Mise en œuvre d'un collecte distribuée au moyen du développement d'une plate-forme de gestion pour des réseaux de capteurs sans fil

Thierry Duhal

Rapport de stage

Mise en œuvre d'une collecte distribuée au moyen du développement d'une plate-forme de gestion pour des réseaux de capteurs sans fil

Thierry DUHAL

Année 2013–2014

Stage de 2e année réalisé chez Inria

Maître de stage : Emmanuel NATAF

Encadrant universitaire : Martin QUINSON

Déclaration sur l'honneur de non-plagiat

Je soussigné(e),

Nom, prénom : DUHAL, Thierry

Élève-ingénieur(e) régulièrement inscrit(e) en 2e année à TELECOM Nancy

Numéro de carte de l'étudiant(e) : 31212261

Année universitaire : 2013–2014

Auteur(e) du document, mémoire, rapport ou code informatique intitulé :

Mise en œuvre d'une collecte distribuée au moyen du développement d'une plate-forme de gestion pour des réseaux de capteurs sans fil

Par la présente, je déclare m'être informé(e) sur les différentes formes de plagiat existantes et sur les techniques et normes de citation et référence.

Je déclare en outre que le travail rendu est un travail original, issu de ma réflexion personnelle, et qu'il a été rédigé entièrement par mes soins. J'affirme n'avoir ni contrefait, ni falsifié, ni copié tout ou partie de l'œuvre d'autrui, en particulier texte ou code informatique, dans le but de me l'accaparer.

Je certifie donc que toutes formulations, idées, recherches, raisonnements, analyses, programmes, schémas ou autre créations, figurant dans le document et empruntés à un tiers, sont clairement signalés comme tels, selon les usages en vigueur.

Je suis conscient(e) que le fait de ne pas citer une source ou de ne pas la citer clairement et complètement est constitutif de plagiat, que le plagiat est considéré comme une faute grave au sein de l'Université, et qu'en cas de manquement aux règles en la matière, j'encourrais des poursuites non seulement devant la commission de discipline de l'établissement mais également devant les tribunaux de la République Française.

Fait à Nancy, le 25 août 2014

Signature :

Remerciements

Tout au long de ce stage, j'ai été encadré par M. Emmanuel NATAF, que je tiens particulièrement à remercier pour son encadrement toujours positif, sa disponibilité, ainsi que ses conseils pertinents.

J'adresse également mes remerciements à M. Anthony DEROCHE, stagiaire dans l'équipe MADYNES, avec lequel j'ai travaillé auparavant en binôme lors du projet qui a mené à ce stage.

Je remercie enfin tous les membres de l'équipe MADYNES, et plus précisément M. Thibault CHOLEZ, maître de stage de M. DEROCHE, ainsi que M. Hubert NGANKAM KENFACK, stagiaire encadré par M. NATAF, pour les échanges riches que nous avons pu avoir.

Résumé

Dans le cadre de ma formation d'ingénieur à TELECOM Nancy, j'ai effectué mon stage de deuxième année chez Inria au laboratoire LORIA à Nancy. Ce stage de type « technicien » constitue ma première expérience en collaboration avec une équipe de recherche dans le domaine de l'informatique. S'étalant sur la période du 2 juin au 10 août 2014, ce stage fait suite à un projet de découverte de la recherche, que j'ai réalisé dans l'équipe MADYNES pendant le premier semestre 2014.

Durant de ce stage axé sur les réseaux de capteurs sans fil, j'ai travaillé sur l'acquisition des données captées depuis plusieurs points de collecte (au lieu d'un seul habituellement). J'ai également entrepris l'extension d'une plate-forme de collecte déjà existante, dans le but de superviser et contrôler un réseau de capteurs sans fil.

Le fait de travailler dans un laboratoire de recherche m'a permis de découvrir ce domaine concrètement, notamment en collaborant avec des chercheurs, des doctorants et d'autres stagiaires, mais aussi en assistant à des conférences, autre aspect du travail du chercheur.

Mots-clés : Réseaux de capteurs sans fil, supervision, collecte distribuée

Abstract

As a part of my studies at the engineering school TELECOM Nancy I performed an internship at Inria in the LORIA Laboratory in Nancy. This internship as a technician was my first experience in collaboration with a computer science research team.
This course set from the 2nd of June to the 10th of August, 2014 refers to a research exploratory project which I performed with the MADYNES team during the 1st semester of my school year.

During this course wireless sensor networks-oriented I worked on data collection from multiple sink - instead of a unique one traditionnally. I also undertook the expansion of an existing collection platform in order to monitor and control a wireless sensor network.

Working in a research laboratory allowed me to concretely understand this domain. I collaborated with researchers, PhD students and other interns and I also attended research conferences which represent another part of what the researchers do.

Keywords : Wireless sensor networks, monitoring, multi-sink

Sommaire

Remerciements ii

Résumé iii

Abstract iii

Sommaire iv

Introduction 1

1 Présentation de l'entreprise 2

1.1 L'employeur : Inria 2

1.2 Le laboratoire d'accueil : LORIA 2

1.3 L'équipe de recherche : MADYNES 3

2 Analyse du problème 4

2.1 Génèse du projet 4

2.1.1 Objectifs poursuivis 4

2.1.2 Première contribution 4

2.2 Collecte des données 4

2.3 Supervision et contrôle 5

3 Le réseau de capteurs sans fil 7

3.1 Matériel à disposition 7

3.2 Logiciels existants 7

3.2.1 Un système d'exploitation léger : Contiki 7

3.2.2 Un simulateur : Cooja 8

3.3 L'algorithme de routage RPL 8

4 Réalisation d'une collecte distribuée 11

4.1 Principe 11

4.2 Solutions envisageables 11
4.3 Mise en oeuvre 12
4.4 Résultats des expériences 13

5 Extension de la plate-forme vers la gestion de réseau 14
5.1 Supervision 14
5.1.1 Les données 14
5.1.2 La passerelle 14
5.1.3 L'API 15
5.2 Contrôle des nœuds à distance 16
5.2.1 Les commandes 16
5.2.2 Diffusion des commandes 16

Conclusion 19

Bibliographie / Webographie 20

Liste des illustrations 21

Annexes 23

A Organigramme d'Inria 23

B Organigramme du LORIA 24

C Le simulateur Cooja 25

D Résultats des expériences 26
D.1 Topologie du réseau 26
D.2 Taux de perte 28

E Exemple de code Java 29

F Commandes 30

G Diffusion des commandes 31

Introduction

De nos jours, la plupart des objets qui nous entourent ont la vocation d'être connectés. On pense notamment aux montres, voitures et lunettes, qui font partie de notre quotidien. Étant équipés d'une puce ou d'un capteur, ils se mettent ainsi à produire des données.

Dans ce contexte de « l'Internet des Objets » (Internet of Things), l'innovation est importante, et de nombreuses applications voient le jour. On peut citer les domaines d'applications tels que la santé (t-shirt connecté mesurant la respiration et le pouls), la planification urbaine (environnement urbain durable, éclairage public), le militaire (réseaux de drones aériens), la domotique (gestion du chauffage et de l'éclairage, alarmes de sécurité).

Ces nouvelles applications font souvent appel à des réseaux de capteurs sans fil. Le besoin de collecter et traiter les données captées se fait alors ressentir. Cependant, la fiabilité et la sécurité de ces réseaux n'est pas toujours garantie.

Mon travail durant ce stage s'est ainsi porté sur la gestion de ces réseaux de capteurs sans fil. Avant de détailler mes contributions, je présenterai l'environnement de travail dans lequel j'ai évolué, ainsi que le contexte matériel et logiciel de ce stage.

1. Présentation de l'entreprise

1.1 L'employeur : Inria

Créé en 1967, Inria (anciennement institut national de recherche en informatique et en automatique) est un établissement public à caractère scientifique et technologique (EPST). Cet organisme public de recherche est dédié aux sciences du numérique. Ses domaines de recherche sont au nombre de cinq :

- Santé, biologie et planète numériques
- Mathématiques appliquées, calcul et simulation
- Perception, cognition, interaction
- Réseaux, systèmes et services, calcul distribué
- Algorithmique, programmation, logiciels et architectures

Ces domaines regroupent au total 168 équipes-projets composées de 1677 chercheurs de l'institut ainsi que 1772 universitaires ou chercheurs d'autres organismes. Les équipes-projets d'Inria sont réparties dans huit centres de recherche implantés en France (Bordeaux, Grenoble, Lille, Nancy, Paris - Rocquencourt, Rennes, Saclay, Sophia).

C'est dans le centre de recherche situé à Nancy que j'ai effectué mon stage de deuxième année. J'ai notamment eu l'occasion d'assister à la « tournée d'adieux » du PDG d'Inria, Michel COSNARD, qui a souligné l'importance de la formulation : travailler *chez* Inria et non pas *à l'* Inria. L'accent est ainsi mis sur l'ensemble des personnes qui travaillent conjointement, plutôt que sur l'aspect matériel ou géographique.

