UNIVERSITÉ
DE REIMS
CHAMPAGNE-ARDENNE

UFR Sciences Exactes et Naturelles

Gestion des noeuds isolés en LoRa

Auteur :
HERARD Joffrey

Responsables :
Olivier FLAUZAC
Florent NOLOT
Philipe COLA

2017-2018

Résumé

La problémablique de notre étude portant sur la gestion des noeuds isolés en *LoRa*. Nous expliquerons les problèmes que peut présenter une architecture *LoRaWAN* et proposerons diverses solutions pour les résoudre. Avant de procéder à l'implémentation des ces diverses solutions nous exécuterons des simulations sur le modèle de communication *LoRa* avec les algorithmes présentés auparavant. Par la suite, il a été expliquer comment programmer cette solution dans un capteur via l'utilisation de capteurs *LoPy* voir même *Discovery* basé sur les puces ST32.

Table des matières

Annexes **25**

Chapitre 1

Présentation du sujet

1.1 Sujet

Tout d'abord pour comprendre l'étendu du sujet il est nécéssaire de bien comprendre tout d'abord l'ensemble des termes qui le décrivent.

Définition 1. *Noeuds isolés : 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 :*

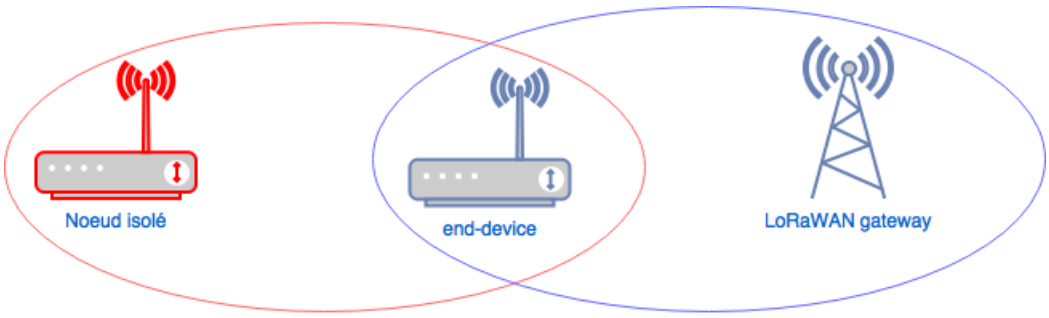

FIGURE 1.1 – Problème des noeuds isolés

Les objectifs sont donc de gérer les noeuds isolés. Le tout en étant capable de faire et de respecter un ensemble de choses :

1. Récupérer les informations détenue par les capteurs/noeuds isolé.
2. Gérer les noeuds isolés de sortes a ne pas leur faire gaspiller de l'energie trop souvent.
3. Faire en sorte que du point de vue du serveur réseaux et de SPOT les noeuds isolés soit actif.
4. Développer des classes/libraires pour utiliser l'algorithme de manière transparente.

Le prochain chapitre décrit le fonctionnement ainsi que l'architecture d'un Réseaux LoRaWAN dit classique. C'est à dire le réseaux avant l'implémentation d'un algorithme comme celui présenté au travers de ce rapport.

Chapitre 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*.

2.0.1 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 .

2.0.2 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.

2.0.3 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…

2.0.3.1 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*.

