HAL
open science

Évaluation et amélioration des plates-formes logicielles pour réseaux de capteurs sans-fil, pour optimiser la qualité de service et l'énergie

Kévin Roussel

▶ **To cite this version:**

Kévin Roussel. Évaluation et amélioration des plates-formes logicielles pour réseaux de capteurs sans-fil, pour optimiser la qualité de service et l'énergie. Autre [cs.OH]. Université de Lorraine, 2016. Français. ⟨NNT : 2016LORR0051⟩. ⟨tel-01754653⟩

HAL Id: tel-01754653

https://hal.univ-lorraine.fr/tel-01754653v1

Submitted on 30 Mar 2018

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Autorisation HAL

HAL Authorization

UNIVERSITÉ DE LORRAINE

AVERTISSEMENT

Ce document est le fruit d'un long travail approuvé par le jury de soutenance et mis à disposition de l'ensemble de la communauté universitaire élargie.

Il est soumis à la propriété intellectuelle de l'auteur. Ceci implique une obligation de citation et de référencement lors de l'utilisation de ce document.

D'autre part, toute contrefaçon, plagiat, reproduction illicite encourt une poursuite pénale.

Contact : ddoc-theses-contact@univ-lorraine.fr

LIENS

Code de la Propriété Intellectuelle. articles L 122. 4
Code de la Propriété Intellectuelle. articles L 335.2- L 335.10
http://www.cfcopies.com/V2/leg/leg_droi.php
http://www.culture.gouv.fr/culture/infos-pratiques/droits/protection.htm

École doctorale IAEM Lorraine

Évaluation et amélioration des plates-formes logicielles pour réseaux de capteurs sans-fil, pour optimiser la qualité de service et l'énergie

THÈSE

présentée et soutenue publiquement le 3 juin 2016

pour l'obtention du

Doctorat de l'Université de Lorraine

(mention informatique)

par

Kévin Roussel

Composition du jury

Président :	François Charpillet	Directeur de Recherches INRIA
Rapporteurs :	Thierry Val	Professeur des Universités
	Thomas Noël	Professeur des Universités
Examinateurs :	Thomas Watteyne	Chargé de Recherches INRIA
	Emmanuel Baccelli	Chargé de Recherches INRIA
Encadrants :	Ye-Qiong Song	Professeur des Universités
	Olivier Zendra	Chargé de Recherches INRIA
Invité :	Jean-Philippe Blanchard	Responsable du Pôle Innovation du Crédit Agricole S.A.

INRIA Nancy Grand-Est
Laboratoire Lorrain de Recherche en Informatique et ses Applications — UMR 7503

Version 3.1416 du 30 juin 2016

Mis en page avec la classe thesul.

À mes parents,
À mes grand-parents.

Remerciements

Je tiens, dans la présente section, à exprimer toute ma gratitude envers tous ceux qui ont permis, directement ou indirectement, à ce travail de thèse de Doctorat d'arriver à son terme.

Je remercie donc tout d'abord le Professeur Ye-Qiong SONG, et le Docteur Olivier ZENDRA, mes encadrants, qui m'ont supporté à tout point de vue durant toute la durée de cette thèse, « contre vents et marées » pourrait-on dire.

J'adresse également ma profonde reconnaissance aux Professeurs Thierry VAL et Thomas NOËL, pour l'œil exigeant mais également bienveillant avec lequel ils ont accepté d'évaluer les présents travaux de thèse. Ma reconnaissance va aussi aux Docteurs Thomas WATTEYNE et Emmanuel BACCELLI, ainsi qu'au Professeur François CHARPILLET pour l'intérêt qu'ils ont bien voulu porter aux dits travaux, respectivement en tant qu'examinateurs et président du jury de soutenance.

Je remercie aussi l'équipe Madynes, ses dirigeants (Olivier FESTOR puis Isabelle CHRISMENT) et tous ses membres pour leur compréhension et leurs encouragements dans les moments difficiles qui sont régulièrement survenus lors de cette période.

Je me dois également d'exprimer ma reconnaissance au Service des Ressources Humaines de l'INRIA au sein duquel s'est déroulée la présente thèse, notamment François THAVEAU et Patricia VENTURIN, et toutes les personnes intervenues à la demande de ces derniers pour me sortir de l'impasse : il s'agit notamment du Dr. Marie-Hélène GLOC (médecin du travail) et Rachel GRÉGOIRE (assistante sociale), ainsi que des membres du cabinet parisien ARIHM. Parmi ces derniers, j'adresse mes respectueux remerciements au Dr. Gisèle BIRCK pour son écoute et ses conseils, et surtout ma reconnaissance amicale à Timy CASSEREAU pour sa présence, son soutien et sa confiance sans failles et sans interruptions.

Plus généralement, je dois également mes remerciements à l'Université de Lorraine et plus précisément au LORIA où s'est déroulée cette thèse, sans qui rien n'aurait bien évidemment été possible. J'adresse également mes remerciements particuliers au Dr. Jean-Philippe BLANCHARD, coordinateur du projet LAR qui a financé ma thèse, et n'a jamais ménagé ses efforts mais aussi sa sympathie et son énergie durant toute la durée du dit projet.

J'adresse également mes sentiments amicaux et reconnaissants aux confrères, doctorants et ingénieurs, qui m'ont cotoyé, soutenu et souvent encouragé durant ces trois années, entre autres : Élian AUBRY, Abdallah DIB, François DESPAUX, Iñaki FERNÁNDEZ PÉREZ, Éric FINICKEL, Gaëtan HUREL, Anthéa MAYZAUD, Saïd SEDDIKI, Wazen SHBAIR, Mohamed TLIG, Evangelia TSIONTSIOU, Shouguo ZHUO... ainsi que tous les amis moniteurs de l'ENSEM. Je ne vous oublie pas, et vous souhaite bonne chance (pour ceux qui doivent encore finir leur propre thèse) et bonne continuation et succès (à tous).

Enfin, j'adresserai les dernières paroles de cette section à mes parents et à feu mes grands-parents, à qui cette thèse est dédiée. Puissent ces modestes travaux vous apporter un peu de fierté, comme je souhaite qu'ils apportent quelques avancées — aussi modestes soient-elles — à la science et à l'informatique.

informatiques mathématiques

Loria

Laboratoire lorrain de recherche
en informatique et ses applications

Sommaire

Table des figures **xi**

Liste des tableaux **xv**

1 Introduction **1**

2 Contexte et problématique **7**

2.1 Réseaux de capteurs et actionneurs sans-fil 7

2.1.1 Définitions 7

2.1.2 Constitution d'une *"mote"* 8

2.1.3 Les réseaux de capteurs sans-fil et leur trafic 10

2.1.4 Spécificités des WSN 12

2.2 La Qualité de Service (QdS) 14

2.2.1 Notion de QdS 14

2.2.2 Critères de QdS 15

2.2.3 Stratégies d'assurance de la QdS 15

2.2.4 Niveaux de service 16

2.2.5 QdS dans les réseaux de capteurs sans-fil 16

2.3 Applications des WSN (et de l'IoT) 17

2.4 Contexte 19

2.4.1 Applications d'e-santé des WSN 19

2.4.2 Le projet LAR 19

2.5 Problématique 21

2.5.1 Exposé de la problématique 21

2.5.2 Stratégie préconisée 21

3 Analyse critique de l'état de l'art **25**
3.1 Le protocole IEEE 802.15.4 25
3.1.1 Couche physique 26
3.1.2 Couche MAC 29
3.2 Protocoles MAC 34
3.2.1 Protocoles MAC synchrones 35
3.2.2 Protocoles MAC asynchrones LPL 37
3.2.3 Protocoles MAC asynchrones LPP 40
3.2.4 Protocoles MAC à ordonnancement temporel 43
3.2.5 Protocoles MAC multicanaux 45
3.2.6 Protocoles hybrides avancés 46
3.2.7 Discussion : les protocoles MAC / RDC 60
3.3 Systèmes d'exploitation dédiés 61
3.3.1 Rappels sur les notions de multi-tâche 61
3.3.2 TinyOS 63
3.3.3 Contiki 65
3.3.4 Nano-RK 66
3.3.5 RIOT OS 68
3.3.6 Autres OS spécialisés 70
3.3.7 OS temps-réel classiques / généralistes 72
3.3.8 Discussion : les systèmes d'exploitation dédiés 74
3.4 Conclusion : protocoles MAC / RDC et OS spécialisés 75

4 Plates-formes logicielles : évaluation, problèmes et améliorations **79**
4.1 Contiki : développement et limitations 79
4.1.1 Documentation minimaliste 80
4.1.2 Limitations techniques 81
4.1.3 Fonctionnalités temps-réel insuffisantes 86
4.2 RIOT OS : découverte et contributions 87
4.2.1 La plate-forme logicielle RIOT OS 87
4.2.2 Contributions au projet RIOT OS 88
4.2.3 Évolution du système RIOT OS 95
4.3 La nouvelle pile réseau de RIOT : une analyse critique 95
4.3.1 Description 95

4.3.2 Avantages 98
4.3.3 Inconvénients et manques 99
4.3.4 Discussion : la pile « gnrc » 100
4.4 Discussion : plates-formes logicielles, contributions et conclusions . . 101

5 Évaluation et comparaison des implantations de deux protocoles MAC / RDC : ContikiMAC et S-CoSenS 103

5.1 Premières plates-formes matérielles 103
5.2 Implantation de S-CoSenS sous RIOT OS : précision de la synchronisation entre noeuds 105
5.3 Évaluation des performances : comparaison avec ContikiMAC 109
5.3.1 Configuration des expériences 109
5.3.2 Taux de réception des paquets 110
5.3.3 Délai total (« de bout-en-bout ») de transmission des trames 114
5.3.4 Consommation d'énergie : *"Duty Cycles"* 117
5.3.5 Stabilité et contraintes mémoire 120
5.3.6 Influence de l'optimisation des implantations 121
5.3.7 Comparaison avec essais sur matériel 123
5.4 Améliorations potentielles des protocoles MAC /RDC 125
5.4.1 Propositions d'améliorations algorithmiques 125
5.4.2 Évaluation de ces propositions : difficultés et inadéquation des simulations 127
5.5 Discussion : évaluation et comparaison de protocoles MAC / RDC, contributions et conclusion 132

6 Validation des expérimentations sur plates-formes réelles 135

6.1 Motivation : limitations et inexactitudes des simulations 136
6.1.1 Rappels sur Cooja et MSPSim 136
6.1.2 Inexactitudes des délais sous MSPSim 138
6.1.3 Risques et conséquences potentiels 143
6.1.4 Discussion : limites des simulations et émulations 146
6.2 Validation sur matériel : moyens et objectifs 148
6.2.1 Besoins en instrumentation 148
6.2.2 Choix de plate-forme matérielle 149
6.2.3 Travaux de validations prévus et contributions attendues . . . 150

6.3 Travaux et mise en œuvre 151
6.3.1 Amélioration du pilote radio : proposition d'API étendue . . 151
6.3.2 Problèmes techniques rencontrés 154
6.3.3 Situation actuelle 163
6.4 Discussion : validation, contributions et conclusion 164

7 Conclusions et perspectives 167
7.1 Conclusions générales 167
7.2 Contributions de la thèse : résumé 169
7.3 Perspectives 171
7.3.1 Perspectives à court terme 171
7.3.2 Perspectives à long terme 173

A Publications et Réalisations 175

Bibliographie 179

Glossaire 191

Table des figures

2.1 *Schéma fonctionnel d'une mote WSN / IoT classique.* 9
2.2 *Principales catégories d'application des réseaux de capteurs sans-fil. Source : [Akyildiz and Vuran, 2010], figure 2.1.* 17

3.1 *Comparaison entre la pile 802.15.4 et les piles OSI et Internet.* 26
3.2 *Schéma topologique d'un réseau de capteurs sans-fil (WSN). (D'après [Song, 2013])* 30
3.3 *Organigramme de l'algorithme de la méthode CSMA/CA. (Source : [IEEE 802.15.4, 2011], figure 11)* 32
3.4 *Cycle de fonctionnement du protocole MAC du standard IEEE 802.15.4 en mode "beacon". (Source : [Huang et al., 2013])* 33
3.5 *Adaptative listening entre trois noeuds suivant le protocole S-MAC. (D'après [Song, 2013])* 36
3.6 *Comparaison du fonctionnement de S-MAC et T-MAC. (D'après [Song, 2013])* 37
3.7 *Schéma de fonctionnement de B-MAC (protocole LPL typique). (D'après [Song, 2013])* 38
3.8 *Schéma de fonctionnement du protocole X-MAC. (D'après [Song, 2013])* 39
3.9 *Fonctionnement du protocole ContikiMAC. (D'après [Song, 2013])* . 40
3.10 *Survenue de « collisions cachées » entre deux paires de noeuds menant chacune une transmission distincte, empêchant la réussite des deux communications. (D'après [Song, 2013] et [Sun et al., 2008])* 41
3.11 *Principe de transmission de données entre un noeud émetteur (**A** ou **S**) et un noeud récepteur (**B** ou **R**), selon la méthode RI-LPP (Receiver-Initiated Low Power Probing). (Source : [Song, 2013] et [Huang et al., 2013])* 42
3.12 *Illustration de la méthode BEB (Binary Exponential Backoff) employée par RI-MAC pour résoudre les problèmes de collisions dûs à l'envoi simultané de trames par plusieurs émetteurs. (Source : [Song, 2013] d'après [Sun et al., 2008])* 42
3.13 *Comparaison entre protocoles "Receiver-Initiated" (LPP) et "Sender-Initiated" (LPL) face à la montée en charge du trafic réseau. (Source : [Song, 2013] d'après [Österlind et al., 2012])* 43

3.14 *Comparaison de l'efficacité entre méthodes basée sur la contention (CSMA) et sur l'ordonnancement (TDMA) en fonction du débit réseau. (Source : [Song, 2013])* 44
3.15 *Principe de base du protocole CoSenS. (Source : [Nefzi, 2011])* 47
3.16 *Principe d'un cycle de fonctionnement d'un noeud coordinateur S-CoSenS. (D'après [Nefzi, 2011])* 49
3.17 *Transmission d'une trame avec le protocole S-CoSenS.* 50
3.18 *Comparaison de la gestion de la montée en charge du trafic réseau entre **(1)** un protocole uniquement basé sur la contention, et **(2)** iQueue-MAC, qui a recours à une période basée sur TDMA de durée variable, adaptée à la charge réseau. (Source : [Song, 2013])* 51
3.19 *Structure d'une trame iQueue-MAC. (Source : [Song, 2013])* 52
3.20 *Structure d'un "beacon" iQueue-MAC. (Source : [Song, 2013])* 52
3.21 *Structure d'un cycle iQueue-MAC. (Source : [Song, 2013])* 53
3.22 *Exemple de transmission de trames entre deux noeuds simples et un routeur avec le protocole iQueue-MAC. (Source : [Song, 2013])* . . . 53
3.23 *Modification du fonctionnement du protocole X-MAC par le mécanisme T-AAD. (Source : [Papadopoulos et al., 2014])* 56
3.24 *Fonctionnement du protocole avancé M-ContikiMAC, optimisé pour les noeuds mobiles des WSN. (Source : [Papadopoulos et al., 2015b])* 57
3.25 *Fonctionnement du protocole avancé ME-ContikiMAC. (Source : [Papadopoulos et al., 2015b])* 58
3.26 *Structure d'un cycle du protocole MaCARI. (Source : [Agha et al., 2009] figure 3)* 59
3.27 *Taxonomie des différents protocoles MAC conçus pour les réseaux de capteurs sans-fil (travaux académiques). Source : [Huang et al., 2013]* 61
3.28 *Principe de base du traitement des évènements temps-réel par les différents modèles de gestion multi-tâche.* 63

4.1 *Représentation fonctionnelle des dépendances au sein de l'architecture de Contiki OS. (Source : [Udit, 2012])* 85

5.1 *Schéma fonctionnel de notre PAN virtuel de test. (Légende : **R** = noeud routeur ; **S** = "sink")* 106
5.2 *Copie d'écran d'une des simulations de notre premier jeu de tests sous Cooja.* 107
5.3 *Zoom sur la partie centrale de la timeline de la figure 5.2 page 107.* . 108
5.4 *Résultats pour le taux de réception de paquets (PRR) pour un cycle MAC / RDC de 8 Hz / 125 ms.* 111
5.5 *Résultats pour le taux de réception de paquets (PRR) pour un cycle MAC / RDC de 16 Hz / 62 ms.* 112
5.6 *Résultats pour le taux de réception de paquets (PRR) pour un cycle MAC / RDC de 32 Hz / 31 ms.* 112
5.7 *Résultats combinés pour le taux de réception de paquets (TRP) avec S-CoSenS, en fonction de la durée de la subframe.* 113

5.8 *Résultats combinés pour le taux de réception de paquets (TRP) avec ContikiMAC, en fonction du channel check rate.* 113
5.9 *Résultats pour les délais moyens de transmission d'une trame pour un cycle MAC / RDC de 8 Hz / 125 ms.* 114
5.10 *Résultats pour les délais moyens de transmission d'une trame pour un cycle MAC / RDC de 16 Hz / 62 ms.* 115
5.11 *Résultats pour les délais moyens de transmission d'une trame pour un cycle MAC / RDC de 32 Hz / 31 ms.* 115
5.12 *Résultats combinés pour les délais moyens de transmission d'une trame avec S-CoSenS, en fonction de la durée de la subframe.* 116
5.13 *Résultats combinés pour les délais moyens de transmission d'une trame avec ContikiMAC, en fonction du channel check rate.* 116
5.14 *Comparaison de l'activité radio (duty cycle) sur les noeuds routeurs fonctionnant respectivement sous ContikiMAC et S-CoSenS, avec des subframes de 31 ms.* 119
5.15 *Comparaison de l'activité radio (duty cycle) sur les noeuds-feuilles fonctionnant respectivement sous ContikiMAC et S-CoSenS, avec des subframes de 31 ms.* 119
5.16 *Résultats obtenus pour le taux de réception de paquets (TRP) avec S-CoSenS, en fonction de la durée de la subframe, sur des motes IoT-LAB WSN430.* 124
5.17 *Résultats obtenus pour les délais moyens de transmission d'une trame avec S-CoSenS, en fonction de la durée de la subframe, sur des motes IoT-LAB WSN430.* 124
5.18 *Influence du mécanisme de "phase lock" sur l'activité radio (duty cycle) induite par S-CoSenS, avec des subframes de 31 ms.* 129
5.19 *Influence du mécanisme de "phase lock" sur l'activité radio (duty cycle) induite par S-CoSenS et ContikiMAC, avec respectivement des subframes de 125 ms et des CCI de 8 Hz.* 131

6.1 *Schéma fonctionnel de notre seconde configuration de test. (Cette topologie est reprise de la publication [Zhuo et al., 2013] en figure 15.)* 151
6.2 *Diagramme complet de la machine à états finis de l'AT86RF231. (Source : [DataSheet AT86RF231, 2009] figure 7-8.)* 155
6.3 *Schéma logique de l'interface entre MCU et AT86RF231 sur une mote IoT-LAB M3. (D'après [DataSheet AT86RF231, 2009] figure 6-1.)* . . 156
6.4 *Disposition physique du PAN de test employé sur IoT-LAB (noeuds de type M3).* 160

7.1 *Capture d'écran d'un "sniffer" de paquets Zigbee / 802.15.4 lors de l'exécution de RIOT avec le module* `csma_sender`*. (Crédit : Shuguo Zhuo, travail en cours.)* 172

Liste des tableaux

2.1 *Comparaison entre réseaux traditionnels et WSN. (d'après [Dargie and Poellabauer, 2010] table 1.2)* 13

3.1 *Principales plates-formes logicielles utilisables en 2015 dans le cadre des réseaux de capteurs sans-fil.* 76

4.1 *Liste des fonctions des pilotes radio nouvelle génération (« gnrc »).* . . 96
4.2 *Liste des évènements radio interceptables (« gnrc »).* 96
4.3 *Liste des options — « capacités » potentielles — des émetteurs / récepteurs radio (« gnrc »).* 97
4.4 *Liste des erreurs potentiellement renvoyées (« gnrc »).* 98

5.1 *Débits réseaux utiles programmés sur l'ensemble des noeuds-feuilles.* 109
5.2 *Statistiques de duty cycle pour ContikiMAC.* 118
5.3 *Statistiques de duty cycle pour S-CoSenS.* 118
5.4 *Delais observés pour le chargement d'une trame de 110 octets dans le buffer d'envoi de la radio CC2420 d'une mote Zolertia Z1, en fonction de l'OS et de l'implantation du pilote SPI.* 122
5.5 *Statistiques de duty cycle pour S-CoSenS avec optimisation de type "phase-lock" au niveau des noeuds-feuilles.* 129
5.6 *Statistiques de duty cycle pour ContikiMAC avec un cycle long : CCI de 8 Hz (valeur par défaut), soit un cycle de 125 ms.* 130
5.7 *Statistiques de duty cycle pour S-CoSenS au niveau des noeuds-feuilles, avec une subframe longue de 125 ms.* 130

6.1 *Delais observés par le chargement d'un paquet de 110 octets dans le buffer d'envoi de la radio CC2420 d'une mote Zolertia Z1, en fonction de l'OS et de l'implantation du pilote SPI, par simulation et par expérimentation sur matériel.* 136
6.2 *Delais observés pour le chargement d'un paquet dans le buffer d'envoi du CC2420 d'une mote SkyMote/TelosB, avec différentes configurations logicielles.* 140
6.3 *Delais observés pour le chargement d'un paquet dans le buffer d'envoi du CC2420 d'une mote Zolertia Z1, avec différentes configurations logicielles.* 141

6.4 *Poids relatif du chargement du buffer d'envoi de l'émetteur / récepteur radio dans le temps de transmission d'un paquet.* 144
6.5 *Liste des fonctions de l'API complémentaires de « capacités ».* 152
6.6 *Liste des paramètres rajoutés par l'API complémentaires de « capacités ».* 153
6.7 *Liste des erreurs pouvant être retournées par les fonctions de l'API complémentaires de « capacités ».* 153
6.8 *Types d'interruptions de l'AT86RF231. (Source : [DataSheet AT86RF231, 2009] table 6-9.)* 157
6.9 *Nombre de trames envoyées avec succès par chaque noeud-feuille, sur 500 envoyés.* 160

Chapitre 1

Introduction

Les capteurs et actionneurs sans-fil (*"Wireless Sensors and Actuators"*) — qui sont l'objet même des travaux de cette thèse — sont en fait des « nano-ordinateurs » embarqués, regroupant unité centrale, interface réseau sans-fil (radio), et divers périphériques leur permettant d'interagir avec leur environnement (capteurs et actionneurs), sur une carte électronique dont la taille ne dépasse en général pas celle d'une carte de crédit.

Une de leur spécificités majeures est de dépendre de batteries (piles ou autres) pour leur alimentation. Cette batterie étant parfois difficile voire impossible à changer, ces nano-ordinateurs doivent être conçus et programmés pour ***minimiser au maximum leur consommation d'énergie*** : leur durée de fonctionnement, et parfois même leur durée de vie, en dépend.

Ces nano-ordinateurs sont couramment appelés **capteurs sans-fil** — les appareils équipés d'actionneurs étant nettement moins nombreux et utilisés —, **nœuds** ou par le terme anglo-saxon ***"mote"***.

Le terme de « noeud » est ici particulièrement révélateur, car ces appareils sont par nature destinés à fonctionner en réseau, via le médium radio. Ce sont ces réseaux de noeuds qui sont appelés **Réseaux de Capteurs Sans-Fil** (***"Wireless Sensor Networks"*** ou **WSN**, selon le terme anglo-saxon).

L'architecture et le fonctionnement de ces capteurs sans-fil et de leurs réseaux, et les défis qu'ils posent, seront définis et expliqués de façon détaillée en section 2.1.1 page 7 du prochain chapitre.

Ces réseaux de capteurs sans-fil, en s'interconnectant entre eux et avec les réseaux globaux (WAN : *Wide Area Network*), ont permis l'apparition de la notion plus récente d'**Internet des Objets** (**IoT** : *Internet of Things*), « organisme » dont ils constituent les cellules.

Il s'agit à l'heure actuelle d'un sujet de recherche et de développement extrêmement vaste, actif et prometteur. Le développement rapide de cet IoT permet l'apparition et la mise en place d'une multitude d'applications nouvelles.

Ces applications, de plus en plus variées, riches et complexes, augmentent encore l'intérêt de plates-formes logicielles (c'est-à-dire de **systèmes d'exploitation**, en anglais *"Operating System"* : **OS**) fiables, fonctionnelles, performantes et adaptées aux noeuds de ces WSN constituant le fondement de l'IoT.

Les domaines d'application des réseaux de capteurs sans-fil, et par extension de l'Internet des Objets, sont extrêmement étendus. Deux livres [Dargie and Poellabauer, 2010] [Akyildiz and Vuran, 2010] détaillent différentes applications déjà existantes et exploitées.

On peut notamment citer :

— des applications militaires,
— des applications industrielles,
— des applications environnementales,
— des applications domotiques,
— des applications à la santé.

Ces deux derniers domaines d'application des WSN sont ceux auxquels nous nous intéressons spécifiquement dans la présente thèse.

Nous disposons notamment, au LORIA / INRIA Nancy Grand-Est, d'un projet d'appartement intelligent pour l'assistance à la personne [LORIA / INRIA Nancy Grand-Est, 2009]. Ce projet, constamment en cours de développement et de perfectionnement, est utilisé de façon intensive par les différentes équipes de recherche du site, pour le développement et le test d'applications diverses, aussi bien académiques qu'industrielles, principalement pour l'aide au maintien à domicile des personnes âgées et / ou dépendantes. Ce projet relève à la fois de l'application domotique et de l'application de santé, tout comme le projet LAR que nous allons détailler dans le chapitre 2.

Comme la plupart des systèmes informatiques connectés, les capteurs sans-fil ont recours à des piles protocolaires pour gérer l'envoi et la réception de données. Dans leur cas, il s'agit d'émettre et de recevoir ces données sous forme de trames transmises sur le médium radio — par définition peu fiable, et souvent sujet à des perturbations. La plupart des WSN actuels accèdent au médium radio selon le standard IEEE 802.15.4 [IEEE 802.15.4, 2011].

Pour pallier les problèmes liés à l'instabilité de ce médium radio, de nombreux protocoles MAC (*"Media Access Control"*) ont été développés. Le protocole 802.15.4 en propose lui-même deux versions, mais leurs limitations ont poussé la communauté académique et industrielle à développer de nombreux protocoles alternatifs, basés sur des principes et des techniques différents.

Pour faciliter la programmation et l'exploitation de ces systèmes spéciaux que sont les capteurs sans-fil, il existe des systèmes d'exploitation spécifiques, prenant en compte leurs spécificités : capacités très limitées, fonctionnement sur batterie faisant de la consommation énergétique un enjeu majeur, communication par radio. Nombre de ces plates-formes logicielles dédiées ont été conçues et sont exploitées à l'heure actuelle, mais les plus utilisés au moment où nous écrivons ces lignes sont Tiny OS [Levis et al., 2005] et surtout Contiki OS [Dunkels et al., 2004]. Ces plates-formes logicielles sont fournies avec leurs propres piles réseau intégrées, qui sont donc la cheville ouvrière du fonctionnement concret des communications sur les réseaux de capteurs sans-fil.

Si de nombreux travaux de recherche ont, comme nous l'avons dit, été menés pour développer des protocoles MAC de plus en plus performants pour exploiter au mieux le médium radio et contourner ses limitations, tout en préservant au maximum les ressources énergétiques des capteurs sans-fil, les résultats de ces travaux n'ont malheureusement guère été concrètement implantés et diffusés à l'heure actuelle dans les piles réseaux des systèmes d'exploitation spécialisés : celles-ci ne comportent le plus souvent que le protocole MAC du standard IEEE 802.15.4, plus éventuellement quelques protocoles simples et / ou anciens ne représentant nullement l'état de l'art en la matière.