1.2 Le laboratoire d'accueil : LORIA

Créé en 1997, le laboratoire lorrain de recherche en informatique et ses applications (LORIA) est une unité mixte de recherche associant Inria, le CNRS, ainsi que l'Université de Lorraine (UMR 7503). C'est aujourd'hui l'un des plus grands laboratoires en Lorraine. L'organigramme du LORIA est disponible en annexe B. Le laboratoire comporte au total 28 équipes de recherche. Cela représente plus de 500 personnes réparties en 5 départements :

1. Algorithmique, calcul, image et géométrie
2. Méthodes formelles
3. Réseaux, systèmes et services
4. Traitement automatique des langues et des connaissances
5. Systèmes complexes et intelligence artificielle

L'équipe de recherche MADYNES, dans laquelle j'ai effectué mon stage, se positionne dans le 3e département : Réseaux, systèmes et services. Ce département traite principalement des problématiques concernant la gestion et la coopération des réseaux modernes, l'informatique parallèle et distribuée, ainsi que les aspects « temps réel ». Les différentes équipes de ce département sont :

- ALGORILLE - Algorithmes pour la Grille
- MADYNES - Supervision des réseaux et des services dynamiques
- ORCHIDS - Recherche opérationnelle pour les systèmes de décision hybrides et complexes
- SCORE - Service et coopération

Afin de faire face ces problématiques, les équipes conçoivent des algorithmes, des protocoles, des logiciels, ainsi que des plates-formes de recherche permettant de mener diverses expérimentations.

En ce qui concerne l'ambiance de travail, les locaux du LORIA contiennent un certain nombre d'espaces de détente. Cela favorise les échanges entre membres d'équipes différentes. J'ai ainsi eu l'opportunité de partager mon travail et surtout découvrir celui des autres, en discutant avec des stagiaires et des doctorants des équipes telles que ORPAILLEUR, CASSIS ou encore SCORE.
Tout au long de mon stage, j'ai eu l'occasion d'assister à de nombreuses conférences, ce qui m'a fait découvrir d'autres sujets intéressants, tels que le protocole OpenFlow, le pishing (hammeçonnage), la gestion des masses de données (Big data). Ces conférences m'ont permis de ne pas rester cantonné à mon stage, j'ai ainsi pu apprécier la vie du LORIA dans sa globalité.

1.3 L'équipe de recherche : MADYNES

L'équipe de recherche dans laquelle j'ai effectué se stage se prénomme MADYNES (de l'acronyme anglais Management of Dynamic Networks and Services). Créée en 2002, cette équipe est principalement spécialisée dans le domaine des réseaux informatiques. Elle s'intéresse notamment à l'étude de nouvelles approches pour configurer, maintenir et sécuriser les protocoles de communications au niveau de l'Internet futur (l'Internet des Objets). Ses principaux axes de recherche sont :

— L'évaluation et la mise en oeuvre d'architectures de communication distribuées exploitant le modèle pair-à-pair, le routage applicatif, et de nouvelles approches pour la représentation de l'information de gestion
— La modélisation et le benchmarking des infrastructures de supervision
— La sécurité : nouveaux protocoles de distribution de clefs et infrastructures pour le respect de l'anonymat et la vie privée
— La mesure et l'analyse : l'instrumentation automatique, le paramétrage, le monitorage et les modèles de mesure de qualité de services de bout-en-bout

Plus récemment, les réseaux de capteurs sans fil font l'objet de nombreux travaux concernant la sécurité, la gestion et les projets de déploiement.

2. Analyse du problème

2.1 Génèse du projet

2.1.1 Objectifs poursuivis

Ce stage s'inscrit dans le contexte de recherche des réseaux de capteurs sans fil. En vue de réaliser des expériences pour des travaux de recherche, le besoin d'une plate-forme de gestion se fait ressentir. La gestion des nœuds sous-entend d'une part la supervision, c'est-à-dire la surveillance du bon fonctionnement des nœuds, et d'autre part le contrôle, ou plus précisément le fait d'agir à distance sur le comportement des nœuds du réseau. La plate-forme ainsi développée pourrait être utilisée pour de nombreuses applications (la détection d'attaques par exemple).

2.1.2 Première contribution

Le choix de ce stage a été grandement motivé par un projet de découverte de la recherche, que j'ai réalisé dans l'équipe MADYNES pendant le premier semestre 2014, en binôme avec M. Anthony DEROCHE. Lors de ce projet, nous avons développé en Java EE une plate-forme pour les réseaux de capteurs sans fil. Cette dernière permet la collecte des données via une passerelle (programme Java) qui fait le lien entre le réseau de capteurs et l'application serveur (nous utilisons GlassFish 4 comme serveur d'applications). Les données ainsi collectées sont stockées dans une base de données relationnelle. Au moyen d'une API, nous récupérons ces données afin de les afficher sur différentes interfaces graphiques. Nous pouvons suivre en temps réel l'évolution des données (la température, la luminosité) grâce à une WebSocket. Nous pouvons aussi afficher la topologie du réseau sur une carte de l'endroit où le déploiement a lieu. Nous utilisons le framework CSS Zurb Foundation [1] pour un visuel agréable et confortable pour l'utilisateur. Axée sur la collecte et l'affichage des données, cette plate-forme ne permet pas de faire de la gestion. Néanmoins elle constitue le socle de mon travail pour ce stage.

2.2 Collecte des données

Lors d'un déploiement, les motes sont répartis de manière à assurer la couverture complète d'une une zone prédéfinie. Par « mote » j'entends un « nœud capteur » du réseau. Afin de collecter les données captées par les nœuds ainsi dispersés, il est nécessaire de connecter un mote sur une machine disposant d'un port USB, on le nomme alors « sink » dans la mesure où il joue le rôle de « puit », c'est-à-dire qu'il est chargé de récupérer toutes les données envoyées par les autres nœuds du réseau (senders). Dans le cas d'un déploiement à grande échelle, ces derniers peuvent être

relativement éloignés du sink, et ne peuvent donc pas lui envoyer directement leurs données. Chaque nœud doit donc jouer à la fois le rôle d'émetteur et de récepteur, en ce sens qu'il peut être amené à relayer les paquets de ses voisins (tous les motes se situant dans sa portée de communication). Certains paquets doivent donc faire plusieurs sauts avant d'arriver au sink, comme illustré sur la figure 2.1 ci-dessous.

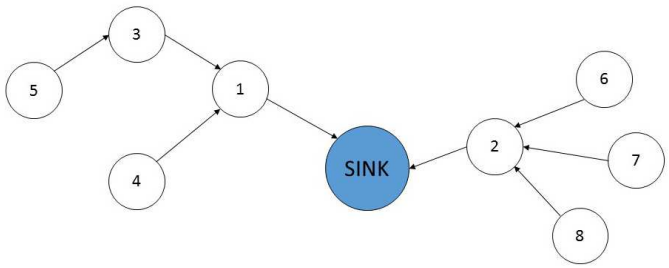


FIGURE 2.1 – Réseau comportant du multi-saut

Cependant cette topologie pose certains problèmes. Tout d'abord, les nœuds voisins du sink subissent toute la charge du réseau, dans le sens où ils relayent les paquets de tous les autres nœuds. C'est le cas des nœuds 1 et 2 sur la figure 2.1. Or au niveau du mote, ce qui lui consomme le plus d'énergie est le module radio. Ces nœuds voisins du sink, relativement peu nombreux, voient leur batterie d'épuiser plus rapidement. Ces « trous énergétiques » ont pour effet de réduire la durée de vie du réseau.

Deuxièmement, les paquets envoyés depuis des nœuds éloignés sont d'autant plus sujets à des pertes (notamment dus à des brouillages lors des communications). Lors d'un déploiement réalisé auparavant à TELECOM Nancy, nous avons constaté des taux de perte de paquets en moyenne deux fois plus important pour des nœuds très éloignés du sink.

Enfin, le fait de disposer d'un unique sink fait de lui un point critique en terme de défaillance et de sécurité. Afin de pallier à ses problèmes, une bonne solution consiste à instaurer plusieurs points de collecte (multi-sink). Cette répartition de la collecte apporte de la fiabilité dans la mesure où les responsabilités sont distribuées. La mise en œuvre de cette solution sera détaillée par la suite.

2.3 Supervision et contrôle

En terme de supervision, nous avons besoin que les nœuds nous envoyent des informations relatives au réseau. Ces informations permettraient par exemple d'observer le changement de la topologie, ou encore calculer le taux de perte de paquets. Pour ce faire, il est pratique d'instaurer des flux de supervision, séparé du flux de données classique des capteurs. Cela permet de gérer différemment ces flux, et nécessite donc d'adapter la plate-forme de collecte existante.

Jusqu'à présent, tous les flux d'informations circulent vers le haut, c'est-à-dire des nœuds vers le sink, et du sink vers l'application serveur grâce à la passerelle. Après avoir fait cette constatation, la volonté d'envoyer des informations aux nœuds m'est venue. Envoyer des informations qui « redescendent » permettraient de contrôler à distance le comportement des nœuds, et cela à des fins diverses. Cependant, cela nécessite l'extension de la plate-forme actuelle en une plate-forme de gestion. Je détaillerai cette partie de mon travail après avoir présenté le contexte matériel et logiciel de ce stage.

3. Le réseau de capteurs sans fil

3.1 Matériel à disposition

FIGURE 3.1 – Un nœud du réseau

Conçus par l'université de Californie à Berkeley, ce « mote » est de type TelosB et se prénomme MTM-CM5000-MSP [2]. Il est principalement composé d'un microcontrôleur MSP430, d'une puce radio TI-CC2420 (permettant l'émission et la réception), d'un capteur de température et d'humidité, d'un capteur de luminosité pour le domaine du visible, d'un autre pour l'infrarouge, d'un bouton paramètrable, d'un bouton reset, de trois LEDs, d'un port USB. Pour fonctionner, un mote a besoin de deux piles de 1,5V, sa durée de vie est donc limitée lors d'un déploiement. En outre, il possède 10kB de mémoire vive et seulement 48kB de mémoire flash, ce qui peut s'avérer gênant, nous en reparlerons.