2.0.3.2 Les différentes couches d'un réseaux LoRaWAN

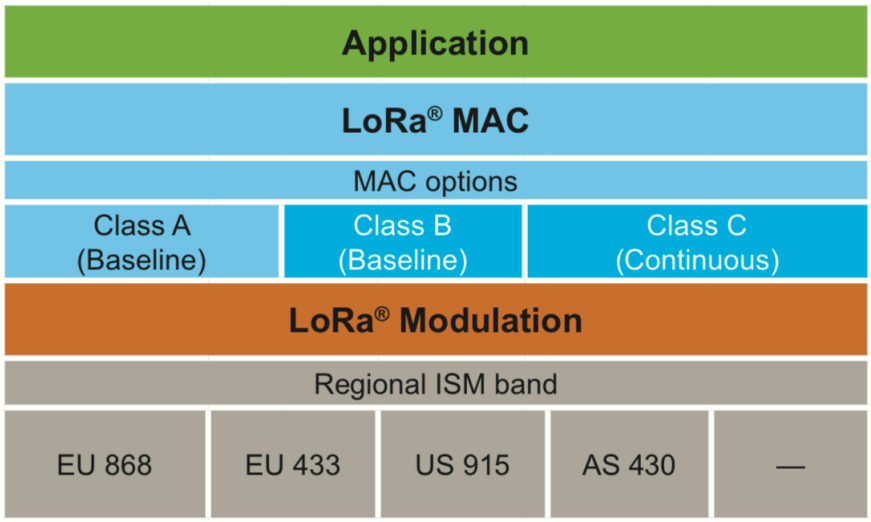

FIGURE 2.1 – Les différentes couches d'un réseau LoRaWAN

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...

2.0.3.3 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.

2.0.3.4 Architecture d'un réseaux LoRaWAN

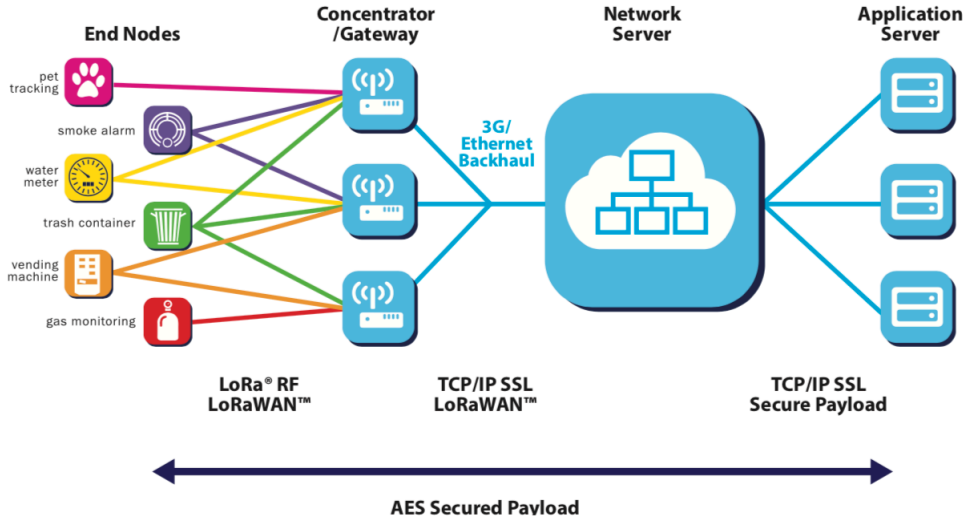

FIGURE 2.2 – Architecture d'un réseaux LoRaWAN

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*.

2.0.3.5 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.

Chapitre 3

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 qui décrive le fonctionnement du mode d'échanges classique entre une LGW et un IN, afin des les appliquer en simulation, puis de réaliser la mise en oeuvre/applications de ceux-ci.

3.1 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.

3.1.1 les messages IN -> 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

3.1.2 les messages LGW -> 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` → `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

3.2 Liste des fonctions utilisées

3.2.1 Fonction d'émission

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

3.2.2 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`

3.3 Algorithme 1 - 1

3.3.1 Algorithme des `IN`

Algorithm 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
```

Algorithm 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
```

3.3.2 Algorithme des `LGW`

Algorithm 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
```

Algorithm 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
```

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

Ici le seul changement qui opère est celui d'enregistrement de plusieurs noeuds isolés.

