une valeur aléatoire pour chaque device le serveur de réseaux garde une trace du DevNonce utilisé par les devices dans le passé. Le serveur de réseaux ignore tout les requêtes de join avec un DevNonce inconnu.

Méssage Join-accept Le serveur renvoie un message join-accept seulement si le device est autorisé à rejoindre le réseaux actuel. Aucune réponse n'est donné si le device n'est pas autorisé .Le message join-accept contient AppNonce composé de 3 octets, un identifiant réseau (NetID), une adresse de périphérique (DevAddr), un délai entre TX et RX (RxDelay) et une liste optionnelle de fréquences de canal (CFList) pour le réseau auquel le device se joint. L'option CFList est spécifique à une région .

Activation par Personallisation Le device contient directement le DevAddr, NwkSKey et AppSKey à la place de DevEUI,APPEUI et AppKey.

2.3.7 Classe B : Beacon

Introduction Cette section décrit la couche *LoRaWAN* Classe B optimisée pour les appareils alimentés par batterie qui peuvent être mobiles ou montés à un emplacement fixe. Les terminaux doivent mettre en œuvre une opération de classe B lorsqu'il est nécessaire d'ouvrir des fenêtres de réception à des intervalles de temps fixes afin d'activer les messages downlink. L'option *LoRaWAN* Classe B ajoute une fenêtre de réception synchronisée sur le device. L'une des limitations de *LoRaWAN* Class A est la méthode ALOHA d'envoi de données à partir du périphérique ; il ne permet pas de connaître le moment de la réaction lorsque l'application client ou le serveur souhaite s'adresser aux périphérique. L'objectif de la classe B est d'avoir un device disponible pour la réception à une heure prévisible, en plus des fenêtres de réception qui suivent la transmission aléatoire d'un méssage uplink du périphérique de classe A. La classe B est obtenue en envoyant à la passerelle un beacon sur une base régulière pour synchroniser tous les devices dans le réseau de sorte que l'appareil puisse ouvrir une courte fenêtre de réception supplémentaire (appelée "ping slot") à un moment prévisible pendant un intervalle de temps périodique.

Principe de la synchronisation du réseaux initialiser en downlink Pour qu'un réseau puisse prendre en charge les terminaux de classe B, toutes les passerelles doivent diffuser de manière synchrone un beacon fournissant une référence de synchronisation aux terminaux. Sur la base de cette référence de temporisation, les devices peuvent périodiquement ouvrir des fenêtres de réception, appelées ci-après des intervalles, qui peuvent être utilisées par l'infrastructure de réseau pour initier une communication downlink. Un message downlink utilisant l'un de ces emplacements est appelée un slot-ping. La passerelle choisie pour initier cette communication downlink est sélectionnée par le serveur de réseau sur la base des indicateurs de qualité de signal de la dernière liaison uplink de l'appareil . Pour cette raison, si un périphérique se déplace et détecte une modification de l'identité annoncée dans le beacon reçue, il doit envoyer envoyé en uplink afin que le serveur puisse mettre à jour la base de données du chemin de routage descendant. Tous les périphériques démarrent et rejoignent le réseau en tant que périphériques de la classe A. L'application peut alors décider de passer à la classe B.

Trame de liaison montante en mode Classe B Les trames en uplink en mode Classe B sont identiques aux uplink de classe A à l'exception du bit RFU dans le champ FCtrl de l'en-tête de la trame. Dans l'uplink de classe A, ce bit est inutilisé (RFU).

Format de la trame physique Un Ping en downlink utilise le même format qu'une trame downlink de classe A, mais pourrait suivre un plan de fréquence de canal différent.

Messages MAC en Unicast et Multicast Les messages peuvent être «unicast» ou «multicast». Les messages monodiffusion sont envoyés à un seul terminal et les messages multidiffusion sont envoyés à plusieurs terminaux. Tous les périphériques d'un groupe de multidiffusion doivent partager la même adresse de multidiffusion et les mêmes clés de chiffrement. La spécification *LoRaWAN* Classe B ne spécifie aucun moyen pour configurer à distance un tel groupe de multidiffusion ou pour distribuer de manière sécurisée le matériel de clé de multidiffusion requis. Cela doit être effectué pendant la personnalisation du noeud ou via la couche d'application.

Acquisition du beacon et tracking Avant de passer de la classe A à la classe B, le device doit d'abord recevoir un beacon pour aligner sa référence d'horloge interne avec le réseau. Une fois en classe B, le périphérique final doit périodiquement rechercher et recevoir un beacon pour annuler toute dérive de son d'horloge interne, par rapport à la synchronisation du réseau. Un périphérique de classe B peut être temporairement incapable de recevoir des beacons (hors de portée des passerelles réseau, présence d'interférences, ..). Dans ce cas, le terminal doit élargir progressivement ses fenêtres de réception de beacon et de ping pour prendre en compte une éventuelle dérive de son horloge interne.

Temps de fonctionnement minimum sans balise En cas de perte de balise, un dispositif est capable de maintenir une opération de classe B pendant 2 heures (120 minutes) après avoir reçu le dernier beacon. Cette opération de classe B temporaire sans beacon est appelée "beacon-less". Il se repose donc sur la propre horloge de l'appareil. Pendant le fonctionnement sans beacon, les slots de réception unicast, multicast et beacon sont tous progressivement étendus pour s'adapter à la possible dérive de l'horloge de l'appareil .

Extension de l'opération beacon-less à la réception Pendant cet intervalle de temps de 120 minutes, la réception de tout beacon dirigée vers l'appareil prolonge de 120 minutes supplémentaires, car elle permet de corriger toute dérive temporelle et de réinitialiser la durée des créneaux de réception.

Minimiser la dérive du timing Les dispositifs peuvent utiliser la périodicité précise du beacon (lorsqu'elle est disponible) pour calibrer leur horloge interne et réduire ainsi l'imprécision de la fréquence d'horloge initiale. Comme les oscillateurs de synchronisation présentent un décalage de fréquence prévisible, l'utilisation d'un capteur de température pourrait permettre une minimisation supplémentaire de la dérive de la synchronisation.

Synchronisation d'un emplacement en liaison descendante

Définitions Pour fonctionner correctement dans la classe B, l'appareil doit ouvrir des créneaux de réception à des instants précis par rapport aux beacon de l'infrastructure.L'intervalle entre le début de deux beacon successif est appelé "beacon period". La transmission de la fenêtre du beacon est alignée avec le début de l'intervalle BEACON_RESERVED. Chaque beacon est précédée d'un intervalle de temps de garde où aucun emplacement de ping ne peut être placé. Ceci afin d'assurer qu'une downlink initiée pendant une fenêtre de réception juste avant l'heure de garde aura toujours le temps de se terminer sans entrer en collision avec la transmission du beacon. L'intervalle de temps utilisable pour l'intervalle de ping s'étend donc de la fin de l'intervalle de temps réservé aux beacon jusqu'au début de l'intervalle du beacon suivant.

Emplacement aléatoire Pour éviter les collisions systématiques ou les problèmes de congestion, l'index des créneaux horaires est aléatoire et modifié à chaque période de beacon. Les paramètres suivants sont utilisés : A chaque période du beacon, le terminal

DevAddr	Device 32 bit network unicast or multicast address
pingNb	Number of ping slots per beacon period. This must be a power of 2 integer: *pingNb* = 2^k where 1 <= k <=7
pingPeriod	Period of the device receiver wake-up expressed in number of slots: *pingPeriod* = 2^{12} / *pingNb*
pingOffset	Randomized offset computed at each beacon period start. Values can range from 0 to (pingPeriod-1)
beaconTime	The time carried in the field **BCNPayload**.Time of the immediately preceding beacon frame
slotLen	Length of a unit ping slot = 30 ms

FIGURE 5 – Options de la classe B

et le serveur calculent un nouveau décalage pseudo-aléatoire pour aligner les créneaux de réception. Un chiffrement AES avec une clé fixe est utilisé :

Key = 16 x 0x00

Rand = aes128_encrypt(Key, beaconTime | DevAddr | pad16)

pingOffset = (Rand[0] + Rand[1]x 256) modulo pingPeriod

CID	Command	Transmitted by		Short Description
		End-device	Gateway	
0x10	***PingSlotInfoReq***	x		Used by the end-device to communicate the ping unicast slot data rate and periodicity to the network server
0x10	***PingSlotInfoAns***		x	Used by the network to acknowledge a PingInfoSlotReq command
0x11	***PingSlotChannelReq***		x	Used by the network server to set the unicast ping channel of an end-device
0x11	***PingSlotFreqAns***	x		Used by the end-device to acknowledge a ***PingSlotChannelReq***command
0x12	***BeaconTimingReq***	x		Used by end-device to request next beacon timing & channel to network
0x12	***BeaconTimingAns***		x	Used by network to answer a ***BeaconTimingReq*** uplink
0x13	***BeaconFreqReq***		x	Command used by the network server to modify the frequency at which the end-device expects to receive beacon broadcast
0x13	***BeaconFreqAns***	x		Used by the end-device to acknowledge a BeaconFreqReq command