A noter que dans le réseau de capteurs sans-fil que nous souhaitons déployer, chaque mote communique avec ses voisins via le protocole de communication 802.15.4 sur la bande de fréquence 2,4 GHz.

3.2 Logiciels existants

3.2.1 Un système d'exploitation léger : Contiki

Créé en 2002 par Adam Dunkels, membre d'une équipe du centre suédois de recherche scientifique (Swedish Institute of Computer Science), le système d'exploitation Contiki est écrit en langage C et est disponible gratuitement sous license BSD, c'est à dire qu'il peut être utilisé et modifié sans restriction. Étant conçu spécialement ce type de petits appareils en réseau, Contiki a l'avantage d'être léger. Il se charge d'allouer les ressources nécessaires à l'exécution du programme « flashé »

sur un mote. Étant écrit en langage C, Contiki se veut être très portable. En effet il peut être embarqué sur de nombreuses plates-formes telles que wismote, z1, avr-raven, ou encore sky (qui correspond aux motes que nous utilisons ici). La dernière version de Contiki est disponible sur le dépôt suivant [3].

Par ailleurs, Contiki permet de « simuler » du multi-threading grâce au concept des protothreads [4]. Afin de permettre l'exécution de plusieurs tâches « en parallèle », les protothreads peuvent se mettre en attente d'un évènement particulier (paquet reçu, compte à rebours expiré). Lorsque l'évènement arrive, ils reprennent leur exécution. Au niveau mémoire, cette solution se veut être légère dans la mesure où les protothreads partagent le même contexte d'exécution, contrairement au multi-threading.

3.2.2 Un simulateur : Cooja

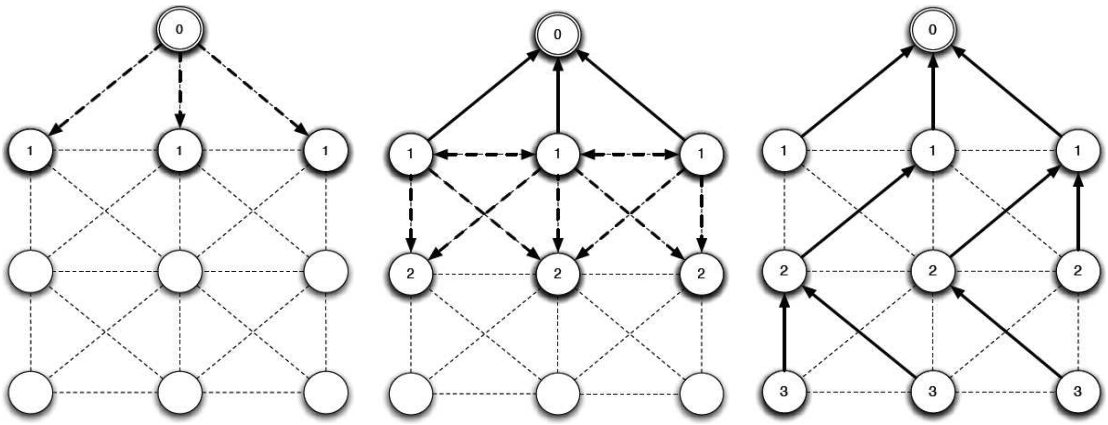
Lors de ce stage, un outil m'a été particulièrement utile, il s'agit du simulateur Cooja [5]. Ce dernier est fourni avec Contiki. Il permet de simuler un réseau de motes virtuels. On peut alors tester rapidement un code écrit en langage C, sans avoir besoin de flasher de vrais motes, ce qui est pratique pour déboguer du code. Grâce à ce simulateur, il est possible de répartir un nombre quelconque de nœuds sur une zone donnée. On visualise alors en temps réel (ou accéléré) la topologie du réseau, ainsi que les routes empruntées lors des transitions de paquets. Une capture d'écran de l'interface de Cooja est visible en annexe C.

3.3 L'algorithme de routage RPL

Rappelons que dans un réseau de capteurs sans fil, les nœuds du réseau ne sont pas forcément en mesure de communiquer directement avec le sink. Chaque paquet émis doit pouvoir passer par un chemin pour arriver jusqu'au sink. Le protocole de routage RPL (Routing Protocol for Low power and Lossy Networks) résout cette problématique [6]. ContikiRPL est l'implémentation dans Contiki du standard RPL, défini par la RFC 6550 [7]. L'algorithme RPL permet l'établissement d'une route entre un nœud émetteur et le sink. Il assure également une topologie dynamique du réseau, dans le cas où un mote est ajouté, déplacé, retiré, ou défaillant.

Du point de vue de son fonctionnement, RPL construit un DODAG (de l'anglais Destination Oriented Directed Acyclic Graph), c'est-à-dire un graphe de nœuds orienté et acyclique. Orienté dans le sens où chaque nœud envoie ses paquets vers le sink, et acyclique en ce sens que RPL garantit l'absence de boucles au sein du graphe. A titre indicatif, il est possible d'avoir plusieurs instances RPL au dessus du même réseau physique. Une instance RPL peut contenir plusieurs DODAG. Cependant, un nœud ne peut appartenir qu'à un seul DODAG au sein d'une même instance.

Afin de construire ce graphe, le sink commence par envoyer à tous ses voisins (les motes se situant dans sa portée) des messages appelés DIO (DODAG Information Object), montré par le schéma 3.2(a). Un message DIO contient tout un ensemble d'informations tel que l'identifiant de l'instance RPL, l'identifiant du DODAG, le numéro de version actuel du DODAG, le rang du nœud qui diffuse le message. Il est utile de préciser que chaque nœud possède un rang dans le graphe, qui représente son éloignement par rapport au sink. Plus le nœud est éloigné du sink, plus son rang est élevé. Lors de la réception d'un DIO, un nœud va se servir de rang pour choisir son parent, c'est-à-dire le nœud par lequel ses paquets seront envoyés jusqu'au sink. Il ne pourra jamais choisir pour parent un autre nœud ayant un rang plus élevé que le sien. C'est ce qui garantit l'absence de boucles.



(a) Étape 1 (b) Étape 2 (c) Étape 3

FIGURE 3.2 – Construction du DODAG

Au moment où un nœud reçoit des DIO, il choisit le meilleur parent parmi un ensemble de parents candidats. Le choix du meilleur parent peut se faire selon différentes métriques (nombre de sauts minimum, qualité de la communication, batterie restante). Il se met ensuite à renvoyer des DIO à tous ses voisins, comme le montre le schéma 3.2(b).

Lorsque les messages DIO ont été reçus par tous les nœuds, le graphe de la figure 3.2(c) est construit. Chaque nœud ayant choisi son parent « préféré », il existe donc une route allant de n'importe quelle source vers le sink.

Afin de maintenir le DODAG, les nœuds s'envoient des messages DIO de manière périodique. La transmission de ces messages est régulée par l'algorithme « Trickle » [8]. Au départ, chaque nœud peut émettre dans un intervalle de temps initial I. Lorsque l'intervalle I est expiré, l'algorithme Trickle double cet intervalle tant qu'aucun problème n'est détecté. L'intervalle augmentant de façon exponentielle, ce mécanisme permet de limiter les envois et les répétitions de messages lorsque la topologie du réseau converge. Cependant, si un nœud « perd » son parent, ou bien détecte un changement de numéro de version du DODAG, l'intervalle I est remis à sa valeur initiale. Les nœuds se remettent alors à échanger des DIO rapidement pour reconstruire le DODAG.

Lorsque des nœuds disparaissent du réseau (défaillance, batterie épuisée), la topologie de ce dernier peut ne pas s'adapter correctement : certains nœuds n'empruntent pas un chemin optimal jusqu'au sink par exemple. Dans ce cas, RPL fournit un mécanisme de « réparation globale » du réseau (global repair). Le sink commence par incrémenter le numéro de version du DODAG présent lors de l'envoi de DIO. Lorsque les nœuds reçoivent des DIO, ils acceptent uniquement les DIO dont le numéro de version est supérieur ou égal au numéro courant. Ce numéro de version garantit que les informations circulant dans le réseau sont bien à jour, dans la mesure où les anciens DIO se font « écraser » par les plus récents. Par la suite, chaque nœud effectue un « local repair », c'est-à-dire qu'il efface son parent de sa table de routage, et qu'il se donne un rang infini afin de ne pas être choisi en tant que parent (il « empoisonne » ses fils). La reconstruction du graphe se fait alors naturellement, de la même manière que la première fois.

4. Réalisation d'une collecte distribuée

4.1 Principe

Jusqu'à présent, nous avons seulement considéré des réseaux ne comportant qu'un seul sink. Cependant, nous avons évoqué précédemment les problèmes liés à ce type de topologie. En instaurant plusieurs points de collecte, le réseau est partitionné. Par conséquent, le nombre de sauts depuis un nœud jusqu'à un sink est en moyenne réduit au sein du réseau (en supposant une répartition correcte des sinks sur une zone donnée). Chaque nœud se trouvant relativement proche d'un sink en moyenne, leur consommation énergétique est réduite dans la mesure où ils relayent peu de paquets [9]. En outre, nous étudierons l'impact d'une collecte distribuée sur les taux de perte au moyen d'une expérimentation réelle.