Algorithm 5 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 :        if (inNOTintabRegisteredIN) then
18 :            tabRegisteredIN.append(in)
19 :        end if
20 :        init_var()
21 :        sendLora(freq_send, data_request(in, freq_listen, slot, duration, freq_next, udef)
22 :        (msg, t) = listen(freq_listen, in, data_response, timer)
23 :        if (msg! = udef) then
24 :            send_lora_data(id + " : " + local_data + "; " + in + " : " + msg.data)
25 :        else
26 :            send_lora_data(id + " : " + local_data + "; " + in + " : " + udef)
27 :            flush_var()
28 :        end if
29 :    end if
30 : end while
```

3.4 Chronogrammes

3.4.1 Phase de découverte et d'enregistrement

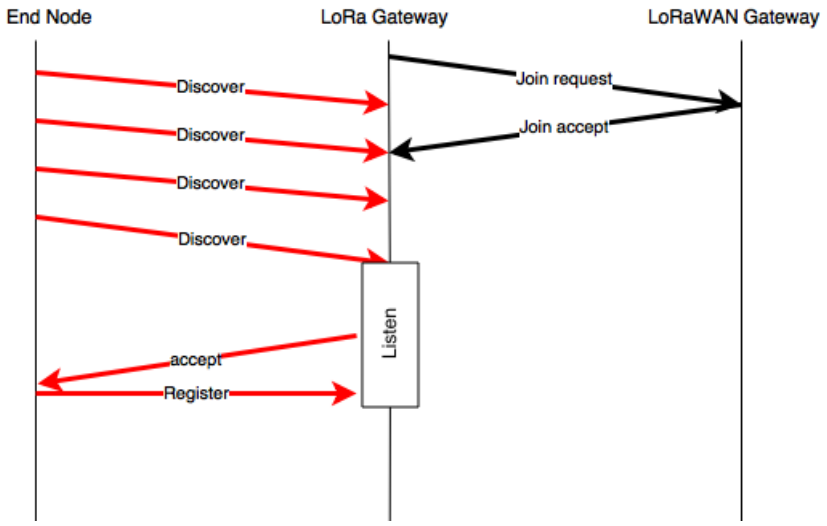

FIGURE 3.1 – Phase de découverte

3.4.1.1 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.

3.4.1.2 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**.

3.4.1.3 Coté LoRaWAN Gateway

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

3.4.2 Phase de collecte

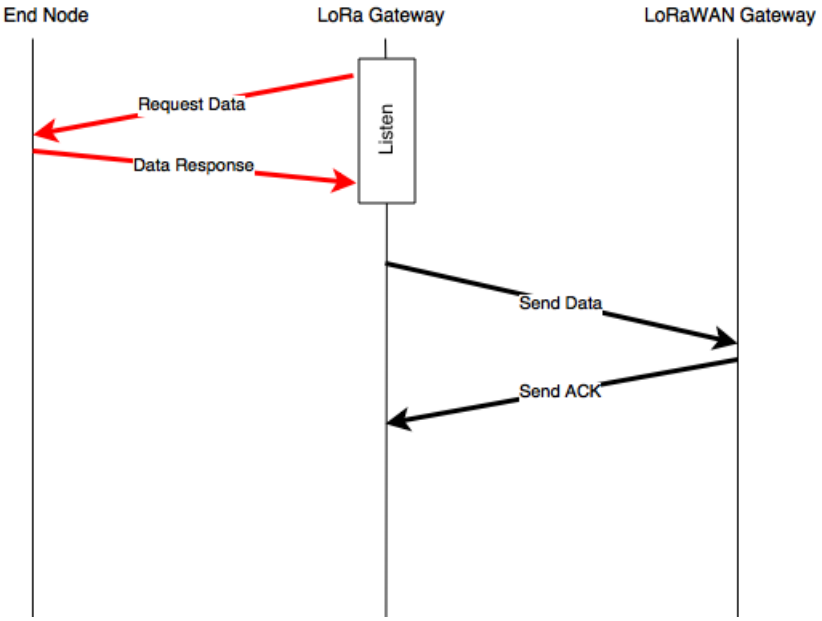

FIGURE 3.2 – Phase de collecte

3.4.2.1 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**.

3.4.2.2 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** .