La seule exception notable à cette situation est la présence en standard du protocole ContikiMAC [Dunkels, 2011] dans la pile réseau des versions récentes Contiki OS. Ce protocole, s'il est récent et performant, repose toutefois sur des principes de fonctionnement somme toute classiques (bien qu'optimisés) impliquant certaines limites. De nombreux travaux d'implantation de protocoles différents — notamment sur leurs principes de fonctionnement — restent encore à entreprendre.

Ainsi, à l'heure actuelle, les couches basses des piles réseau spécialisées pour les capteurs sans-fil, notamment celles intégrées aux plates-formes logicielles dédiées, constituent un « goulot d'étranglement » pour la performance des communications entre *motes*, les couches MAC (et plus généralement les couches basses) semblant avoir été peu prioritaires dans les efforts de développement et d'implantation des piles réseau des OS pour WSN.

Cette thèse a ainsi pour but de permettre d'obtenir des améliorations significatives pour ces couches basses, tant sur le plan des performances que de l'optimisation de la consommation énergétique, via un effort de recherche et d'implantation des piles réseau dédiées, notamment en exploitant au mieux les fonctionnalités offertes par les plates-formes logicielles spécialisées dans les capteurs sans-fil.

Si les OS les plus utilisés que sont Tiny OS [Levis et al., 2005] et Contiki OS [Dunkels et al., 2004] n'offrent pas les fonctionnalités nécessaires, d'autres plates-formes logicielles moins répandues, mais plus performantes et récentes, offrent notamment des mécanismes avancés de gestion des interruptions, un modèle multitâche préemptif, et des fonctionnalités temps-réel (notamment en exposant au mieux — via une API adaptée — les *timers* matériels présents dans les microntrôleurs équipant les capteurs sans-fil). Ces fonctionnalités sont notamment très utiles pour améliorer la qualité de service (QdS) des réseaux. Parmi les systèmes pour capteurs sans-fil offrant de telles capacités, on pourra entre autres citer Nano-RK [Eswaran et al., 2005] ou RIOT OS [Hahm et al., 2013].

Un autre mécanisme lié à l'OS particulièrement intéressant pour l'économie d'énergie est la présence d'un noyau fonctionnant en mode *"tickless"*, c'est-à-dire permettant de ne faire fonctionner l'appareil que quand cela est strictement nécessaire.

Nous nous proposons d'exploiter toutes ces fonctionnalités avancées offertes par ces OS dédiés pour tenter d'implanter l'état de l'art en matière de protocoles MAC, et ainsi obtenir de meilleurs résultats en termes de performances de communication et d'économies d'énergie.

Notons que pour des raisons juridiques aussi bien que techniques — possibilité de modifier et d'améliorer le cœur et les différents composants du système selon nos besoins — nous n'envisagerons dans la présente thèse uniquement l'utilisation des systèmes d'exploitation — et plus généralement des logiciels — à licence libre et *open source*.

Objectifs

Les travaux de la présente thèse ont ***les principaux objectifs*** suivants :

1. Un ***état de l'art des différents protocoles MAC créés par la recherche*** académique ou industrielle, et surtout un passage en ***revue des principaux systèmes d'exploitation utilisés dans le cadre des réseaux de capteurs sans-fil***, en analysant leurs fonctionnalités, déterminant ainsi ***quels sont les mieux adaptés au développement de couches basses*** (notamment MAC) avancées et performantes.
2. Le ***choix de la plate-forme logicielle la mieux adaptée*** pour le développement de ces couches basses, avec notamment une ***analyse critique des piles réseau — notamment de l'API et des drivers radio— de ces plates-formes spécialisées (Contiki et RIOT OS)***.
3. Une ***implantation du protocole S-CoSenS sur le système RIOT OS*** — s'inscrivant dans un effort plus global destiné à ***fournir une couche MAC performante à la pile réseau de cette plate-forme logicielle*** — suivi d'une ***comparaison avec l'implantation standard de ContikiMAC*** sur Contiki OS, notamment en présence d'un trafic réseau intense ; et enfin la ***proposition d'idées d'améliorations algorithmiques à apporter aux protocoles MAC***.
4. La ***validation sur plates-formes réelles*** des résultats de nos expérimentations — effectuées jusqu'alors par simulation / émulation — suite à la ***découverte d'un problème d'inexactitude temporelle dans l'outil de simulation Cooja / MSPSim***, avec une analyse des problèmes rencontrés, et la fourniture d'autant de détails techniques et des pistes de résolution possibles pour aider à la résolution ultérieure des difficultés rencontrées.

Structure

Le présent manuscrit de thèse est organisé de la façon suivante :

— Après ce présent chapitre d'introduction, le chapitre 2 page 7 donne les définitions techniques nécessaires à la bonne compréhension du sujet, développe les différentes applications possibles des WSN et de l'IoT, puis présente le contexte de la thèse, et enfin la problématique que celle-ci se propose de résoudre, en commençant à détailler nos pistes de travail.

— Le chapitre 3 page 25 présente l'état de l'art sur le protocole IEEE 802.15.4 sur lequel reposent les réseaux de capteurs sans-fil actuels ; une présentation des différents axes de recherche et des exemples significatifs de protocoles MAC

développés par la communauté pour suppléer aux limitations du protocole MAC du standard 802.15.4 ; et enfin, nous faisons une première contribution sous la forme d'une revue (*"survey"*) des systèmes d'exploitation (OS) spécialisés dans le domaine des WSN, en détaillant sucessivement leurs points forts et leurs limitations, et par là-même leur adaptation au développement de protocoles MAC avancés et performants.

— Le chapitre 4 page 79 détaille notre recherche d'une plate-forme logicielle (OS) adaptée à nos travaux de recherche sur les protocoles MAC à hautes performances, montre nos contributions au développement de la plate-forme performante et novatrice (RIOT OS) que nous avons choisie, et propose dans ce cadre une étude critique de sa nouvelle pile réseau de RIOT OS (« gnrc »).

— Le chapitre 5 page 103 montre les résultats de nos premières expériences en comparant les implantations d'un protocole hybride, S-CoSenS, mis au point au sein de notre équipe, à celle du protocole ContikiMAC, référence largement utilisée dans la communauté. Ces comparaisons montrent en particulier le comportement de ces deux implantations de protocoles face à une montée en charge intensive du trafic réseau. Nous proposons également plusieurs techniques susceptibles d'améliorer la robustesse des protocoles MAC / RDC, notamment en complétant l'interface avec la couche 1 (pilotes des émetteurs / récepteurs radio) afin d'influer dynamiquement sur des paramètres liés à l'écoute du médium.

— Le chapitre 6 page 135 présente tout d'abord les inexactitudes d'ordre temporel que nous avons découvertes dans les résultats fournis par Cooja, l'un des simulateurs de WSN les plus utilisés ; nous y montrons nos contributions sous la forme d'une analyse des limitations de ce dernier comme outil d'évaluation de performances, et des conséquences possibles sur la validité et la justesse des travaux basés sur ces simulations (y compris nos propres travaux) ; nous fournissons enfin des pistes sérieuses quant aux causes du problème, et aux moyens de le contourner ou d'y remédier. Sont ensuite détaillés les travaux de validation prévus sur matériel pour valider de façon indiscutable nos précédentes expériences, ainsi que pour tester la montée en charge de S-CoSenS sur un réseau de forte taille. Nous continuons en décrivant nos premières expériences sur la plate-forme matérielle de test choisie, le *testbed* IoT-LAB. Nous détaillons enfin les problèmes techniques nous ayant empêché de terminer de mener à bien ces travaux, en tentant de fournir le maximum de pistes techniques pour faciliter leur résolution future.

— Enfin, le chapitre 7 page 167 termine ce manuscrit de thèse en présentant nos conclusions générales, et en discutant des perspectives pouvant faire suite à nos travaux.

Chapitre 2

Contexte et problématique

2.1 Réseaux de capteurs et actionneurs sans-fil

2.1.1 Définitions : technologie des réseaux de capteurs et actionneurs sans-fil

Les avancées spectaculaires des dernières décennies en micro-électronique, notamment dans la domaine de la miniaturisation, de l'augmentation constante de la puissance des circuits intégrés, ainsi que dans le domaine de la communication sans-fil — principalement par des moyens radio — ont permis l'émergence d'appareils électroniques miniatures (Microsystèmes électromécaniques : MEMS) capables d'interagir avec leur environnement physique, de traiter des données et de communiquer entre eux et avec d'autres systèmes informatiques sans besoin de recourir à des câbles (communication radio). Ces mêmes avancées de la technologie permettent la fabrication de tels appareils à de très faibles coûts.

Ces appareils miniatures portent le nom de **capteurs / actionneurs sans-fil** — le terme « capteurs sans-fil » étant le plus souvent utilisé pour désigner tous ces appareils, l'utilisation d'actionneurs agissant sur leur environnement étant à l'heure actuelle plus rare — et sont également souvent appelés par le terme anglais ***"motes"***.

Une des principales spécificités de ces appareils est d'être alimentés par des batteries de faible puissance : piles AAA, ou piles « boutons » au lithium par exemple. L'énergie est donc un facteur très limitant sur une *mote*.

Ces appareils, lorsqu'ils sont connectés les uns aux autres, forment des **réseaux de capteurs sans-fil** (WSN : *Wireless Sensor Networks*). Ces réseaux, utilisent à la base des technologies spécifiques — notamment un protocole radio dédié différent de celui, par exemple, des téléphones cellulaires ou du *WiFi* équipant les ordinateurs portables — offrant en contrepartie d'une faible consommation énergétique un débit et une portée très limités. Les réseaux formés par ces motes sont nommés **PAN *(Personal Area Network)***, en référence à la très faible portée des émetteurs / récepteurs radio mis en action.

Une évolution plus récente dans le développement des WSN est l'emploi dans les couches supérieures de leur piles réseau, notamment la couche réseau (niveau 3), du

protocole IPv6, adapté pour les appareils de faible puissance (6LoWPAN [Mulligan, 2007] : *IPv6 over Low-power Wireless Personal Area Networks*), permettant ainsi l'utilisation d'UDP ou d'ICMP pour la couche transport (niveau 4), et des applications basées sur des standards classiques d'internet comme HTTP. L'interconnection et la fusion des WSN avec les réseaux « classiques » composant Internet — réseau global ou **WAN** (***Wide Area Network***) — a donné naissance à la notion **d'Internet des Objets** (**IoT** : *Internet of Things*) ; notion dont l'intérêt et les applications augmentent de façon exponentielle. Le développement et la diffusion de ces applications sont rendus d'autant plus faciles par le coût très faible de ces *motes* constituant les noeuds des WSN et donc de l'IoT.

2.1.2 Constitution d'une *"mote"*

La structure d'une *mote* peut être résumée par le schéma montré en figure 2.1 page ci-contre.

Les motes constituant les **noeuds** des WSN — et par extension de l'IoT — sont des appareils extrêmement compacts : ils sont conçus autour d'un circuit intégré central regroupant le processeur principal (CPU) et plusieurs périphériques de base intégrés : *timers*, convertisseurs A/D et D/A, contrôleurs E/S regroupant des broches à usage général, des bus SPI et I^2C, etc. Ces circuits intégrés centraux sont appelés **microcontrôleurs** (**MCU** : *Micro Controller Unit*).

Il faut être bien conscient que ces MCU disposent d'une puissance de calcul et d'un espace mémoire extrêmement limités — non seulement comparés aux ordinateurs personnels (PC) actuels, mais également en regard d'appareils mobiles tels que les *smartphones* et les tablettes. Leur puissance est en fait plutôt comparable, pour les modèles les moins chers (et donc parmi les plus couramment employés), à celle des ordinateurs personnels du début des années 1980 (Apple II, Commodore VIC et 64, Atari 800XL, Amstrad CPC, etc.).

Outre ce MCU central, une *mote* comprend en général seulement quelques **capteurs** (*"sensors"*) — et parfois, plus rarement quelques **actionneurs** (*"actuators"*) — leur permettant d'interagir avec leur environnement, en captant et mesurant des phénomènes physiques environnants — température, pression, humidité, présence d'individus, radioactivité... ; l'ensemble des capteurs envisageables est quasiment sans limite — et en agissant sur cet environnement pour les *motes* équipées d'actionneurs (par exemple : alarme, signaux lumineux, etc.)

De nombreuses *motes* possèdent, pour des raisons pratiques, des périphériques supplémentaires (externes au MCU) : on citera par exemple de la mémoire Flash supplémentaire pour le stockage permanent de données, ou des ports série / USB pour se connecter à un PC — principalement à des fins de déboguage.

Une *mote* est bien évidemment équipée d'un émetteur / récepteur radio, pour pouvoir communiquer en réseau. Comme dit ci-dessus, ces appareils sont pour la plupart extrêmement limités en énergie ; ces émetteurs / récepteurs radio utilisent donc un standard conçu pour consommer très peu d'énergie, en contrepartie d'un débit très limité (comparable aux premiers modems analogiques du début des années 1990). La standard spécifiquement conçu dans cette optique utilisé par la plupart de

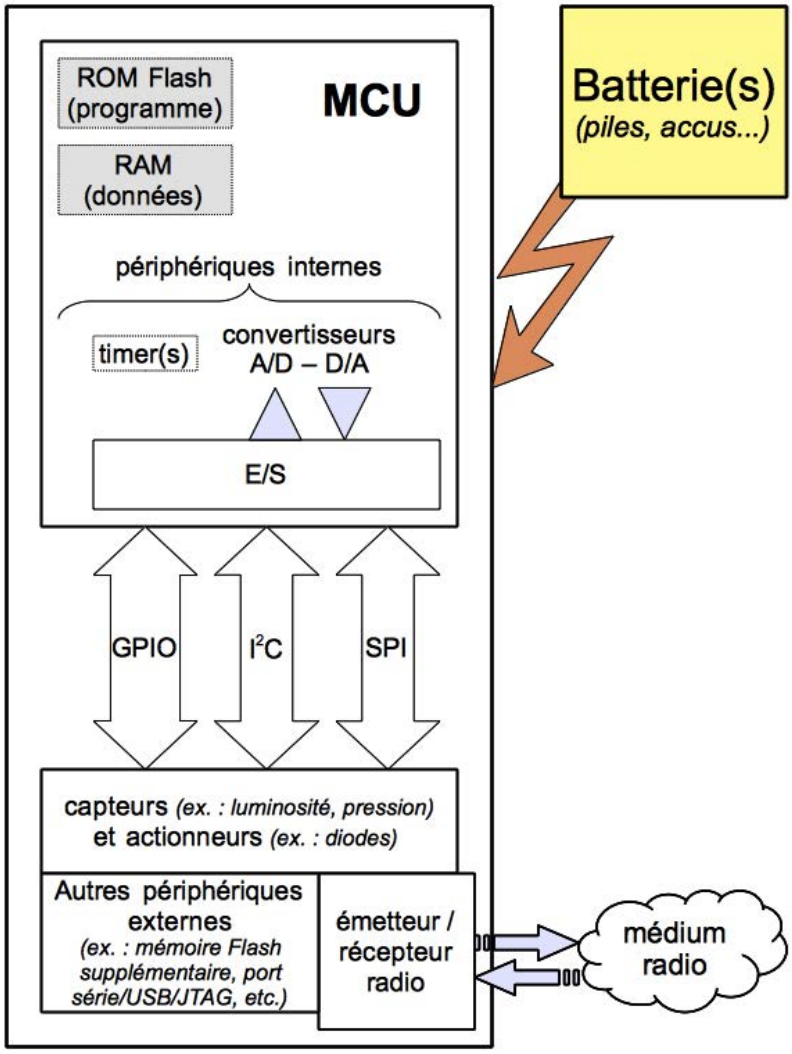

FIGURE 2.1 – *Schéma fonctionnel d'une mote WSN / IoT classique.*

ces radios est le standard IEEE 802.15.4 [IEEE 802.15.4, 2011] (ne couvrant que les couches les plus basses de la pile réseau) que nous décrirons ultérieurement dans le présent manuscrit en section 3.1 page 25. Notons que certains MCUs récents, dans un souci d'efficacité et d'intégration, intègrent directement un émetteur / récepteur radio sur la même puce.

Enfin, une *mote* est bien évidemment équipée d'une ou plusieurs batteries pour son alimentation. Certains modèles de développement / déboguage offrent également la possibilité d'être alimentés par une source d'énergie externe (secteur).

Les middlewares pour WSN. Avant de nous intéresser en détail au fonctionnement technique des couches de bas niveau des WSN, nous allons très brièvement évoquer un autre sujet de recherche actif dans le domaine des WSN.

Notons en effet que l'absence de standard officiel pour les WSN au-delà des deux couches les plus basses fournies par le standard IEEE 802.15.4 (niveaux OSI 1 : PHYsique, et 2 : *Media Access Control*, comme nous le verrons dans la section 3.1 page 25 de l'état de l'art) a amené à un développement assez anarchique des piles, protocoles et autres solutions pour les couches plus hautes. À l'heure où nous écrivons ces lignes, aucune des solutions conçues pour travailler par-dessus les réseaux 802.15.4 (telles ZigBee, etc.) ne s'est réellement imposée comme standard, fut-il de fait.

On voit donc, selon les applications prévues pour les différents WSN, se multiplier des solutions souvent incompatibles dans le déploiement des différents réseaux de capteurs sans-fil sur le terrain.

Un domaine de recherche dans les WSN est donc la conception de ***"middlewares"*** (« intergiciels ») permettant une interaction efficace entre tous ces WSN de conception différente et le réseau global (Internet), pour tenter de faire de l'IoT une réalité tangible, fiable et donc utilisable industriellement. Un article de référence (*"survey"*) recensait déjà nombre de projets de ce type lors de la décennie précédente [Wang et al., 2008].

Nous n'avons au cours de cette thèse malheureusement quasiment pas eu le temps de nous pencher sur cette problématique, si ce n'est au début pour travailler très brièvement sur le *middleware* MPIGate [Chehaider et al., 2013]. Les travaux de la présente thèse concernent, à l'exception de la publication cité dans la phrase précédente, uniquement les couches basses des piles réseau des plates-formes spécialisées pour WSN.

2.1.3 Les réseaux de capteurs sans-fil et leur trafic

Les réseaux de capteurs sans-fil (WSN) sont, comme nous l'avons vu ci-dessus dans le descriptif des motes, des réseaux à bas débit et à faible portée, en contrepartie d'une consommation d'énergie très limitée.

Le trafic réseau est, dans ce genre de réseau, typiquement faible, avec éventuellement des pointes de débit. Les transmissions entre noeuds d'un WSN peuvent, par nature, être déclenchées selon trois modalités différentes :

Périodique : un noeud capteur analyse son environnement à intervalles réguliers,

en tire une valeur exploitable (par exemple : température, pression, etc.), et l'émet sur le réseau à l'aide de son émetteur / récepteur radio. L'intervalle de temps entre deux mesures / émissions est à la discrétion des concepteurs du WSN et / ou des programmeurs de la *mote*.

Évènementiel : un noeud capteur, lors d'une analyse de son environnement, capte une valeur — ou une variation de valeur — anormale ou atypique, et envoie alors un message d'alerte sur le réseau. Là encore, l'intervalle des valeurs normales, ou le type d'évènement susceptible de déclencher l'envoi d'une alerte est à la discrétion des concepteurs du WSN et/ou des programmeurs de la *mote*.

Par requête : un intervenant extérieur au WSN émet, via la station de base[1] une requête vers une *mote* donnée, qui y répond par le déclenchement d'un actionneur — si le noeud en est équipé — et / ou par l'envoi d'une réponse, correpondant le plus souvent à une valeur analysée par un des capteurs de la *mote* en question. Ici aussi, la nature des requêtes et des réponses à y apporter sont à la discrétion des concepteurs du WSN et/ou des programmeurs de la *mote*.

Ces trois modalités ne sont bien évidemment pas exclusives : un même WSN peut utiliser les trois types de fonctionnement en fonction des besoins du moment.

Prenons l'exemple, pour aborder le sujet de l'e-santé qui concerne cette thèse, d'un WSN destiné à monitorer les signes vitaux d'un patient sous surveillance (par exemple : un malade cardiaque) :

— En temps normal, les capteurs présents sur le patient relèvent ses signes vitaux (pulsations cardiaques, pression sanguine, oxymétrie de pouls, mouvements respiratoires, etc.) de façon régulière et envoient ainsi (via l'IoT) les résultats à une base de données spécialisée consultable par les médecins.

— En cas d'anomalie, (par exemple : chute de pression sanguine, ou des pulsations), les capteurs envoient sans attendre un message d'alerte, qui pourra être relayé aux médecins traitant le patient ou même, selon la gravité, au SAMU. Un mécanisme d'IoT permettant d'envoyer des messages (type SMS) à des secouristes présents à proximité du patient pourrait également être envisagé.

— En cas de situation critique (par exemple : fibrillation cardiaque), les messages d'alerte envoyés plus tôt auront alerté les services médicaux d'urgence, qui pourront en attendant l'arrivée physique des secours envoyer (toujours via l'IoT) une requête au WSN présent sur le patient : par exemple pour ordonner la mise en action d'un défibrillateur cardiaque portatif (ou d'un *"pace-maker"*) connecté au WSN porté par le patient.

Les possibilités offertes par les WSN — ne serait-ce que sur le seul domaine médical vu dans cet exemple — sont donc extrêmement larges.

Il importe toutefois, pour connaître les limites de ces réseaux de capteurs sans-fil, d'en connaître les spécificités. C'est le sujet que nous allons maintenant aborder dans la prochaine section.

1. La **station de base** est le noeud (*mote* ou PC ou tout autre appareil connecté) servant de lien entre un réseau de capteurs sans-fil et le reste de l'Internet : il s'agit, pour faire simple, des passerelles de l'Internet des Objets (IoT). On les nomme également souvent ***"sinks"***.

2.1.4 Spécificités des WSN

Par rapport aux réseaux traditionnnels, les réseaux de capteurs sans-fil ont des spécificités, décrites dans la table 2.1 page suivante.

Les principaux points déterminants concernant ces spécificités sont les suivants :

— Les capteurs sans-fil sont des appareils dont le coût doit rester le plus faible possible. Cela influe sur les capacités très limitées de ces appareils, et aussi potentiellement sur leur fiabilité.

— Les réseaux de capteurs sans-fil sont généralement des réseaux ***ad-hoc***, c'est-à-dire dont la structure n'est pas planifiée. Leur mode de fonctionnement est ainsi fondamentalement local, aucune gestion centralisée n'est en général possible.

— Les réseaux de capteurs sans-fil sont en général conçus pour servir une et une seule application donnée au cours de leur existence (point commun avec l'informatique / électronique embarquée). Ils peuvent être amenés à opérer dans des conditions environnementales difficiles voire hostiles.

— Un point commun avec les réseaux traditionnels est la montée en charge. Un réseau de capteurs sans-fil peut résulter de l'interconnexion de centaines de PANs, et regrouper ainsi des milliers de *motes*. De telles tailles de réseau sont rarement atteintes avec les réseaux sans-fil les plus courants.

— Les noeuds de ces réseaux pouvant tomber en panne sans pouvoir être remplacés (défaillance d'un composant, ou d'une batterie non remplaçable), la structure de tels réseaux doit pouvoir « survivre » à ces pannes et les gérer.

— Ce sont des réseaux sans-fil, dont le médium radio est peu fiable par nature (par rapport aux réseaux câblés), sujet à la diffusion, et dont l'état est variable au cours du temps.

— Enfin, et c'est peut-être le plus important, la consommation d'énergie est une contrainte extrêmement forte dans les réseaux de capteurs sans-fil, bien plus que dans n'importe quel autre type de réseau. Cette contrainte influe directement et lourdement sur la conception des WSN et de leurs noeuds.

Pour répondre à ces différents problèmes, quatre notions liées à ***l'auto-gestion*** sont mises en avant dans [Dargie and Poellabauer, 2010] comme caractéristiques désirables pour les WSN :

***"Self-organization"*, auto-organisation :** est la capacité d'adapter les paramètres de la configuration du réseau en fonction de son état et de son environnement.

***"Self-optimization"*, auto-optimisation :** est la capacité de monitorer et d'optimiser l'utilisation des ressources limitées du réseau.

***"Self-protection"*, auto-protection :** est la capacité de reconnaître les intrusions et attaques et de s'en protéger.

***"Self-healing"*, auto-réparation :** est la capacité de découvrir, d'identifier la cause et de réagir aux pannes et défaillances du réseau.

Réseaux traditionnels	**Réseaux de capteurs sans-fil**
Réseaux généralistes : conçus pour servir toutes les applications possibles.	***Réseaux spécifiques :*** conçus dans un but unique pour servir une application bien définie.
Les principaux objectifs sont ***la performance du réseau et ses latences***. La consommation d'énergie n'est pas une préoccupation majeure.	***L'énergie est une contrainte centrale*** dans la conception d'un réseau de capteurs sans-fil et de chacun de ses noeuds.
Les réseaux sont ***conçus et déployés selon des plans prédéterminés***.	Le déploiement, la structure des réseaux et l'utilisation des ressources sont souvent déterminés de manière **ad-hoc** ***(sans planification préalable)***.
Les réseaux et leurs composants opèrent dans des ***environnements contrôlés*** et des ***conditions environnementales modérées***.	Les réseaux de capteurs sans-fil opèrent souvent dans des ***environnements et des conditions hostiles***.
La maintenance et la réparation sont des opérations courantes et ***le réseau et ses composants sont typiquement faciles d'accès.***	***L'accès physique aux capteurs sans-fil est souvent difficile voire même impossible***.
Le coût des composants des réseaux peut être élevé, selon le niveau de performances visé.	***Le coût des capteurs sans-fil doit rester très faible***, d'où le recours à des composants bon marché aux performances (et parfois à la fiabilité) limitées.
La défaillance d'un composant du réseau est réglée par ***des opérations de maintenance et de réparation***.	La défaillance d'un ou plusieurs noeuds est prévisible et ***gérée dans la conception même du réseau***.
La connaissance de l'état global du réseau est typiquement possible et ***la gestion centralisée est possible***.	La plupart des décisions sont ***prises au niveau local sans intervention d'une gestion centralisée***.

TABLE 2.1 – *Comparaison entre réseaux traditionnels et WSN. (d'après [Dargie and Poellabauer, 2010] table 1.2)*

Ces différentes notions amènent à la capacité de ces réseaux de capteurs sans-fil d'assurer un niveau de service suffisant pour les applications ayant des contraintes fortes quant à la qualité et le débit de leurs flux de données. Ce domaine, la **Qualité de Service (QdS)** (*QoS* en anglais) va être l'objet d'étude de la prochaine section 2.2.

2.2 La Qualité de Service (QdS)

Note : cette section est inspirée des informations issues de [Nefzi, 2011] et de [Comment ça marche QoS, 2014]

2.2.1 Notion de QdS

Le terme **QdS** (« **Qualité de Service**, en anglais QoS : *"Quality of Service"*) désigne la capacité à fournir un service, ici un support de télécommunications, conforme à des exigences de fonctionnement acceptable à assurer par le fournisseur du service envers l'utilisateur.

Appliquée aux réseaux à commutation de paquets (réseaux basés sur l'utilisation de routeurs) la QoS désigne l'aptitude à pouvoir garantir le non-dépassement d'un niveau acceptable de baisse de qualité, défini contractuellement, pour un usage donné (voix sur IP, vidéo-conférence, etc.).

En effet, contrairement aux réseaux à commutation de circuits, tels que le réseau téléphonique commuté, où un circuit de communication est dédié pendant toute la durée de la communication, il est impossible sur Internet de prédire le chemin emprunté par les différents paquets.

Cette incertitude est encore plus forte sur les réseaux sans-fil, où le médium radio lui-même est sujet à la diffusion, dont l'état et l'encombrement peuvent varier fortement et rapidement au cours du temps. Ce médium est, comparé aux câbles des réseaux informatiques « classiques », bien moins fiable.