FIGURE 6 – Commandes de la classe B

Beaconing (Option de classe B)

Couche physique de la balise En plus de relayer les messages entre les terminaux et les serveurs de réseau, toutes les passerelles participent à la mise en place de mécanismes de synchronisation en envoyant des beacon à des intervalles fixes réguliers configurables par le serveur réseau (BEACON_INTERVAL). Toutes les balises sont transmises en mode implicite de paquet radio, c'est-à-dire sans entête physique *LoRa* et sans CRC ajouté par la radio. Le préambule de la balise commence par (plus long que par défaut) les symboles non modulés. Cela permet aux dispositifs d'extrémité de mettre en œuvre une recherche par balise à faible puissance. La longueur de la trame du beacon est étroitement liée au fonctionnement de la couche physique radio. Par conséquent, la longueur réelle de la trame peut changer d'une implémentation de région à une autre.

Size (bytes)	3	4	1/2	7	0/1	2
BCNPayload	NetID	Time	CRC	GwSpecific	RFU	CRC

FIGURE 7 – Trame beacon partie 1

Field	NetID	Time	**CRC**	InfoDesc	lat	long	**CRC**
Value Hex	CCBBAA	CC020000	**7E**	0	002001	038100	**55DE**

FIGURE 8 – Trame beacon partie 2

Format de la trame de Beaconing

The content of the **GwSpecific** field is as follow:

Size (bytes)	1	6
GwSpecific	InfoDesc	Info

The information descriptor**InfoDesc** describes how the information field **Info** shall be interpreted.

InfoDesc	**Meaning**
0	GPS coordinate of the gateway first antenna
1	GPS coordinate of the gateway second antenna
2	GPS coordinate of the gateway third antenna
3:127	RFU
128:255	Reserved for custom network specific broadcasts

FIGURE 9 – Champ gwspecific

Beaconing : format du champ GwSpecific

Gateway coordonées GPS : InfoDesc = 0,1 ou 2 L'encodage des coordonées GPS sont diffusé dans la balise : 3 octets de lat et 3 octets et Lng, respectivement latitude et longitude

Timing précis sur le Beaconing Les beacons sont envoyé toute les 128 secondes à partir de minuit UTC

2.3.8 Classe C : Continiously listening

Les devices implémentant l'option de classe C sont utilisés pour des applications qui ont une énergie plus que suffisante et n'ont donc pas besoin de minimiser le temps de réception. Les terminaux de classe C ne peuvent pas implémenter l'option de classe B. Le périphérique de classe C écoutera les paramètres des fenêtres RX2 aussi souvent que possible. Le device écoute sur RX2 quand il n'est pas (a) envoyant ou (b) recevant sur RX1, selon la définition de la classe A. Pour ce faire, il ouvrira une petite fenêtre sur les paramètres RX2 entre la fin de la transmission d'un uplink et le début de la fenêtre de réception RX1 et basculera sur les paramètres de réception RX2 dès que la fenêtre de réception RX1 sera fermée ; la fenêtre de réception RX2 reste ouverte jusqu'à ce que l'appareil envoie un autre message.

Temps de la deuxième fenêtre de réception pour la classe C Les périphériques de classe C implémentent les mêmes deux fenêtres de réception que les périphériques de classe A, mais ils ne ferment pas la fenêtre RX2 tant qu'ils n'ont pas besoin d'envoyer à nouveau. Par conséquent, ils peuvent recevoir une liaison descendante dans la fenêtre RX2 à tout moment. Une courte fenêtre d'écoute sur la fréquence RX2 et le débitde donnée est également disponible entre la fin de la transmission et le début de la fenêtre de réception RX1.

Multicast liaison descendante De même que pour la classe B, les périphériques de classe C peuvent recevoir des trames downlink multicast. L'adresse de multidiffusion et la clé de session de réseau associée et la clé de session d'application doivent provenir de la couche d'application. Les mêmes limitations s'appliquent aux trames downlink multicast de classe C :

* Ils ne sont pas autorisés à transporter des commandes MAC, ni dans le champ FOpt, ni dans le payload sur le port 0, car une liaison descendante multicast n'a pas la même robustesse d'authentification qu'une trame unicast. * Les bits ACK et ADRACKReq doivent être à zéro. Le champ MType doit porter la valeur de Down Data Unconfirmed Down. * Le bit FPending indique qu'il y a plus de données de multidiffusion à envoyer. Étant donné qu'un périphérique de classe C conserve son récepteur actif la plupart du temps, le bit FPending ne déclenche aucun comportement spécifique du devices.

3 Problème

Il y a un problème qui se révele assez vite. En effet, le *LoRaWAN* est basé sur un modèle d'infrastructure. Par exemple, quand est-il d'un noeud qui n'est pas visible par une *LoRaWAN* Gateway ? Il y a deux cas de figures simple : le premier cas est celui ou le noeud n'est visible d'aucun device et d'aucune gateway *LoRaWAN*. Dans ce cas il n'y a rien à faire le noeud est isolé de manière permanente. Le deuxième cas de figure est celui qui nous intéresse particulièrement ici, en effet le noeud isolé est visible par un end-device qui lui est à porté d'une *LoRaWAN* Gateway tout comme l'illustre le schéma suivant :

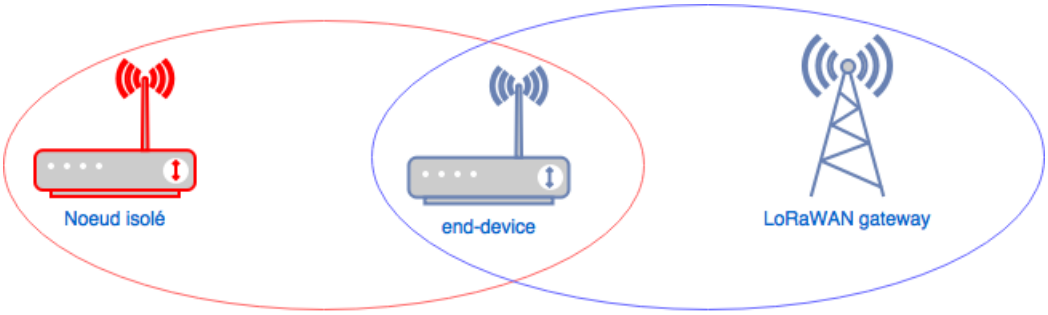

FIGURE 10 – Problème des noeuds isolé

4 Solutions

Il y a plusieurs pistes de solutions afin de résoudre ce problème. Un end node visible par un noeud isolé peut etre redefinie comme une passerelle → On définit donc une *LoRa* Gateway (noté LGW à travers le document) Il y a donc une redéfinition des acteurs :

— LoRaWAN Gateway (LWGW)
— LoRa Gateway
— Isolated Node (IN)

On aurais la possibilité d'avoir deux modes :

— Mode proxy : La LoRaGateway fait un relai en aveugle des données des noeuds Isolés .
— Mode concentrateur : La LoRaGateway collecte les données des Noeuds Isolés.

Il ya tout de même quelque problèmes pour le mode Proxy. En effet il y a une certaine compléxité de la mise en oeuvre. Que ce soit au niveau de l'écoute et la captures des échanges par des end-nodes mais aussi de la capture par le End Node de message du Noeud isolé, ou encore de la capture par le End Node de message à destination du Noeud isolé. Il y a aussi une certaine compléxité au niveau de la gestion des synchronisation en fonction des classes energetique des objets. En classez C, le problème se revele pendant les phases démission du End-Node. En classe A et B lors dez la gestion multiple des slots de réception par le un End Node. Il y a un derniere aspect qui est pratique, en effet les bibliothèques, on écoute et collecte des message sans en etre le destinataire. Il y aurais deux solutions pour pallier à cela, soit la mise en place d'un mode promiscuité, soit d'un recodage complet de la spécification.

Ici nous allons nous intéréssé surtout à l'ensemble des algorithmes soumis par M.Flauzac qui décrive le fonctionnement du mode Concentrateur, afin des les appliquer en simulation, puis de réaliser la mise en oeuvre/applications de ceux-ci.