4.2 Solutions envisageables

Du point de vue de l'algorithme de routage RPL, il est intéressant de remarquer qu'il n'y a pas de duplication des paquets envoyés par un nœud en particulier. En effet, ce dernier envoie ses données uniquement à son parent, et il ne peut avoir qu'un seul parent au sein d'une même instance RPL. Afin de mettre en place cette collecte distribuée, il existe différentes solutions.

Plusieurs DODAG

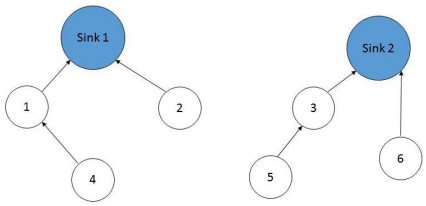


FIGURE 4.1 – Exemple de réseau constitué de 2 DODAG

La solution la plus naturelle consiste à utiliser le fait qu'une instance RPL peut contenir plusieurs DODAG. La racine de chaque DODAG est alors un sink, comme le montre la figure 4.2. Chaque nœud enregistre les différentes structures de DODAG dans un tableau dont la taille est définie lors de la compilation. Cela lui permet d'appeler des fonctions qui comparent les différents DODAG, afin de choisir le meilleur. Le choix du parent préféré se fait une fois le DODAG rejoint. Cette solution n'est pas évolutive dans la mesure où l'on doit connaître le nombre de sinks à l'avance, ce qui définit la taille du tableau précédent. Il y a autant de DODAG que de sinks. Cette solution s'avère être coûteuse en matière de calculs et de mémoire.

Sink virtuel

Contrairement à la solution précédente, le principe ici est de n'utiliser qu'une seule instance RPL et qu'un seul DODAG. Pour ce faire, nous allons utiliser la notion de « sink virtuel » [10], citée au paragraphe 3.1.3 de la RFC6550. Le sink virtuel joue le rôle de racine du DODAG.

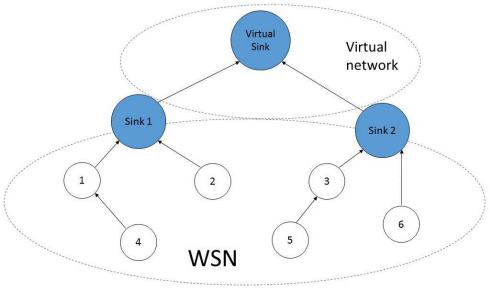


FIGURE 4.2 – Illustration du sink virtuel au sein d'un réseau multi-sink

Pour ce faire, tous les sinks partagent la même adresse IPv6, qui est aussi celle du sink virtuel. Les nœuds envoyent ainsi leurs données en anycast, c'est-à-dire qu'il existe un ensemble de récepteurs finaux (les sinks), mais un seul est choisi en fonction de l'algorithme de routage RPL. L'ensemble des sinks doit être synchronisé (« dire » la même chose par l'envoi de DIO), c'est ce qui simule le sink virtuel. Les nœuds du réseau sont ainsi bernés, ils n'ont pas la connaissance d'un réseaux composé de plusieurs sinks. Les différents sinks sont vus comme des fils du sink virtuel, cependant ils ne transmettent pas d'informations à ce sink virtuel, ce dernier n'existant pas en réalité.

Cette solution a l'avantage d'être évolutive, on peut facilement ajouter un sink dans le réseau. Au niveau mémoire flash, cette solution est moins coûteuse que la précédente. En effet, chaque nœud ne stocke qu'une seule structure de DODAG, et appelle uniquement les fonctions permettant de choisir son meilleur parent à l'intérieur du graphe. J'ai choisi de mettre en œuvre cette solution.

4.3 Mise en oeuvre

L'intégration du multi-sink se fait de manière transparente pour notre plateforme. En effet, chaque sink est branché sur le port USB d'une machine qui fait tourner notre passerelle. Cette dernière lit sur le port USB, récupère les données du sink, et transmet ensuite les données collectées au serveur par l'intermédiaire d'une requête HTTP (POST). L'application serveur supporte l'existence de plusieurs passerelles connectées. Les données sont regroupées dans la base de données.

En revanche, on rencontre un problème au niveau du « global repair ». En effet, lorsque qu'on initie ce mécanisme depuis un seul sink (on peut appeler une fonction depuis l'appui d'un bouton physique sur le mote), les autres sinks reçoivent également les DIO du premier sink. Ces DIO contiennent un numéro de version de DODAG plus récent. Dans l'implémentation ContikiRPL, un sink recevant ce type

de DIO se met à ré-incrémenter le numéro de version, et attire ainsi tous les nœuds du réseau. De ce fait, les numéros de versions sont constamment incrémentés. On boucle ainsi de manière infinie, chaque sink se bat pour "récupérer" le réseau, et la topologie du réseau ne converge jamais.

Le changement que j'ai apporté ici consiste à accepter le numéro de version lorsqu'un sink reçoit un DIO avec un numéro plus récent. Lorsqu'on initie un global repair sur un seul sink, tous les autres sinks vont accepter le numéro de version, et le réseau sera homogène. Aucun sink ne sera « délaissé » du fait d'un numéro de version périmé. Cependant cette solution n'est pas la meilleure, puisqu'il serait préférable de déclencher le global repair depuis le sink virtuel, c'est-à-dire incrémenter le numéro de version du DODAG sur chaque sink. Cette synchronisation des sinks sera apportée par l'application serveur, je détaillerai cette partie de mon travail dans la partie suivante.

4.4 Résultats des expériences

Étudions l'impact du multi-sink sur les taux de perte. Nous considérons ici les paquets de données, contenant entre autres la température, la luminosité, l'humidité, et notamment un numéro de séquence associé au paquet. Tout d'abord, il est nécessaire de calculer les taux de perte de paquets pour chaque nœud. Les numéros de séquence manquants correspondent à des paquets perdus. Il suffit de les sommer et de diviser par le nombre de paquets théoriquement envoyés. De même, pour calculer le taux de perte global du réseau, je somme tous les paquets perdus que je divise par la somme de tous les paquets théoriquement envoyés par les senders.

J'ai réalisé deux expériences au sein du LORIA. Ces deux expériences ont duré une journée chacune. Elles se sont déroulées dans les mêmes conditions (placement identique des nœuds, envois de paquets se faisant toutes les 2 minutes), à la seule différence que la première ne comporte qu'un sink, tandis que la deuxième est faîte avec deux sinks. Dans le cas d'un unique sink, on remarque sur l'annexe D.1 que les paquets des motes les plus éloignés doivent en général faire 3 sauts pour arriver jusqu'au sink. Dans le cas du multi-sink, la majorité de ces derniers sont à 1 voir 2 sauts d'un sink (cf. annexe D.2). On remarque que la topologie est différente, on distingue ici clairement les deux « sous-arbres » qui se sont créés. J'ai utilisé le framework KineticJS [11] pour représenter les sinks sous forme d'étoiles.

Du point de vue des taux de perte, je les affiche sous forme d'histogramme grâce à la librairie JavaScript Highcharts [12]. Les Taux de perte sont visibles en annexe. On constate un taux de perte global de l'ordre de 10% pour du mono-sink, et 4% pour du multi-sink. Bien évidemment ces taux de perte sont dépendants du placement des sinks, ce qui sort du cadre de mon étude. Il est intéressant de remarquer que, dans les deux expériences, les motes ayant les taux de perte les plus élevés sont les motes qui sont les plus éloignés d'un sink. Le multi-sink a donc un impact positif sur les taux de perte de paquets. Par ailleurs, ces déploiements se sont faits sur un seul étage. En tirant avantage du multi-sink, il est désormais possible d'effectuer des déploiements sur plusieurs étages, de manière à couvrir tout un bâtiment.

5. Extension de la plate-forme vers la gestion de réseau

En vue de répondre aux problématiques énoncées dans la partie 2.3, j'ai contribué à l'extension de la plate-forme.

5.1 Supervision

5.1.1 Les données

Selon les applications souhaitées, le besoin de données précises provenant des motes se fait ressentir. On peut citer par exemple :

— l'énergie restante, afin de détecter une batterie bientôt épuisée
— l'ensemble des voisins de chaque nœud, pour détecter certaines attaques
— le RSSI (Received Signal Strength Indication), pour faire de la géolocalisation de motes

La réponse à ce besoin est de définir des types de paquets. À l'échelle des motes, cela se traduit par l'ajout d'un entier en tête de paquet. À titre indicatif, le flux de données classique correspond au type 0, le paquet de géolocalisation au type 10, nous pourrions également avoir un flux de données de sécurité de type 20. Les écarts d'entiers sont prévus pour d'éventuelles applications futures. À titre d'exemple, j'ai ajouté un type 31 qui contient l'identifiant du sink (qui correspond à son adresse MAC convertie en entier 16 bits), ainsi que le numéro de version du DODAG actuel. Cela me permet d'enregistrer les sinks dans la table Mote de la base de données et de les afficher ensuite sur une carte.

Par ailleurs, du point de vue du code C flashé dans un mote, nous avons accordé une attention particulière vis-à-vis de la propreté et l'évolutivité du code. En effet, tout le code permettant de gérer un type de paquet est regroupé sous la forme d'un module. Les modules sont indépendants les uns des autres. Il est ainsi aisé d'ajouter, de modifier, ou de supprimer un module précis. En outre, les types peuvent être gérés depuis l'interface admin. Il est nécessaire de préciser le nombre minimum de données que le flux doit contenir. Cette contrainte permet d'éviter des cas d'erreurs lors du traitement du flux.