Ainsi, rien ne garantit qu'une communication nécessitant une régularité du débit pourra avoir lieu sans encombre. C'est pourquoi il existe des mécanismes, dits mécanismes de QoS, permettant de différencier les différents flux réseau et réserver une partie de la bande passante pour ceux ayant une importance particulière.

Plus formellement, la recommandation E.800 de septembre 2008 de l'Union Internationale des Télécommunications (UIT) définit la QdS comme étant « l'ensemble des caractéristiques d'un service de télécommunication qui lui permerrent de satisfaire aux besoins explicites et aux besoins implicites de l'utilisateur du service » ; une caractéristique étant définie comme une « propriété (qualitative ou quantitative) qui aide à faire la distinction entre les individus d'une population donnée ».

En termes pratiques, une caractéristique du standard E.800 de l'UIT correspond à un critère de QdS. Le but à atteindre étant d'assurer une valeur minimale en-deça de laquelle ne pas tomber pour respecter un contrat de qualité avec l'utilisateur.

2.2.2 Critères de QdS

Les principaux critères permettant d'apprécier la qualité de service sont les suivants :

Perte de paquets (en anglais *packet loss*) : elle correspond à la non-délivrance (perte) de paquets de données, la plupart du temps dûe à un encombrement du réseau.

Débit (en anglais *bandwith*) — encore appelée **bande passante** — il définit le volume maximal d'information (bits) pouvant transiter par unité de temps.

Latence, délai, ou temps de réponse (en anglais *delay*) : elle caractérise le retard entre l'émission et la réception d'un paquet de données.

Gigue (en anglais *jitter*) : elle représente la fluctuation du signal numérique, dans le temps ou en phase.

Déséquencement (en anglais *desequencing*) : correspond à une modification de l'ordre d'arrivée des paquets.

Le rôle d'une politique ou d'un mécanisme de QdS est de toujours garder une valeur acceptable (égale ou supérieure à une valeur-seuil de qualité minimale) pour ces différents critères.

(À noter qu'à côté de ces principaux critères de QdS, d'autres peuvent être pris en compte, comme par exemple la durée de vie de chaque noeud, le MTBF, le respect du coût, etc.)

2.2.3 Stratégies d'assurance de la QdS

Les stratégies classiques appiquées pour assurer une QdS optimale peuvent se diviser en deux grandes catégories :

— Les méthodes de contrôle *a priori* ou ***préventives*** du trafic réseau que sont :
  > les politiques de gestion de files d'attente, ou ordonnancement, permettant la mise en place de la différentiation de service ;
  > le lissage (ou mise en forme) du trafic, consistant à contrôler le volume du trafic entrant dans le réseau, la plupart du temps selon les méthodes du seau percé ou du seau à jetons ;
  > le contrôle du trafic, et sa variante extrême, le contrôle d'admission, consistant à refuser le trafic entrant en fonction de certains critères.

— Les méthodes de contrôle *a posteriori* ou ***réactives*** du trafic réseau :
  > le contrôle de congestion, acceptant tout le trafic arrivant en conditions normales, et diminuant le débit ou supprimant des paquets lors de la survenue d'une congestion ; le mécanisme de fenêtre de congestion de TCP / IP fait partie de ces méthodes ;
  > le choix dynamique des routes, pour assurer la meilleure QdS possible en fonction de l'état courant d'un réseau : ce mécanisme dépend du protocole de routage de la pile réseau.

D'une façon générale, les méthodes de contrôle réactives semblent se montrer mieux adaptées aux réseaux sans-fil, notamment aux WSN, les méthodes proactives s'adaptant mal au médium radio et ses caractéristiques.

2.2.4 Niveaux de service

Le terme **niveau de service** (en anglais *Service level*) définit le niveau d'exigence pour la capacité d'un réseau à fournir un service point à point ou de bout en bout avec un trafic donné. On définit généralement trois niveaux de QdS :

Service garanti (en anglais *guaranteed service* ou *hard QoS*), consistant à réserver des ressources réseau pour certains types de flux. Le principal mécanisme utilisé pour obtenir un tel niveau de service est RSVP (*Resource reSerVation Protocol*, en français « Protocole de réservation de ressources »).

Service différencié (en anglais *differenciated service* ou *soft QoS*), permettant de définir des niveaux de priorité aux différents flux réseau sans toutefois fournir une garantie stricte.

Meilleur effort (en anglais *best effort*), ne fournissant aucune différenciation entre plusieurs flux réseaux et ne permettant aucune garantie. Ce niveau de service est ainsi parfois appelé (abusivement) *lack of QoS*.

2.2.5 QdS dans les réseaux de capteurs sans-fil

Les données venant d'être exposées jusqu'ici dans la présente section 2.2 concernent la QdS en général : elles s'appliquent à tous les types de réseaux.

Pour les réseaux de capteurs sans-fil, la nature même du réseau, avec des noeuds aux capacités très limitées, une contrainte énergétique très forte, un médium radio non-fiable, et une possibilité de défaillance de noeuds, rend les notions de service garanti ou différencié impossibles à atteindre. ***La Qualité de Service, dans le domaine des WSN, est donc quasiment toujours une garantie du type "best effort"***.

Concernant les critères de QdS importants et gérés au niveau des couches MAC / RDC des WSN, on s'intéresse principalement au ***taux de perte de paquets*** et au ***délais de transmission***. Les autres critères étant soit définis par les standards gérant le médium radio — par exemple : le débit (théorique) de 250 Kbit/s pour le standard 802.15.4 sur la bande 2,4 GHz —; soit dépendants de conditions externes sur lesquelles aucune intervention n'est possible (gigue, débit réel diminué par des interférences radio) ; soit réglés par les couches supérieures des piles réseau (déséquencement des paquets).

Enfin, le caractère extrêmement contraignant de la limitation énergétique fait de cette ***obligation d'économie d'énergie un critère à part entière de la QdS*** — bien que celui-ci soit contradictoire avec les autres critères de QdS. En effet, les *motes* étant souvent dans des situations où le changement de batterie est difficile ou même impossible, conserver la batterie opérationnelle le plus longtemps possible revient à maintenir le noeud correspondant « en vie » le plus longtemps possible. Une *mote* à

cours d'énergie est en effet souvent une *mote* perdue définitivement, donc une perte de fonctionnalité — potentiellement sévère — pour le WSN correspondant.

En résumé, on peut dire que pour les WSN, le lien entre Qualité de Service « fonctionnelle » (bande passante, fiabilité et rapidité des transmissions) et économie d'énergie est tout à la fois contradictoire, indissociable et crucial. L'optimisation de la QdS revient ainsi toujours à trouver ***le meilleur compromis, l'équilibre optimal*** entre tous les facteurs cités dans cette présente section 2.2.5, en fonction de l'application voulue.

Après toutes ces définitions techniques, nous allons maintenant dans la section 2.3 suivante passer en revue les différents domaines d'applications des WSN. Comme nous allons le voir, ceux-ci sont nombreux et variés.

2.3 Applications des WSN (et de l'IoT)

Akyildiz et Vuran, dans leur livre [Akyildiz and Vuran, 2010], considèrent cinq grands types d'applications aux WSN (voir figure 2.2) :

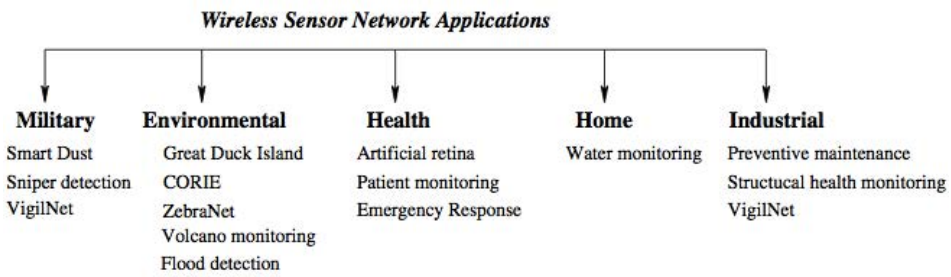

FIGURE 2.2 – *Principales catégories d'application des réseaux de capteurs sans-fil. Source : [Akyildiz and Vuran, 2010], figure 2.1.*

Applications militaires : les buts sont multiples, comme le contrôle des forces alliées, de leur équipement et de leurs munitions, la surveillance du champ de bataille, la reconnaissance de l'ennemi et du terrain, l'évaluation des dégâts, ou la détection d'attaques non conventionnelles. Plusieurs exemples sont cités dans [Akyildiz and Vuran, 2010] : le projet *Smart Dust* du DARPA (un pionnier dans les WSN), ou un système de détection de *sniper* [2].

Applications industrielles : ici aussi, les applications sont nombreuses : la supervision des processus de fabrication et la vérification de la qualité de production, la localisation et la surveillance de l'équipement industriel, la maintenance préventive des usines (surtout de grande taille) et des conditions de travail ou d'opération du matériel industriel, etc. Un exemple cité dans [Akyildiz and Vuran, 2010] est le projet FabApp [Krishnamurthy et al., 2005], dont le but est d'évaluer les vibrations du matériel industriel lors de son fonctionnement

2. Démontrations vidéos sur cette page Web : `http://bbn.com/boomerang`

pour prévenir pannes et accidents (tant dans une usine de fabrication de semi-conducteurs que dans un pétrolier).
On pourra également citer des applications de contrôle de l'état structurel des bâtiments, comme par exemple la surveillance active du *Golden Gate Bridge* (San Francisco) [Kim et al., 2007].
Citons aussi l'assistance au domaine minier : l'objectif étant de détecter les évènements dangereux pour les mineurs (poches de gaz, responsables de « coups de grisou ») ainsi que de quantifier les émissions de méthane produites par les mines de charbon (le méthane étant un gaz à effet de serre, celui-ci influe sur le changement climatique actuel).

Applications environnementales : de nombreux projets sont réalisables, comme le suivi des mouvements d'animaux, la détection de feux de forêts ou d'inondations, la recherche météorologique ou géophysique, l'étude de la pollution, etc. Plusieurs exemples sont cités dans [Akyildiz and Vuran, 2010], parmi lesquels : *ZebraNet* [Zhang et al., 2004] un système de surveillance au long cours de la population de zèbres, déployé au Kenya. Autre exemple : un système de surveillance des volcans [Werner-Allen et al., 2006] — également cité dans [Dargie and Poellabauer, 2010] — le but étant ici de détecter à l'avance les signes avant-coureurs des éruptions, pour mieux prévenir celles-ci, ou d'étudier le fonctionnement de volcans sous-marins, pour mieux comprendre les phénomènes d'activité volcanique. On peut aussi citer un système de détection précoce d'inondations dans les pays en voie de développement [Basha et al., 2008].
Les réseaux de capteurs sans-fil servent également à l'agriculture de précision, en permettant d'analyser précisément l'état des sols, afin de mieux utiliser les ressources à utiliser (eau, engrais, pesticides, herbicides) pour optimiser les récoltes. Plusieurs projets prototypes ont déjà été déployés dans ce but.
L'environnement urbain peut également en bénéficier, par exemple par des applications de contrôle actif du trafic routier aux États-Unis [Knaian, 2000], ou encore le projet PipeNet [Stoianov et al., 2007] mis en place dans de nombreuses villes américaines pour monitorer les réseaux d'égoûts.

Applications à la santé : le domaine de la santé fait partie du contexte de la présente thèse, et nous y reviendrons donc plus longuement dans une section suivante (section 2.4.1 page suivante).

Applications domotiques : On cite dans [Akyildiz and Vuran, 2010] un système nommé NAWMS [Kim et al., 2008b] destiné à détecter les gaspillages d'eau et leur origine précise, afin d'en informer les usagers pour les aider à réduire leur facture d'eau.

En étudiant les applications des WSN et de l'IoT citées dans [Dargie and Poellabauer, 2010] et [Akyildiz and Vuran, 2010], on voit que ces applications ne manquent pas, et sont appelées à continuer à se développer de façon exponentielle.

Nous allons maintenant dans la section 2.4 suivante nous focaliser sur le contexte de la présente thèse.

Ce contexte est celui du domaine d'application de la santé pour les WSN. Nous y étudierons tout particulièrement le projet LAR (*"Living Assistant Robot"*), dont cette thèse fait partie, dans la section 2.4.2.

2.4 Contexte

2.4.1 Applications d'e-santé des WSN

L'utilisation des réseaux de capteurs sans-fil pour des applications dans le domaine de la santé est un domaine déjà bien établi et très actif : les PAN (*Personal Area Network*) dédiés à être installés sur un patient — par exemple pour suivre son état de santé — sont même désignés par un acronyme spécifique : **BAN *("Body Area Network")***.

Les applications des WSN au domaine de la santé sont également, comme dans les autres domaines, très nombreuses et variées : un article complet de référence [Alemdar and Ersoy, 2010] recensait en 2010 les nombreux projets d'exploitation des réseaux de capteurs sans-fil liés au domaine de la santé.

Parmi les projets les plus ambitieux, les deux livres de [Dargie and Poellabauer, 2010] et de [Akyildiz and Vuran, 2010] citent tous deux des projets de rétines artificielles [Schwiebert et al., 2001] [*US Department of Energy*, 2011] combinant une caméra CCD externe couplée à des nano-noeuds sans-fil implantés ***in vivo*** sur la rétine du patient. Ces projets visent à apporter une solution à des maladies actuellement incurables, comme la DMLA (Dégénérescence Maculaire Liée à l'Âge) ou la rétinite pigmentaire (maladie génétique).

Sans aller jusqu'à des projets aussi avancés technologiquements (et organiquement invasifs) de nombreux projets de surveillance à domicile de patients fragiles et de réponse aux cas d'urgence sont actuellement mis en place.

Rappelons notamment l'existence du projet **d'informatique située**, consistant en une plate-forme d'« appartement intelligent pour l'assistance à la personne » [LORIA / INRIA Nancy Grand-Est, 2009], utilisant intensivement les WSN, que l'on peut classer à la fois dans les applications domotiques et les applications à la santé. Ce projet, hébergé au LORIA / INRIA Nancy Grand-Est, implique directement (entre autres) notre équipe — Madynes —, et l'un des encadrants de la présente thèse — Y.-Q. Song.

2.4.2 Le projet LAR

Un autre de ces projets fait également usage des WSN, combinant aussi — comme nous l'avons dit auparavant au chapitre 1 — aspect domotique et aspect santé, est le **projet LAR** (*"Living Assistant Robot"*), dont la présente thèse fait partie.

Le projet LAR a pour objectif d'aider au maintien à domicile des personnes âgées et / ou dépendantes, et de tenter de retarder le plus possible le moment où ces dernières doivent être placées en institutions spécialisées, ce qui représente un coût financier et surtout humain considérable.

Dans ce but, le projet vise à fournir des outils de suivi à domicile de la personne dépendante et de son activité, de développer pour cela des outils de perception, de modélisation de l'activité humaine, et d'interaction avec la personne. Comme le nom du projet l'indique, il est question de fournir (du moins dans le cas les plus lourds) un robot d'assistance spécialement conçu par le projet ; mais une grande partie du travail fourni (détection de présence, de mouvement, de chutes, etc.) sera fourni par un ou plusieurs WSN installés chez le patient, le robot étant également porteur de capteurs et actionneurs et donc membre à part entière de ce(s) WSN. Ces WSN sont également reliés à l'IoT, les données médicales recueillies étant destinées à être stockées dans des bases médicales spécialisées, accessibles de façon sécurisée aux personnels de soins (médecins, infirmiers, etc.).

Le principe central du projet LAR est la mise en œuvre d'un écosystème modulaire de services (comme nous venons de le voir : capteurs de données de toutes natures, robots, mais aussi services à la personne, etc.), lesquels services opèrent au sein d'une architecture orientée services (*"Service Oriented Architecture"* ou **SOA**) centrée sur un orchestrateur (alias *"Enterprise Message Bus"*) Microsoft BizTalk.

L'un des objectifs fondamentaux du projet est de commercialiser ce service au coût le plus faible possible, afin de le rendre accessible au plus grand nombre ; c'est pourquoi le recours à des WSN (avec leurs *motes* à faible coût) est un point clé du programme. L'une des idées directrices du projet est de rendre l'offre modulaire, et de ne fournir que les services nécessaires à l'état courant du patient : les cas les plus légers n'auront ainsi que l'installation d'un WSN « léger » et peu intrusif à leur domicile, l'évolution de l'état du patient pouvant ensuite amener à recourir à des modules supplémentaires ; le robot étant l'élément le plus évolué mais aussi le plus coûteux.

Plusieurs partenaires académiques et industriels sont impliqués dans ce projet : la partie académique est représentée par le LORIA / INRIA Nancy Grand-Est en l'équipe Madynes (dont nous faisons partie) pour les WSN, et l'équipe Larsen — anciennement Maia — pour la partie robotique et reconnaissance des mouvements ; les partenaires industriels sont Diatelic (pour la gestion de la base de données médicales et son accès sécurisé), Robosoft S.A. (pour la conception et la fabrication du robot), et le Pôle Innovation du Crédit Agricole S.A. (qui coordonne le projet). Tous ces partenaires ont formé un consortium — dont le Crédit Agricole a pris la tête — pour répondre à l'appel d'offres « e-santé » lancé en 2011 par la DGCIS — depuis devenue DGE (Direction Générale des Entreprises) —, lequel consortium a remporté l'appel et ainsi fondé le projet LAR. Le contrat correspondant a été signé en 2012, et le projet LAR a été financé par la Banque Publique d'Investissement (BPIFrance).

2.5 Problématique

2.5.1 Exposé de la problématique

Dans le domaine des réseaux de capteurs sans-fil, les piles réseau spécialisées constituent un domaine de recherche très actif depuis maintenant une quinzaine d'années. On citera actuellement, par exemple dans le domaine des couches hautes, les protocoles de routage (RPL et ses nombreuses variantes et évolutions), ou encore l'activité actuelle autour des CCN (*"Content-Centric Networks"*).

Pour les couches basses, de nombreux travaux de recherche ont également été effectués sur les protocoles MAC (la couche MAC représentant le niveau 2 dans le modèle OSI), pour suppléer le protocole du standard 802.15.4 et ses limitations, comme nous le verrons au chapitre 3 page 25. Toutefois, beaucoup de ces études se sont principalement consacrées à l'étude théorique de ces protocoles, se contentant souvent — à des fins de tests — de simulations puremement théoriques (avec des outils comme OPNET, ns-3, OMNeT++, voire même MATLAB), ou au mieux par des émulations de *motes* virtuelles (avec des outils comme TOSSIM [Levis et al., 2003] ou Cooja [Österlind et al., 2006]).

L'implantation de ces couches basses, dans les plates-formes logicielles réelles — c'est à dire les systèmes d'exploitation dédiés aux réseaux de capteurs sans-fil — n'a pas fait l'objet d'un effort de recherche poussé et systématique. Le principal but était de minimiser la consommation d'énergie de ces couches basses, généralement en ayant recours aux protocoles et aux implantations les plus simples possibles.

L'un des seuls efforts remarquables de conception et d'implantation dans ce domaine, le protocole ContikiMAC — développé spécifiquement pour la plate-forme logicielle Contiki OS — a ainsi comblé un manque, et est devenu le standard de fait, du moins dans la littérature sur le domaine des réseaux de capteurs sans-fil. Depuis, on ne peut qu'observer un certain *statu quo* dans ce domaine. Si de nombreux protocoles MAC à hautes performances ont été conçus (comme nous le verrons dans la section 3.2 page 34 de l'état de l'art), aucun d'entre eux n'a été implanté et largement diffusé avec un OS spécialisé comme l'est ContikiMAC.

2.5.2 Stratégie préconisée

2.5.2.1 Besoins identifiés

Pour avancer dans ce domaine, et notamment pouvoir exploiter les nouvelles *motes* plus puissantes — par exemple celles basées sur des microcontrôleurs à base ARM Cortex-M — rencontrées de plus en plus souvent (en sus des plates-formes classiques à base MSP430 ou AVR), nous voyons plusieurs besoins à combler :

1. ***Faire évoluer les plates-formes logicielles (OS) spécialisées dans les réseaux de capteurs sans-fil, en leur ajoutant des fonctionnalités nécessaires au développement de piles réseau plus performantes.***
   La programmation sans OS ni plate-forme d'aucune sorte (programmation *"bare metal"*) nous semble à proscrire, car les logiciels ainsi créés ne sont nullement

portables, et doivent être réécrits à chaque changement de matériel, lequel dans ce domaine (comme dans toute l'électronique et l'informatique) évolue très vite. L'utilisation d'une plate-forme logicielle facilite aussi la programmation d'applications, ainsi que les travaux d'évaluation et de comparaison.

2. ***Expérimenter, de façon intensive et approfondie, ces plates-formes logicielles spécialisées et leurs piles réseau*** afin de les optimiser et de les fiabiliser, tout spécialement leurs couches basses.

3. ***Tester le comportement et la résilience de ces piles réseau spécialisées (et notamment de leurs couches basses) face à des charges réseau intenses.*** En effet, la plupart des réseaux de capteurs sans-fil actuels sont conçus pour gérer des trafics faibles à modérés. ***Lors de situations exceptionnelles — tout spécialement dans le domaine médical — on peut imaginer que ces WSN puissent avoir à gérer des trafics intenses, que ce soit de façon ponctuelle (par exemple : gestion d'un patient à domicile en situation d'urgence vitale) ou continue (comme dans une maison médicalisée ou un hôpital où de nombreux WSN traiteront constamment et de façon concurrente de fortes quantités de données).*** Nous souhaitons dans cette thèse évaluer, et si nécessaire améliorer et optimiser, le fonctionnement des couches basses — niveaux 1 et surtout 2 du modèle OSI — de ces piles réseau spécialisées face à de telles situations.

Nous nous proposons donc, dans cette thèse, de nous pencher sur les implantations de ces couches basses, au sein d'un ou plusieurs systèmes d'exploitation spécialisés offrant les fonctionnalités nécessaires, et de les optimiser. Nous souhaitons pour cela utiliser l'approche décrite dans la section 2.5.2.2 suivante.

2.5.2.2 Approche mise en œuvre

Lorsque l'on programme les appareils miniatures qui constituent les noeuds des WSN, il est nécessaire de s'adapter aux spécificités et aux limitations de ces appareils. Ainsi, outre leur puissance très limitée, le principal objectif lorsque l'on programme des *motes* est de réduire autant que possible leur consommation énergétique. Le but est de faire durer leur(s) batterie(s) aussi longtemps que possible, pour des raisons économiques mais également pratiques : il est parfois difficile — et même quasiment impossible — de changer les batteries de certaines de ces *motes*, à cause de leur localisation (par exemple : en haut d'immeubles, sous des routes, etc).

De tous les différents composants constituant une *mote*, l'élément le plus consommateur d'énergie est, de loin, l'émetteur / récepteur radio (comparer, par exemple, les consommations énergétiques d'un émetteur / récepteur radio comme le TI CC2420 [DataSheet CC2420, 2007] et d'un microcontrôleur comme le MSP430F1611 [DataSheet MSP430F1611, 2011], qui sont les deux principaux composants de la famille répandue de motes que sont les TelosB / SkyMotes [DataSheet TelosB, 2006]). En conséquence, pour limiter la consommation énergétique de ces appareils, un premier point-clé est d'utiliser cet émetteur / récepteur radio uniquement quand cela est

nécessaire, en le gardant éteint — ou en mode « sommeil » aussi souvent que possible. L'élément logiciel responsable de contrôler cet émetteur / récepteur radio de façon adéquate est le niveau 2 de la pile réseau logicielle, constituant les couches **MAC** (*Media Access Control*) et **RDC** (*Radio Duty Cycle*). Cela implique aussi le plus souvent d'intervenir sur la couche de niveau 1 : la couche **PHY** ou physique, c'est-à-dire le pilote de l'émetteur / récepteur radio, afin que ce dernier fournisse les fonctionnalités nécessaires aux couches supérieures (notamment la couche MAC / RDC) de la façon la plus efficace possible.

Une stratégie efficace d'économie d'énergie pour les motes repose ainsi sur la recherche du meilleur compromis entre d'une part la réduction de la fraction de temps durant laquelle la radio est active ("duty cycle"), et d'autre part le maintien de la plus haute efficacité possible du réseau sans-fil (QdS). Ce compromis s'atteint en développant de nouveaux protocoles MAC / RDC « intelligents », s'adaptant dynamiquement et automatiquement au trafic réseau en cours d'exécution.

Pour implanter de nouveaux protocoles MAC / RDC à hautes performances, il est nécessaire de pouvoir réagir aux évènements — par exemple : expirations de délais ou arrivées de paquets, prenant souvent au niveau de la programmation la forme d'interruptions — avec une bonne réactivité (latence la plus faible possible) et avec flexibilité. De tels protocoles reposent sur un *timing* précis pour assurer une synchronisation efficace entre les différentes *motes* et autres appareils équipés de radio amenés à faire partie des PANs, permettant ainsi de faire fonctionner les émetteurs / recepteurs radio de ces appareils ***uniquement*** lorsque cela est nécessaire.

Le deuxième élément le plus consommateur d'énergie dans une *mote*, après l'émetteur / récepteur radio, est le microcontrôleur (MCU) au coeur de cette *mote*. Tous les MCUs actuels offrent des « modes à basse consommation d'énergie », consistant à « éteindre » à la demande les différents circuits intégrés au MCU, à commencer par le coeur CPU lui-même. Le principal moyen de minimiser la consommation d'énergie d'un MCU est ainsi de désactiver ses fonctionnalités selon les besoins du moment, en ne les réactivant que quand elles sont nécessaires au fonctionnement courant de l'application : cela revient effectivement à mettre le MCU en sommeil aussi souvent et aussi complètement que possible, tout en continuant d'exécuter de façon optimale l'application voulue.

Rappelons toutefois que le contexte de notre thèse (applications médicales) nous poussera toujours à priviligier d'abord la QdS, même si l'optimisation de la consommation d'énergie doit en souffrir. Cette approche est constante dans ce travail de thèse ; elle est notamment ce qui nous pousse à tester les couches basses des piles réseau en situation de trafic intense.

Dans les deux cas, pour la gestion de l'émetteur / récepteur radio comme pour celle du MCU, il est nécessaire de pouvoir utiliser les *timers* matériels et les interruptions de façon optimale, pour pouvoir mettre en sommeil et réactiver ces circuits de manière efficace afin d'atteindre le meilleur compromis entre QdS et économies d'énergie, là encore de façon dynamique et automatique.

Être capable d'utiliser les *timers* et les interruptions de façon efficace et avec un minimum de difficultés implique l'utilisation d'un système d'exploitation (OS) spécialisé dans les capteurs sans-fil — une plate-forme logicielle adaptée — qui nous offrira en outre les bénéfices de la portabilité de nos applications, et des fonctionnalités multitâches facilitant la programmation (revoir le premier point abordé plus haut dans la section 2.5.2.1 page 21).

La recherche de cette plate-forme logicielle la mieux adaptée à nos besoins — outre une section consacrée aux OS spécialisés dans le chapitre sur l'état de l'art (section 3.3 page 61) — fera l'objet d'un chapitre dédié (chapitre 4).

Mais auparavant, nous allons nous pencher sur l'état de l'art concernant les domaines concernés par cette thèse dans le chapitre 3 suivant.

Chapitre 3

Analyse critique de l'état de l'art

Nos travaux ont, d'un point de vue pratique, principalement concerné les couches basses des piles réseau de systèmes d'exploitation spécialisés pour les réseaux de capteurs sans-fil. Ces réseaux fonctionnent à l'heure actuelle principalement avec le protocole IEEE 802.15.4 [IEEE 802.15.4, 2011], même si l'utilisation de technologies alternatives (tel le BLE — *Bluetooth Low Energy* également appelé *Bluetooth Smart*) commence à émerger.