Mode Concentrateur : Une fois que toutes les données sont reçues, la Gateway assemble les données et envoie à la *LoRaWAN* gateway.
Mode Proxy : Une fois que toutes les données sont reçues, la Gateway usurpe chaque isolated node et envoie les données à la *LoRaWAN* gateway. Pour finir par envoyé ces propres données.

3.4.2.3 Coté LoRaWAN Gateway

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

Chapitre 4

Simulation

4.1 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.

4.2 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 :

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

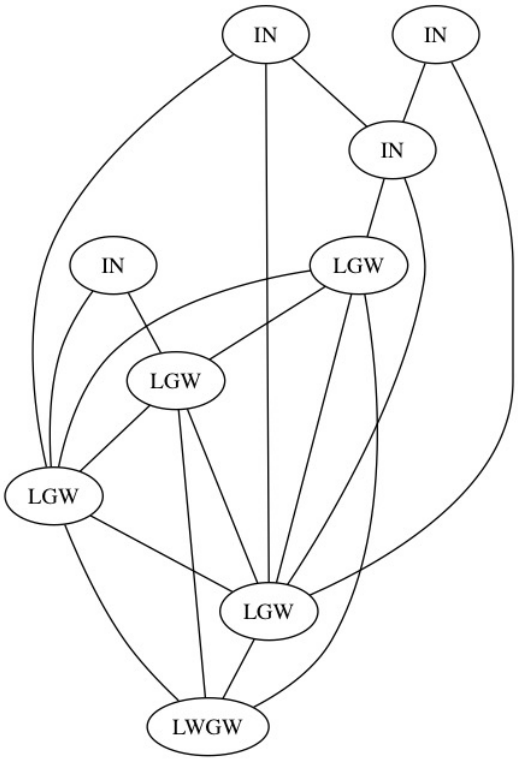

FIGURE 4.1 – Exemple de Graphe de sortie du script

4.3 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.

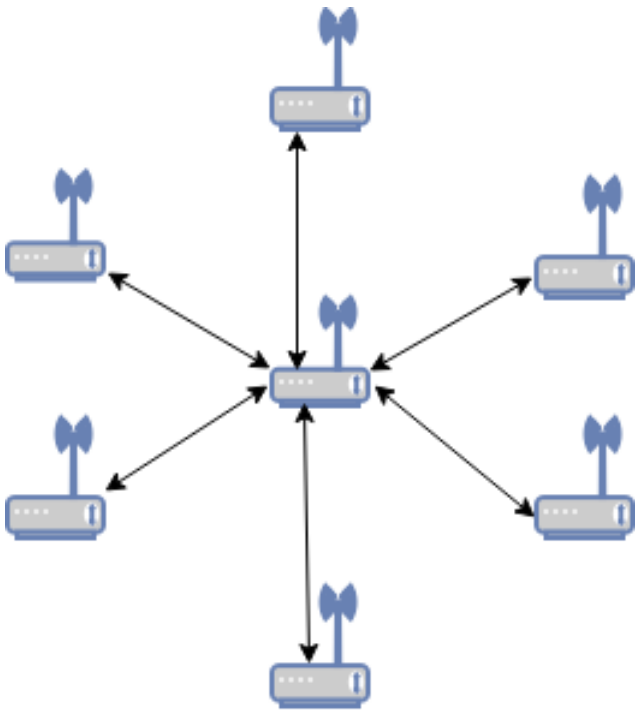

FIGURE 4.2 – Exemple de communication radio

4.4 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.

4.4.1 Simulation sur une chaine

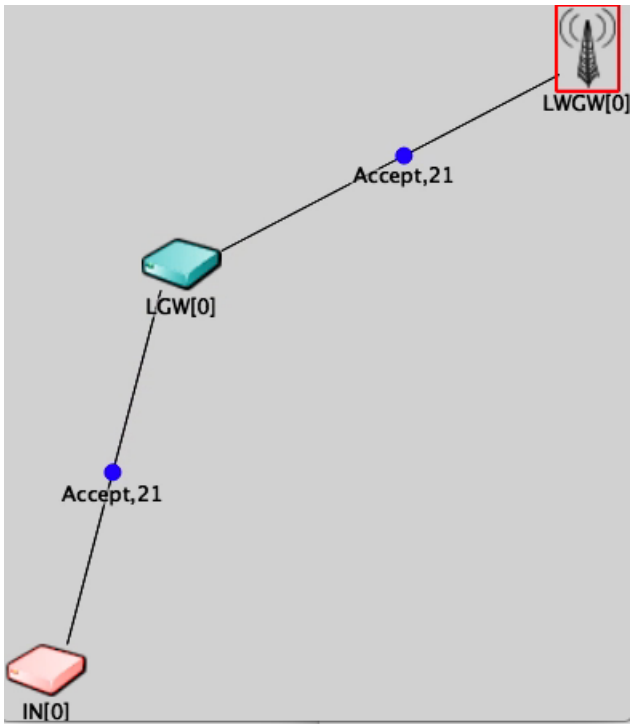

FIGURE 4.3 – Exemple de chaine

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.

4.4.2 Simulation avec 3 IN sur une LGW

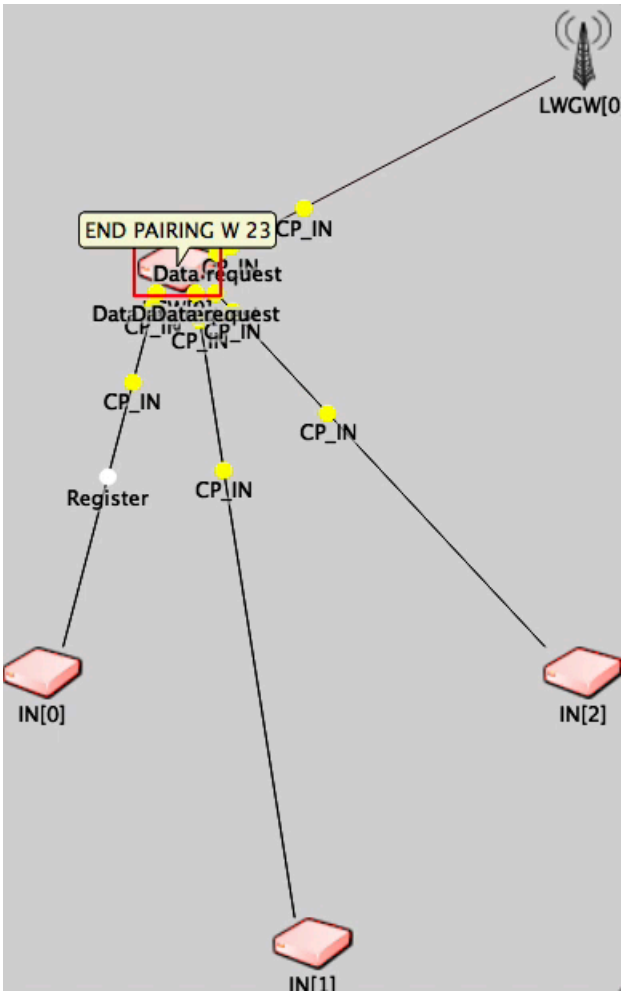

FIGURE 4.4 – Exemple avec 3 Isolated Nodes

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.

4.4.3 Simulation à plus grande echelle