5.1.2 La passerelle

Rappelons que nous disposons d'un programme Java qui fait office de passerelle entre le sink et l'application serveur. Jusqu'à présent, cette passerelle ne gère que le flux de données classique (type 0). Elle récupère les données du sink arrivant sur le port USB, et les transmet ensuite au serveur par l'intermédiaire d'une requête

POST (HTTP) sur une URL bien précise. Ces données sont ensuite traitées, stockées, et éventuellement affichées.

Désormais, nous disposons de flux différents, que nous pouvons qualifier de flux de supervision. Afin de prendre en compte ces changements, j'ai été amené à modifier la passerelle. Cette dernière poste dorénavant les données sur une URL différente en fonction du type du paquet. Par l'intermédiaire d'un fichier de configuration, elle connaît l'association type - URL. Ce fichier est facilement modifiable et aucun changement au niveau du code n'est nécessaire. Il est également possible d'envoyer des données différentes sur des serveurs distincts. La passerelle joue en quelque sorte le rôle d'un switch, dans la mesure où elle aiguille les paquets vers des destinations différentes en fonction de leur en-tête.

5.1.3 L'API

Lors du projet antérieur à ce stage, nous avions décidé de mettre à disposition les données envoyées par les noeuds du réseau à travers une API REST (REpresentational State Transfer), ce qui permettrait à de futurs développeurs de créer des applications graphiques utilisant ces données. Notre API (Application Programming Interface) est un service web permettant l'accès aux ressources depuis une interface cliente. C'est la façade qui fait le lien entre le front-end (interfaces graphiques) et le back-end (serveur, base de données). L'API permet de récupérer toutes les données et informations souhaitées par le biais de requêtes sur différentes URL, et renvoie des réponses de type JSON (JavaScript Object Notation).

Afin d'assurer la gestion des motes, j'ai été amené à étendre l'API. Pour ce faire j'ai ajouté certaines URL, derrière lesquelles des méthodes Java sont appelées. A titre d'exemple, le code Java permettant de supprimer un mote de la base de données est disponible en annexe E. Cette fonction est appelée lorsqu'on réalise une requête POST sur l'URL de la forme http ://domainName/iot_lab/rest/info/motes/remove. Les données associées au mote sont supprimées en même temps que celui-ci. Dans le but de faciliter cette gestion, suppression de motes, j'ai ajouté une interface graphique dans la partie administration (nécessite d'avoir un compte admin, les utilisateurs « normaux » n'y ont pas accès). Celle-ci est visible en annexe F.1.

5.2 Contrôle des nœuds à distance

5.2.1 Les commandes

L'objectif que je me suis fixé ici est de pouvoir modifier le comportement des nœuds du réseau, sans avoir à intervenir physiquement sur les motes. Dans ce but, j'ai mis en place toute une architecture que je vais vous présenter.

Tout d'abord, le contrôle des nœuds s'effectue par l'envoi de commandes. Ces commandes permettent de réaliser des actions diverses. J'ai implémenté les principales fonctionnalités, correspondants aux besoins actuels. Il est tout à fait possible d'en ajouter. La liste des commandes disponibles est visible sur l'Interface web de l'envoi de commandes. En voici le détail :

- START ALL : tous les modules (geolocation, data) sont démarrés dans chaque sender. Il est également possible de démarrer uniquement certains modules (par défaut, tous les modules sont lancés).
- STOP ALL : arrête tous les modules, c'est-à-dire que les flux de données de ce module ne sont plus envoyés depuis les senders. De la même manière que pour le START, il est possible d'arrêter seulement un module (STOP GEOLOC par exemple).
- GLOBAL REPAIR : effectue un global repair sur tous les sinks connectés.
- SINKS INFO : permet de récupérer l'identifiant de tous les sinks connectés, ainsi que leur numéro de version du DODAG. Cette commande est envoyée par le serveur dès qu'une passerelle se connecte. Les sinks sont ainsi directement enregistrés dans la base de données.
- DATA_PERIOD T : permet de changer la période d'envoi des données de type 0, où T est la fréquence d'émission en secondes, comprise entre 1 et 65535.
- TX_POWER P : permet de changer la puissance d'émission d'un nœud (et donc sa portée) avec P entier inclus entre 1 et 31. Cette commande peut être utile pour simuler un grand nombre de sauts sur une zone restreinte.

5.2.2 Diffusion des commandes

L'architecture globale de la diffusion des commandes est visible en annexe G.

Du client vers les sinks

Partons du début. Un client (administrateur) se rend sur l'interface de gestion du réseau avec son navigateur web et veut agir sur le comportement des nœuds du réseau. Supposons qu'il veuille arrêter ses senders pendant la nuit, afin d'économiser leur consommation énergétique. Il clique alors sur la commande STOP, en choisissant ALL dans la liste déroulante. Si aucune passerelle (aucun sink) n'est actuellement connectée, le client reçoit un message d'échec. Dans le cas contraire, on l'informe que sa commande a bien été transmise.

Un gestionnaire de commande, présent sur le serveur, va traiter sa commande. Cette dernière est transformée en JSON et envoyée à toutes les passerelles connectées. Cela permet la synchronisation dont nous avions besoin en ce qui concerne les sinks. La communication entre une passerelle et le serveur se fait via le protocole WebSocket [13].

WebSocket est un standard du web et un protocole de la couche application, apparu avec HTML5 et standardisé par l'IETF (Internet Engineering Task Force) dans la RFC 6455 en 2011. Il vise à construire un canal bidirectionnel full-duplex entre un serveur web et un client au dessus du protocole de transport TCP (Transmission Control Protocol). Dans notre cas, cela permet l'établissement d'une connexion permanente entre la passerelle et le serveur. Il est important de préciser que c'est la passerelle qui initie le processus de connexion sur le serveur (handshake) et non pas l'inverse, car la machine qui fait tourner la passerelle peut être protégée derrière un pare-feu. Le serveur gère le protocole WebSocket grâce à l'implémentation intégrée dans Java EE 7.

Lorsqu'une passerelle reçoit la commande, elle doit l'envoyer au sink. J'ai dû adapter la passerelle, de sorte qu'elle puisse communiquer avec le sink, mais « dans l'autre sens » cette fois-ci. Cela est rendu possible en redirigeant la sortie standard vers le port USB auquel est connecté le sink.

Des sinks vers les nœuds

Du côté du sink, ce dernier est perpétuellement en attente d'évènements provenant du port USB. Lorsqu'une commande arrive, il la « parse » et vérifie qu'elle est conforme avant d'effectuer une quelconque action. La diffusion de certaines commandes, telles que SINKS INFO ou GLOBAL REPAIR, s'arrête au niveau des sinks. Pour les autres commandes, elles doivent être transmises à tous les nœuds du réseau.

Atteindre tous les nœuds n'est pas chose facile, dans la mesure où l'on cherche à envoyer des informations vers le « bas ». Les nœuds ne communiquant en unicast qu'avec leur parent, on ne peut pas tirer profit du DODAG créé par l'algorithme de routage RPL. De même, faire du broadcast ici ne permet pas d'atteindre tous les nœuds. En effet, chaque nœud se comportant comme un routeur, il délimite les domaines de broadcast. Une fois le message reçu, il ne le transmet pas à ses voisins (expérimentation réalisée sous Cooja).

En revanche, le multicast peut être une solution. Le principe est que tous les nœuds du réseau rejoignent le même groupe multicast (adresse multicast IPv6 commune). Le multicast permettant de faire du multi-saut, c'est-à-dire le fait qu'un nœud transmette un paquet à ses voisins après l'avoir reçu, a été implémenté dans Contiki [14]. Cependant, utiliser ce multicast nécessite d'inclure un certain nombre de modules présents dans Contiki. Or la mémoire flash étant limité, il m'a été impossible de flasher un mote en incluant ces modules. J'ai alors adapté le multicast en le modifiant légèrement.

Lorsque le sink veut transmettre une commande, il la transmet à tous ses voisins. Une fois la commande reçue, un nœud effectue les actions nécessaires en appelant diverses fonctions : arrêt d'un module, changement de la fréquence d'émission des données, etc. Il retransmet ensuite la commande à tous ses voisins. Ce mécanisme de répétitions finit toujours par s'arrêter, dans la mesure où un compteur limite le nombre de répétitions qu'un nœud peut effectuer pour une même commande. Nous ne sommes pas assurés de la bonne réception de la commande par tous les nœuds du réseau, mais nous pouvons grandement l'espérer grâce à ce mécanisme de répétitions qui s'inspire de l'algorithme Trickle.

Afin de différencier les commandes, j'ai ajouté un numéro de version de commande en en-tête de chaque commande, que j'abrègerai « cvn » (de l'anglais command version number). L'utilité de ce cvn est de ne pas « inonder » le réseau dans le cas où plusieurs commandes se succèdent très rapidement. Ce cas de figure peut se produire lorsque plusieurs clients envoient des commandes en même temps, qui peuvent même être contradictoires (START et STOP).

De façon analogue au « dvn » (DODAG version number), lorsqu'un nœud reçoit un cvn plus petit que le cvn de la commande précédemment reçue, il n'accepte pas la commande et ne la retransmet donc pas à ses voisins. Initialement, le cvn est à 0 dans les nœuds. La première commande envoyée depuis le serveur comporte un cvn égal à 1.