Notre objectif était de réaliser et d'améliorer l'implantation logicielle de la couche MAC (*Medium Access Control*, protocole d'accès au médium, en l'occurence le canal radio 802.15.4 voulu). Pour des raisons techniques évidentes, nous avons également été amenés à travailler sur la couche PHY logicielle sous-jacente (consistant en les pilotes des émetteurs / récepteurs radio des appareils concernés), ainsi que sur l'interface entre ces deux couches.

Le présent chapitre va ainsi s'articuler en 3 sections :

— la première va brièvement rappeler les bases du protocole IEEE 802.15.4, dont la couche physique a servi de base à tous nos travaux ;

— la seconde résumera les différentes familles et technologies des protocoles MAC applicables aux réseaux 802.15.4 ;

— quant à la troisième et dernière, elle reprendra les différents systèmes d'exploitation spécialisés dans les systèmes embarqués et notamment dans les réseaux de capteurs sans-fil. Cette troisième section sera plus analytique et critique, et s'intéressera particulièrement aux fonctions offertes par ces systèmes aux programmeurs, tout spécialement pour le développement des couches bases des piles réseau.

Aucune de ces sections ne prétend à l'exhaustivité, mais cherche à donner les données nécessaires et suffisantes pour comprendre les bases à partir desquelles nous avons effectué nos travaux.

3.1 Le protocole IEEE 802.15.4

Le standard IEEE 802.15.4 définit les couches basses (PHYsique et MAC) de réseaux sans-fil destinés à relier des appareils sur une très faible distance et avec un

faible débit (on parle de LR-WPAN : *Low-Rate Wireless Personal Area Network*, en contrepartie d'une consommation d'énergie très modeste.

Apparu dans sa première version en 2003, il a été révisé en 2006 puis en 2011, et complété par divers amendements (802.15.4a, 4b, 4c, 4d et à l'heure actuelle 4e).

Ce standard ne définissant que les deux couches les plus basses de la pile réseau, il est le plus souvent utilisé de concert avec d'autres protocoles définissant les couches hautes. On peut notamment utiliser des piles protocolaires basées sur IP (*Internet Protocol*) — telle que 6LoWPAN [Mulligan, 2007] — ou non — comme ZigBee. Toutes ces piles réseau reposent ainsi sur le standard 802.15.4.

La figure 3.1 résume de façon schématique la comparaison entre le standard IEEE 802.15.4, et les deux piles de protocoles réseaux complètes et couramment utilisées que sont le modèle OSI et le modèle Internet (souvent appelé abusivement « TCP/IP »).

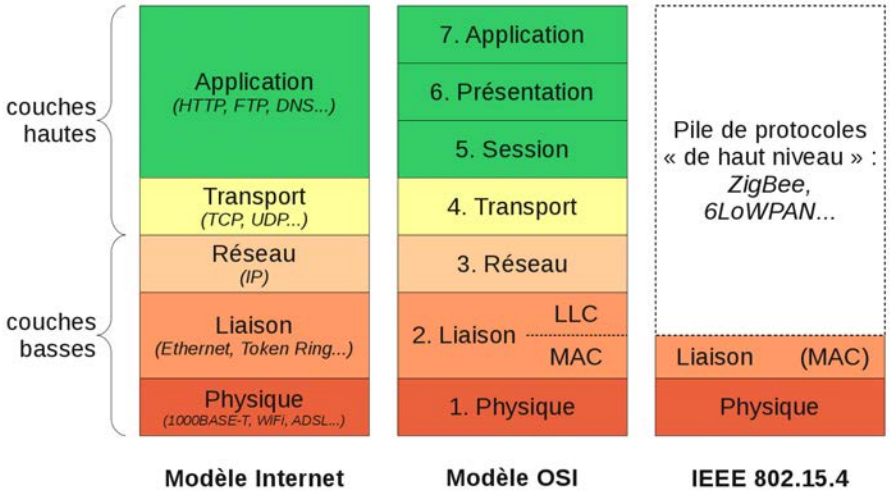

FIGURE 3.1 – *Comparaison entre la pile 802.15.4 et les piles OSI et Internet.*

3.1.1 Couche physique

La couche physique (PHY) 802.15.4 offre les fonctionnalités suivantes :

Bandes de fréquences. De base, le standard offre trois bandes : une fréquence (canal) unique à 868 MHz disponible en Europe ; une bande autour de 915 MHz offrant d'abord 10 canaux (puis étendue à 30) disponible en Amérique du Nord ; et la bande ISM de 2,4 GHz offrant 16 canaux et disponible dans le monde entier.

À l'origine (2003), seule la bande à 2,4 GHz permettait d'atteindre le débit maximal de 250 kbps, les basses fréquences (868 et 915 MHz) étaient elles limitées à de faibles débits (20 à 40 kbps) en contrepartie d'une portée plus grande (due à un moindre affaiblissement). Le choix d'une bande de fréquence

correspondait ainsi à un compromis entre débit et portée, en sus d'un choix dicté par des considérations géographiques.

La révision de 2006, grâce à l'introduction de nouvelles méthodes de modulation du signal radio, a permis aux bandes à 868 et 915 MHz d'atteindre des débits de 100 et 250 kbps. Le choix d'une bande radio dépend donc désormais surtout du déploiement géographique envisagé pour les réseaux considérés.

En outre, les différents amendements ont ajouté de nouvelles bandes de fréquences disponibles (315, 430 et 780 MHz disponibles en Chine ; 950 MHz disponible au Japon). Ces mêmes amendements apportèrent également des méthodes de modulation de signal supplémentaires, permettant notamment l'utilisation de bandes à très hautes fréquences (jusqu'à 10 GHz).

L'amendement 802.15.4a a égalemement ajouté la notion d'*Ultra-Wide Band* (UWB), lequel offre, outre de plus hauts débits, la possibilité de mesurer notamment précisément les temps de vol (c-à-d. les délais de communication entre deux noeuds).

Les différents émetteurs / récepteurs radio (RF) à la norme 802.15.4 ne peuvent pas fonctionner sur toutes les fréquences du standard : leur choix dépend donc de la bande de fréquences souhaitée. Parmi les plus utilisés à l'heure actuelle, on peut par exemple citer :

— La famille Texas Instruments (TI) ChipCon (CC) 1000/1100. Cette famille regroupe des émetteurs / récepteurs pour les bandes à 868 et 915 MHz.
— La famille TI CC 2400/2500, regroupant des émetteurs / récepteurs pour la bande à 2,4 GHz.
— La famille Atmel AT86RF230, regroupant également des émetteurs / récepteurs pour la bande à 2,4 GHz.

Nous avons au cours de nos travaux utilisé des appareils dotés de composants radio appartenant aux deux dernières familles citées (CC2420 et AT86RF231/233), et fonctionnant donc sur la seule bande ISM à 2,4 GHz.

Notons également que l'on trouve de plus en plus fréquemment des microcontrôleurs intégrant directement un émetteur / récepteur radio dans la même puce : par exemple, le STM32W108 de ST Microelectronics, le SAMR21 ou la famille ATmegaRFR2 [DataSheet ATmegaRFR2, 2014] d'Atmel.

Mode de liaison. Le couche physique du standard IEEE 802.15.4 est une ***liaison half-duplex***. Cela signifie que si la communication entre deux appareils à la norme IEEE 802.15.4 est bien ***bidirectionnelle*** — chaque appareil peut assumer le rôle d'émetteur ***et*** celui de récepteur — il est ***impossible à une seule radio au standard IEEE 802.15.4 d'émettre et de recevoir simultanément***. La transmission d'informations entre deux *motes* dans les réseaux de capteurs sans-fil se fait donc ***de façon alternée***, chaque noeud pouvant à chaque instant donné soit émettre, soit recevoir des données ; mais jamais les deux en même temps (si le noeud ne possède qu'un seul émetteur / récepteur radio).

L'un des rôles majeurs de la couche directement supérieure (MAC) est justement de déterminer à quels instants la radio doit être en mode émission, et à

quels autres instants en réception — et aussi éventuellement à quels instants elle est désactivée (pour économiser de l'énergie).

Notons que ce fonctionnement ***half-duplex*** n'est pas propre à ce standard 802.15.4. Le standard IEEE 802.11 (plus connu sous le nom de *"WiFi"*) est lui aussi un standard *half-duplex*. Ce mode de fonctionnement est en fait mieux adapté aux communications sans-fil, utilisant le médium radio, lequel est par nature sujet à la diffusion, aux variations rapides d'état et à d'autres facteurs nuisant à la qualité de transmission. (Les réseaux câblés sont la plupart du temps en *full-duplex*, c-à-d. peuvent transmettre et recevoir simultanément, généralement en consacrant — au moins — un fil à chaque sens de communication.)

Ce mode de fonctionnement *half-duplex* a des conséquences pratiques : un émetteur / récepteur radio doit régulièrement passer du mode émission au mode réception et inversement. Une telle procédure n'est pas instantanée, chaque puce radio ayant un délai de retournement indiqué dans sa *datasheet*. Le standard IEEE 802.15.4 — dans sa version de 2011 — indique un délai maximal pour passer du mode émission au mode réception et inversement, que toutes les radios conformes au standard ne sont pas censées dépasser : constante ***aTurnaroundTime***, égale à 12 symboles (soit 192 μsec. pour la plupart des puces radio émettant sur la bande de 2,4 GHz).

Format de trames. Le standard définit un format — très simple — de trame physique (*"frame"*), dont la taille est limitée à 127 octets. Ces 127 octets incluant les entêtes des couches supérieures (dont la couche MAC), la taille disponible pour les données proprement dites (charge utile ou *"payload"*) est en général nettement inférieure.

Modes d'adressage. Deux modes d'adressage complémentaires sont pris en charge :
- un adressage court sur 16 bits (plus un identifiant de réseau — identifiant PAN — de 16 bits également) dont l'intérêt est de limiter la taille des entêtes de trames ;
- un adressage long (dit « adressage IEEE ») sur 64 bits permettant *a priori* directement une identification unique de chaque appareil.

Détection du médium. Le standard inclut la détection d'énergie (ED : *Energy Detection*) sur le médium radio, et par extension la vérification de la disponibilité du canal radio (CCA : *Clear Channel Assessment*).

Qualité de service. Le standard prend en charge la définition et l'indication de la qualité de liaison (LQI : *Link Quality Indicator*) pour une transmission de trame donnée. Le RSSI (*Received Signal Strength Indicator*), s'il n'est quévoqué dans le glossaire du standard 802.15.4, est également la plupart du temps pris en charge au niveau physique (émetteurs / récepteurs radio).

Toutes ces propriétés sont implantées de façon matérielle par les différents émetteurs / récepteurs radio conformes au standard 802.15.4 — qu'il s'agisse de composants autonomes, ou de circuits intégrés à des microcontrôleurs ou *System-on-Chip* (SoC).

La couche PHY, au niveau logiciel, notamment dans un système d'exploitation spécialisé, consiste donc en un jeu de primitives (implantées dans des pilotes) permettant d'exploiter ces émetteurs / récepteurs radio. On peut envisager cette couche comme une couche d'abstraction matérielle (HAL : *Hardware Abstraction Layer*) vis-à-vis de la couche MAC et des couches supérieures de la pile réseau du système.

3.1.2 Couche MAC

Située immédiatement au-dessus de la couche physique, la couche MAC du standard 802.15.4 repose sur la méthode CSMA/CA (*Carrier Sense Multiple Access with Collision Avoidance*). Une version différente de cette technologie est déjà utilisée (sous le même nom) dans le standard IEEE 802.11 (alias *"WiFi"*).

La couche MAC standard peut utiliser cette méthode selon deux modalités différentes, selon l'utilisation ou non de *"beacons"* (ou balises [1]) pour synchroniser les différents noeuds et identifier les différents réseaux (PAN). Sans *"beacons"*, la couche MAC standard emploie toujours des cycles de fonctionnement (*"duty cycles"*) fixes, sans prendre en compte l'utilisation réelle du réseau et le débit des données transmises. Le mode « avec *beacons* » permet d'ajuster certains paramètres de fonctionnement [Khssibi, 2015]. Certains éléments restent toutefois immuables, même dans ce mode (par exemple les 16 *slots* de temps dans la *superframe*).

Le standard 802.15.4 est à la base conçu pour faire fonctionner des réseaux ayant une topologie en étoile ; pour cela, le standard permet de différencier les noeuds en deux types :

— les noeuds à fonctionnalités complètes (FFD : *Full Function Devices*), capables de jouer aussi bien le rôle de « noeud simple » occupant une des extrêmités du réseau, que celui plus évolué de coordinateur de réseau (en mode « avec *beacons* ») ;
— les noeuds à fonctionnalités réduites (RFD : *Reduced Function Devices*), limités au rôle de « noeuds simples »

Dans notre terminologie, un « noeud simple » est un noeud dont la défaillance ne met pas en péril le fonctionnement de l'ensemble, au contraire d'un routeur ou d'un coordinateur dont la seule panne entraîne la perte de tout un réseau.

Un exemple de base d'un schéma topologique d'un réseau de capteurs sans-fil (WSN) est montré figure 3.2 page suivante. Celui présenté ici est composé de quatre sous-réseaux (PAN) différents, possédant chacun un routeur jouant également le rôle de coordinateur de son PAN. Les noeuds simples (« feuilles » de l'arborescence) ne communiquent qu'avec leur propre coordinateur, ces derniers se chargeant de transmettre les données de PAN en PAN de façon adéquate.

Ces réseaux (PAN) peuvent eux-mêmes être reliés entre eux de façon maillée et/ou arborescente (selon les protocoles de routage utilisés par dessus 802.15.4).

1. Pour des raisons pratiques, nous emploierons dans la suite de ce manuscrit le terme anglo-saxon *"beacon"*, plus utilisé dans la littérature.

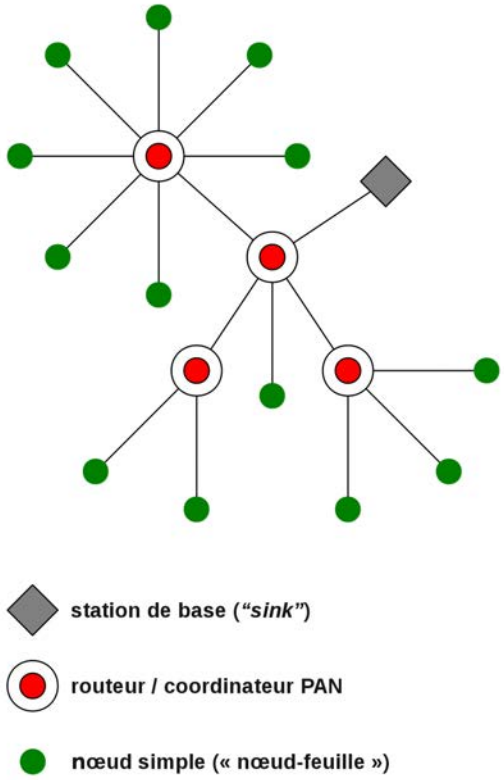

FIGURE 3.2 – *Schéma topologique d'un réseau de capteurs sans-fil (WSN). (D'après [Song, 2013])*

3.1.2.1 CSMA/CA

Cette méthode consiste à éviter les collisions potentielles sur le médium en vérifiant préalablement la disponibilité du canal radio et, en cas d'encombrement, en attendant pendant un délai aléatoire avant de retenter une écoute.

Plus précisément, l'implantation de CSMA/CA dans le standard 802.15.4 en version de base « non slottée » est la suivante :

1. un noeud souhaitant émettre une trame doit d'abord attendre pendant un délai dont la durée est ***aléatoirement*** choisie entre 0 et $2^{BE} - 1$ unités de temps nommées **BP** (*"Backoff Period"*) . La valeur d'une BP est une constante dépendant de la bande choisie pour la couche physique (elle vaut 320 microsecondes pour la bande ISM à 2,4 GHz). La valeur BE est nommée *"Backoff Exponent"*, et est initialisée à une valeur définie dans le standard sous le nom de ***macMinBE*** (3 par défaut).
2. une procédure de CCA est alors effectuée : le CCA consiste à écouter le medium radio durant un délai spécifique — nommé DIFS (*Distributed Inter-Frame Space*) —, si le medium reste libre pendant ce délai, le noeud peut alors émettre sa trame ;

3. si le médium radio est encombré, on revient à la première étape d'attente d'un délai aléatoire, en ayant toutefois incrémenté la valeur de BE, dans la limite de la valeur maximale ***macMaxBE*** définie par le standard (5 par défaut). Toutefois, au bout d'un nombre maximal d'essais, défini par le standard sous le nom de ***macMaxCSMABackoffs*** (4 par défaut), l'envoi de la trame est considéré comme échoué par la couche MAC.

La méthode CSMA/CA peut également fonctionner en mode « slotté », de façon à s'adapter aux protocoles MAC où les transmissions doivent se caler sur des intervalles de temps (*"slots"*) bien définis. Dans ce cas, un nouveau paramètre, nommé *"Contention Window"* (***CW***) intervient pour s'assurer que le médium radio a été détecté libre (CCA) un certain nombre de fois avant de commencer une émission ; cette étape supplémentaire a notamment pour but de faciliter la transmission des messages d'acquittement signalant la bonne réception d'une trame (ACK). Par défaut, le standard fixe la valeur initiale de CW (CW_0) à 2.

L'organigramme décrivant le fonctionnement de la méthode CSMA/CA est schématisé dans la figure 3.3 page suivante. Dans cette figure, la branche de gauche présente la version « slottée », employée notamment dans le mode *"beacon"* du protocole MAC du standard IEEE 802.15.4 ; tandis que la branche de droite présente la version simple « non slottée » qui est en général utilisée pour les protocoles non synchronisés (tels que LPL, LPP, cf. suite du chapitre).

En sus de la méthode CSMA/CA, notons qu'il existe également un mécanisme obligatoire d'acquittement des trames. Ainsi, à la réception de chaque trame de données, le noeud de destination doit renvoyer un acquittement à l'envoyeur ; si après envoi d'une trame de données, le noeud émetteur ne reçoit pas d'acquittement dans un délai voulu (*"timeout"*), l'envoi de la trame est considéré comme un échec.

3.1.2.2 Mode *"non-beacon"*

Le mode de fonctionnement le plus simple de la couche MAC du standard 802.15.4 repose exclusivement sur l'emploi de la méthode CSMA/CA « non slottée » décrite section 3.1.2.1 page ci-contre. Il n'y a aucune notion de cycle de fonctionnement : le coordinateur est constammant en fonctionnement (la plupart du temps en écoute), tandis que les noeuds simples n'accèdent au médium radio que lorsqu'ils ont besoin d'envoyer ou recevoir des données.

La réception de données s'effectue grâce à un bit nommé *"Frame Pending"* de l'entête MAC 802.15.4, via lequel le coordinateur indique à un de ses noeuds-feuilles qu'il attend de lui envoyer une trame lui étant destinée. Le noeud destinataire peut alors envoyer une trame spéciale dite « de commande » (*"Data Request"*) pour déclencher la réception proprement dite.

Ce mode est bien adapté aux réseaux dans lesquels les noeuds simples émettent des données de façon sporadique (c'est-à-dire passent la quasi-totalité de leur temps en « sommeil »), et où les données transmises n'ont pas un caractère urgent (ce mode sans *"beacon"* n'offrant aucune garantie d'accès au canal pour une période donnée).

Il est ainsi possible d'avoir des noeuds simples fonctionnant sur une batterie

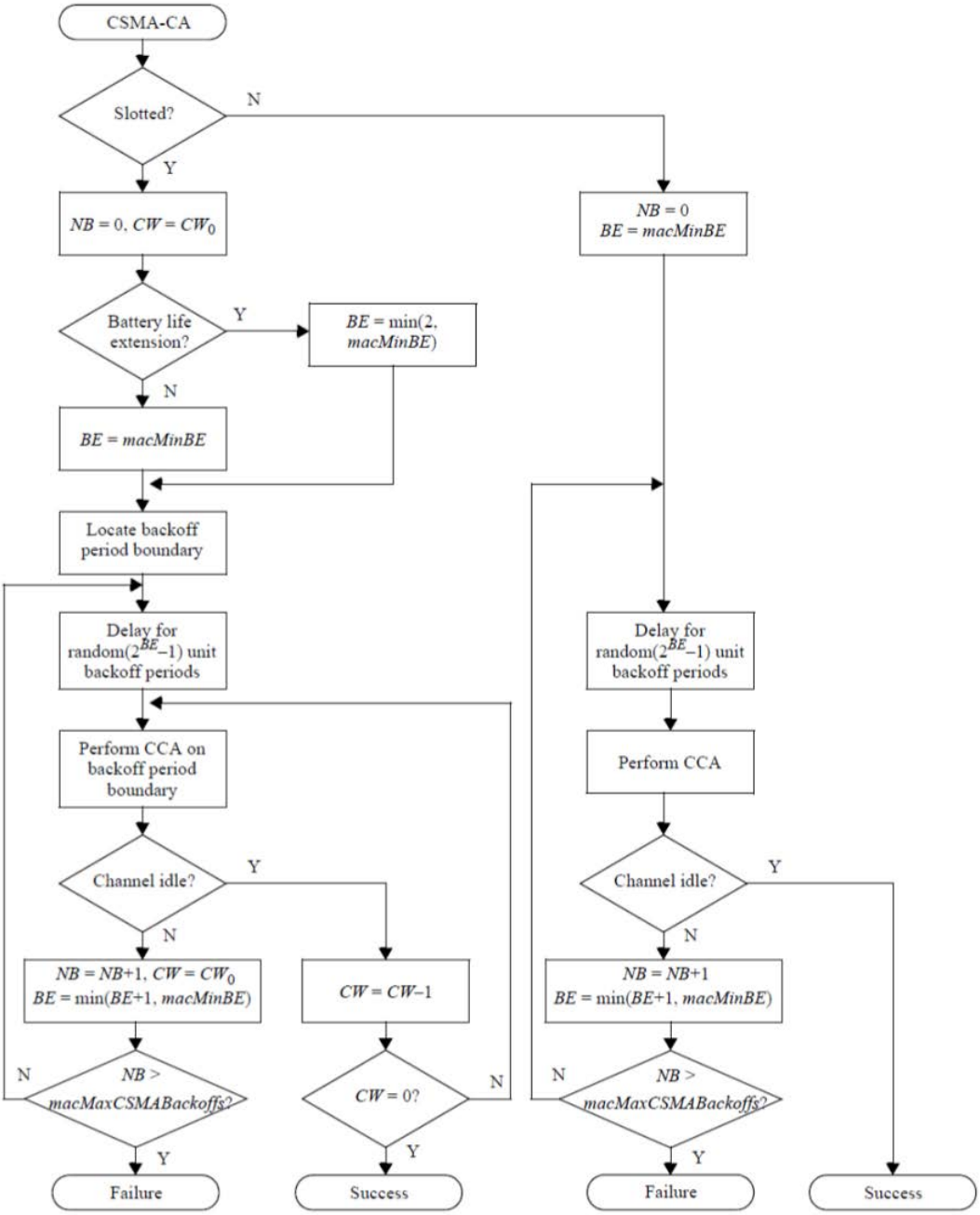

FIGURE 3.3 – *Organigramme de l'algorithme de la méthode CSMA/CA. (Source : [IEEE 802.15.4, 2011], figure 11)*

dont l'autonomie pourra être longue. Par contre, le noeud central jouant le rôle de coordinateur doit lui être constamment à l'écoute, ce qui impose la contrainte de le faire fonctionner sur une source d'énergie constante (secteur).

3.1.2.3 Mode *"beacon"* (ou mode balisé)

Le second mode de fonctionnement de la couche MAC standard impose au noeud central, jouant le rôle de coordinateur de PAN (compatible avec celui de routeur, ce qui représente l'immense majorité des cas), l'envoi régulier de *"beacons"* à chaque début de cycle de fonctionnement (*"duty cycle"*).

Ces *"beacons"* servent à synchroniser les différents noeuds du PAN (ainsi qu'à identifier ce dernier).

Dans ce mode, un cycle de la couche se décompose en :

l'envoi d'un *"beacon"* par le noeud coordinateur du PAN, cet envoi n'utilisant pas la méthode CSMA/CA, mais la diffusion directe à tous les noeuds à l'écoute (*"broadcast"*), aucun autre noeud n'étant en effet censé être autorisé à émettre spontanément hors cycle ; le *"beacon"* est, dans ce mode, en général la trame comportant le fameux bit *"Frame Pending"* pour permettre l'envoi de données aux noeuds-feuilles ;

une CAP (*Contention Access Period*) période durant laquelle les noeuds simples émettent vers le noeud central, ou recoivent les données leur étant destinées depuis celui-ci, en mode CSMA/CA ;

une CFP (*Contention Free Period*) période durant laquelle il est possible de garantir l'accès au médium radio pour un noeud (ou plusieurs noeuds consécutifs) ;

une période de sommeil durant laquelle le PAN est mis en inactivité ce qui permet aux différents noeuds, y compris le noeud central, de désactiver leur émetteur / récepteur radio pour économiser leur énergie.

L'ensemble constitué par la CAP et la CFP d'un cycle donné constitue la **supertrame** (en anglais *"superframe"*, par opposition à la periode de sommeil).

Le cycle de fonctionnement du protocole MAC standard 802.15.4 en mode *"beacon"* est représenté dans la figure 3.4.

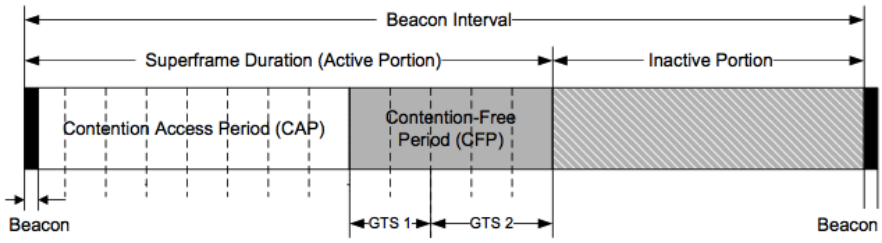

FIGURE 3.4 – *Cycle de fonctionnement du protocole MAC du standard IEEE 802.15.4 en mode "beacon". (Source : [Huang et al., 2013])*

Cette supertrame est divisée en 16 *"time slots"*, lequels sont répartis entre CAP et CFP par le noeud coordinateur à chaque nouveau cycle, en fonction des besoins du trafic réseau. Les *"time slots"* alloués à la CFP sont nommés **GTS** (*Guaranteed Time Slots*) ; le standard définit la procédure permettant à un noeud simple de demander au noeud coordinateur de PAN de lui allouer un ou plusieurs GTS au cycle suivant.

On voit ainsi que ce mode offre plusieurs avantages par rapport au mode *"non-beacon"* :

— le noeud coordinateur peut établir à lui seul son réseau (PAN) ;
— la présence d'une CFP permet de prévoir l'envoi de données importantes, dont le transport nécessite un accès garanti au médium radio sans risque de collision ;
— la CFP permet également d'envisager l'envoi de lots de trames successifs (pour la transmission de données de grande taille), de façon plus robuste que par l'utilisation de la méthode CSMA/CA.

On voit que la division de la période d'activité (*superframe*) en CAP et CFP est un moyen d'adapter le fonctionnement de la couche MAC au trafic sur le médium radio. Par défaut, cette adaptabilité est limitée par le fait que la durée d'un cycle, de la *superframe*, et donc le rapport entre ces deux durées, sont des constantes fixées lors de la configuration du PAN. Des travaux ont récemment été menés pour adapter dynamiquement ces paramètres au trafic réseau [Khssibi, 2015], le standard ne le prévoyant pas, mais ne l'interdisant pas non plus.

3.1.2.4 802.15.4e

Le standard IEEE 802.15.4 continue à évoluer, et les groupes de travail dédiés de l'IEEE ne cessent de le réviser et de le compléter — via des amendements — au cours du temps.