4.1 Algorithme

L'ensemble des messsages du système sont de la forme :

```
< message_type , source , destination , data >
```

On distingue donc plusieurs messages dans le système. On considère qu'à chaque message correspond une fonction portant le nom du message, initialisant le type et la source du message, et prenant en paramètre la destination et les données.

4.1.1 les messages $IN \rightarrow LGW$

- les messages `discover`
  - message mis en place pour la découverte d'une `LGW` par un `IN`
  - `destination = udef` : en broadcast
  - `data = udef` : aucune info
- les messages `pair`
  - message d'apairage d'un IN sur une LGW
  - `data = udef` : aucune info
- les messages `data_response`
  - réponse à une demande de données de la part d'une LGW

4.1.2 les messages $LGW \rightarrow IN$

Pour l'ensemble des messages LoRa issus de la `LGW` la partie data est structurée ainsi :

- `answer_frequency` : fréquence sur laquelle l'`IN` doit répondre
- `next_slot` : délai d'ici la prochaine fenêtre d'écoute
- `next_duration` : temps fixé de la prochaine fenêtre d'écoute
- `next_frequency` : fréquence de la prochaine fenêtre d'écoute
- `data` : espace de données spécifiques à l'échange

Le message devient donc :

```
< message_type , source , destination , answer_frequency , next_slot ,
                next_duration , next_frequency , data>
```

Les différents messages `LGW` $\rightarrow$ `IN` sont donc :
- les messages `candidate`
  - message de réponse d'une `LGW` après réception d'un `discover` d'un `IN`

  - `data = udef` : aucune info
- les messages `data_request`
  - message de demande de données
  - `data = udef` si une seule donnée disponible ou `data = requested_data` dans le cas de donnés multiples

4.2 Liste des fonctions utilisées

4.2.1 Fonction d'émission

```
void sendLora(frequency , message)
```

4.2.2 Fonction de réception

la fonction `listen` écoute sur la fréquence `frequency` un temps définit par `time`. Le prototype de cette fonction est :

```
(message,time) listen(frequency , source , message_type , time_listen)
```

les valeurs des paramètres de cette fonction sont :
- `frequency` : fréquence d'écoute
- `source` : id de l'émetteur du message
  - `source = udef` : écoute de tous les nœuds sur la fréquence définie
- `message_type` : type de message attendu
  - `message_type = udef` : écoute de tous les types de messages
- `time_listen` : durée de la fenêtre de réception
  - `time_listen = udef` : fenêtre infinie

Valeurs de retour :

- `message` message reçu
  - passage du message dans sa totalité
  - `message == udef` : pas de réception respectant les contraintes
- `time` temps restant basé sur `time_listen`
  - `time == udef` : dans le cas de `time_listen = udef`

Algorithme 1 Initialisation des variables de communication

```
1 : procedure init_var(msg)
2 :     lgw ← msg.source
3 :     freq_send → msg.answer_frequency
4 :     next_time ← msg.next_slot
5 :     timer ← msg.next_duration
6 :     freq_listen ← msg.next_frequency
7 : end procedure
8 :
9 : procedure flush_var( )
10 :    lgw ← udef
11 :    msg ← udef
12 :    next_time ← udef
13 :    timer ← timer_disco
14 :    freq_listen ← freq_disco
15 :    freq_send ← freq_disco
16 : end procedure
```

Algorithme 2 Algorithme IN 1-1

```
1 : while (true) do
2 :     flush_var()
3 :                          ▷ ——————————————————————— phase d'apairage
4 :     while (msg = udef) do
5 :         sendLora(freq_listen, discover(udef, udef))
6 :         (msg, t) = listen(freq_send, udef, candidate, timer + rnd())
7 :     end while
8 :     initVar(msg)
9 :     sendLora(freq_send, pair(lgw, udef))
10 :
11 :                                  ▷ ——————————————————————— Phase d'échanges
12 :    while (lgw! = udef) do
13 :        sleep(next_time)
14 :        (msg, t) = listen(freq_send, lgw, data_request, timer)
15 :        if msg! = udef then
16 :            initVar(msg)
17 :            sendLora(freq_send, date_response(lgw, local_data)
18 :        elseflush_var()
19 :        end if
20 :    end while
21 : end while
```

4.3 Algorithme 1 - 1

4.3.1 Algorithme des `IN`

4.3.2 Algorithme des `LGW`

Algorithme 3 Initialisation des variables de communication

```
1 : procedure init_var( )
2 :     freq_send → chose()
3 :     timer ← chose()
4 :     freq_listen ← chose()
5 :     freq_next ← chose()
6 : end procedure
7 :
8 : procedure flush_var( )
9 :     timer ← timer_disco
10 :    freq_listen ← freq_disco
11 :    freq_send ← freq_disco
12 :    in ← udef
13 : end procedure
```

Algorithme 4 Algorithme lgw 1-1

```
1 : LoRaWAN_join()
2 : flush_var()
3 : while (true) do
4 :     if (in == udef) then
5 :         (msg, t) = listen(freq_listen, udef, discover, timer)
6 :     end if
7 :     if (msg! = udef) then
8 :         in ← msg.source
9 :         init_var()
10 :        sendLora(freq_send, candidate(in, freq_listen, slot, duration, freq_next, udef)
11 :        (msg, t) = listen(freq_listen, in, pair, timer)
12 :        if (msg == udef) then
13 :            flush_var()
14 :        end if
15 :    end if
16 :    if (in! = udef) then
17 :        init_var()
18 :        sendLora(freq_send, data_request(in, freq_listen, slot, duration, freq_next, udef)
19 :        (msg, t) = listen(freq_listen, in, data_response, timer)
20 :        if (msg! = udef) then
21 :            send_lora_data(id + " : " + local_data + ";" + in + " : " + msg.data)
22 :        else
23 :            send_lora_data(id + " : " + local_data + ";" + in + " : " + udef)
24 :            flush_var()
25 :        end if
26 :    end if
27 : end while
```

4.3.3 Algorithme k/1 : k IN <-> 1 LGW

Algorithme 5 k/1 : k IN <-> 1 LGW

Precondition : x and y are packed DNA strings of equal length n

```
1 : function DISTANCE(x, y)
2 :     z ← x ⊕ y                                  ▷ ⊕ : bitwise exclusive-or
3 :     δ ← 0
4 :     for i ← 1 to n do
5 :         if z_i ≠ 0 then
6 :             δ ← δ + 1
7 :         end if
8 :     end for
9 :     return δ
10 : end function
```

4.4 Chronogrammes

4.4.1 Phase de découverte et d'enregistrement

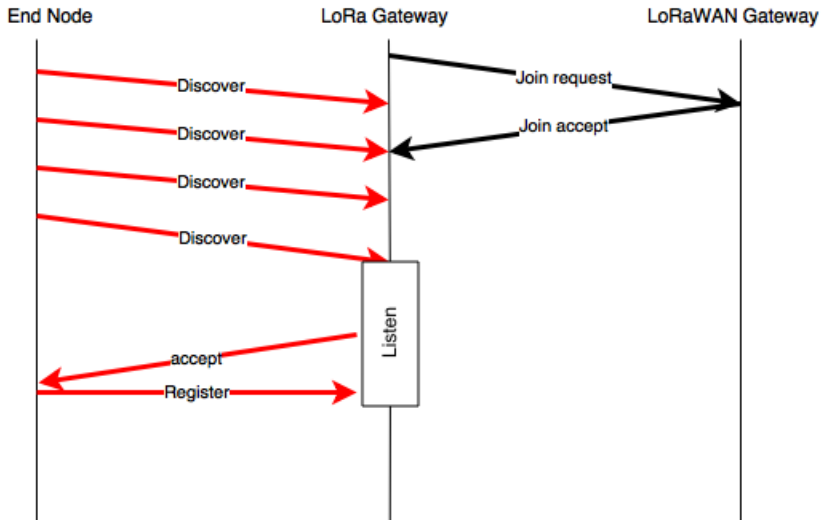

FIGURE 11 – Phase de découverte

Coté Isolated node Au début le noeud isolé se reveille, il envoie des messages **Discover**. Si une *LoRa* Gateway répond par un message **Accept** le noeud répond alors avec un message **Register** pour confirmer son appairage avec la *LoRa* Gateway.

Coté LoRa Gateway La *LoRa* Gateway avant toute opération avec n'importe quel élement de l'environnement doit s'appairer avec une gateway *LoRaWAN* avec la requete **Join Request**.