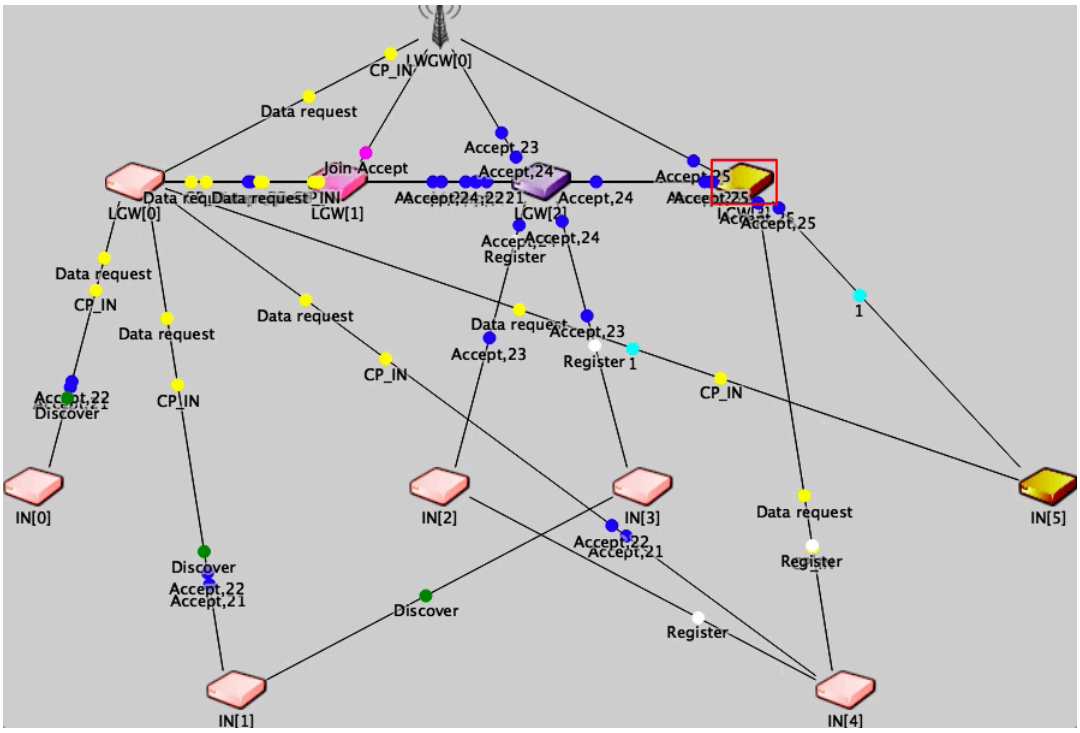

FIGURE 4.5 – Exemple à grande echelle

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 .

4.4.4 Générations des graphes

Un ensemble de 100 graphes contenant 1000 noeuds on été simulé en respectant cette configuration.

- Variation du nombre de nœuds isolés par passerelle entre 1 et 4.
- Variation du nombre de passerelle maximum par puits entre 1 et 4.
- Variation du nombre de données à récupérer par jour entre 1 et 24 messages.
- Variation sur la méthode de réassemblage des données collectées par les nœuds isolés (ie, envoyé au puits à chaque réception (méthode NA) ou réassemblé après fusion de tous les messages (méthode A))
- Variation de la vitesse de chaque lien entre deux nœuds.

Ce qui nous fait un total de 12800 graphes à créer. Pour réaliser cette création, nous avons écrit un algorithme basé sur la probabilité d'existence d'un lien entre deux nœuds. C'est-à-dire qu'au début nous créons le premier puits(LWGW). À celui-là regarde son degré (lequel est zéro au début) de la passerelle accrochée à lui. On tire une probabilité et si elle est supérieure à celle établie en entrée de l'algorithme on ajoute une passerelle(LGW) et le lien entre le puits et celui-ci. Nous faisons de même avec les nœuds isolés. Pour résumer, nous avions : cent graphiques avec un maximum d'un nœud isolé par passerelle et un maximum d'une passerelle par puits. Cette configuration sera abrégée 1 (IN) -1 (GW) $rightarrow$ 1-1.