Le numéro de version de commande peut aller de 1 à 65534, le dernier numéro (65535) étant réservé pour la commande de « reset du cvn », qui remet à 1 le cvn côté serveur, et à 0 le cvn côté sinks et senders. Cette commande est utile dans le cas où le serveur redémarre. En effet, son cvn revient à 1 par défaut, tandis que les sinks et senders peuvent en être au numéro de commande 17, et donc ne plus prendre en compte les commandes à venir. Lorsque le serveur redémarre, la connexion WebSocket est perdue, la passerelle essaie alors de se reconnecter de manière périodique. C'est pourquoi on envoie cette commande de remise à zéro du cvn lorsque la passerelle se (re)connecte.

Enfin, il est important de souligner que la communication entre un client et les sinks est extrêmement fiable, ce qui n'est pas le cas des échanges au niveau des nœuds du réseau. C'est pour cette raison que le mécanisme de répétitions a été instauré.

Conclusion

Au terme de ce stage, j'ai atteint les objectifs qui m'étaient fixés, à savoir la mise en œuvre du multi-sink pour les réseaux de capteurs sans fil utilisant l'algorithme RPL, ainsi que l'extension de la plate-forme de collecte en une plate-forme de gestion, permettant de superviser et contrôler les nœuds du réseau.

Par ailleurs, une grande importance a été accordée quant à l'architecture de la plate-forme. La mise en place de patrons de conception a rendu le développement propre et évolutif dans la mesure où j'ai facilement pu modifier certains points de la plate-forme sans que cela entraîne d'importants changements. Au delà de cet aspect de développement, j'ai été amené à consulter des publications scientifiques, me faisant découvrir une partie du monde de la recherche.

Ce stage m'a permis d'approfondir mes connaissances dans de nombreux domaines informatiques, à savoir le logiciel embarqué, le réseau, le génie logiciel ainsi que les bases de données. J'ai pu mettre en pratique les technologies autour des web services avec Java EE. Cette technologie étant très populaire en entreprise, cela rend ce stage encore plus intéressant, et cela ne peut être que bénéfique pour ma vie professionnelle future. Je peux affirmer que travailler avec une équipe de recherche m'a beaucoup enrichi professionnellement, tant sur le plan technique que sur le plan humain.

Enfin, la plate-forme actuelle constitue le noyau de base pour de possibles extensions. Elle pourra être utilisée pour des travaux de recherche futurs, mais également pour des projets de déploiement auprès d'industriels.

Bibliographie / Webographie

[1] Documentation de Zurb Foundation. http://foundation.zurb.com/docs/. 4

[2] Documentation technique d'un mote. http://www.advanticsys.com/shop/mtmcm5000msp-p-14.html. 7

[3] Code source de Contiki. https://github.com/contiki-os/contiki. 8

[4] Adam Dunkels. Protothreads. http://dunkels.com/adam/pt. 8

[5] Get Started with Contiki and Cooja. http://www.contiki-os.org/start. 8

[6] Gaddour Olfa and Koubâa Anis. RPL in a nutshell : A survey . *ELSEVIER*, July 2012. 8

[7] Winter T. and Thubert P. RPL : IPv6 Routing Protocol for Low-Power and Lossy Networks, March 2012. http://tools.ietf.org/html/rfc6550. 8

[8] Levis P., Clausen T., Hui J., Gnawali O., and Ko J. The Trickle Algorithm, March 2011. http://tools.ietf.org/html/rfc6206. 9

[9] Shah-Mansouri Vahid and W.S. Wong Vincent. Bounds for Lifetime Maximization with Multiple Sinks in Wireless Sensor Networks. *IEEE*, August 2007. 11

[10] Carels David, Derdaele Niels, De Poorter Eli, Vandenberghe Wim, Moerman Ingrid, and Demeester Piet. Support of multiple sinks via a virtual root for the RPL routing protocol. *EURASIP Journal on Wireless Communications and Networking*, June 2014. 12

[11] Framework HTML5 Canvas JavaScript : KinecticJS. http://kineticjs.com/. 13

[12] Librairie Highcharts. http://www.highcharts.com/demo. 13

[13] Tutoriel WebSocket Java. http://docs.oracle.com/javaee/7/tutorial/doc/websocket.htm. 17

[14] George Oikonomou. Exemple d'utilisation du multicast dans Contiki. https://github.com/adamdunkels/contiki-fork/tree/master/examples/ipv6/multicast. 17

Liste des illustrations

2.1 Réseau comportant du multi-saut 5

3.1 Un nœud du réseau 7
3.2 Construction du DODAG 9

4.1 Exemple de réseau constitué de 2 DODAG 11
4.2 Illustration du sink virtuel au sein d'un réseau multi-sink 12

A.1 Organigramme d'Inria 23

B.1 Organigramme du LORIA 24

C.1 Simulation du multi-sink sous Cooja 25

D.1 Topologie avec un seul sink 26
D.2 Topologie avec deux sinks 27
D.3 Avec un seul sink 28
D.4 Avec deux sinks 28