Les amendements 802.15.4a, 4b, 4c et 4d ont successivement amené de nouvelles bandes de fréquences et méthodes d'encodage du signal radio, l'amendement IEEE 802.15.4a ayant notamment amené la notion d'UWB, comme dit plus haut en section 3.1.1 page 26

Un amendement récent au standard, l'amendement 802.15.4e, adopté en 2012, a ajouté une nouvelle couche MAC nettement plus complexe, incluant notamment des mécanismes de multiplexage temporel ***et*** fréquentiel des transmissions radio. Nous reviendrons sur cet amendement 802.15.4e dans la section 3.2.5 page 45 consacrée aux protocoles MAC multicanaux.

3.2 Protocoles MAC

Outre les couches MAC « officielles » proposées par le standard 802.15.4, la communauté scientifique a proposé de nombreux protocoles alternatifs destinés à surpasser les limitations du protocole standard, notamment sa version simple reposant sur la seule méthode CSMA/CA.

Contrairement au mode simple, et de façon similaire au mode balisé décrit section 3.1.2.3 page 33, ces protocoles MAC alternatifs reposent sur la notion de *"duty cycle"*. Toute la difficulté pour la mise au point de ces protocoles consiste donc à fixer des ***points de rendez-vous*** entre les différentes noeuds pour assurer correctement leur synchronisation.

On peut ainsi diviser ces divers protocoles en différentes familles, selon les méthodes utilisées pour fixer ces points de rendez-vous :

— les protocoles MAC synchrones, employant des mécanismes de synchronisation explicites entre noeuds pour la transmission de trames : S-MAC et T-MAC en sont deux exemples ;
— les protocoles MAC asynchrones basés sur l'écoute à basse énergie (LPL : *Low Power Listening*) : B-MAC, X-MAC et ContikiMAC en sont trois exemples (ce dernier étant aujourd'hui très largement utilisé) ;
— les protocoles MAC asynchrones basés sur l'émission à basse énergie (LPP : *Low Power Probing*) : RI-MAC en étant l'exemple le plus connu ;
— les protocoles MAC basés sur l'ordonnancement temporel : LMAC, AI-LMAC en sont deux exemples ;
— les protocoles MAC multicanaux : différents protocoles ont exploré cette voie, mais surtout, l'extension du standard IEEE 802.15.4e fait désormais appel au multiplexage fréquentiel des transmissions, via un mécanisme nommé TCSH.

Nous allons dans cette section étudier ces différents types de protocoles de façon consécutive, à chaque fois en se penchant sur un ou deux protocoles représentatifs de chaque famille (notre but n'étant ici encore pas d'être exhaustif).

Enfin, nous étudierons plus en détail plusieurs protocoles MAC avancés : la famille CoSenS et iQueue-MAC, conçus et développés au sein du LORIA et de l'INRIA Nancy, sur lequels nous avons focalisé nos travaux.

La présente section 3.2 reprend la présentation effectuée dans les supports de cours de Y.-Q. Song sur les systèmes communicants contraints [Song, 2013], ainsi que des données présentées dans l'article de référence (*"survey"*) de P. Huang, L. Xiao et al [Huang et al., 2013] et la thèse de B. Nefzi [Nefzi, 2011].

3.2.1 Protocoles MAC synchrones

Cette famille de protocoles MAC est basée sur la synchronisation explicite des cycles de fonctionnement de noeuds voisins. Il s'agit de la solution la plus évidente pour permettre la communication entre appareils, mais cela implique des coûts supplémentaires (en temps et en complexité) pour effectuer cette synchronisation. Ce type de protocoles n'ayant pas de difficultés pour établir des communications entre noeuds, ils peuvent faire l'objet d'optimisations pour augmenter le débit de données et réduire les délais de transmission.

3.2.1.1 S-MAC

Un premier exemple de protocole synchrone est le protocole S-MAC [Ye et al., 2002]. Celui-ci est basé sur la méthode CSMA/CA complétée par l'envoi de signaux

indiquant qu'un noeud a des données à envoyer (RTS : *Ready To Send*) et qu'un noeud est prêt à recevoir (CTS : *Clear To Send*). Ce mode de fonctionnement est celui utilisé par le standard 802.11 ("Wi-Fi").

Un cycle de fonctionnement sous le protocole S-MAC est divisé en une période active et une période inactive. La période active est la seule pendant laquelle un noeud peut envoyer et recevoir des données, la période inactive correspondant à la désactivation de l'émetteur / récepteur radio pour économiser l'énergie. La période active est elle-même divisée en une période de synchronisation, et une période d'échange de données.

Dans ce protocole, chaque noeud choisit ses périodes actives et inactives en fonction de ses voisins. Le premier noeud à démarrer est le seul choisissant librement ses périodes, et annonce ensuite régulièrement ses périodes par l'envoi de signaux de synchronisation (SYNC) durant une période dédiée (au début de la période active). Les noeuds démarrant ensuite vont alors adapter leurs propres périodes sur celles annoncées par les noeuds précédents. Ce mécanisme d'adaptation porte le nom *"d'adaptative listening"*.

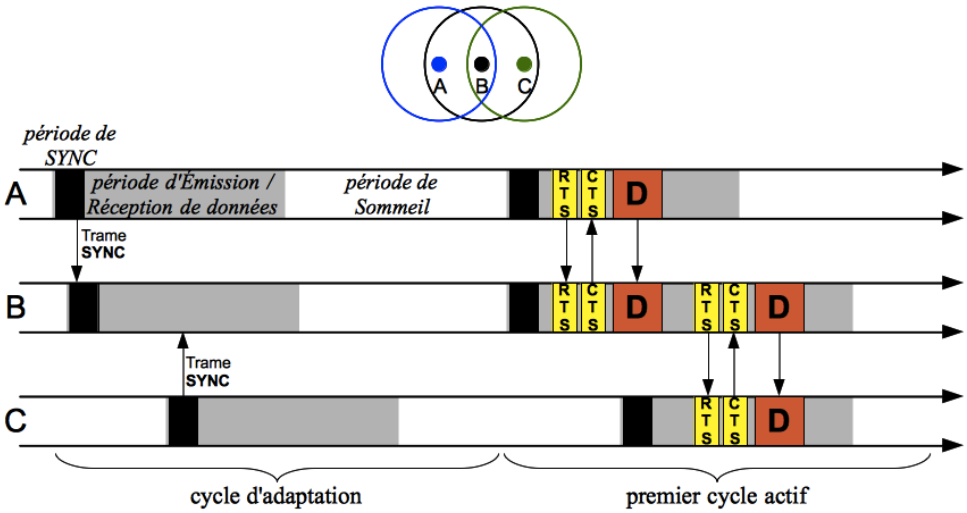

FIGURE 3.5 – *Adaptative listening entre trois noeuds suivant le protocole S-MAC. (D'après [Song, 2013])*

La figure 3.5 montre un exemple de synchronisation entre plusieurs noeuds sous S-MAC. Le noeud B, démarrant en dernier, adapte sa période active à celle de ses deux voisins A et C. A et C étant hors de portée l'un de l'autre, ne vont par contre pas se synchroniser entre eux.

Ce protocole est donc basé sur la définition de rendez-vous entre émetteur et récepteur. L'utilisation de nombreux signaux pour coordonner les transmissions (SYNC, RTS, CTS) entraîne un surcoût (*"overhead"*) élevé diminuant d'autant la bande passante disponible pour les données, même si S-MAC est capable d'envoyer des trames par lots (*"send burst"*), après une seule séquence RTS/CTS, pour accélérer le traitement de données volumineuses.

Ce protocole ayant des cycles de fonctionnement fixes et un mécanisme de synchronisation assez lourd, son adaptabilité au trafic sur le réseau est très faible.

Notons enfin que toute synchronisation entre appareils différents est susceptible d'être victime du phénomène de dérive des horloges (dû aux inévitables différences de fonctionnement entre les horloges internes de chaque appareil). S-MAC utilisant des périodes actives assez longues, il est peu sensible aux perturbations dues à ce phénomène.

3.2.1.2 T-MAC

Le protocole T-MAC [Dam and Langendoen, 2003] est une amélioration de S-MAC. Dans T-MAC, la durée de la période active n'est plus fixée à l'avance, mais chaque noeud reste éveillé jusqu'à ce qu'aucun signal ne le concernant n'ait été entendu pendant un certain délai (notion de *"timeout"*). Ce schéma permet un meilleure adaptabilité au trafic réseau, en permettant notamment une période de sommeil plus longue en cas de faible trafic, d'où une meilleure économie d'énergie. La figure 3.6 permet de comparer le fonctionnement de base des protocoles S-MAC et T-MAC. On notera que ce dernier adapte sa période d'éveil à l'intensité du trafic réseau.

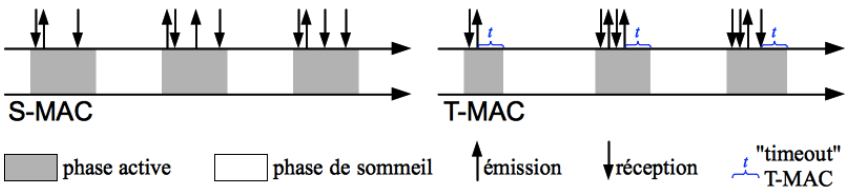

FIGURE 3.6 – *Comparaison du fonctionnement de S-MAC et T-MAC. (D'après [Song, 2013])*

T-MAC reprend également la capacité de S-MAC à effectuer des envois de trames par lots (après une unique paire de signaux RTS/CTS) pour un traitement plus efficace des données volumineuses.

3.2.2 Protocoles MAC asynchrones LPL

Dans un protocole MAC asynchrone, chaque noeud garde sa propre base de temps autonome. L'absence de synchronisation entre noeuds voisins permet (en théorie) d'éviter d'employer des phases de synchronisation systématiques à chaque cycle, donc à chaque appareil d'avoir un *"duty cycle"* plus réduit, et ainsi d'économiser son énergie, mais rend plus délicat l'établissement de communications entre noeuds. Toute la difficulté dans la mise au point d'un protocole asynchrone est de trouver une méthode efficace en ce sens.

La famille des protocoles dits LPL (*"Low Power Listening"*, ou écoute à faible puissance) repose sur le principe suivant : chaque noeud passe la quasi-totalité de son temps en sommeil, c'est-à-dire avec sa radio désactivée ; pour déterminer si un message lui est destiné, il va de façon cyclique activer sa radio et vérifier si le canal

radio est occupé (la procédure consistant à écouter le médium radio pour vérifier s'il est libre est appelée CCA : *Clear Channel Assessment*).

Les protocoles LPL vont différer sur la méthode d'envoi des données par les noeuds émetteurs : il s'agit en effet de s'assurer que le noeud destinataire remarquera qu'une transmission lui est destinée — et donc d'entrer en contact avec ce dernier lorsqu'il effectue son CCA — tout en essayant de minimiser l'énergie consommée par le noeud émetteur (en limitant le temps où la radio de ce dernier doit émettre).

3.2.2.1 B-MAC

Un premier exemple de protocole LPL est B-MAC (Berkeley MAC) [Polastre et al., 2004]. Dans ce protocole, un noeud devant envoyer des données émet un très long préambule (une trame ne contenant aucune donnée utile et ne servant qu'à signaler une future émission de données). Ce préambule est très long car son émission doit durer plus longtemps que l'intervalle entre deux CCAs consécutifs du récepteur. De plus, ce préambule ayant une taille fixe, le récepteur devra, lorsqu'il l'aura détecté, attendre sa fin avant de pouvoir recevoir ses données proprement dites. Ce mode de fonctionnement est représenté dans la figure 3.7.

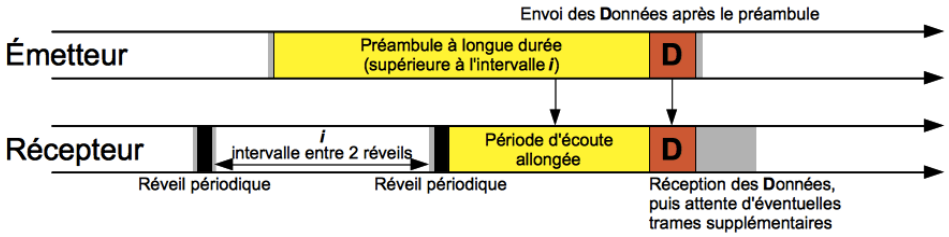

FIGURE 3.7 – *Schéma de fonctionnement de B-MAC (protocole LPL typique). (D'après [Song, 2013])*

On voit que cette procédure entraîne une dépense d'énergie maximale pour le noeud émetteur (à cause de la nécessité d'envoyer ce très long préambule) ainsi qu'un encombrement important du canal radio, augmentant le risque de collisions. La taille fixe du préambule comme de l'intervalle entre CCAs consécutifs rend également le protocole très peu adaptable au trafic réseau.

3.2.2.2 WiseMAC

Notre second exemple de protocole LPL, nommé WiseMAC [El-Hoiydi and Decotignie, 2004], emploie une technique consistant à « apprendre » les cycles de fonctionnement des noeuds voisins d'un même PAN. Pour ce faire, ce protocole repose d'abord sur l'envoi de longs préambules (comme B-MAC), auquels les noeuds récepteurs du PAN répondent par un acquittement lors de leur phase cyclique de réveil (*"channel sampling"*). Une fois les cycles de tous les noeuds du PAN connus, WiseMAC se distingue alors de B-MAC, en recourant à l'envoi de préambules bien plus courts, ce qui est possible en démarrant les transmissions au moment adéquat (en

incluant une certaine marge de sécurité pour éviter les problèmes dûs à la dérive des horloges entre appareils). La possibilité, après un temps d'apprentissage, d'utiliser des préambules raccourcis améliore considérablement la consommation d'énergie des différents noeuds — aussi bien des émetteurs ayant moins de temps à passer à émettre, que des récepteurs n'ayant plus à écouter inutilement des préambules trop longs — et optimise l'utilisation du médium radio en le libérant pour la transmission de données « utiles », augmentant ainsi le débit maximal utile réel du réseau.

3.2.2.3 X-MAC

Notre troisième exemple de protocole LPL, X-MAC [Buettner et al., 2006], optimise nettement la procédure d'envoi, en remplaçant les long préambules de B-MAC par de très courts trames-préambules intégrant en outre l'adresse du noeud destinataire, comme on peut le voir dans la figure 3.8. Cette technique améliore non seulement la dépense d'énergie par le noeud émetteur ainsi que le taux d'occupation du médium, mais permet également d'éviter de garder inutilement éveillés des noeuds tiers, le destinataire d'une future émission étant clairement identifié par ces courts préambules. Enfin, X-MAC inclut également l'envoi d'un acquittement préalable par le noeud récepteur (assimilable à un signal CTS) avant l'envoi des données proprement dites.

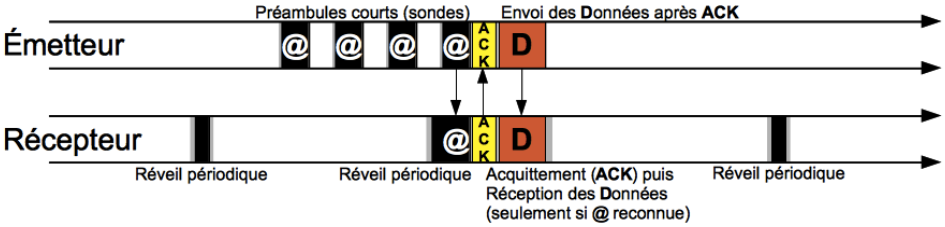

FIGURE 3.8 – *Schéma de fonctionnement du protocole X-MAC. (D'après [Song, 2013])*

3.2.2.4 ContikiMAC

Notre quatrième exemple de protocole LPL est ContikiMAC [Dunkels, 2011]. Celui-ci est nettement plus récent que les précédents, et a été conçu spécifiquement pour fonctionner sous Contiki OS (système que nous présenterons dans la section 3.3.3 page 65).

Ce protocole, par rapport à X-MAC, simplifie encore la procédure d'envoi des données, en supprimant totalement la notion de préambule : lors d'une transmission, c'est désormais la trame de données elle-même qui est émise de façon répétitive, jusqu'au prochain réveil du noeud récepteur. Une fois la trame de données bien reçue, le destinataire envoie un acquittement (ACK standard 802.15.4) pour mettre fin à l'émission de la trame. Le fonctionnement de ContikiMAC est représenté dans la figure 3.9 page suivante.

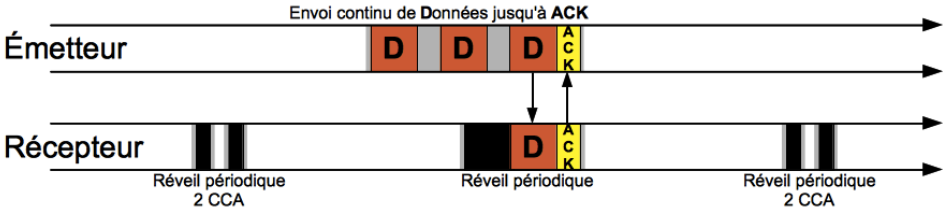

FIGURE 3.9 – *Fonctionnement du protocole ContikiMAC. (D'après [Song, 2013])*

Autre particularité : chaque noeud effectue ***deux*** CCA à chaque réveil cyclique, et ne retombe en sommeil que si ces deux CCA ont indiqué la non-occupation du canal radio. Ces deux CCA, séparés par un délai calculé à partir du délai standard entre émission de trames — DIFS — et la taille minimale d'une trame 802.15.4, permettent de consommer moins d'énergie qu'une longue écoute. Ces deux CCA ont ainsi la durée minimale nécessaire pour déterminer ou non l'occupation du médium radio.

Enfin, signalons que les versions récentes de ContikiMAC bénéficient d'une optimisation supplémentaire : après un envoi réussi, le noeud émetteur mémorise la période de réveil du destinataire, afin d'envoyer les trames suivants au meilleur moment et ainsi limiter au maximum les transmissions inutiles. Cette optimisation est nommée *"phase lock"* par les concepteurs de ContikiMAC.
On peut rapprocher cette technique de *"phase lock"* du mécanisme d'apprentissage des cycles des noeuds observé antérieurement dans le protocole WiseMAC.

3.2.3 Protocoles MAC asynchrones LPP

Si les protocoles LPL sont conçus pour avoir une meilleure adaptabilité au trafic que les protocoles synchrones (comme S-MAC ou T-MAC), ils souffrent néanmoins de plusieurs inconvénients :

— une efficacité energétique non optimale : B-MAC, avec ses très longs préambules, est clairement insuffisant dans ce domaine ;
— l'envoi de préambules (ou de trames dans le cas de ContikiMAC) avant la « transmission réelle » occupe inutilement le medium, augmentant ainsi les délais de transmission, les risques de collision, et diminuant le débit réel maximal du trafic ;
— l'inadéquation aux trafics élevés — par exemple lors de « pics » ou « pointes » de transmissions — à cause de la méthode CSMA/CA sous-jacente ;
— la risque de « collisions cachées » entre noeuds : ce terme regroupe les cas où plusieurs paires de noeuds émetteurs/récepteurs, ayant chacune indépendemment des données à se transmettre, se gênent mutuellement si elles sont partiellement à portée l'une de l'autre. La figure 3.10 page suivante illustre la survenue de ces problèmes : les deux récepteurs sont à portée des deux émetteurs, ces derniers étant par contre incapables de se détecter mutuellement, ce qui provoque une collision entre les trames transmises par chacun des deux émetteurs.

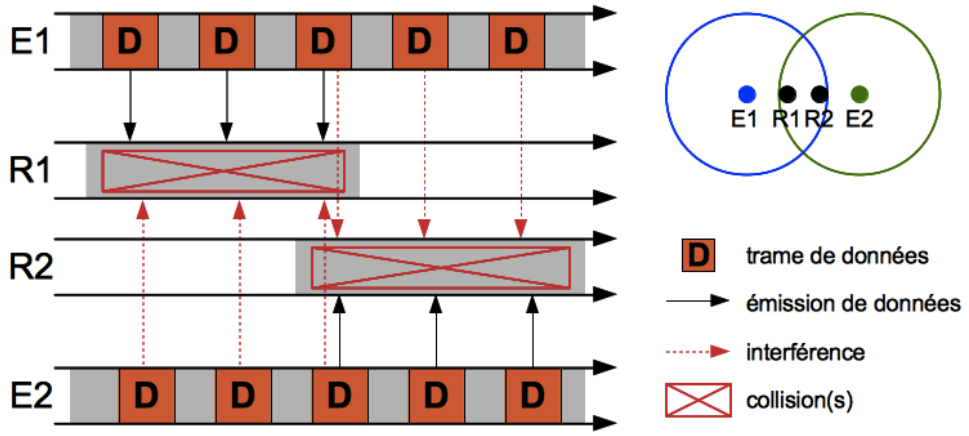

FIGURE 3.10 – *Survenue de « collisions cachées » entre deux paires de noeuds menant chacune une transmission distincte, empêchant la réussite des deux communications. (D'après [Song, 2013] et [Sun et al., 2008])*

Pour tenter de résoudre ces problèmes, un autre famille de protocoles MAC asynchrones a été conçue. Elle repose sur l'idée fondamentale de laisser les noeuds destinataires démarrer les transmissions. On parle donc de communications initiées par le recepteur, chaque noeud émettant cycliquement des *"beacons"* (ou *"probes"* : sondes) signalant quand il est prêt à recevoir (RI-LPP : *Receiver Initiated Low Power Probing*).

3.2.3.1 RI-MAC

L'exemple de protocole LPP que nous allons examiner ici est RI-MAC [Sun et al., 2008]. Son principe de fonctionnement est le suivant : chaque noeud d'un réseau se réveille périodiquement, et émet un *"beacon"* signalant qu'il est prêt à recevoir des données. Un noeud ayant des données à transmettre va donc écouter le médium radio et attendre de recevoir un *"beacon"* issu du destinataire voulu. Dès ce *"beacon"* reçu, l'émetteur envoie sa trame de données. Le noeud récepteur confirmera ensuite la bonne réception de cette trame par l'envoi d'un nouveau *"beacon"*, lequel joue à la fois le rôle d'acquittement de la trame reçue (ACK) ***et*** de signal pour le lancement d'une nouvelle transmission. Grâce à ce double rôle des *"beacons"*, l'envoi rapide de trames par lots est possible (par exemple pour gérer les transmissions de grandes quantités de données). Le principe de fonctionnement de la méthode LPP est représentée dans la figure 3.11 page suivante.

Par rapport aux protocoles LPL, ce principe de fonctionnement permet de réduire l'occupation moyenne du médium radio par un couple de noeuds pour arriver à un point de rendez-vous. En outre, les phénomènes de collisions cachées entre noeuds sont également moins fréquents.

L'un des principaux problèmes du principe LPP survient lorsque plusieurs noeuds ont simultanément besoin d'envoyer des données au même destinataire : à la récep-

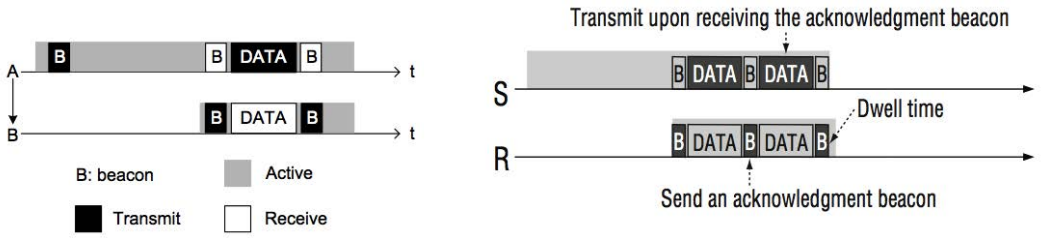

FIGURE 3.11 – *Principe de transmission de données entre un noeud émetteur (**A** ou **S**) et un noeud récepteur (**B** ou **R**), selon la méthode RI-LPP (Receiver-Initiated Low Power Probing). (Source : [Song, 2013] et [Huang et al., 2013])*

tion du *"beacon"* émis par ce dernier, les différents noeuds émetteurs vont tous émettre leurs trames de données, ce qui provoquera inévitablement une collision. La probabilité que plusieurs noeuds cherchent à envoyer des données au même noeud au même moment est d'autant plus forte que le trafic du réseau concerné devient important.

Pour pallier ce problème, RI-MAC ajoute au principe LPP de base une notion de *"backoff"* intégrée aux *"beacons"* : ces derniers incluent en effet une durée maximale de « silence » parmi laquelle les noeuds émetteurs devront choisir aléatoirement un délai à respecter entre la réception de ce *"beacon"* et l'envoi de leurs données. Au début, cette fenêtre de *"backoff"* est nulle ; à chaque collision détectée, cette fenêtre va voir sa valeur augmenter rapidement pour minimiser les risques de nouvelle collision. Cette procédure est appelée BEB (*Binary Exponential Backoff*), et est illustrée dans la figure 3.12.

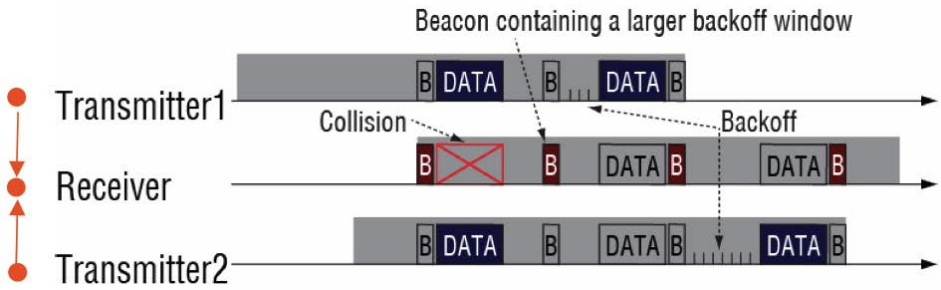

FIGURE 3.12 – *Illustration de la méthode BEB (Binary Exponential Backoff) employée par RI-MAC pour résoudre les problèmes de collisions dûs à l'envoi simultané de trames par plusieurs émetteurs. (Source : [Song, 2013] d'après [Sun et al., 2008])*

On peut noter que cette procédure de BEB est similaire au mécanisme de *"backoff"* utilisé par la méthode CSMA/CA, également dans le but de résoudre les cas de collisions.

Grâce à ces mécanismes de fonctionnement, RI-MAC peut obtenir de meilleurs résultats que X-MAC quand le trafic réseau est élevé, tout en ayant un cycle de fonctionnement similairement bas (donc aussi économe en énergie) quand le trafic est faible.

De façon générale, on constate que les protocoles basés sur le principe RI (*Receiver Initiated*) sont nettement plus performants quand le trafic réseau est intense et/ou soumis à des interférences, là où les protocoles laissant l'initiative de la transmission aux émetteurs vont « étouffer » le réseau avec des collisions de trames de données. Cela est nettement visible dans le schéma présenté dans la figure 3.13. On y voit que les protocoles *"Receiver-Initiated"* (LPP) ont toujours un net avantage, d'autant plus flagrant que le débit augmente.

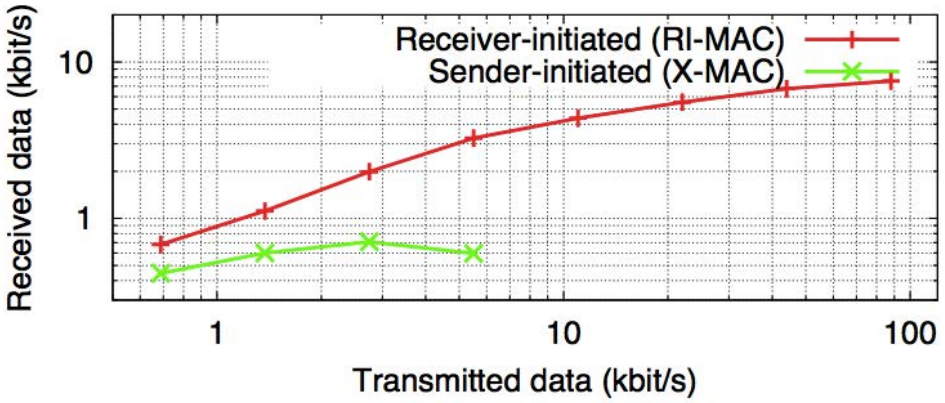

FIGURE 3.13 – *Comparaison entre protocoles "Receiver-Initiated" (LPP) et "Sender-Initiated" (LPL) face à la montée en charge du trafic réseau. (Source : [Song, 2013] d'après [Österlind et al., 2012])*

Néanmoins, l'utilisation du médium radio par la méthode LPL (même améliorée par RI-MAC) est toujours loin d'être optimale, le recours aux *backoffs* aléatoires pour éviter les collisions provoquant des « temps morts » où le réseau doit rester inutilement silencieux.