Coté LoRaWAN Gateway La gateway *LoRaWAN* executé un déroulement normal. Elle répond aux requetes join avec un message **Join Accept**

4.4.2 Phase de collecte

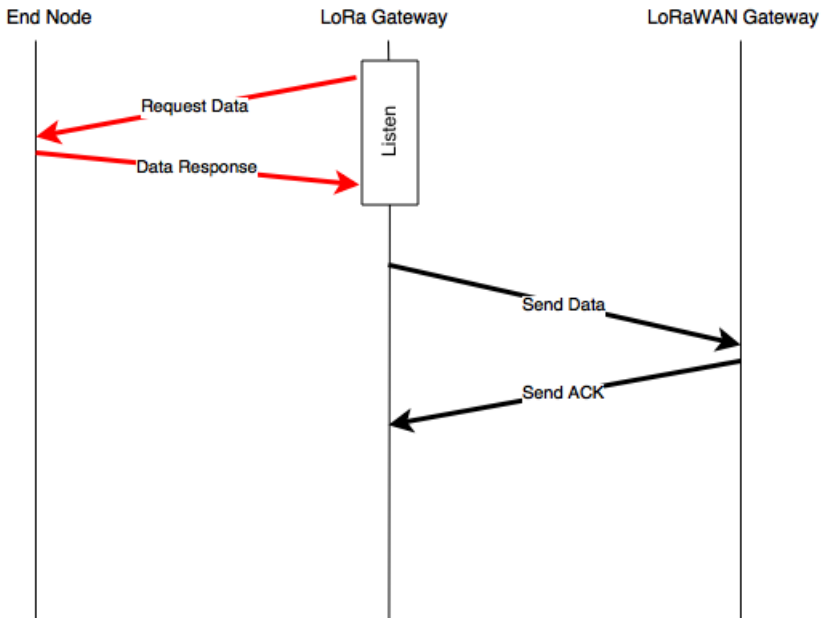

FIGURE 12 – Phase de collecte

Coté Isolated node Le noeud se reveille à un slot prévu entre la *LoRa* Gateway et le noeud. Tout ceci afin de recevoir la demande de donnée de la part de la gateway *LoRa*. Le noeud répond avec les données dans un message nommé **Data Response**.

Coté LoRa Gateway La *LoRa* Gateway va se reveiller pour collecter les données des noeuds qui lui sont accroché. Elle envoit donc un message **Request Data** . Une fois que toutes les données sont reçues, la Gateway assemble les données et envoie à la *LoRaWAN* gateway.

Coté LoRaWAN Gateway La *LoRaWAN* Gateway accuse la reception des données envoyées par la gateway *LoRa*. **Send ACK**

5 Simulation

5.1 Le simulateur : Omnet++

OMNeT ++ est une bibliothèque et une structure de simulation C ++ extensible, modulaire et basée sur des composants, principalement utilisé pour la construction de simulateurs de réseau. Le terme "réseau" est entendu dans un sens plus large qui inclut les réseaux de communication câblés et sans fil, les réseaux sur puce, les réseaux de mise en file d'attente, et ainsi de suite. Les fonctionnalités spécifiques au domaine telles que la prise en charge de réseaux de capteurs, de réseaux ad-hoc sans fil, de protocoles Internet, de modélisation de performances, de réseaux photoniques, etc., sont fournies par des frameworks de modèles, développés en tant que projets indépendants. OMNeT ++ offre un IDE basé sur Eclipse, un environnement d'exécution graphique et une foule d'autres outils. Il existe des extensions pour la simulation en temps réel, l'émulation de réseau, l'intégration de base de données, l'intégration SystemC et plusieurs autres fonctions. Même si OMNeT ++ n'est pas un simulateur de réseau en soi, il a acquis une grande popularité en tant que plate-forme de simulation de réseau dans la communauté scientifique ainsi que dans les milieux industriels, et a constitué une importante communauté d'utilisateurs. Afin de réaliser un ensemble de test/simulation nous avons utilisé Omnet pour ce projet.

Le code des fichiers C^{++} est mis à disposition en Annexes.

5.2 Génération des graphes

Nous avons choisi d'utiliser iGraph, afin de generer des graphes aléatoires. iGraph est une collection de bibliothèque pour créer et manipuler des graphiques et analyser des réseaux. Il est écrit en C et existe également en tant que paquets Python et R. Il existe de plus une interface pour Mathematica. Le logiciel est largement utilisé dans la recherche universitaire en sciences de réseau et dans des domaines connexes. iGraph a été développé par Gábor Csárdi et Tamás Nepusz. Le code source des paquets iGraph a été écrit en C. iGraph est disponible gratuitement sous GNU General Public License Version 2.

La génération se fait en plusieurs étapes :

1. Récupération des arguments : Nombres de *LoRaWAN* Gateway, *LoRa* Gateway et de Isolated node.
2. Création du graphe et ajout du nombre de sommets dont on a besoin.
3. On connecte les *LoRaWAN* à une *LoRaWAN* gateway, par définitions ils sont enregistré à au moins une gateway *LoRaWAN*.
4. On connecte des Isolated Node de manière aléatoire à une LoRa Gateway (au moins une)

5. Ensuite on applique un deuxième tour d'association de noeud à des gateway basé sur une méthode Erdös Renyi : On définit la probabilité d'existence entre deux nœuds, pour chaque couple de nœud, on tire au sort un nombre, si le nombre est inférieur on met le lien.
6. On écrit le fichier .ned qui décrit la configuration du graphe crée.

```
fic.write("\t\t\t" + ""+nomSommet0+"["+str(num_sommet0)+"].channelsO++" + " --> "
+"{delay="+str(delay1)+"ms;}" + " --> " +
""+nomSommet1+"["+str(num_sommet1)+"].channelsI++" + ";\n")
fic.write("\t\t\t" + ""+nomSommet0+"["+str(num_sommet0)+"].channelsI++" + " <-- "
 +"{delay="+str(delay2)+"ms;}" + " <-- " +
   ""+nomSommet1+"["+str(num_sommet1)+"].channelsO++" + ";\n")
```

Par exemple voici le résultat graphique réalisé avec GraphViz :

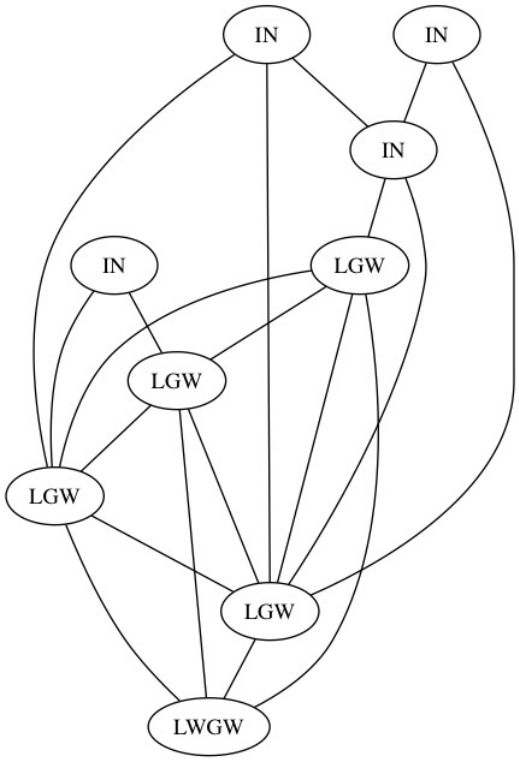

FIGURE 13 – Exemple de Graphe de sortie du script

Le code de génération de graphe aléatoire est mis à disposition dans son intégralité en Annexes.

5.3 Représentation des échanges radio

Dans cette version purement algorithmique sur Omnet++, la radio est definit par sa porté qui est donc la possibilité de communiqué avec un device. Si une arrête existe alors une communication est possible. Afin d'être plus réaliste, les arrêtes on pour attribut une latence, qui est générer aléatoirement au moment de la création du graphe. La radio étant des ondes quand un device envoie un message il l'envoie sur tous les liens qui lui sont attribué.