F.1 Interface web de l'envoi de commandes 30

G.1 Architecture globale de la diffusion des commandes 31

Annexes

A. Organigramme d'Inria

FIGURE A.1 – Organigramme d'Inria

B. Organigramme du LORIA

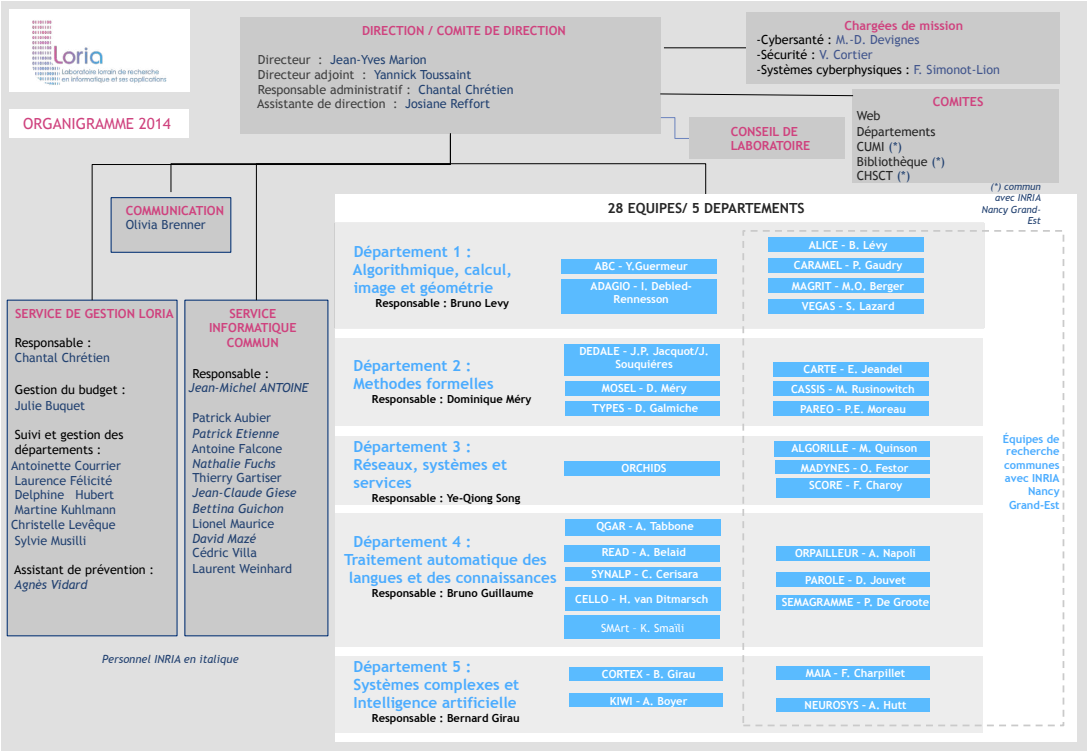


FIGURE B.1 – Organigramme du LORIA

C. Le simulateur Cooja

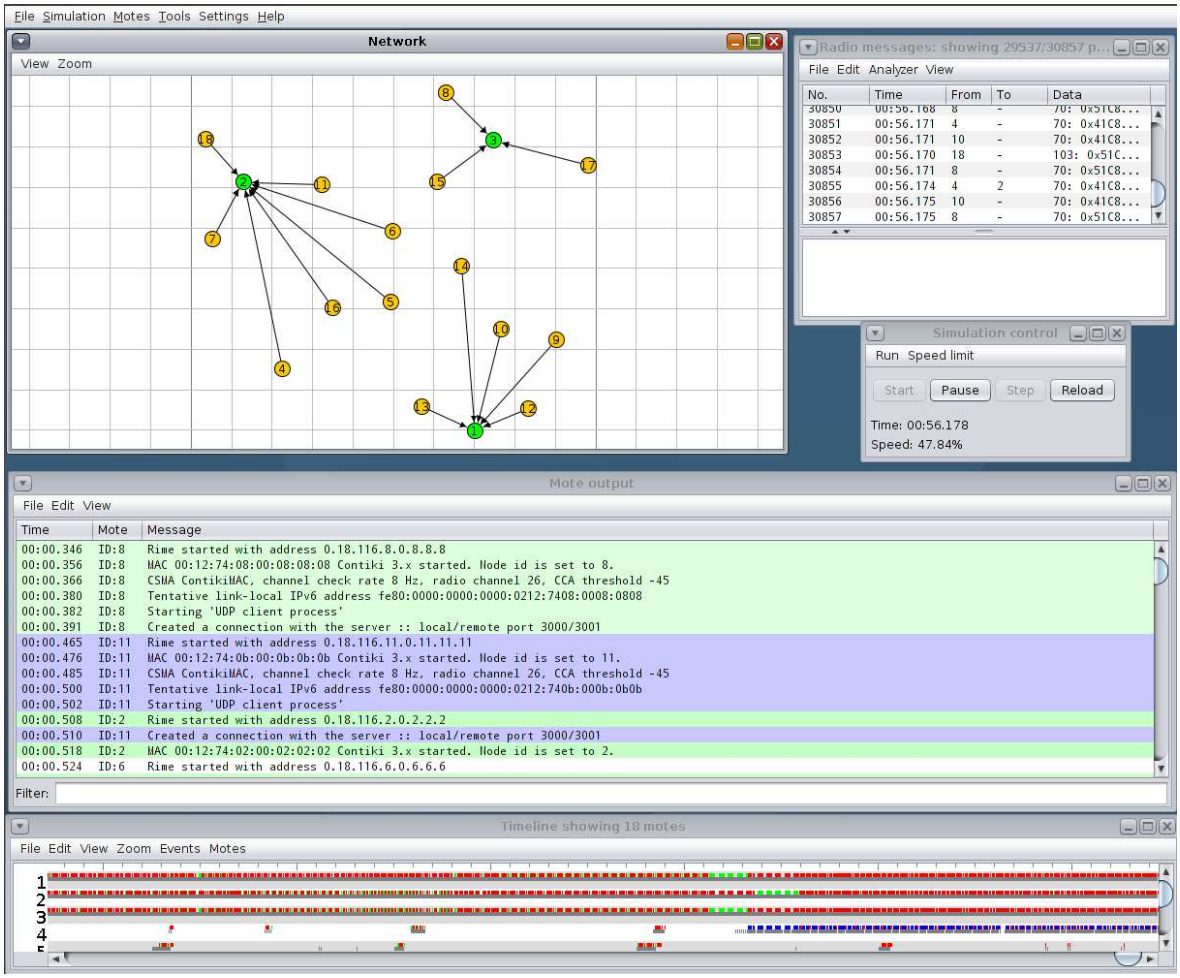


FIGURE C.1 – Simulation du multi-sink sous Cooja

D. Résultats des expériences

D.1 Topologie du réseau

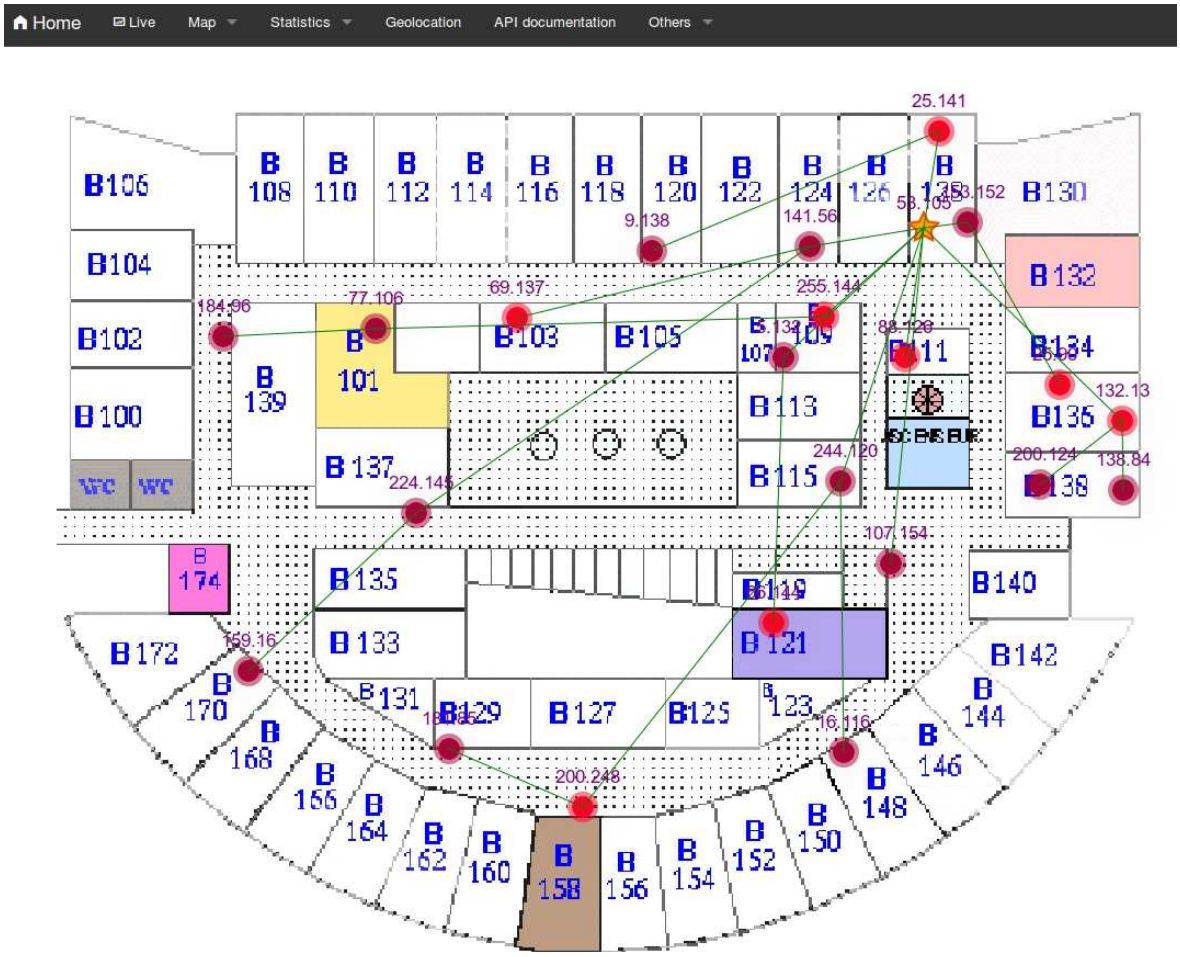


FIGURE D.1 – Topologie avec un seul sink

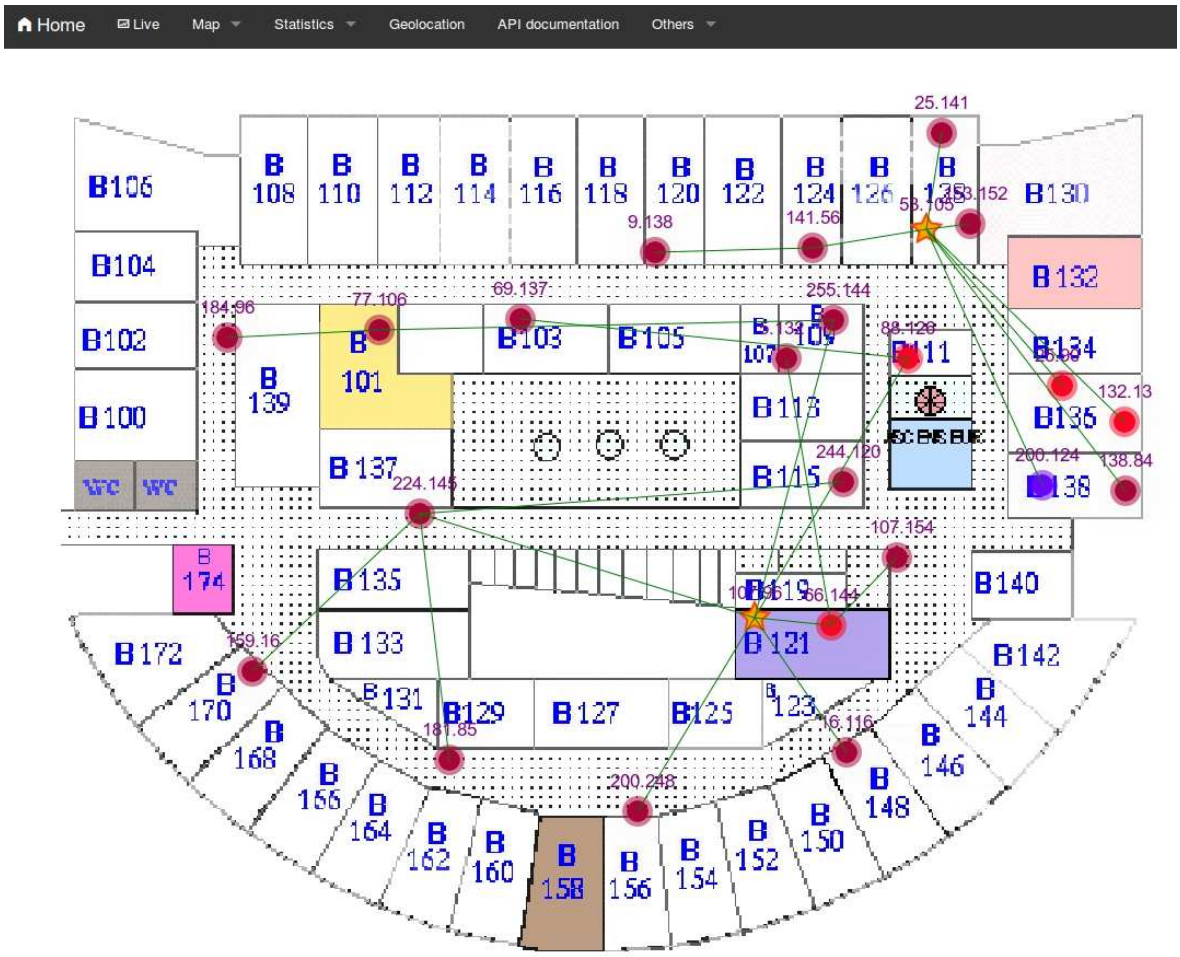


FIGURE D.2 – Topologie avec deux sinks

D.2 Taux de perte

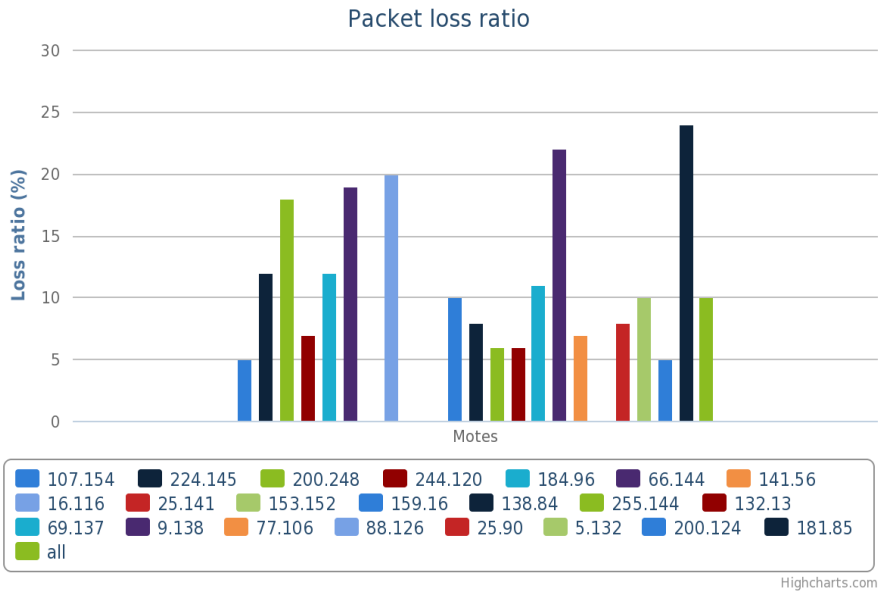


FIGURE D.3 – Avec un seul sink

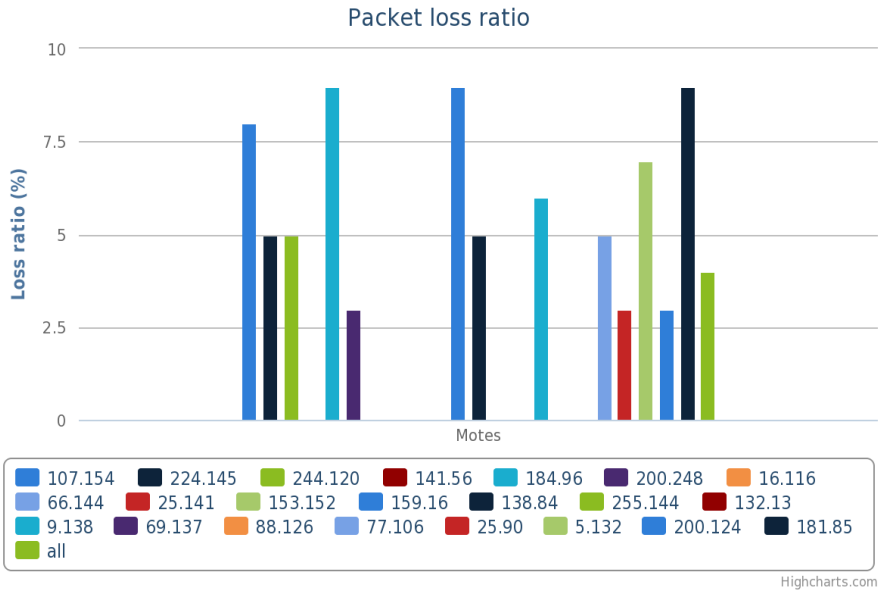


FIGURE D.4 – Avec deux sinks

E. Exemple de code Java

```
@Path("info")
public class InfoResource extends RestResource {

        @POST
        @Produces("application/json")
        @Consumes(MediaType.APPLICATION_FORM_URLENCODED)
        @Path("motes/remove")

        public String removeMote(@FormParam("mote_id") int mote_id) {
                String message = "";