3.2.4 Protocoles MAC à ordonnancement temporel

Tous les protocoles que nous avons vus jusqu'ici sont basés sur la ***contention*** : les trames de données sont émises sur un médium radio non réservé, ce qui implique un risque de collision et donc de perte de données (ce qui nécessite en général un mécanisme de vérification et de réémission des données si besoin est).

À l'inverse, les protocoles MAC dit ***ordonnancés*** définissent strictement l'occupation du médium radio, en allouant une partie précise de la bande passante à chaque transmission potentielle, éliminant ainsi les collisions.

La méthode employée pour parvenir à cette fin est basée sur le multiplexage temporel : chaque transmission se voit allouer le canal radio durant un intervalle de temps bien défini. Cette méthode de fonctionnement est nommée **TDMA** (*Time Division Multiple Access*). Son fonctionnement est très similaire à la CFP observée plus haut dans le protocole MAC du standard IEEE 802.15.4 en « mode *"beacon"* ».

— Les protocoles basés sur la méthode TDMA sont considérés comme les plus efficaces pour le traitement de trafics réseaux intenses, car ce sont ceux qui

permettent d'exploiter le plus efficacement le médium radio, et donc d'approcher le plus de son débit maximal théorique.

— Par contre, la nécessité de réserver au préalable les différents intervalles de temps, quelle que soit l'intensité du trafic réseau, entraîne un surcoût d'organisation (*"overhead"*), ce qui rend les protocoles basés sur la contention plus efficaces pour les faibles trafics. Les réseaux de capteurs sans-fil actuels n'ayant le plus souvent à gérer qu'un faible trafic (avec éventuellement quelques pointes), cela explique que les protocoles basés sur la contention restent actuellement les plus utilisés.

Tout ceci est résumé de façon claire et synthétique dans le schéma présenté dans la figure 3.14. On y voit que la contention (CSMA) gère mieux les faibles trafics, tandis que l'ordonnancement (TDMA) est plus efficace pendant les débits intenses (ou les pointes de trafic réseau).

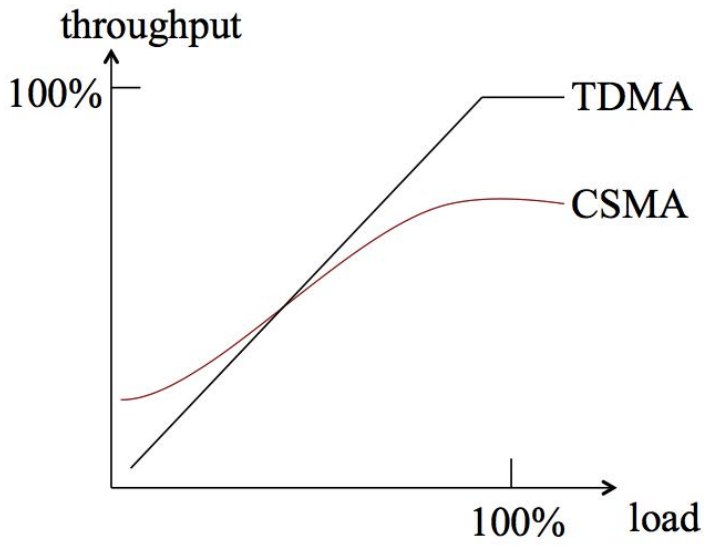

FIGURE 3.14 – *Comparaison de l'efficacité entre méthodes basée sur la contention (CSMA) et sur l'ordonnancement (TDMA) en fonction du débit réseau. (Source : [Song, 2013])*

En outre, toujours à cause de la nécessité d'organisation préalable du multiplexage temporel du médium radio, les protocoles basés sur l'ordonnancement sont en général du type synchrone, bien mieux adapté.

3.2.4.1 LMAC

Le protocole LMAC [Hoesel and Havinga, 2004] est un exemple typique de protocole basé sur la méthode TDMA. Chaque cycle de fonctionnement (ou « trame ») est divisé en *"slots"*, représentant une unité minimale de temps qui sera réservée à un noeud donné. Chaque noeud du réseau se verra attribuer un *slot* donné au sein de chaque trame réseau.

Durant son *slot* de temps, un noeud doit d'abord émettre un message de contrôle — indiquant s'il a des données à transmettre, et si oui quel est leur destinataire — suivi s'il y a lieu du trame de données à émettre.

Tous les noeuds du réseau doivent ainsi avoir leur émetteur / récepteur radio activé pour écouter le message de contrôle émis au début de chaque *slot*. Les noeuds n'étant pas destinataires d'un message durant ce *slot* peuvent ensuite désactiver leur radio (donc tous les noeuds si aucune donnée n'est à émettre durant ce *slot*).

À noter que l'allocation des slots ne se fait pas de façon centralisée : au démarrage du réseau, chaque noeud doit choisir un *slot* parmi ceux disponibles, et le réserver pour les trames suivantes en y émettant un message de contrôle.

3.2.4.2 AI-LMAC

Le protocole AI-LMAC [Chatterjea et al., 2004] est une extension de LMAC permettant à un noeud de réserver plusieurs *slots* au sein de chaque trame. Le choix du nombre de *slots* à allouer à chaque noeud n'est pas faite de façon distribuée (comme pour la réservation), mais par le coordinateur du réseau en fonction du trafic sortant prévu pour chaque noeud.

3.2.5 Protocoles MAC multicanaux

Une dernière catégorie de protocoles MAC repose sur la capacité de la couche physique 802.15.4 d'exploiter plusieurs canaux (c'est-à-dire plusieurs fréquences) radio différents. Ainsi, tout comme il est possible de multiplexer les différentes transmissions dans le temps, cette fonctionnalité de la couche physique permet d'effectuer un multiplexage sur les fréquences radio : en assignant des canaux différents aux différents noeuds d'un réseau, il devient possible de gérer des transmissions de données parallèles.

Par analogie avec l'acronyme TDMA vu section 3.2.4 page 43, cette méthode de multiplexage sur les fréquences radio est appelée **FDMA** (*Frequency Division Multiple Access*).

Les deux principales difficultés pour la conception de tels protocoles sont :

— la distribution des différents canaux aux différents noeuds d'un réseau ; et

— la gestion efficace de la communication entre ces différents canaux.

Le deuxième point est rendu particulièrement difficile, car les émetteurs / récepteurs radio conçus pour le standard 802.15.4 ne sont pas capables d'écouter simultanément plusieurs fréquences. En outre, le standard 802.15.4 limitant la taille des trames physiques à 127 octets, un mécanisme d'attribution des canaux aux différents noeuds peut rapidement entraîner un surcroît de charge (*"overhead"*) intolérable pour un réseau de ce type.

Différents protocoles ont été proposés pour contourner ces difficultés, la tendance actuelle étant de combiner multiplexage temporel et multiplexage fréquentiel : on parle ainsi de protocoles TDMA/FDMA. Parmi ces protocoles, on peut citer MC-LMAC employant cette méthode pour gérer les envois de reales de données, ainsi que Y-MAC et MuChMAC employant une méthode de « saut de fréquence » (*"channel*

hopping") pour permettre aux différents noeuds de recevoir des trames de données de plusieurs canaux différents.

Ces protocoles multicanaux n'ayant pas été traités lors des présents travaux de thèse, nous ne les étudierons pas plus en détail ici ; l'étude de référence (*"survey"*) de [Huang et al., 2013] consacrant toute sa section V à l'étude de ces protocoles, nous invitons le lecteur souhaitant plus de détails sur ce sujet à s'y reporter.

Il est par contre plus important de noter que l'amendement **802.15.4e** du standard, adopté en 2012, consacré à l'amélioration de la couche MAC officielle du standard IEEE [IEEE 802.15.4e, 2012], inclut un mécanisme nommé *"Time Slotted Channel Hopping"* ou **TSCH**. Plus qu'une simple amélioration, il s'agit tout simplement d'une nouvelle couche MAC, totalement compatible avec la couche physique définie dans les précédentes versions du standard 802.15.4, employant multiplexage temporel ***et*** multiplexage fréquentiel pour parvenir à une consommation d'énergie minimale et une fiabilité de transmission maximale.

Une version dédiée de la pile IPv6 destinée à fonctionner par-dessus cette nouvelle couche MAC et son mécanisme TSCH est déjà en cours de conception par l'IETF, sous le nom de projet « 6TiSCH » [Thubert et al., 2013]. Cette nouvelle pile protocolaire est notamment destinée à être supportée par la pile réseau avancée **OpenWSN** [Watteyne et al., 2012], dont nous reparlerons dans plus loin dans le présent chapitre (section 3.3.5 page 68 sur RIOT OS et section 3.3.7.1 page 72 sur FreeRTOS).

Notons toutefois que cette couche MAC est largement plus complexe que celles définies dans les versions précédentes du standard (et que nous avons étudiées section 3.1.2 page 29). Sa mise en oeuvre pourrait par conséquent poser des difficultés sur les appareils à fonctionnalités réduites (RFD) susceptibles de faire partie d'un réseau de capteurs sans-fil. La description de cette nouvelle couche MAC complexe sort du cadre du présent manuscrit de thèse et ne sera donc pas entreprise ici.

Malgré ces limitations, le standard IEEE 802.15.4e proposant désormais une couche MAC officielle reposant sur le paradigme TDMA/FDMA, ce mode de fonctionnement sera sans doute appelé à devenir incontournable dans l'implantation des réseaux de capteurs sans-fil du futur.

Après l'étude des principales catégories « standard » de protocoles MAC, et d'exemples représentatifs de ces dernières, nous allons maintenant nous focaliser sur les protocoles avancés ayant fait l'objet des travaux de la présente thèse.

3.2.6 Protocoles hybrides avancés

Des recherches sur la conception de protocoles MAC ont également été menées au sein du centre INRIA Nancy Grand-Est et du LORIA. Ces recherches ont notamment abouti à la conception de plusieurs protocoles évolués, qui ont été à la base des travaux menés dans la présente thèse. Nous allons dans la présente section examiner successivement ces différents protocoles sur lesquels nous nous sommes focalisés.

Nous aborderons également brièvement — pour compléter notre tour d'horizon des protocoles avancés — dans une autre sous-section les travaux d'autres équipes de recherche sur d'autres protocoles MAC avancés.

Nous évoquerons aussi la couche MAC contenue dans une pile protocolaire intégrée et complète, issue d'un effort de développement suivant une approche dite multi-couches (en anglais *"cross-layer"* dans une dernière sous-section. De telles approches sortent toutefois du cadre de cette thèse, laquelle se focalise sur les seules couches basses, et ne font par conséquent l'objet d'aucun travail dans cette thèse.

3.2.6.1 CoSenS et S-CoSenS

Ce protocole, conçu par B. Nefzi et Y.-Q. Song [Nefzi and Song, 2010] [Nefzi and Song, 2012], fait partie de la classe des protocoles basés sur la contention. Il est décrit dans le chapitre 3 de la thèse de B. Nefzi [Nefzi, 2011].

Le but de ce protocole est d'améliorer les performances de la couche MAC notamment concernant la qualité de service, tout en limitant les coûts liés à l'implantation de cette couche, ce qui exclut le recours à la méthode TDMA (celle-ci nécessitant une configuration minutieuse du réseau, une précision très fine de la synchronisation entre noeuds, et une adaptation difficile aux changements de trafic sur le réseau).

Le protocole CoSenS reprend donc la méthode CSMA/CA, en contrôlant les instants où se déroulent les transmissions et où le médium radio est utilisé. Le principe de base est de séparer temporellement la période où un noeud reçoit des trames de données (période de réception) de celle où ce même noeud émet les trames ainsi reçues (période de retransmission). Ceci est schématisé dans la figure 3.15. On voit que chaque cycle CoSenS est partagé entre une période de réception (ou période d'écoute : WP) durant laquelle le noeud collecte les trames reçues, suivi d'une période de transmission (TP) durant laquelle tous les trames collectées dans la file d'envoi sont transmises vers la destination adéquate. (Les trames / paquets collectés pouvant éventuellement être traités par les couches supérieures de la pile de façon arbitraire entre réception et émission).

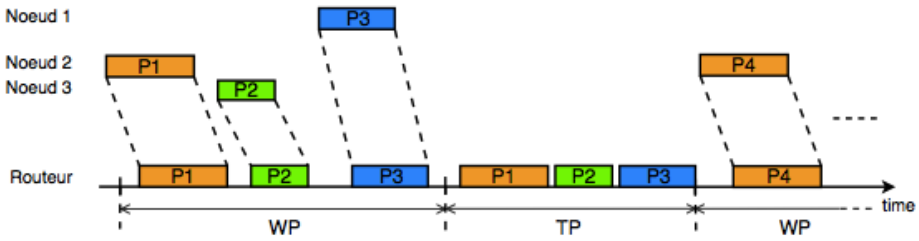

FIGURE 3.15 – *Principe de base du protocole CoSenS. (Source : [Nefzi, 2011])*

Il a été démontré que ce protocole apporte une amélioration sensible des performances et de la qualité de service — notamment au niveau du taux de succès de transmission des trames, du débit réseau maximal supporté, ainsi que du délai de transmission de bout-en-bout des données — par rapport au protocole MAC de base du standard 802.15.4.

Son principal défaut est par contre de nécessiter que tous les noeuds gardent leur émetteur / récepteur radio allumé en permanence, ce qui représente un sérieux désavantage concernant l'économie d'énergie.

C'est pourquoi ce protocole a été amélioré pour inclure une période de sommeil dans le cycle de fonctionnement des noeuds. Le protocole résultant est nommé **S-CoSenS**, et c'est celui sur lequel nous nous sommes principalement basés lors de nos travaux de thèse. Il s'agit donc à la fois d'un protocole MAC et d'un protocole RDC (*Radio Duty Cycle* : gérant le cycle d'allumage et d'extinction de l'émetteur / récepteur radio), qui plus est adapté aux phases de routage. Il est décrit dans le chapitre 5 de la thèse de B. Nefzi [Nefzi, 2011].

Tout comme CoSenS, il est basé sur la méthode CSMA/CA, et est ainsi comparable à la couche MAC 802.15.4 en « mode sans *"beacon"* ». L'idée de base reste de retarder la retransmission des trames reçues, en divisant chaque cycle de fonctionnement d'un noeud coordinateur en trois périodes :

— une période de sommeil (SP : *Sleeping Period*) où l'émetteur / récepteur radio est désactivé pour économiser l'énergie du noeud ;
— une période d'écoute ou période d'attente (WP : *Waiting Period*) durant laquelle le médium radio est écouté pour collecter les trames de données 802.15.4 arrivantes ;
— et une période de transmission (TP : *Transmission Period*) durant laquelle les trames reçues et mis en attente durant la période d'écoute sont réémises, si possible en lot (*"burst"*).

Le principal avantage de S-CoSenS est son capacité à s'adapter dynamiquement à l'intensité du trafic radio en temps réel, en calculant pour chaque cycle radio (commun à tout un PAN fonctionnant sous S-CoSenS) la durée des périodes de sommeil (SP) et d'écoute (WP), en fonction du nombre de trames retransmises durant les cycles précédents.

Notons que l'ensemble constitué de la période de sommeil (SP) et de la période de réception (WP) d'un même cycle est appelé **subframe** ; il s'agit de la partie du cycle de S-CoSenS dont la durée est calculée et connue *a priori*. À l'inverse, la durée de la période d'envoi (TP) ne peut être déterminée qu'à son tout début, car cette durée dépend directement de la quantité de trames de données reçues avec succès durant la période d'écoute (WP) précédente.

En outre, le calcul de la durée de la période d'envoi WP se fait via un algorithme de « moyenne glissante », où la durée de WP pour chaque cycle est calculée ainsi :

$$\overline{\mathrm{WP}_n} = \alpha \cdot \overline{\mathrm{WP}_{n-1}} + (1-\alpha) \cdot \mathrm{WP}_{n-1}$$
$$\mathrm{WP}_n = \max(\mathrm{WP}_{min}, \min(\overline{\mathrm{WP}_n}, \mathrm{WP}_{max}))$$

où $\overline{\mathrm{WP}_n}$ et $\overline{\mathrm{WP}_{n-1}}$ sont respectivement la durée moyenne de WP au n^{eme} et $(n-1)^{\text{eme}}$ cycle, tandis que WP_n et WP_{n-1} sont les durées réelles de WP respectivement au n^{eme} et $(n-1)^{\text{eme}}$ cycles ; α est un paramètre dont la valeur est arbitrairement choisie entre 0 et 1, représentant le poids relatif de l'historique dans le calcul des durées ; WP_{min} et WP_{max} étant les limites minimales et maximales imposées à la durée de WP par le programmeur.

La synchronisation locale entre un routeur S-CoSenS et ses noeuds-feuilles, au sein d'un même PAN, se fait grâce à une trame *"beacon"* émis par le routeur au début de chaque cycle. Ce *"beacon"* inclut les durées (en microsecondes) choisies pour la période de sommeil (SP) et la période de réception (WP) au cours du cycle dont le *"beacon"* marque le début.

La représentation schématique d'un cycle complet S-CoSenS, du point de vue d'un routeur gérant un PAN, est décrite dans la figure 3.16. On distingue la période de sommeil (SP), la période d'écoute / réception (WP) et la période de retransmission (TP). L'ensemble des deux premières périodes constitue la *subframe*, dont la durée est calculée à l'avance, en fonction du trafic réseau observé jusqu'alors.

La figure 3.16 montre clairement l'adaptation du cycle S-CoSenS à un trafic réseau respectivement moyen, intense et faible.

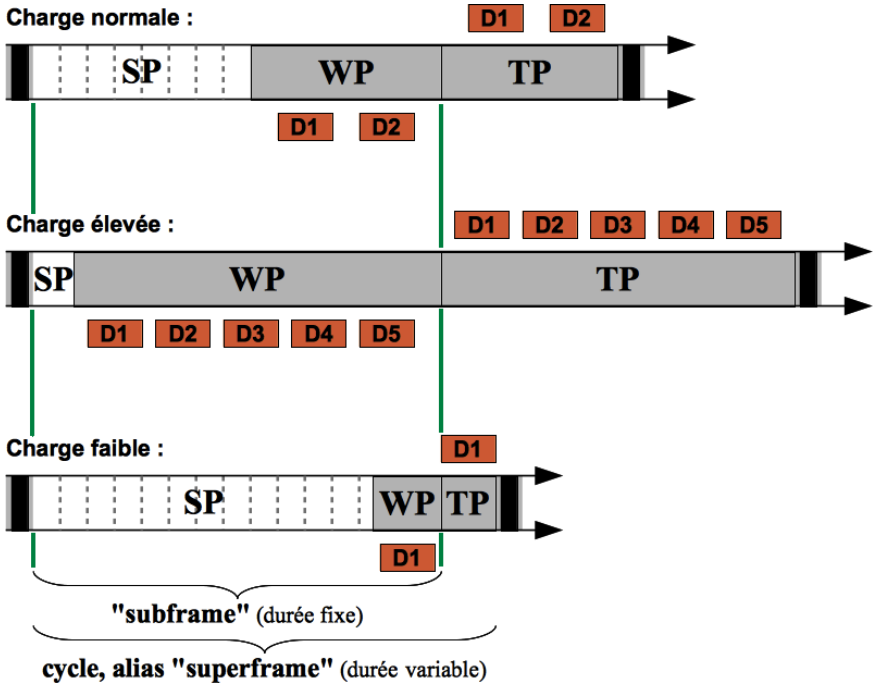

FIGURE 3.16 – *Principe d'un cycle de fonctionnement d'un noeud coordinateur S-CoSenS. (D'après [Nefzi, 2011])*

Cette synchronisation locale au sein d'un PAN S-CoSenS, faite grâce à l'envoi cyclique d'un *"beacon"*, repose donc sur le paradigme LPP (*Low Power Probing*, principe détaillé plus haut). D'un autre côté, la synchronisation et la communication entre routeurs S-CoSenS appartenant à des PANs différents se fait grâce à de courtes périodes d'éveil et d'écoute se déroulant durant la période de sommeil (SP), et repose donc sur le paradigme LPL (*Low Power Listening*, comme pour X-MAC par exemple), pour des raisons de simplicité.

À noter que le cycle S-CoSenS complet (SP + WP + TP) ne concerne que les noeuds jouant le rôle de routeur et de coordinateur de PAN.

En effet, une propriété intéressante de S-CoSenS est que les noeuds terminaux ou noeuds-feuilles (c'est-à-dire ceux n'ayant pas un rôle de routeur ou de coordinateur de réseau) peuvent garder leur émetteur / récepteur radio constamment éteint, tant qu'ils n'ont pas de trame à envoyer. Quand un de ces noeuds-feuilles doit envoyer une trame de données, il démarre sa radio, écoute et attend le premier *"beacon"* émis par un routeur, puis émet sa trame en utilisant la méthode CSMA/CA au début de la période de réception (WP) annoncée dans le *"beacon"* qu'il a reçu. Ce noeud-feuille peut éteindre sa radio pendant le délai courant entre le *"beacon"* et la WP attendue (c'est-à-dire durant le délai correspondant à la période de sommeil SP du routeur ayant émis le *"beacon"*), émet sa trame durant la WP voulue, puis peut retourner à l'état de sommeil une fois sa trame transmise avec succès.

Toute cette procédure de transmission est résumée dans la figure 3.17. Dans cette figure, les parties grises représentent les périodes où un noeud fait fonctionner sa radio, les blocs oranges correspondent aux transmissions de trames, tandis que les parties blanches sont celles où le noeud met sa radio en sommeil pour économiser son énergie. Le noeud simple émetteur se réveille quand une trame doit être émise, attend le *"beacon"* issu du routeur, se synchronise alors pour émettre la trame durant la période de réception (WP) du routeur, puis retourne en mode sommeil. Le routeur peut ensuite réemettre la trame vers la destination adéquate durant sa période de transmission (TP).

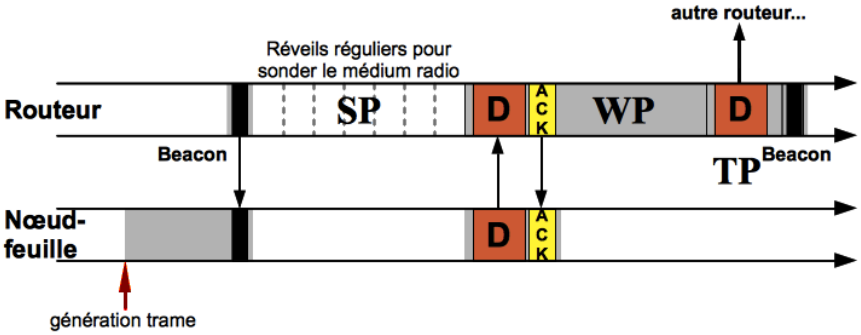

FIGURE 3.17 – *Transmission d'une trame avec le protocole S-CoSenS.*

Si ce protocole représente une amélioration certaine par rapport au protocole MAC de base du standard 802.15.4 et aux protocoles LPL et LPP classiques — comme nous le verrons dans le chapitre 5 de la présente thèse —, il continue néanmoins de souffrir des limitations intrinsèques liées au principe de contention sur lequel il est basé : lors de trafics réseaux intenses, la qualité de service (taux de trames transmises avec succès, délais de transmission) chute irrémédiablement. Ces problèmes sont mieux pris en charge par les protocoles basés sur l'ordonnancement (TDMA) mais ceux-ci posent d'autres problèmes (complexité, lourdeur d'organisation). Des recherches ont donc été menées pour contourner ces problèmes, et ont abouti à la mise au point de nouveaux protocoles plus performants, dont celui que nous allons maintenant détailler dans la section 3.2.6.2 suivante.

3.2.6.2 iQueue-MAC

Ce protocole, de conception récente et ambitieuse [Zhuo et al., 2013], est un protocole ordonnancé hybride, utilisant la contention (méthode CSMA/CA) et l'ordonnancement temporel (TDMA) en fonction du trafic réseau courant.

Nous avons vu en effet plus haut qu'il a été démontré que la méthode CSMA est plus efficace pour le traitement des faibles trafics, tandis que TDMA est nettement plus appropriée pour supporter les trafics intenses.

Si le protocole MAC standard 802.15.4 en « mode *"beacon"* » fait déjà appel à une approche hybride (avec les notions de CAP et CFP), il est limité par la configuration totalement statique de ses paramètres de fonctionnement.

À l'inverse, iQueue-MAC est conçu pour s'adapter dynamiquement et en temps réel au trafic du réseau, de façon à adopter à chaque instant la meilleure configuration pour optimiser qualité de service et consommation d'énergie (via la maîtrise du *duty cycle* et de ses différentes périodes, dont les durées sont dynamiques).

La figure 3.18 montre comment iQueue-MAC gère la montée en charge du trafic réseau de façon totalement différente d'autres protocoles plus classiques basés sur la seule contention (comme par exemple S-CoSenS).

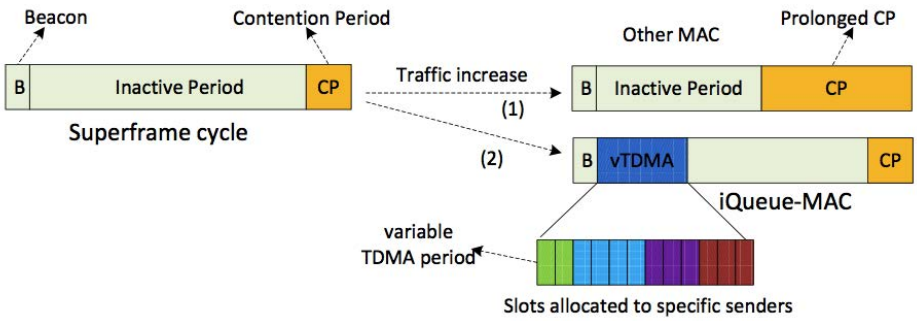

FIGURE 3.18 – *Comparaison de la gestion de la montée en charge du trafic réseau entre **(1)** un protocole uniquement basé sur la contention, et **(2)** iQueue-MAC, qui a recours à une période basée sur TDMA de durée variable, adaptée à la charge réseau. (Source : [Song, 2013])*

Les idées principales qui sous-tendent la conception d'iQueue-MAC sont les suivantes :

— L'utilisation de la contention (CSMA/CA) quand le trafic est faible, et de l'ordonnancement temporel (TDMA) quand le trafic devient intense.
— La séparation des noeuds sans-fil en deux catégories : les noeuds simples (terminaux ou noeuds-feuilles) et les routeurs.
— Les noeuds simples (tout comme avec la famille CoSenS) ne s'éveillent que quand ils ont des données à envoyer, et passent donc la majeure partie de leur temps en sommeil pour économiser leur énergie.
— Les routeurs sont les noeuds exécutant véritablement la totalité du mécanisme d'iQueue-MAC.

FIGURE 3.19 – *Structure d'une trame iQueue-MAC. (Source : [Song, 2013])*

Header	Slots#	Nodes#	ID list	Slot Allocation list

FIGURE 3.20 – *Structure d'un "beacon" iQueue-MAC. (Source : [Song, 2013])*

— Le protocole base son comportement et sa configuration sur la connaissance de la quantité de trames à envoyer (c'est-à-dire la longueur de la file d'envoi de trames) pour chaque noeud du réseau.