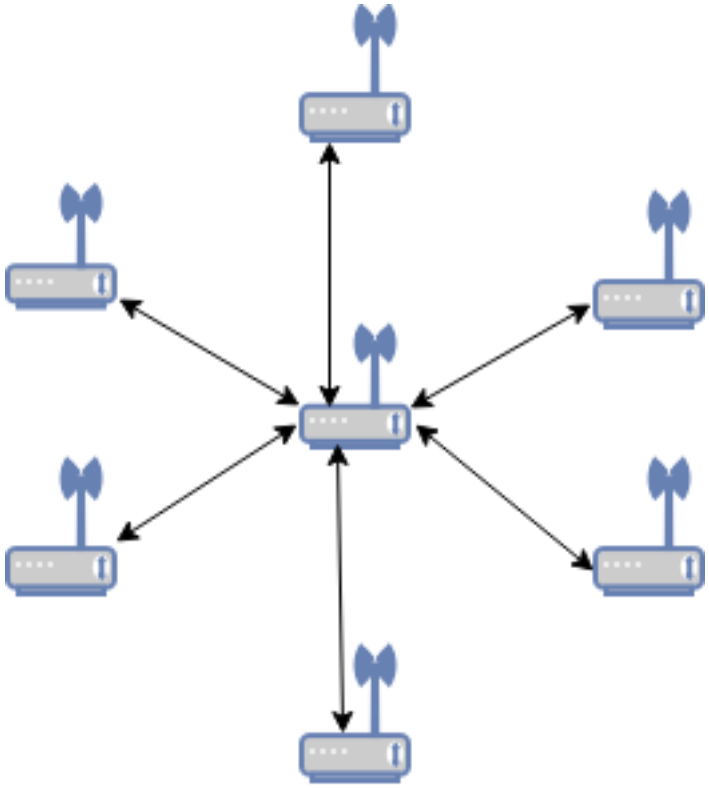

FIGURE 14 – Exemple de communication radio

5.4 Exemples de simulations graphiques

Afin de simplifié, les messages contiennent le nom du message et possedent un code couleur propre et unique. Une *LoRa* Gateway possède une couleur propre à son groupe (elle même et les Isolated nodes associés), les INs ont aussi cette couleur.

5.4.1 Simulation sur une chaine

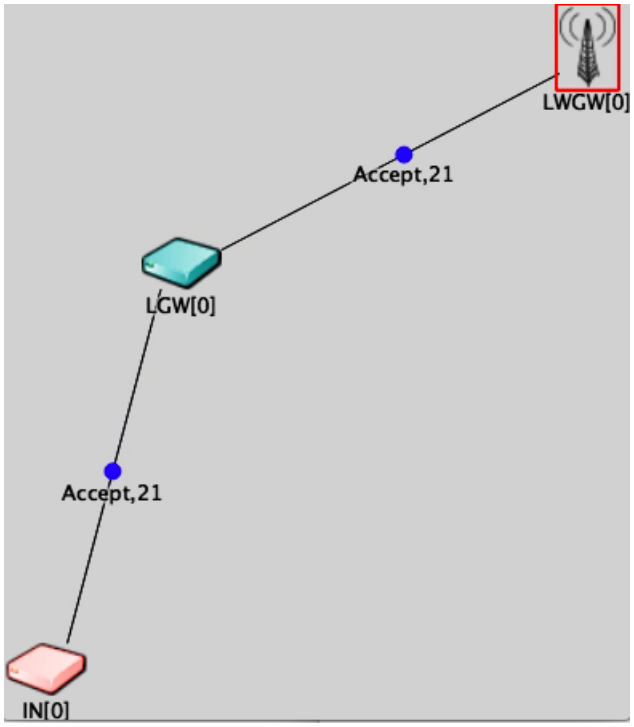

FIGURE 15 – Exemple de chaine

Ceci est une image sortie de la simulation graphique, la première réalisé, un exemple typique de chaine avec seulement un élement de chaque composant. C'est le premier exemple de test réalisé. Ce fut aussi le terrain de devellopement pour affiner le comportement individuel et verifier le fonctionnement correct de chaque composant. Cette simulation ci seras l'exemple typique du première algorithme.

5.4.2 Simulation avec 3 IN sur une LGW

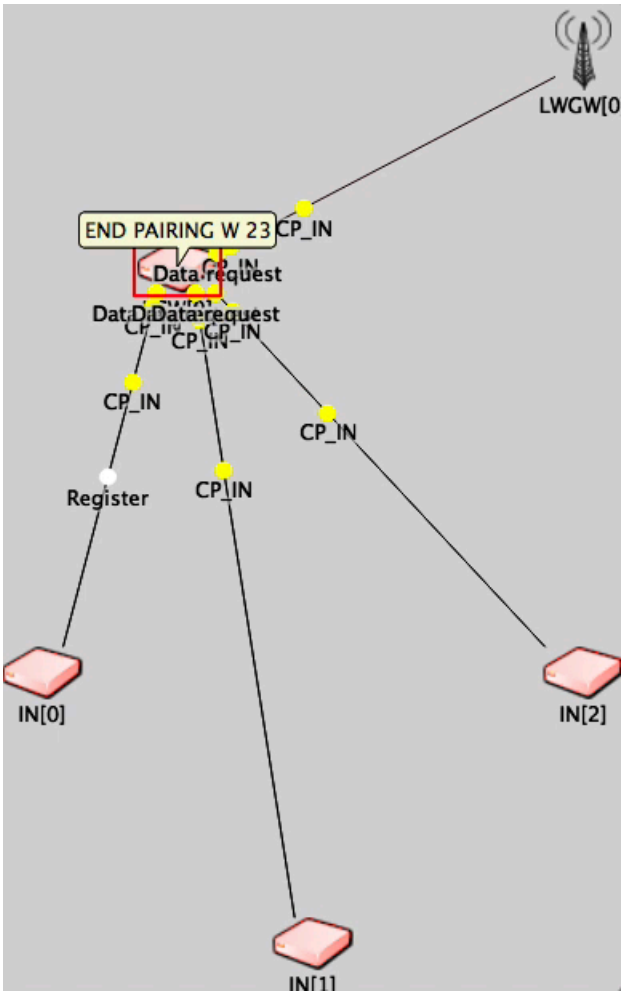

FIGURE 16 – Exemple avec 3 Isolated Nodes

Ceci est le deuxième exemple de simulation pour gérer l'enregistrement multiple.Ici nous avons donc toujours une *LoRaWAN* gateway qui est connecté a une *LoRa* Gateway qui doit gérer trois Isolated Node Cette simulation ci seras l'exemple typique du deuxième algorithme.

5.4.3 Simulation à plus grande echelle

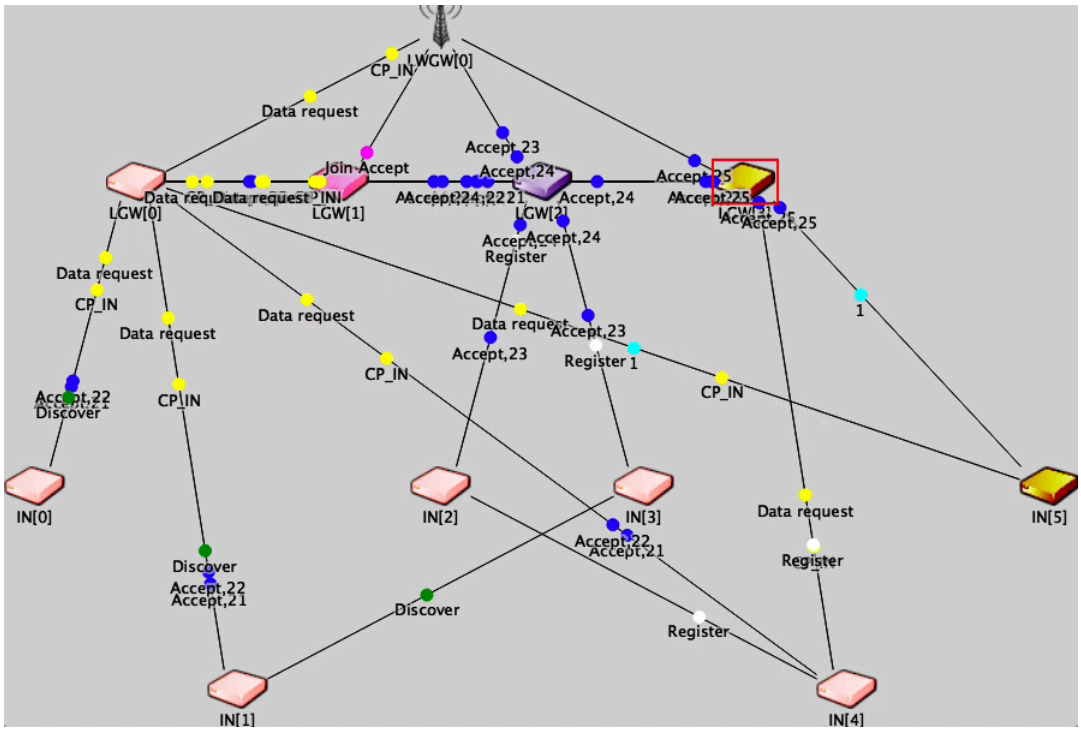

FIGURE 17 – Exemple à grande echelle

Ceci est le troisème exemple de simulation pour gérer l'enregistrement multiple ainsi que l'unicité d'un enregistrement sur une Gateway *LoRa* d'un device.Ici nous avons donc toujours une *LoRaWAN* gateway qui est connecté a quatre *LoRa* Gateway qui doit gérer X Isolated Node Cette simulation ci seras l'exemple typique du deuxième algorithme. Mais aussi d'un troisème cas de figure possible. Cette exemple est surtout présent pour montrer et pousser la simulation à être plus proche de la réalité. L'exemple présenté ci-contre fait aussi transparaitre les exemples ou les communications sont difficiles. Un noeud peut pendant un certain temps ne faire que envoyer des messages **Discover** très longtemps. Pour plusieurs raisons, notamment celle-ci :

— Problème de latence sur le reseaux de communication.
— Problème de bruit radio (Plusieurs emissions sont réalisé en même temps de la part de plusieurs device, que ce soit sur le device de recepetion ou d'émission provocant la corrupition des messages)
— Problème de phase réception, en effet des slots sont prévue ainsi que deux fenêtre de réception dans la specification *LoRa*. Il se peut que deux device pas encore synchroniser est du mal à l'être .