                Boolean success = false;
                JsonObjectBuilder builder = Json.createObjectBuilder();

                try {
                        dao.getMoteDAO().remove(dao.getMoteDAO()
                                .getReference(mote_id));
                        builder.add("mote_id", mote_id);
                        success = true;
                        message = "Mote has been successfully removed";
                } catch (DAOException e) {
                        success = false;
                        message = e.getMessage();
                }
                logManager.info(request.getSession(), "Info", message,
                        LogManager.getIP(request));

                return builder.add("success", success)
                        .add("message", message)
                        .build().toString();
        }
        //...
}
```

F. Commandes

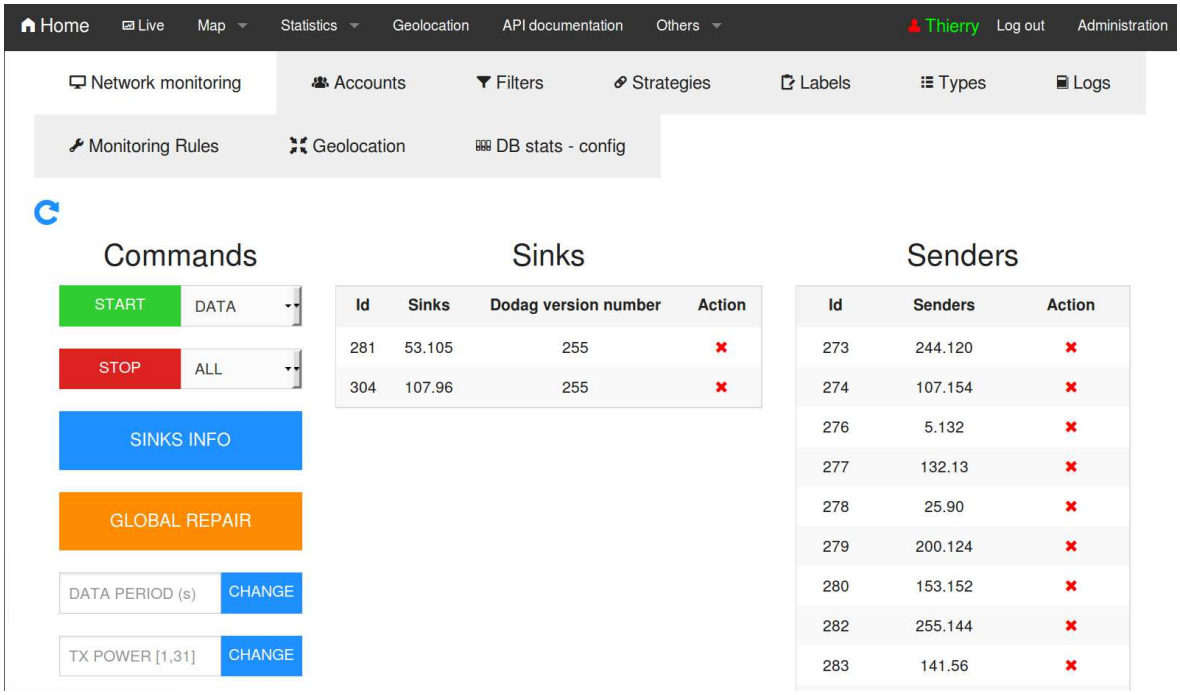


FIGURE F.1 – Interface web de l'envoi de commandes

G. Diffusion des commandes

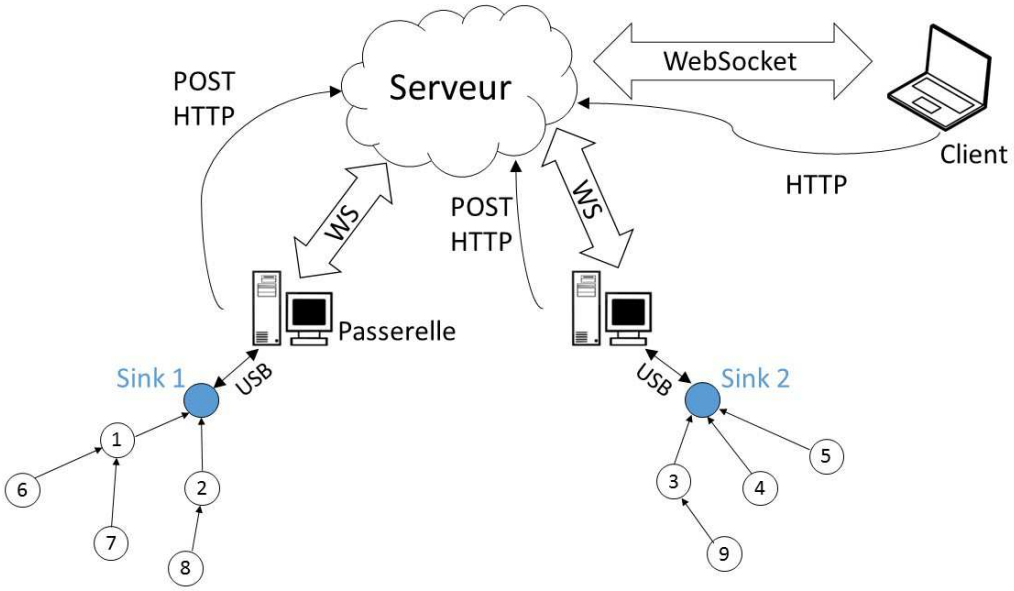


FIGURE G.1 – Architecture globale de la diffusion des commandes