En effet, chaque noeud simple, lorsqu'il envoie une trame, insère le niveau d'occupation de sa file d'envoi (juste après l'entête de la trame). Le routeur est donc à chaque cycle capable d'évaluer la charge que devra soutenir le réseau lors du cycle suivant. La structure d'une trame iQueue-MAC est représentée figure 3.19. On notera l'ajout du taux d'occupation de la file d'envoi du noeud émetteur entre l'entête MAC et la charge utile (*"payload"*) de la trame proprement dite.

Chaque cycle iQueue-MAC (également appelé **superframe**, est représenté figure 3.21 page ci-contre. La *subframe* est la partie du cycle dont la durée est calculée (donc connue) à l'avance ; elle comporte la période TDMA, dont la durée varie en fonction de l'intensité du trafic réseau, et dont les *slots* temporels sont alloués aux différents noeuds en fonction de la quantité de trames qu'ils ont à émettre (ces quantités ayant été transmises durant la période de contention du cycle précédent).

Un cycle iQueue-MAC se décompose donc en les phases suivantes :

B : *"beacon".* Celui-ci permet la synchronisation des différents noeuds du PAN : il indique la durée et l'affectation des différents *slots* de la période TDMA, ainsi que la durée de la période de sommeil. L'ensemble regroupant les slots de temps TDMA et la période de sommeil est appelé ***subframe*** : c'est la partie du cycle iQueue-MAC dont la durée est calculée et allouée à l'avance. La structure d'un *"beacon"* iQueue-MAC est détaillée figure 3.20. Toutes les données nécessaires pour calculer la durée de la *subframe*, de la section TDMA variable, et l'allocation des différents *slots* de temps TDMA aux différents noeuds demandeurs sont présentes.

SF : *SubFrame* Cette période contient les slots de temps TDMA dont l'allocation aux différents noeuds a été annoncée dans le *"beacon"*; ces slots TDMA sont suivis de la période de sommeil où la radio du routeur est éteinte pour économiser l'énergie. Notons que comme dans d'autres protocoles tels que X-MAC ou S-CoSenS, de courts moments d'activation de la radio ont lieu durant la période de sommeil, afin de pouvoir détecter et recevoir des messages en provenance de routeurs d'autres PANs.

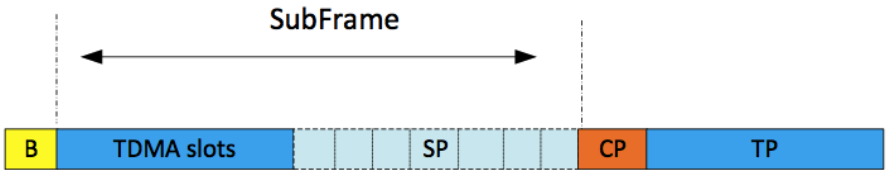

FIGURE 3.21 – *Structure d'un cycle iQueue-MAC. (Source : [Song, 2013])*

CP : *Contention Period* Période de réception basée sur le principe de la contention (CSMA/CA) où chaque noeud du PAN est autorisé à transmettre une et une seul trame — s'il a plus d'une trame de données à transmettre, l'indicateur de remplissage de sa file d'envoi situé au début de la charge utile (*"payload"*) de la trame lui fera allouer le nombre de *slots* nécessaires durant la *subframe* du cycle suivant. La durée de la CP n'est pas prédéterminée : le routeur écoute jusqu'à la survenue d'un *timeout* d'inactivité radio, ce qui est censé donner assez de temps à chaque noeud ayant des données à émettre (principe similaire à celui vu plus haut pour T-MAC).

TP : *Transmission Period* La retransmission des trames par le routeur — vers le routeur suivant ou le concentrateur final / station de base du réseau sans fil (*"sink"*) — se fait par lots (en mode *"burst"*), comme par exemple pour les protocoles T-MAC ou RI-MAC vus plus haut : une fois la première trame envoyée avec succès, les trames suivantes sont envoyées avec un minimum *d'overhead*.

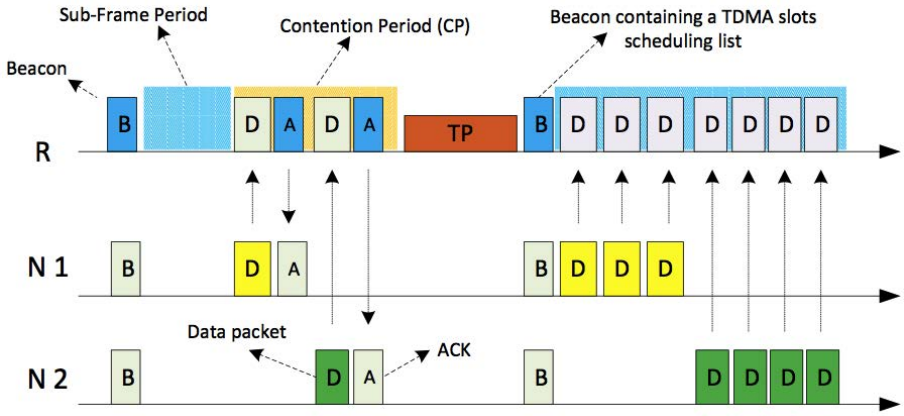

FIGURE 3.22 – *Exemple de transmission de trames entre deux noeuds simples et un routeur avec le protocole iQueue-MAC. (Source : [Song, 2013])*

Donnons ici un exemple de déroulement d'une transmission d'un lot de trames depuis deux noeuds simples vers un routeur iQueue-MAC, tel que montré dans la figure 3.22 page précédente :

1. Soit un routeur R, et deux noeuds N1 et N2, ayant respectivement 4 et 5 trames de données à transmettre ; on suppose qu'aucune autre transmission n'est en cours à ce moment-là dans ce PAN.
2. Pendant le premier cycle, N1 et N2 reçoivent le *"beacon"* de synchronisation. Aucun slot TDMA n'étant alloué, ils attendent tous deux la période de contention CP, et émettent chacun leur première trame de données ; la méthode CSMA/CA étant utilisée, ils reçoivent un acquittement pour l'envoi de ces premières trames.
3. Le routeur termine son cycle par sa période de transmission TP, et le premier cycle se termine.
4. Le routeur ayant reçu la taille des files d'envoi des deux noeuds lors de la période de contention précédente, il alloue donc le nombre de *slots* TDMA nécessaire aux différents noeuds : 3 pour N1, et 4 pour N2. Une fois les durées nécessaires calculées, le second cycle commence, et un nouveau *"beacon"* est envoyé.
5. Les noeuds N1 et N2 reçoivent ce *"beacon"* et se synchronisent pour l'envoi de leurs données durant la *subframe*
6. Les trois premiers *slots* TDMA étant alloués à N1, celui envoie dès le début de la *subframe* ses trames restantes. Sa file d'envoi étant désormais vidée, il peut retourner en mode sommeil, jusqu'à ce qu'il ait par la suite de nouvelles données à envoyer.
7. Les quatre *slots* TDMA suivants étant alloués à N2, il peut lui aussi envoyer à son tour les trames de données restant dans sa file d'envoi, et à son tour passer en mode sommeil pour économiser son énergie.
8. Toutes les données ayant besoin d'être transmises sont désormais arrivées au routeur, qui peut terminer sa *subframe* en période de sommeil, puis retransmettre ces trames de façon adéquate durant la période de transmission (TP) qui terminera ce second cycle (mais n'est pas montrée sur la figure 3.22 page précédente).

Les différentes expériences menées dans la publication [Zhuo et al., 2013] ont montré la nette supériorité d'iQueue-MAC en termes de qualité de service — taux de trames transmises avec succès, délai total de transmission — dans toutes les configurations, vis-à-vis de protocoles tels que RI-MAC ou CoSenS (lequels avaient eux-mêmes déjà prouvé leur supériorité sur les protocoles LPL classiques couramment utilisés). Cette supériorité est notamment flagrante pour les charges réseaux intenses, ou pour le traitement de pics soudains de trafic. Ce protocole semble donc se poser en concurrent sérieux pour le nouveau protocole 802.15.4e ; ce dernier ayant pour avantage sa capacité de traitement multicanal, tandis que iQueue-MAC semble plus à même d'exploiter au mieux le débit maximal théorique d'une unique fréquence

radio, ce qui rend sa conception et son implantation plus simple — avantage non négligeable lorsqu'il s'agit d'implanter un protocole MAC sur un appareil à fonctionnalités limitées (« RFD » standard IEEE 802.15.4).

3.2.6.3 Autres évolutions de protocoles basés sur la contention

Outre les protocoles avancés développés par notre équipe que nous avons décrit ci-dessus dans les sections 3.2.6.1 à 3.2.6.2 pages 47–51, d'autres équipes de recherche continuent également d'apporter de nouvelles idées et techniques pour améliorer les performances et/ou diminuer la consommation énergétique des protocoles MAC / RDC.

Nous citerons ici, à titre d'exemple, les travaux menés à l'Université de Strasbourg pour améliorer les protocoles basés sur la technique LPL, grâce notamment à un mécanisme nommé T-AAD (*lightweight "Traffic Auto-ADaptations"* [Papadopoulos et al., 2014], conçu pour permettre aux protocoles LPL de mieux gérer les pointes (*"bursts"*) de trafic réseau.

Le mécanisme **T-AAD** consiste à adapter dynamiquement les paramètres de configuration des protocoles MAC en fonction des variations du trafic réseau. Le principal paramètre d'un protocole LPL étant la durée de la période de sommeil (ici appelée ST) entre deux CCA consécutifs — c'est-à-dire en fait la durée du cycle de ces protocoles LPL —, T-AAD se propose d'alterner la durée de ST entre deux extremums, ST_{min} et ST_{max}, selon la charge réseau. ST_{max} est la durée de cycle longue (par exemple, 500 ms), employée par défaut, lorsque la charge réseau est à son niveau « de base ». ST_{min} est la durée de cycle courte (par exemple, 32 ms), utilisée temporairement afin de prendre en charge efficacement les pointes de trafic réseau.

Afin de pouvoir estimer dynamiquement la charge réseau à venir, T-AAD impose à chaque noeud d'ajouter, à chaque paquet émis, le nombre de paquets lui restant à émettre dans sa file d'envoi. Un noeud récepteur employant T-AAD utilise alors cette information pour calculer le temps durant lequel il va passer sa durée de cycle à ST_{min} pour gérer cette charge à venir : cette durée de fonctionnement « intensif » est nommée T_{adapt}. Une fois la durée T_{adapt} écoulée, le noeud revient à la durée de cycle par défaut ST_{max}. (Notons que quand les émetteurs ne produisent qu'une charge réseau modérée, T_{adapt} sera calculée à la valeur 0, et un récepteur gardera alors un cycle standard de durée ST_{max}.)

Le but du mécanisme T-AAD est donc de rendre les protocoles LPL « classiques » (comme X-MAC ou ContikiMAC) auto-adaptatifs à la charge réseau, comme le sont naturellement CoSenS / S-CoSenS et iQueue-MAC.

Notons toutefois qu'il s'agit d'un mécanisme bien plus simple que pour nos deux protocoles, la durée de cycle variant de façon discrète entre deux valeurs, et non de façon continue et précise. L'avantage de T-AAD est d'être ainsi d'une conception bien plus simple, au prix d'une moindre précision dans l'adaptation de la durée des cycles MAC ; toutefois, comme ce mécanisme doit s'adapter à des protocoles n'ayant pas été au départ conçus pour être auto-adaptatifs, une telle simplicité peut être considérée comme mieux adaptée à la cible choisie.

Ce mécanisme T-AAD a été implémenté et testé pour améliorer des protocoles LPL classiques comme X-MAC [Papadopoulos et al., 2014] [Papadopoulos et al., 2015a] (se reporter à la section 3.2.2.3 page 39 pour les détails sur le fonctionnement « standard » de ce protocole). Les changements apportés par le mécanisme T-AAD dans le fonctionnement de X-MAC sont illustrés dans la figure 3.23.

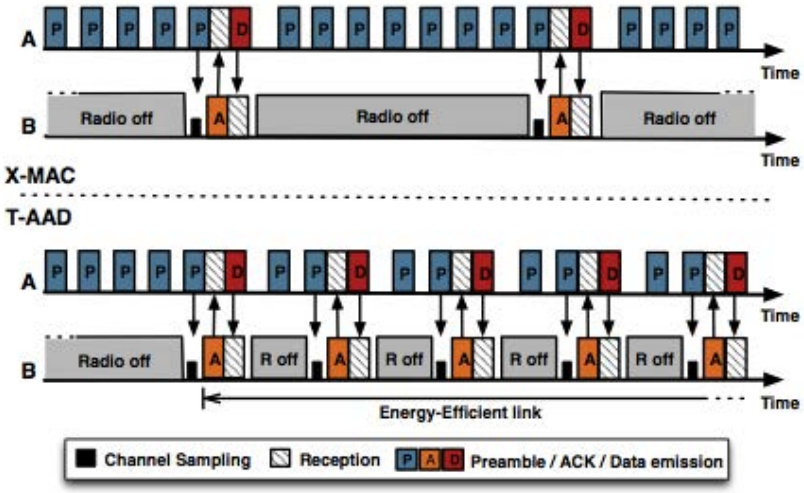

FIGURE 3.23 – *Modification du fonctionnement du protocole X-MAC par le mécanisme T-AAD. (Source : [Papadopoulos et al., 2014])*

Il a été montré dans [Papadopoulos et al., 2014] que ce mécanisme simple — donc relativement facile à implémenter et peu coûteux en occupation mémoire et en temps processeur — a permis d'améliorer significativement les performances (notamment en termes de délais de transmission) et l'efficacité énergétique du protocole X-MAC ainsi modifié. Ces améliorations ont été constatées lors de tests faits sur du matériel réel, à savoir : les noeuds du *testbed* IoT-LAB [FIT IoT-LAB, 2008].

Plus récemment, cette même équipe a publié plusieurs versions améliorées de ContikiMAC (cf. section 3.2.2.4 pour plus de détails sur ce protocole MAC / RDC), destinées à être mieux adaptées aux noeuds mobiles, nommées M-ContikiMAC et ME-ContikiMAC [Papadopoulos et al., 2015b] (M signifiant ici *"Mobile"*, et ME *"Mobile Enhanced"*).

Le protocole **M-ContikiMAC** consiste à utiliser pour émettre ses trames un mode de transmission *"anycast"*, permettant à n'importe quel noeud à portée de recevoir une trame. Contrairement toutefois au mode *"broadcast"* que nous avons vu pour l'envoi de *beacons*, en mode *"anycast"* tout récepteur est tenu de renvoyer une trame d'acquittement, comme pour une transmission classique (également appelée *"unicast"*). Une fois un ou plusieurs récepteurs identifiés grâce aux premiers envois de trames en mode *"anycast"*, les transmissions suivantes de trames de données peuvent se faire en mode *"unicast"* classique suivant le mécanisme standard de ContikiMAC. Ce mode opératoire, bien adapté aux noeuds mobiles ne pouvant compter sur des

protocoles de routage avancés pour découvrir leur entourage, peut être illustré grâce au schéma montré en figure 3.24 (où un noeud mobile communique avec un noeud fixe parmi deux présents dans un WSN donné).

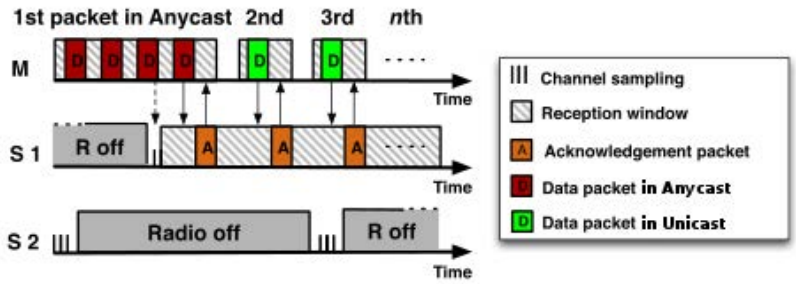

FIGURE 3.24 – *Fonctionnement du protocole avancé M-ContikiMAC, optimisé pour les noeuds mobiles des WSN. (Source : [Papadopoulos et al., 2015b])*

Sur la base de M-ContikiMAC, la même équipe a par la suite développé **ME-ContikiMAC**. Ce dernier protocole a été conçu pour mieux gérer, par rapport à M-ContikiMAC, les problèmes liés à la duplication des trames et à l'optimisation des délais de transmission. Pour ce faire, ME-ContikiMAC envoie désormais en *"anycast"*, pour chercher des récepteurs et établir des « connexions », non plus des trames de données (comme dans M-ContikiMAC), mais des trames de contrôle, dont l'émission est maintenue même en cas de collision au niveau des trames d'acquittement, jusqu'à la réception d'une trame d'acquittement correcte. L'envoi des trames de données se fait ensuite uniquement en *"unicast"*, jusqu'à la fin de la transmission ou la « perte de connexion » (par exemple quand le noeud mobile n'est plus à portée).

Le fonctionnement de ME-ContikiMAC peut être illustré par la figure 3.25 page suivante. On y voit notamment la façon dont ce protocole gère les collisions au niveau des trames d'acquittement, et ce avec un noeud mobile communicant avec trois noeuds fixes.

[Papadopoulos et al., 2015b] met en évidence de nettes améliorations du point de vue de la qualité de service (déduplication de paquets, occupation du médium radio et délais de transmission réduits) et de la consommation énergétique. Ladite publication montre en particulier que les améliorations apportées par ME-ContikiMAC sont sensibles au niveau des WSN ayant des noeuds statiques mais aussi et surtout des noeuds mobiles. (Notons toutefois que cet article ne s'appuie, contrairement aux publications sur T-AAD, que sur des simulations effectuées sous Cooja.)

On voit donc au final que de nombreuses équipes de recherche continuent à explorer activement et sans relâche de nombreuses pistes pour améliorer les couches MAC / RDC des piles protocolaires des WSN, que ce soit par la conception de protocoles entièrement nouveaux — tels CoSenS / S-CoSenS et iQueue-MAC — ou par l'invention de mécanismes astucieux et efficaces pour améliorer les protocoles existants.

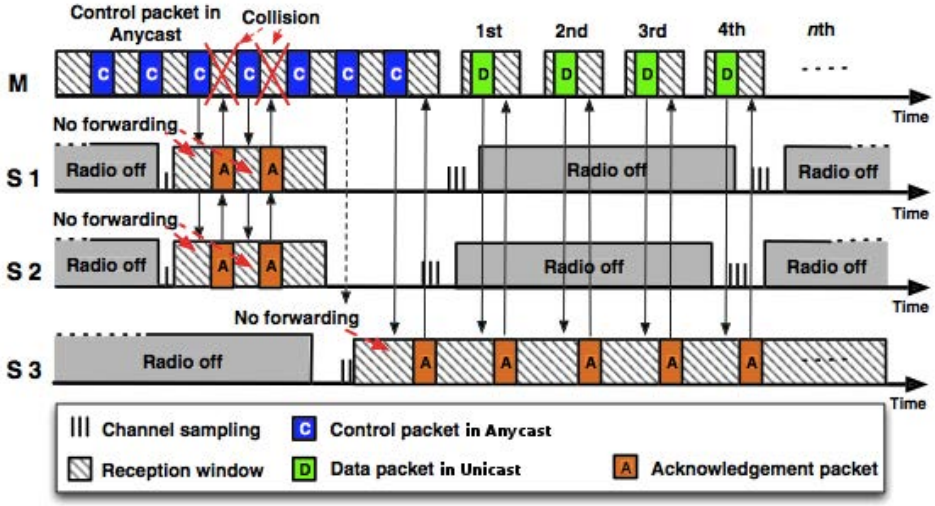

FIGURE 3.25 – *Fonctionnement du protocole avancé ME-ContikiMAC. (Source : [Papadopoulos et al., 2015b])*

Les deux approches peuvent l'une comme l'autre permettre d'obtenir des résultats significatifs en termes d'optimisation de la qualité de service et de la consommation d'énergie.

3.2.6.4 Approches multi-couches : l'exemple de la pile OCARI

Outre le seul développement de protocoles MAC / RDC — c'est-à-dire de solutions relevant exclusivement de la couche 2 du modèle OSI —, plusieurs travaux consistant en le développement de piles protocolaires complètes, pouvant aller jusqu'aux couches du plus haut niveau, ont été menés. On peut noter que de tels travaux ont été menés plus souvent dans un cadre industriel que dans celui de la recherche purement académique, comme c'était majoritairement le cas des protocoles que nous avons vu jusqu'ici dans cet état de l'art.

De telles approches, dites « multi-couches » ou *"cross-layer"*, incluent donc une couche MAC qui, par définition, est intimement liée au reste de la pile, et n'est *a priori* pas conçue pour être implantée de façon indépendante.

Un exemple d'une telle pile protocolaire complète, basée sur la couche physique du standard IEEE 802.15.4 (sur la bande 2,4 GHz uniquement), est OCARI (Optimisation des Communications Ad-hoc pour les Réseaux Industriels) [Agha et al., 2009]. Comme son nom l'indique, il s'agit ici d'un effort de recherche appliquée — mené par une alliance regroupant différents acteurs majeurs, industriels et académiques — destiné à fournir une pile réseau à très haute fiabilité, afin d'offrir une solution pour des applications critiques comme celles liées aux centrales nucléaires ou aux navires militaires.

De cet effort est née une pile protocolaire complète, allant de la couche MAC / RDC (niveau 2 OSI) aux API destinées à la programmation d'applications (assimi-

lables au niveau 7 OSI). Compte-tenu du sujet de notre thèse, nous nous cantonnerons à l'étude de la couche MAC, nommée **MaCARI**, bien que celle-ci soit étroitement liée aux autres couches d'OCARI. Cette couche est notamment prévue pour faciliter les opérations de routage, en coopération étroite avec les couches supérieures d'OCARI.

L'organisation d'un cycle MaCARI se divise en trois périodes, et est représentée sur la figure 3.26.

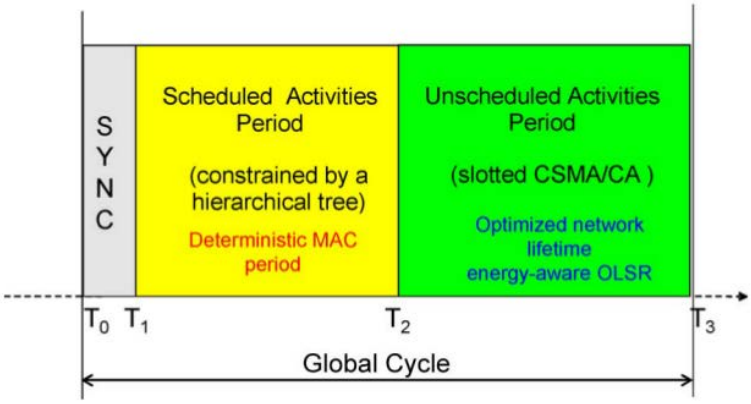

FIGURE 3.26 – *Structure d'un cycle du protocole MaCARI. (Source : [Agha et al., 2009] figure 3)*

Ces trois périodes sont :

1. une période de synchronisation des différents noeuds (T_0–T_1), en gris sur la figure 3.26 ;
2. une période de transmissions pré-déterminées (T_1–T_2), basée sur une méthode TDMA optimisée, assurant une réservation de *slots* évitant toute interférence pour toute transmission, en jaune sur la figure 3.26 ;
3. une période de transmissions « spontanées » (T_2–T_3), basée sur la contention (CSMA/CA slottée), en vert sur la figure 3.26.

Le protocole MaCARI est véritablement un protocole hybride, puisque l'on peut le considérer à la fois comme un protocole synchrone, un protocole basé sur le multiplexage temporel (TDMA), et un protocole basé sur la contention. Il s'agit en outre d'un protocole MAC / RDC complet, l'émetteur / récepteur radio d'un noeud donné pouvant être mis hors fonction pour économiser l'énergie durant la période « déterministe » quand ce noeud n'a pas à communiquer durant son *slot* assigné.

Si le protocole MaCARI est, comme on le voit, très performant et ingénieux, son intégration étroite avec le reste de la pile OCARI — le rendant délicat à porter et à utiliser indépendamment de cette pile — et son orientation très industrielle ne lui ont jusqu'ici pas permis d'apparaître dans des OS pour WSN comme Contiki. De plus, la présence d'une pile fournissant un ensemble très complet de fonctionnalités, pouvant presque faire doublon avec un système d'exploitation proprement dit, rend un tel portage encore plus sujet à caution.

Rappelons enfin que, les piles protocolaires complètes étant au-delà du sujet de la présente thèse, nous n'avons au cours de cette dernière effectué aucun travail avec OCARI ou MaCARI.

3.2.7 Discussion : les protocoles MAC / RDC

En sus de l'évolution du protocole IEEE 802.15.4, et pour tenter de dépasser les limitations de celui-ci, la recherche académique a depuis maintenant plus d'une douzaine d'années proposé de nombreux protocoles MAC alternatifs. Comme nous venons de le voir, plusieurs approches ont été explorées : protocoles basés sur la contention — qu'ils soient synchrones ou asynchrones, ces derniers pouvant être basés sur l'initiation des transmissions par les noeuds émetteurs (LPL) ou par les noeuds récepteurs (LPP) —, protocoles basés sur l'ordonnancement par multiplexage temporel (TDMA) ou fréquentiel (FDMA) des transmissions. Enfin, des protocoles hybrides, exploitant plusieurs de ces approches, ont plus récemment été publiés : nous avons présenté deux de ces protocoles conçus par notre équipe, et l'amendement 802.15.4e du standard IEEE repose lui-même sur une utilisation simultanée des deux techniques de multiplexage temporel et fréquentiel.

Toutes ces approches ont été explorées au cours du temps par diverses équipes. Notre présentation dans la présente section est, comme nous l'avons signalé, loin d'être exhaustive, et la figure 3.27 page ci-contre reprise de l'article de référence de [Huang et al., 2013] montre l'intense activité de recherche ayant eu lieu dans ce domaine. (Il faut de plus ajouter que cette figure date de fin 2011, et ne prend pas en compte les développements les plus récents, comme l'amendement 802.15.4e du standard IEEE.)

Pour le moment, et malgré tous ces développements, les protocoles asynchrones LPL restent, du moins dans le milieu académique, les plus couramment utilisés. Ils sont en effet bien adaptés aux trafics réseaux modérés, ce qui correspond au mode de fonctionnement de la majorité des réseaux de capteurs sans-fil à l'heure actuelle, et permettent également une bonne économie des ressources, tant du point de vue énergétique (économie de la batterie des noeuds) que matériel (simplicité d'implantation, d'où une faible occupation de la mémoire — d'une capacité souvent extrêmement limitée — de ces appareils). Le protocole ContikiMAC, grâce à la forte influence du système d'exploitation Contiki dans le domaine des réseaux de capteurs sans-fil (comme nous allons le voir dans la section 3.3 page suivante) tend actuellement à devenir le standard de fait dans les différents travaux de recherche dans le domaine.

Toutefois, l'adoption de l'amendement 802.15.4e du standard IEEE, avec sa couche MAC évoluée (mécanisme TSCH), et le développement d'une pile réseau complète sur cette base (projet 6TiSCH [Thubert et al., 2013] [Palattella et al., 2014] [Dujovne et al., 2014]) pourraient prochainement changer la donne, et imposer un nouveau standard notamment dans les réseaux industriels exigeants — surtout ceux composés de noeuds puissants pouvant jouer le rôle de FFD (*"Full Function Devices"*). Le développement de la pile réseau avancée OpenWSN est un autre facteur susceptible de faciliter une telle évolution.