6 Programmation

6.1 LoPy

Tout d'abord le lopy est développée par la société Pycom, la plate-forme de développement et de prototypage d'applications IoT LoPy offre la particularité, unique, de rassembler sur une même carte plusieurs technologies de connectivité radio, courte et longue portées : Bluetooth, Wi-Fi, Sigfox et LoRa. Concrètement, la plate-forme LoPy utilise une implantation spécifique et optimisée du langage de script MicroPython.L'objectif de la plate-forme est de permettre aux développeurs de bâtir des applicatifs pour l'Internet des objets avec la possibilité de mettre au point une nanopasserelle LoRa qui peut, selon Pycom, connecter jusqu'à 100 cartes LoPy et donc de créer son propre réseau privé avec une portée allant jusqu'à 5 kilomètres. C'est donc tout naturel que pour envisager une toutes première implémentation de la gestion des noeuds isolé selon les algorithmes prévus. L'intégralité des codes sources ne seras pas présenté, mais il seras présenté rapidement les parties important de chaque composant develloper.

Tout d'abord un type messageLora a été crée. Celui ci contient
- Le nom du message
- Le type de message → un entier
- La frequence du prochain slot
- Dans combien de temps est le prochain slot
- L'identifiant du noeud à l'origine de l'emission du message
- L'identifiant du noeud cible du message
- une eventuelle data
- Temps d'ecoute sur le prochain slot

Sur la prochaine page vous pourrez voir une partie du fichier de ddescription de la classe des message LoRa

```
class messageLoRa:
    messageName="not set yet"
    frequency=0
    slots=0
    id_src=-1
    id_dest=-1
    data=-1
    kind=0
    listeningtime=0
    def __init__(self):
        self.messageName="not set yet"
        self.kind=-1
        self.frequency=0
        self.slots=0
        self.id_src=-1
        self.id_dest=-1
        self.data=-1
        self.listeningtime=0
    def fillMessage(self, data):
        message=data.decode()
        if message != '':
            self.messageName,
            self.kind,
            self.frequency,
            self.slots,
            self.id_src,
            self.id_dest,
            self.data,
            self.listeningtime= message.split(",")
        else:
            print("Received nothing")
```

Tout d'abord la gestion des phase d'écoute seras gere grace à une petite classe TimerL (L comme Listening). Celle ci change la variable d'etat du capteur à l'instant t choisi .

```
class TimerL:
    def __init__(self,timing):
        self.seconds = 0
        self.__alarm = Timer.Alarm(self._seconds_handler, timing, periodic=True)
    def _seconds_handler(self, alarm):
        global isListening
        alarm.cancel() # stop it
        if isListening:
            isListening=False
```

Cette portion de code permet un changement rapide et simple de la frequence sur laquelle ecoute le capteur. Dans les message l'attribut frequence désigne l'entier qui seras geré par cette fonction.

```
def change_frequency(frequency_d):
    current_frequency=lora.frequency()
    if current_frequency != frequency_d:
        print("FREQUENCY WAS CHANGED FROM :"+str(current_frequency)+" TO= ")
        if frequency_d == 1:
            lora.frequency(868000000)
            print("868000000")
        if frequency_d == 2:
            lora.frequency(868100000)
            print("868100000")
        [.....]
            lora.frequency(864100000)
            print("864100000")
        if frequency_d == 6:
            lora.frequency(864300000)
            print("864300000")
        if frequency_d == 7:
            lora.frequency(864500000)
            print("864500000")
```

6.1.1 Isolated Node

Cette partie va être consacré au lopy désigne comme noeud isolé. Concernant le comportement classique et globale. Tout d'abord la fonction de Découverte, qui décrit la phase *Discover* comme décrite plus haut. On recoit donc un message qui si il corespond à un message de type *Accept* on engrange donc dans la phase suivante, celle du registering.

```
def notDiscovered():
    global tryDiscover
    global discovered
    global myLoRa
    global id
    global frequency
    global slot
    print("PHASE NOT DISCOVERED STARTED "+str(tryDiscover))
    s.send('Discover,'+str(1)+','
        +str(frequency)+','+str(slot)+','+
        str(id)+','+str(-1)+','+str(-1)+','+str(-1))
    print("Discover sent by "+str(id))
    data=s.recv(128)
    msg =messageLoRa()
    msg.fillMessage(data)
    if msg.messageName == "Accept":
        myLoRa=msg.id_src
        frequency=msg.frequency
        change_frequency(msg.frequency)
        print("Receive ACCEPT msg")
        discovered=True
    tryDiscover+=1
    print("PHASE NOT DISCOVERED ENDED\n")
```

Dans cette phase registering on envoit le message de type *Register* On recoit donc un message qui si il corespond à un message de type DataReq on engrange donc dans la phase suivante, qui est celle de l'attente d'un slot puis de demande de données.

```
def notRegistered():
    #send some data
    global tryRegister
    global registered
    global myLoRa
    print("PHASE NOT REGISTERED STARTED\n")
    s.send('Register,'+str(3)+','
        +str(frequency)+','+str(slot)+
        ','+str(id)+','+str(myLoRa)+
        ','+str(-1)+','+str(-1))
    print("Register sent")
    # get any data received...
    data=s.recv(128)
    msg =messageLoRa()
    msg.fillMessage(data)
    if msg.messageName == "DataReq" and msg.id_src== myLoRa:
        registered=True
    else:
        tryRegister+=1
    print("PHASE NOT REGISTERED ENDED\n")
```

Pour finir, la fonction d'envoie des données, qui fait partie d'un déroulement classique et qui répond à un message DataReq.

```
def sendData():
    #send some data
    global tryDataReq
    global data
    global myLoRa
    global id
    global slot
    global frequency
    print("PHASE SEND DATA STARTED\n")
    s.send('DataRes,'+str(5)+','
        +str(frequency)+','+str(slot)+','+
        str(id)+','+str(myLoRa)+','+str(data)
        +','+str(-1))
    print("DataResponse sent")
    print("PHASE SEND DATA ENDED\n")
```

```
while True:
    if isListening:
        print("I am awake : my LoRaGW is "
            +str(myLoRa)+" and my slot is "+str(slot))
        pycom.rgbled(0x007f00) # green
        #We are not discovered yet
        while not discovered:
            notDiscovered()
            rnd=Random()
            print("Try Discover in "+str(rnd))
            time.sleep(rnd)
        while not registered and discovered:
            notRegistered()
            rnd=Random()
            print("Try Register in "+str(rnd))
            time.sleep(rnd)
        dataR=s.recv(128)
        msg =messageLoRa()
        msg.fillMessage(dataR)
        if msg.kind=="4":
            sendData()
            slot=int(msg.listeningtime)
            clock = TimerL(float(msg.listeningtime))
            print("I sent my data")
        data+=1
    else:
        pycom.rgbled(0x7f0000) #red
        print("I am sleeping")
        del clock
        time.sleep(slot)
        isListening=True
        clock = TimerL(slot)
```

6.1.2 LoRaGateway

Concernant le composant designant le comportement LoRaGateway. Tout d'abord, la partie attrité a la phase d'appairage. Tout simplement si on recçoit un message *Discover* on renvoit un message *Accept* puis on ajoute au tableau des idRegistered les id des machines en cours d'appairage.

```
def pairing_phase(msg):
    global slot
    global idRegistered
    #print("PAIRING PHASE WITH "+str(msg.id_src)+" STARTED")
    s.send('Accept,'+str(2)+','+
            str(frequency)+','+str(slot)+
            ','+str(id)+','+str(msg.id_src)+
            ','+str(-1)+','+str(slot))
    idRegistered.append(msg.id_src)
    #print("PAIRING PHASE WITH "+str(msg.id_src)+" ENDED")
```