Nous avons donc utilisé l'outil Omnet ++ pour simuler tous ces graphes et chaque cas. Dans ceux-ci nous avons noté pour le noeud isolé le nombre de messages envoyés ainsi que la batterie restante à la fin de la simulation. Pour la batterie, nous avons supposé qu'il avait 6600 mAh ou 23760000 mAs. L'énergie électrique

consommée provient de la documentation LoPy, section WiFi. Nos calculs de prévision sont réalisés comme suit :

L'appareil est considéré pour envoyer X msg (#msg) par jour (sur 24 heures). Le courant d'une émission est Ce = 107,3 mA. Chaque émission dure Te = 2s. Le courant en réception est Cr = 37mA. L'heure à la réception est Tr Le courant de veille est Cv = 0,531 mA. L'appareil est inactif pour Tv = (24 * 3600- #msg * Te-Tr * Cr) Le courant consommé sur une journée est donc : CDay (mA.s) = (#msg * Ceci * Te + Cr * Tr + Cv * Tv)

4.4.5 Résultats

Voici donc un graphique montrant la différence entre le mode d'agrégation et de non-agrégation au niveau du message. On peut remarquer que dans les figures suivantes un aspect linéaire ressort nettement sur la variation

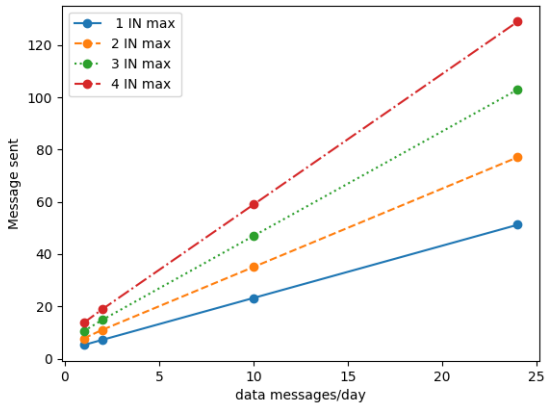

FIGURE 4.6 – Mode : Agrégation (Concentrateur)

du nombre de messages de la part d'une passerelle quel que soit le mode (avec agrégation ou sans agrégation). Ce

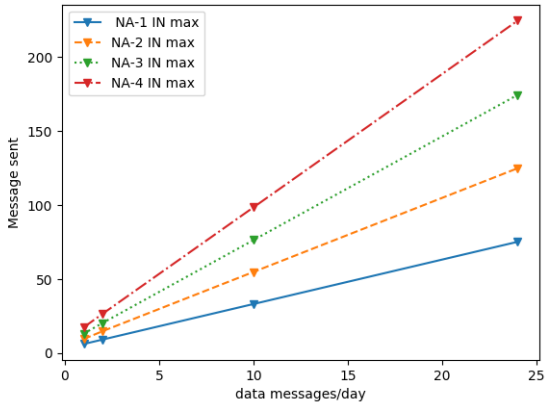

FIGURE 4.7 – Mode : Sans d'agrégation (PROXY)

que nous pouvons également remarquer est le nombre qui, comme prévu, est plus grand et croît beaucoup plus vite sans le mode d'agrégation (Fig. 4, 5, 6). Ce qui peut donc être considéré comme suffisamment significatif. En effet, l'ajout des données de la passerelle à celles des noeuds isolés fait que le nombre de messages en mode cluster est le même que si les IN étaient des GW et donc des noeuds connectés.

L'objectif était de vérifier si l'idée intuitive est correcte : l'architecture avec relais intermédiaire et agrégation renvoie, en nombre de messages et donc de durée de vie du réseau, au modèle sans relais, le modèle ou chaque client parle en direct avec des antennes toujours UP.

Les figures 7 à 10 sont plus complexes. En effet ces graphes représentent par leurs couleurs le nombre de Gateway maximum lié à un puits. Chaque point est un rappel de simulation décrit par le degré maximum de passerelles et de nœuds isolés ainsi que le nombre de messages demandés par le puits par jour. Ici, ces quatre figures sont en mode d'agrégation.