Pour suivre, la fonction de finalisation de l'appairage des données, qui fait partie d'un déroulement classique et qui répond à un message Register.

```
def registering_phase(msg):
    global isRegistered
    global slot
    #print("REGISTERING PHASE WITH "+str(msg.id_src)+" STARTED")
    s.send('DataReq,'+str(2)+','+
            str(frequency)+','+str(slot)+
            ','+str(id)+','+str(msg.id_src)+
            ','+str(-1)+','+str(slot))
    if msg.id_src in isRegistered:
        print("Added before")
    else:
        isRegistered.append(msg.id_src)
```

La routine qui exprime et l'attitude principale de la gateway LoRa qund elle ne recoit aucun message. C'est donc la phase de collecte des informations. Qui ce font en parcourant le tableau d'identifiant des noeuds bien appairé avec la gateway LoRa.

```
def standard():
    print("STANDARD PHASE STARTED")
    global isRegistered
    global slot
    for idDest in isRegistered:
        print(idDest)
        print('DataReq,'+str(2)+','+
        str(frequency)+','+str(slot)+
        ','+str(id)+','+str(msg.id_src)+
        ','+str(-1)+','+str(slot))
        s.send('DataReq,'+str(2)+','+
        str(frequency)+','+str(slot)+
        ','+str(id)+','+str(msg.id_src)+
        ','+str(-1)+','+str(slot))

        dataHarvested = s.recv(128)
        msgH =messageLoRa()
        msgH.fillMessage(dataHarvested)
        rnd=Random()
        print("[FIRST Send] Request data in "+str(rnd))
        print(dataHarvested)
        time.sleep(rnd)
        while msgH.id_src != idDest and msgH.id_dest != id and msgH.kind != "5":
            rnd=Random()
            print("[Try] send Request data in "+str(rnd))
            time.sleep(rnd)
            s.send('DataReq,'+str(2)+','+
                str(frequency)+','+str(slot)+
                ','+str(id)+','+str(msg.id_src)+
                ','+str(-1)+','+str(slot))
            dataHarvested = s.recv(128)
            msgH =messageLoRa()
            msgH.fillMessage(dataHarvested)
    print("STANDARD PHASE ENDED")
```

Voici donc a quoi ressemble le programme global.

```
while True:
    if isListening:
        print("I am awake")
        pycom.rgbled(0x007f00) # green
        data = s.recv(128)
        handle_message(data)
        time.sleep(0.500)
        recolte=standard()
        print(recolte)
        time.sleep(0.500)
        if recolte !="" :
            changetoLW()
            s.setblocking(True)
            send_datatoLWGW(s,recolte)
            s.setblocking(False)
            changetoLoRa(lora)
    else:
        pycom.rgbled(0x7f0000) #red
        print("I am sleeping")
        time.sleep(slot)
        isListening=True
        del clock
        clock = TimerL(slot)
```

Le programme fait donc une recolte des données et les formatent de la manière suivante :

<ID_IN1>,<DATA_IN1> :<ID_IN2>,<DATA_IN2> :....

Les fonctions suivantes : *changetoLW(), send_datatoLWGW(s,recolte), changetoLora(lora)* change la manière de communiqué de la LoRaGateway : soit en LoRa, soit en LoRaWAN, et bien entendu la fonction send, s'occupe de formater la chaine donné en bytes et de l'envoyer.

7 Problèmes rencontrés

Lors du dévellopement sur les LoPy, il y a eu un en semble de problème.

1. Ce premier est du a la lever d'une exception EAGAIN, qui levée quan le système detecte au bout de trois action identique et répété dans un espace de temps assez court. Par exemple lors des tests réalisé nous etions entre les 500ms et 800ms de tolérance de répétion minimale. La mesure n'etant pas facile à cette échelle de temps, cela explique l'intervalle assez large.
2. Le deuxième problème a été logiciel. En effet lorsque la gateway LoRa fait sont premier join request avant d'entamer tout traitement. Le lopy ne garde pas en mémoire qu'il a déjà join une gateway LoRaWAN, en effet la récupération des données des IN se fait en mode.LoRa. Or quand on veut les remonter à la LoRaWAN Gateway on doit à nouveau executer une Join Request.

8 Conclusion

Durant ce projet consacrée à la gestion des noeuds isolé en LoRa, la programmation réalisé sur les LoPy a révéle que la théorie a pu être appliqué. Cependant il n'as pas été inutile de réaliser un ensemble de simulation, afin de voir l'intégralité des messges à prévoir, les eventuels problèmes. Ainsi que la réalisation de certain timer, et d'echange. Mais aussi et surtout pour voir la quantité de message qui sont émit sur une seule et même fréquence. L'idéal serait donc d'attribuer des fréquences pour chaque noeud. Ce qui evite de surcharger une fréquence, même si statistiquement les échanges finiront par s'effectuer. Il est nécéssaire d'écourter au maximum les temps ou le capteur est actif. C'est donc pour cela que pour continuer ce projet il faudrais envisager un changement d'organisation sur les échanges et le nombre de noeud prix en charge sur chaque fréquence et sur chaque LoRaGateway. Il faudrais aussi envisager un changement dans la simulation pour pouvoir réaliser des tests plus poussés. Notamment une vrai implémentation du LoRa dans Omnet++ comme par exemple FloRa. Pour finir cette ouverture, il serait plus que intéréssant de s'atteler à une programmation sur des capteurs Semtech.

Table des figures

Liste des Algorithmes

Annexes

A Code iGraph

```
#! /usr/bin/python

from igraph import *
import sys
import random

#Random number is the probability which have a LoraGateway to have one isolated node
p=random.random()
#Random number is the probability which have a LoraGateway to have another isolated node
p2=random.random()
#Random number is the probability  have an Isolated Node to have a connection with another
p3=random.random()

if(len(sys.argv) ==6):

        #Getting numbers of LoRaWAN gateways and LoRagateways and Isolated nodes
        nb_LWGW = int(sys.argv[1])
        nb_LGW = int(sys.argv[2])
        nb_IN = int(sys.argv[3])
        lower_bound= int(sys.argv[4])
        upper_bound= int(sys.argv[5])
        #nb_LWGW = 1
        #nb_LGW = 3
        #nb_IN = 6

        nbVertex=nb_LWGW+nb_LGW+nb_IN
        #make the graph
        g = Graph()
        #Adding LoRaWAN, LGW, IN
        g.add_vertices(nbVertex)

        #First step and the second step are: Connecting the LoRagateways to the LoRaWAN and

        for i in range(1,nb_LGW+1):
                g.add_edges([(0,i)])
```

```
        for j in range(1,nb_LGW+1):
                if (i != j and i>j):
                        g.add_edges([(i,j)])

#Connecting the Isolated node to the LoRa gateway because an..
#.. Isolated node must have at less one LoRa gateway around
nb_IN_remaining=nb_IN
firtIN=nb_LWGW+nb_LGW
for j in range (nb_LGW+1,nbVertex):
                i=random.randint(nb_LWGW,nb_LGW)

                g.add_edges([(i,j)])

#Connecting another Isolated node to the LoRagateway because random is fun
nb_IN_remaining=nb_IN
firtIN=nb_LWGW+nb_LGW
for j in range (nb_LGW+1,nbVertex):
        i=random.randint(nb_LWGW,nb_LGW)
        pj=random.random()
        if(pj<p2 and -1==g.get_eid(i, j, directed=False, error=False)):
                g.add_edges([(i,j)])

#Last step is consist in making neighborhood between the INs
for j in range (nb_IN,nbVertex-1):
        i=random.randint(nb_IN,nbVertex-1)
        pj=random.random()
        if(pj<p3 and i!=j and -1==g.get_eid(i, j, directed=False, error=False)):
                g.add_edges([(i,j)])

#Changing the name of each vertex
names=[]
names.append("LWGW")
for i in range(1,nb_LGW+1):
        names.append("LGW")
for j in range (nb_LGW+1,nbVertex):
        names.append("IN")


g.vs["label"] =names
g.es[0]
#Writing NED file like the sample provided
```

```
fic=open("randomGrapheLoRa.ned","a")
fic.write("\t submodules:\n");
fic.write("\t\t LWGW["+str(nb_LWGW)+"]: LWGW; \n");
fic.write("\t\t LGW["+str(nb_LGW)+"]: LGW; \n");
fic.write("\t\t IN["+str(nb_IN)+"]: IsoN; \n");
fic.write("\t connections:\n");
for l in g.get_edgelist():
        #we have to know if the node l[0] & l[1] is a LWGW or a LGW or a IN.
        if (l[0] <nb_LWGW and l[0]>=0):
                #It's a LoRaWANGATEWAY
                nomSommet0="LWGW"
                num_sommet0=l[0]
        elif (l[0] <nb_LGW+1 and  l[0]>=nb_LWGW):
                nomSommet0="LGW"
                num_sommet0=l[0]-nb_LWGW
        elif (l[0] <nbVertex+1 and  l[0]>=nb_LGW):
                nomSommet0="IN"
                num_sommet0=l[0]-nb_LGW-1
                if num_sommet0 ==6:
                        print "toto"


        if(l[1] <nb_LWGW and l[1]>=0):
                #It's a LoRaWANGATEWAY
                nomSommet1="LWGW"
                num_sommet1=l[1]
        elif (l[1] <nb_LGW+1 and  l[1]>=nb_LWGW):
                nomSommet1="LGW"
                num_sommet1=l[1]-nb_LWGW
        elif (l[1] <nbVertex+1 and  l[1]>=nb_LGW):
                nomSommet1="IN"
                num_sommet1=l[1]-nb_LGW-1
                if num_sommet1 ==6:
                        print "toto"


        #Writing into the file
        delay1=random.randint(lower_bound,upper_bound)
        delay2=random.randint(lower_bound,upper_bound)

        fic.write("\t\t\t" + ""+nomSommet0+"["+str(num_sommet0)+"].channels0++" + "
```

```
                fic.write("\t\t\t" + ""+nomSommet0+"["+str(num_sommet0)+"].channelsI++" + "
        fic.write("}")
        fic.close()

        #Writing .dot file #Graphviz
        g.write_dot("loraGraph.dot")
else:
        print "6 arguments is needed, the right way to use this python script is :\n"
        print "python loraGraph.py <NUMBER_LORAWANGATEWAY> <NUMBER_LORAGATEWAY> <NUMBER_ISO
```

B Code Omnet++

B.1 Fichier .h

```
/*
 * frequency.h
 *
 *  Created on: 22 janv. 2018
 *      Author: JoffreyHerard
 */

#ifndef FREQUENCY_H_
#define FREQUENCY_H_

#define LOG if(0)
#define DEBUG if(0)

/*EU 863-870MHz ISM Band*/
#define EU_A_1 868.10   /* as 1 in receivePhase */
#define EU_A_2 868.30   /* as 2 in receivePhase */
#define EU_A_3 868.50   /* as 3 in receivePhase */
#define EU_A_4 864.10   /* as 4 in receivePhase */
#define EU_A_5 864.30   /* as 5 in receivePhase */
#define EU_A_6 864.50   /* as 6 in receivePhase */
#define EU_B_1 869.525  /* as 7 in receivePhase*/

#define RECEIVE_DELAY1 1
#define RECEIVE_DELAY2 2
#define JOIN_ACCEPT_DELAY1 5
#define JOIN_ACCEPT_DELAY2 6
#define MAX_FCNT_GAP 16384
#define ADR_ACK_LIMIT 64
#define ADR_ACK_DELAY 32
#define ACK_TIMEOUT 2
#endif /* FREQUENCY_H_ */
```

```
#ifndef __LORA_ISON_H_
#define __LORA_ISON_H_
#include <omnetpp.h>
#include <stdlib.h>     /* srand, rand */
#include <time.h>       /* time */
#include <unistd.h>
#include "messageLoRA.h"
using namespace omnetpp;
using namespace std;
class IsoN : public cSimpleModule
{
  public:
    double getFrequency() const;
    void setFrequency(double frequency);
    int getId() const;
    void setId(int id);
    int getX() const;
    void setX(int x);
    bool isDiscovered() const;
    void setDiscovered(bool discovered);
    int getData() const;
    void setData(int data);
    double getOldPhase() const;
    void setOldPhase(double oldPhase);
    int getSlot() const;
    void setSlot(int slot);
  private:
    int id,time,data,tryDiscover;
    bool discovered,registered;
    double frequency,old_phase;
    int slot;
    int myLoRa;
    string mycolor;
  protected:
    virtual void initialize();
    virtual void handleMessage(cMessage *msg);
    void notListeningHandleMessage(messageLoRA *msg);
    void isListeningHandleMessage(messageLoRA *msg);
};

#endif
```

```
#ifndef __LORA_LGW_H_
#define __LORA_LGW_H_
#include <omnetpp.h>
#include <time.h>
#include <unistd.h>
#include <vector>
#include "messageLoRA.h"
using namespace omnetpp;
using namespace std;
class LGW : public cSimpleModule{
    public:
        bool isDiscovered() const;
        void setDiscovered(bool discovered);
        double getFrequency() const;
        void setFrequency(double frequency);
        int getId() const;
        void setId(int id);
        int getNbIn() const;
        void setNbIn(int nbIn);
        const vector<int>& getIdRegistered() const;
        void setIdRegistered(const vector<int>& idRegistered);
        double getOldPhase() const;
        void setOldPhase(double oldPhase);
        int getSlot() const;
        void setSlot(int slot);
  private:
        int id,time,NbIN,slot,MyLW,nb_harvest;
        vector<int> idRegistered;
        vector<bool> isRegistered;
        double frequency;
        bool discovered;
        double old_phase;
        string mycolor;
  protected:
        virtual void initialize();
        virtual void handleMessage(cMessage *msg);
        void notListeningHandleMessage(messageLoRA *msg);
        void isListeningHandleMessage(messageLoRA *msg);
};
#endif
```

```
#ifndef __LORA_LWGW_H_
#define __LORA_LWGW_H_
#include <omnetpp.h>
#include "messageLoRA.h"

using namespace omnetpp;
using namespace std;
class LWGW : public cSimpleModule
{
public:
    bool isDiscovered() const;
    void setDiscovered(bool discovered);
    const double getFrequency() const;
    void setFrequency(double frequency);
    const vector<int>& getIdRegistered() const;
    void setIdRegistered(const vector<int>& idRegistered);
    const vector<int>& getIdRegisteredLgw() const;
    void setIdRegisteredLgw(const vector<int>& idRegisteredLgw);
    double getOldPhase() const;
    void setOldPhase(double oldPhase);
    int getSlot() const;
    void setSlot(int slot);

  private:
    bool discovered;
    double frequency;
    double old_phase;
    vector<int> idRegistered;
    vector<int> idRegisteredLGW;
    int slot,id;
  protected:
    virtual void initialize();
    virtual void handleMessage(cMessage *msg);
    void notListeningHandleMessage(messageLoRA *msg);
    void isListeningHandleMessage(messageLoRA *msg);
};

#endif
```

```
#ifndef MESSAGELORA_H_
#define MESSAGELORA_H_
#include <omnetpp/cmessage.h>
#include <vector> /*required for slots definition*/
#include <ctime> /*required for slots definition*/
#include "frequency.h"
using namespace std;
class messageLoRA: public omnetpp::cMessage {
    private:
        string messageName;
        double frequency;
        int slots;
        long int id_src;
        long int id_dest;
        bool isolated;
    public:
        messageLoRA();
        messageLoRA(const messageLoRA&);
        virtual messageLoRA *dup() const {return new messageLoRA(*this);}
        virtual ~messageLoRA();
        double getFrequency() const;
        void setFrequency(double frequency);
        const string& getMessageName() const;
        void setMessageName(const string& messageName);
        long int getIdSrc() const;
        void setIdSrc(long int idSrc);
        int getSlots() const;
        void setSlots(int slots);
        long int getIdDest() const;
        void setIdDest(long int idDest);
        bool isIsolated() const;
        void setIsolated(bool isolated);
};
#endif /* MESSAGELORA_H_ */
```

B.2 Fichier .cpp

L'intégralité des codes se trouve dans l'archive jointe au présent rapport.

C Arborescence des fichiers

```
Racine du projet
├── omnetpp.ini
├── Makefile
├── README.md
├── igraph
│   ├── loraGraph.py
│   └── README
├── out
│   ├── clang-debug
│   │   └── src
│   │       └── Fichier de debug
│   └── clang-release
│       └── src
│           └── Fichier de release
├── old
│   └── src
│       └── Ancien fichier source d'une ancienne méthode d'enregistrement des
│           noeuds
├── results
├── lopy
│   ├── README.md
│   ├── test
│   │   └── Fichiers de tests
│   └── current
│       ├── IsolatedNode.py
│       ├── messageLoRa.py
│       ├── LoRaGateway.py
│       └── nota_DO_NOT_DELETE.txt
├── simulations
│   ├── omnetpp.ini
│   ├── network.ned
│   ├── lora.ned
│   ├── run
│   └── package.ned
├── Rapport_PFE
│   └── Rapport.pdf
└── src
    └── Code source de Omnet++
```