D'abord sur chaque graphique, chaque point est une moyenne du nombre de messages envoyés ou de l'état de la batterie le cas échéant, par un nœud isolé, ou par une passerelle si nécessaire. Pour chaque graphe, le bloc

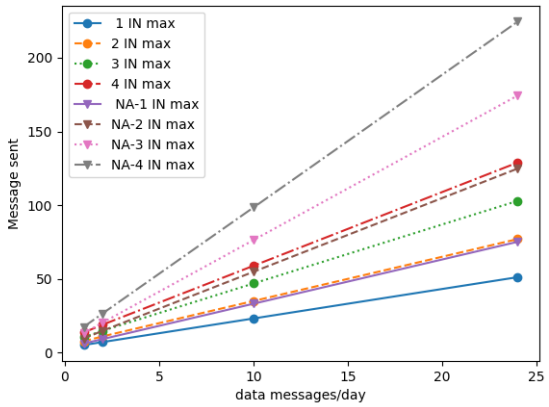

FIGURE 4.8 – Mode : Concentrateur vs Proxy

de 4 couleurs est donc l'ensemble des graphes de degré 1 IN. il faut comprendre que les simulations de 0 à 399 sont celles appelées "1-1, 1-2, 1-3, 1-4". de 400 à 799 "2-1, 2-2, 2-3, 2-4" etc. Nous avons réalisé ces simulations avec comme précédemment dit un nombre de message demandé par les différents puits. En effet nous avons choisi : 1, 2, 10, 24 messages par jour.

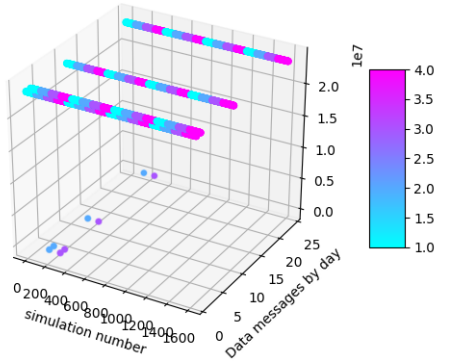

FIGURE 4.9 – Etat de la batterie sur un jour / message de données requis / jour pour une passerelle

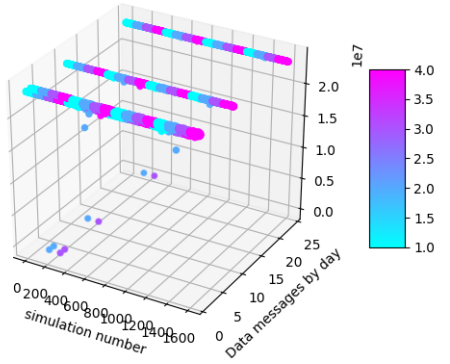

FIGURE 4.10 – Etat de la batterie sur un jour / message de données requis / jour pour un IN

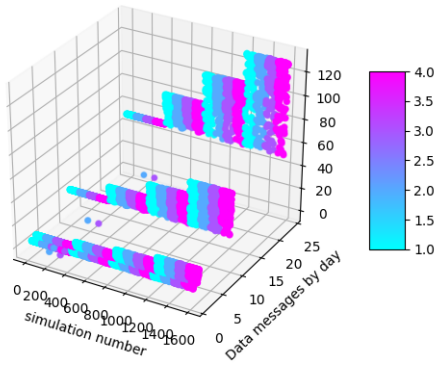

FIGURE 4.11 – Nombre de messages sur un jour / message de données

Chapitre 5

Programmation

Chapitre 6

Complément

Chapitre 7

Bilan

Annexes

Bibliographie

[1] Documentation Docker `https://docs.docker.com/`.