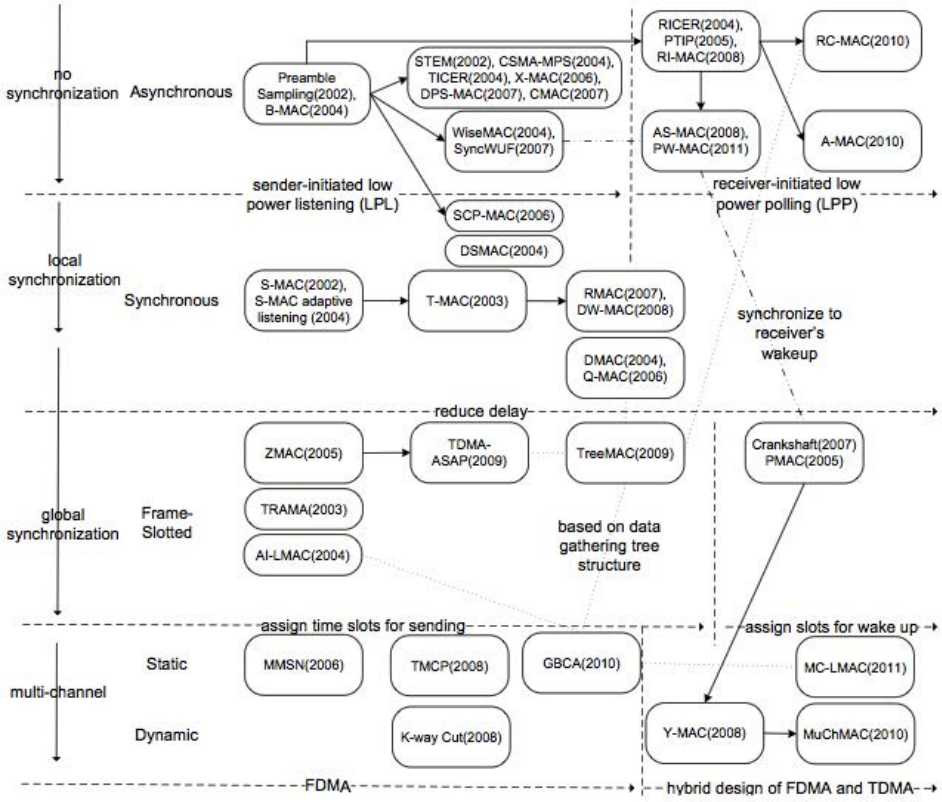

FIGURE 3.27 – *Taxonomie des différents protocoles MAC conçus pour les réseaux de capteurs sans-fil (travaux académiques). Source : [Huang et al., 2013]*

3.3 Systèmes d'exploitation dédiés

Des systèmes d'exploitation spécialisés pour les appareils embarqués spécifiques que sont les noeuds des réseaux de capteurs sans-fils (les *"motes"*) ont été conçus et publiés depuis maintenant plus d'une dizaine d'années.

Rappelons que pour des raisons juridiques aussi bien que techniques — possibilité de modifier et d'améliorer le cœur et les différents composants du système selon nos besoins — nous n'avons étudié et envisagé l'utilisation que des systèmes à licence libre et *open source*.

3.3.1 Rappels sur les notions de multi-tâche

À un moment donné, un micro-contrôleur (du moins la majorité d'entre eux, qui ne disposent que d'un unique « cœur ») ne peut exécuter qu'une tâche à la fois. Pour qu'un système informatique, comme une *mote*, puisse effectuer plusieurs tâches, il est nécessaire de passer régulièrement et très rapidement d'une de ces tâches à une autre. Cela introduit la notion de ***changement de contexte*** : lorsqu'on passe d'une tâche A à une autre tâche B, on sauvegarde l'état de A pour charger celui de B ;

l'état d'une tâche étant nommé son **contexte** (il s'agit des valeurs des registres du processeur, des périphériques, etc.)

L'espace où sont sauvegardés les contextes des différentes tâches est nommé ***pile*** (en anglais : *"stack"*). Il s'agit d'une structure de données variable située en mémoire (RAM donc), suivant le paradigme « dernier entré, premier sorti » (en anglais LIFO : *Last In First Out*).

Notons que la pile n'est pas dédiée aux seuls contextes de tâches : elle sert également de zone de travail en mémoire pour les programmes en cours d'exécution (variables locales, appels de sous-programmes, etc.)

Selon la conception du système d'exploitation, il peut y avoir une seule pile commune à tout le système, ou plusieurs piles dédiées aux différentes tâches, comme nous allons le voir.

On distingue deux principaux modèles conceptuels pour réaliser un système multi-tâche :

le multi-tâche coopératif : dans ce modèle, chaque tâche doit prévoir explicitement dans son code source de « passer la main » (par l'appel à une fonction dédiée du système d'exploitation). Ce modèle rend les systèmes d'exploitation plus simples à concevoir et plus compacts : il est en effet possible dans ce cas d'employer une pile unique pour l'ensemble du système. En contrepartie, la programmation d'applications est plus contraignante : le développeur doit gérer lui-même le basculement d'une tâche à une autre, sinon le système est condamné à mal fonctionner, et même en général à se bloquer (« se planter »). Ce modèle rend donc au final le système moins robuste : une tâche mal programmée (par exemple : ne passant pas la main) peut mettre tout le système hors service.

le multi-tâche préemptif : dans ce modèle, le système d'exploitation alloue lui-même les intervalles d'exécution à chaque tâche, et planifie automatiquement les changements de contexte entre celles-ci (cette capacité d'interrompre une tâche sans collaboration de sa part est nommée **préemption**). Ce modèle implique des systèmes d'exploitation plus évolués et gourmands en ressources : il est en effet nécessaire d'avoir une pile distincte par tâche. En contrepartie, la programmation d'applications est nettement simplifiée : le développeur n'a pas à se soucier des changements de tâches. Le système est en outre nettement plus robuste : une tâche « bloquée » ne va pas systématiquement mettre hors service tout le système.

En outre, la capacité d'interrompre à n'importe quel moment une tâche en cours est un pré-requis pour implanter un système temps réel, qui a besoin de pouvoir réagir à un évènement comme une interruption en un délai limité. C'est pourquoi les systèmes offrant des fonctionnalités temps-réel sont basés sur le modèle préemptif, le multi-tâche coopératif — qui oblige à attendre que la tâche en cours soit prête à passer la main — étant par conception défavorable à ces fonctionnalités.

Cela est illustré par la figure 3.28 page ci-contre. On notera notamment la latence présente en modèle coopératif **A** due à l'attente de la fin de la tâche en

cours, absente du modèle préemptif **B** pouvant interrompre une tâche à tout moment (sauf à être déjà en train de gérer un évènement ou une interruption).

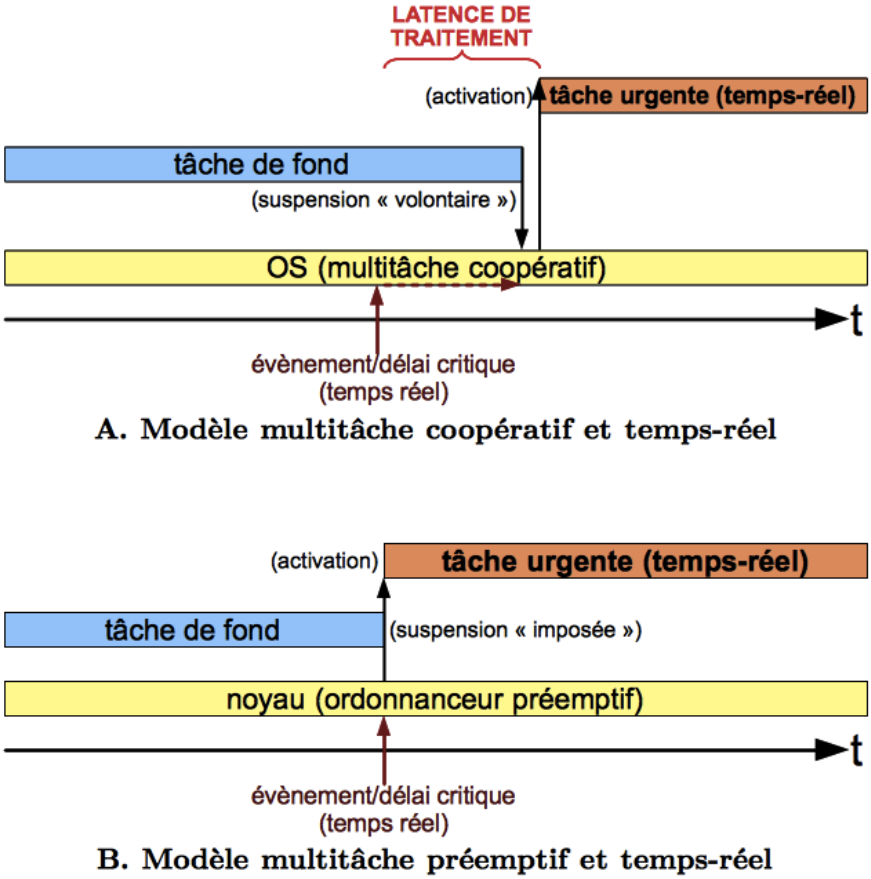

FIGURE 3.28 – *Principe de base du traitement des évènements temps-réel par les diférents modèles de gestion multi-tâche.*

3.3.2 TinyOS

Le premier système ayant connu un large succès dans le domaine des réseaux de capteurs sans-fil est **TinyOS** [Levis et al., 2005]. Il s'agit d'un système *open-source* et libre (sous license BSD), dont la première version stable (version 1.0) a été publiée en septembre 2002. Il est extrêmement léger, et par là-même très bien adapté aux appareils limités que sont les noeuds des réseaux de capteurs sans-fil de la première génération (Mica2, MicaZ [DataSheet MicaZ, 2007], TelosB / SkyMote [DataSheet TelosB, 2006], etc.).

Ce système a permis de nombreuses avancées dans le domaine, comme la possibilité d'utiliser le protocole réseau Internet (IP, y compris dans sa dernière version :

IPv6) ainsi que des protocoles de routage (comme RPL) sur les réseaux au standard IEEE 802.15.4. Il a également introduit la possibilité de simuler des réseaux de *motes* fonctionnant sous TinyOS, grâce au simulateur **TOSSIM** [Levis et al., 2003].

L'un des principaux défauts de ce système est la nécessité d'apprendre un langage spécifique — nommé **nesC** [Gay et al., 2003] — pour pouvoir travailler et développer des applications avec TinyOS. Contrairement à ce que son nom pourrait laisser penser, le langage nesC est très différent du langage C classique et de tous les autres langages informatiques impératifs couramment utilisés — il ressemble plutôt, dans sa philosophie, aux langages de description de matériel utilisés dans la conception de circuits intégrés, comme VHDL ou Verilog. En cela, il peut se révéler difficile à apprendre et à maîtriser pour des programmeurs ayant une formation de développement informatique classique.

La présence de ce langage spécifique au sein du projet TinyOS n'est pas un hasard : TinyOS est en effet construit sur ses propres paradigmes spécifiques. Ce système gère les entrées / sorties (I/O) de façon totalement asynchrone, et fonctionne autour d'une pile unique, depuis laquelle les différents composants constituant une application TinyOS sont appelés en tant que *callbacks* statiquement liés. TinyOS a en effet été conçu pour fonctionner de façon principalement évènementielle, via la programmation de gestionnaires d'interruptions.

La notion de « tâche » de longue durée existe, mais la gestion du multitâche est sous TinyOS particulièrement limitée : les « tâches » sont exécutées l'une après l'autre, suivant une file (FIFO) dont l'ordre ne peut être modifié — l'ordonnanceur de TinyOS n'est pas conçu pour changer dynamiquement l'ordre des tâches à exécuter en fonction du déroulement de l'application et de son environnement. Les tâches ne peuvent en outre être préemptées que par des interruptions (« évènements ») mais jamais par d'autres tâches. Le manuel de TinyOS définit en fait les tâches comme des appels de procédures différés (*"Deferred Procedure Calls"*). Ce fonctionnement permet toutefois de minimiser grandement la consommation d'énergie, toute la *mote* étant en sommeil lorsqu'aucune tâche n'a à s'exécuter (le prochain réveil étant provoqué par « l'évènement » suivant).

Il est clair que le fonctionnement de TinyOS et ses paradigmes sont particulièrement atypiques par rapport aux autres systèmes décrits dans la présente section. Ceci se ressent dans la programmation d'applications, qui nécessite de maîtriser ces notions pour savoir décomposer tout programme en divers « évènements », « tâches » et autres « commandes » (appel d'un composant à un autre) pour créer les composants constituant les applications. La difficulté d'apprentissage et de maîtrise de ce système pour un informaticien formé et habitué aux notions classiques de programmation sera ainsi assez ardue.

L'ensemble des architectures matérielles sur lesquelles TinyOS a été porté semble actuellement se limiter aux familles AVR et MSP430 (un portage sur des architectures plus puissantes comme les Cortex-M est évoqué dans la FAQ du site Web de TinyOS, mais son état d'avancement actuel est inconnu).

Enfin, signalons que TinyOS nécessite l'emploi d'outils de développement dédiés (langage spécifique oblige) pour pouvoir être compilé.

Toutes ces limitations, ajoutées à un rythme de développement relativement lent — la dernière version stable (2.1.2) remonte à août 2012 — ont ces dernières années nui à son adoption, et il n'est désormais plus le système de référence ni le plus utilisé dans le domaine des réseaux de capteurs sans-fil.

3.3.3 Contiki

Le système d'exploitation de référence dans le domaine des réseaux de capteurs sans-fil et par extension de l'Internet des objets (IoT) est **Contiki** [Dunkels et al., 2004]. Il s'agit également d'un système *open source* et libre (license BSD), dont la première version publiée remonte à 2002. Le projet Contiki a été à l'origine de nombreuses avancées : on pourra citer entre autres la pile TCP/IP embarquée **uIP** [Dunkels, 2003] depuis étendue en **uIPv6** (qui est présentée comme la pile IPv6 fonctionnelle la plus légère qui soit), la pile réseau **Rime** [Dunkels, 2007] simplifiée et orientée vers les économies d'énergie, ou encore le simulateur avancé de réseaux de capteurs sans-fil **Cooja** [Österlind et al., 2006].

S'il est légèrement plus exigeant en ressources que TinyOS, Contiki reste très léger et particulièrement bien adapté aux *motes* constituant les réseaux de capteurs sans-fil.

Son principal avantage sur TinyOS est d'être basé sur des paradigmes standards, avec une API reprenant les principes classiquement rencontrés dans le domaine des systèmes d'exploitation ; il est en outre codé en langage C standard : ceci rend son apprentissage et sa maîtrise bien plus facile pour un programmeur ayant un cursus classique.

Contiki est basé sur un noyau événementiel, implantant un modèle multitâche coopératif. Il offre également une pile réseau complète, directement prête à l'emploi.

Il a été porté sur une multitude de plate-formes, comprenant de nombreuses *motes* pour réseaux de capteurs sans-fil, mais aussi de vieux modèles d'ordinateurs personnels — Commodore 64, Apple II, Atari 800XL... — et même certains modèles de consoles de jeux (dans des portages non-officiels).

Des modules optionnels permettent également de fournir des fonctionnalités aussi diverses qu'une interface graphique, un système de fichiers, ou encore une mise à jour du programme (*"firmware"*) durant l'exécution (suivant un processus certes complexe).

Toutes ces caractéristiques et avantages ont largement contribué à la diffusion et à l'adoption massive de Contiki, ce qui fait qu'il s'agit désormais du système d'exploitation de référence dans le monde des réseaux de capteurs sans-fil.

Les développeurs de Contiki ont également été actifs quant au développement de la couche MAC/RDC : nombre de protocoles classiques (comme par exemple X-MAC) ont été implantés dans la pile réseau de ce système ; et un nouveau protocole, ContikiMAC [Dunkels, 2011], a été spécifiquement conçu pour jouer le rôle de protocole RDC par défaut de Contiki (en collaboration avec le protocole CSMA/CA du standard IEEE 802.15.4 en tant que couche MAC). Nous avons abordé ce protocole ContikiMAC dans l'étude des protocoles LPL plus haut en section 3.2.2.4 page 39.

Toutefois, la compacité de Contiki, son optimisation et ses fonctionnalités imposent en contrepartie diverses limitations à ce système.

Contiki est basé sur un modèle multitâche coopératif, et non préemptif : l'ordonnanceur de tâches se déclenche en réaction à des « évènements » (pour reprendre la terminologie de Contiki). Cet ordonnanceur ne peut se déclencher que selon un rythme spécifique, dont la fréquence (intervalle de déclenchement) est déterminée à la compilation. Ce modèle de fonctionnement multitâche est, comme nous l'avons vu ci-dessus en section 3.3.1 page 61, un obstacle à la présence de fonctionnalités temps-réel.

Il faut en outre noter que l'ordonnanceur coopératif de Contiki est conçu pour traiter un type spécifique de tâches nommé **protothreads** [Dunkels et al., 2006]. Ce mécanisme permet de gérer différentes files d'exécution (*"threads"*) sans avoir besoin de maintenir une pile d'exécution séparée pour chacun d'entre eux. Le grand avantage de cette technique est la possibilité d'utiliser une pile unique pour tout un système, diminuant ainsi significativement la quantité de RAM nécessaire, sans pour autant devoir se limiter à un mécanisme d'ordonnancement statique comme dans TinyOS. Ce mécanisme de *protothreads* est d'ailleurs utilisé dans plusieurs autres applications en dehors de Contiki. La contrepartie de ce mécanisme est qu'il est nécessaire de respecter certaines règles rigoureuses quant à l'utilisation de certains éléments du langage C : il est notamment impossible d'utiliser l'instruction `switch` dans certaines parties d'un programme utilisant ces *protothreads*.

Toutes ces limitations du système Contiki ont posé des problèmes non résolus pour nos travaux de thèse, comme nous le verrons dans la section 4.1 page 79 du présent manuscrit.

3.3.4 Nano-RK

Un système spécialisé dans les réseaux de capteurs sans-fil présentant des propriétés très intéressantes, et surtout proches de nos besoins, est **Nano-RK** [Eswaran et al., 2005]. Il a été conçu à l'Université de Carnegie Mellon à partir de 2005.

Il s'agit également d'un système *open source*, publié selon un schéma de double licence permettant l'utilisation de la GNU GPL (*General Public License*) si tel est le choix du développeur. On peut donc le considérer comme un logiciel libre.

Celui-ci fournit de nombreuses fonctionnalités intéressantes :

— Un noyau fonctionnant en mode ***tickless*** (c'est-à-dire sans nécessiter un *timer* le déclenchant de façon régulière).

— Un ***ordonnanceur temps-réel*** avec ***multi-tâche préemptif gérant les priorités***.

  La granularité théorique — permettant de descendre jusqu'à la nano-seconde par l'utilisation de la représentation temporelle POSIX, réimplantée sous le nom de `nrk_time_t` — est excellente. Mais en pratique, la granularité effective garantie pour les interruptions (*"timer tick"*) est de l'ordre de la milliseconde [Nano-RK Time Management API, 2015]. Cela implique une gigue (*"jitter"*) pouvant aller jusqu'à 1000 microsecondes, alors que la durée de base d'une période de *backoff* CSMA/CA est de seulement 320 microsecondes. Les fonc-

tionnalités temps-réel de Nano-RK, aussi avancées et complètes soient-elles, offrent donc une granularité pouvant ne pas être suffisante pour nos travaux.
— Un paradigme de ***réservations de ressources*** très intéressant, permettant par exemple d'accorder respectivement 100 ms de temps d'exécution ou l'envoi de 10 paquets par seconde pour une tâche ou un noeud donnés. Il s'agit en fait d'un principe de quotas, assuré par le noyau système, et permettant de prévoir un « budget énergétique » consommé par un appareil donné pendant une période de temps. Il s'agit d'une spécificité très avancée et fonctionnellement très prometteuse de Nano-RK.
— Une occupation mémoire réduite (bien que légèrement supérieure à celle de TinyOS ou Contiki).
— Une gestion intégrée des erreurs fatales (violations d'espace mémoire ou de *timing*, *watchdog*, chutes d'alimentation énergétique...)
— Le système est écrit en langage C standard, et est donc compilable avec le compilateur GNU GCC classique.
— Une pile réseau légère, comprenant notamment plusieurs protocoles MAC classiques, dont le protocole LPL B-MAC (voir section 3.2.2.1 page 38) ; d'autres protocoles spécifiques et expérimentaux sont proposés, ainsi qu'une fonction simple de passerelle vers un réseau IP nommée *"SLIPStream"*.

Ce système a le principal désavantage d'avoir été porté sur un nombre limité d'architectures de microcontrôleurs : seules les plates-formes MSP430 et AVR sont supportées, avec une activité et une spécialisation nettement plus grandes vis-à-vis de cette dernière. Les principales plates-formes matérielles sur lesquelles Nano-RK est employé sont en effet :

— la *"mote"* MICAz [DataSheet MicaZ, 2007], de conception déjà ancienne (contemporaine de la TelosB/SkyMote) basée sur un MCU ATmega128L [DataSheet ATmega128L, 2011] et le classique TI CC2420 [DataSheet CC2420, 2007] comme radio ;
— la plus récente FireFly3 [FireFly3, 2012], spécifiquement conçue par Carnegie Mellon pour servir de plate-forme matérielle à Nano-RK. Elle a la particularité d'être conçue autour d'un microcontrôleur intégrant un émetteur / récepteur radio, l'ATmega128RFA1 [DataSheet ATmega128RFA1, 2014], et peut accueillir plusieurs cartes d'extension, notamment une carte comportant de nombreux capteurs (température, pression, humidité, accéléromètre, audio, mouvement...).

Ces deux *"motes"*, mises en avant sur le site de Nano-RK, sont toutes deux basées sur l'architecture Atmel AVR. Si cela peut présenter certains avantages — comme la possibilité, mise en avant par le projet, de développer et déboguer directement Nano-RK avec l'outil Atmel Studio — cela implique une forte limitation au niveau de la portabilité, laquelle est pourtant l'un des principaux objectifs recherchés en utilisant un système d'exploitation pour développer sur WSN.

Aucune activité de portage sur d'autres matériels ou architectures ne semble entreprise : cela est fort regrettable, sachant que les *"motes"* basées sur des microcontrôleurs 32 bits (comme ceux d'architecture ARM), plus puissantes, apparaissent

et se répandent de plus en plus.

Cela n'empêche toutefois pas Nano-RK d'être encore activement développé (bien qu'avec des moyens humains visiblement limités, comme beaucoup de projets strictement académiques), et de faire régulièrement l'objet de publications, citons par exemple [Buevich et al., 2013].

Toutefois, malgré ces qualités, nous n'avons pas dans le cadre de cette thèse travaillé avec Nano-RK, car nous avons été amenés à découvrir une autre plate-forme logicielle très intéressante et plus ouverte. Nous allons maintenant aborder cette plate-forme dans la section 3.3.5 ci-dessous.

3.3.5 RIOT OS

Nous avons été amenés à nous intéresser, dans le cadre de nos travaux de thèse, à **RIOT OS** [Hahm et al., 2013].

Ce nouveau système — sa première version a été publiée en 2013 — est également *open source* et publié sous une licence libre (LGPL v2.1), et est spécifiquement conçu pour les noeuds de réseaux de capteurs sans-fil.

Il est développé principalement avec le soutien de la *Freie Universität Berlin*, de l'INRIA, et de la *Hamburg University of Applied Sciences*. Il est à noter que l'équipe de développement de ce projet est particulièrement accueillante et ouverte aux contributions extérieures.

Il fournit les avantages de base que l'on peut attendre d'un OS pour réseaux de capteurs sans-fil, notamment la portabilité (il fonctionne sur de nombreux appareils reposant sur les architectures MSP430, AVR et ARM — notamment la famille des Cortex-M), ainsi qu'un ensemble complet de fonctionnalités, notamment une pile réseau.

Parmi les fonctionnalités offertes, on retrouve de nombreuses capacités avancées des autres OS spécialisés dans les capteurs sans-fil (comme Nano-RK), plus d'autres *a priori* inédites :

- Un ***micro-noyau*** léger, dirigé par les interruptions, fonctionnant par défaut de façon *tickless* (c'est-à-dire sans nécessiter qu'un *timer* le déclenche de façon régulière).
- Un ordonnanceur, intégré au micro-noyau, fonctionnant en mode ***multi-tâche préemptif avec gestion des priorités***.
- Une utilisation optimisée des ***timers matériels grâce à une API dédiée***, permettant de programmer le déclenchement d'actions avec une granularité très fine, de l'ordre de la dizaine de microsecondes.
- Des structures de données de base : piles, files, listes chaînées et circulaires.
- Des mécanismes de communication entre tâches (*threads)* : notamment un système de passage de messages — implanté par l'utilisation de queues (FIFO) de messages, appartenant chacune à un *thread* donné —, ainsi que des *mutex*.
- RIOT est entièrement écrit en ***langage C standard*** ; de plus, contrairement à Contiki, il n'y a aucune restriction quant aux éléments utilisables du langage (telles que les limitations imposées par les *protothreads*).

— Une conception claire et modulaire, rendant le développement avec mais aussi ***dans*** le système plus facile et productive.
— Une gestion des erreurs critiques, que nous avons dans le cadre de cette thèse initiée et contribué à mettre en place.

Les trois premières fonctionnalités citées ci-dessus font de RIOT un système ***temps-réel*** à part entière.

Notons que le mécanisme de gestion des *timers* matériels a évolué pendant l'année 2015. L'ancien mécanisme `hwtimer`, qui était intégré au noyau lui-même, a été supprimé et remplacé par un module nommé `xtimer`.

Toutes les expériences décrites dans ce manuscrit ont, sauf indication contraire, été basées sur une version de RIOT utilisant l'ancien système `hwtimer` intégré au noyau. Celui-ci a été conçu pour offrir la possibilité d'utiliser les *timers* matériels du microcontrôleur, et ce avec une granularité aussi fine que le permettent les *"ticks"* de ces *timers*. Sur les premières *motes* que nous avons utilisé (voir section 5.1 page 103), nous avons ainsi disposé d'une granularité d'environ $30,5$ μs pour programmer le déclenchement de nos évènements.

Le nouveau module `xtimer` permet lui de s'abstraire de la notion de *"tick"*, c'est à dire du délai (en général fixe) entre deux déclenchements successifs d'un *timer* matériel, pour proposer des délais exprimés systématiquement en microsecondes, le module `xtimer` étant conçu pour être implanté sur la base d'un timer matériel dont la cadence est fixée à 1 MHz. Cette implantation lui permet donc d'atteindre, en théorie, une granularité à la microseconde près ; dans les faits, les délais d'exécution des fonctions mêmes de `xtimer`, ainsi que la possibilité de préemption par le noyau (hors gestionnaire d'interruption) génèrent une gigue allant jusqu'à quelques dizaines de microsecondes, soit (dans les cas les moins favorables) une granularité comparable à celle offerte par le mécanisme précédent.

Notons quoi qu'il en soit que la disponibilité en standard d'une granularité temporelle, pour les mécanismes temps-réel de RIOT, de l'ordre de la dizaine de microsecondes (c'est-à-dire largement inférieure à la milliseconde) est une fonctionnalité exceptionnelle : nous ne l'avons retrouvée lors de nos recherches dans aucun autre OS dédié aux WSN (par exemple, dans Nano-RK [Nano-RK Time Management API, 2015] qui dispose pourtant d'un mécanisme de *timing* évolué). Même FreeRTOS (cf. section 3.3.7.1 page 72) n'offre par défaut qu'un mécanisme de *software timer* d'une granularité d'1 ms, l'utilisation des potentialités des *timers* matériels étant laissée à la seule charge du développeur d'applications [FreeRTOS Timer Resolution, 2012].

Le fonctionnement *tickless* du noyau et l'utilisation optimale des *timers* matériels font également de RIOT OS une plate-forme logicielle prometteuse pour le développement d'applications particulièrement optimisées et économes en énergie, ces deux fonctionnalités permettant potentiellement de garder le MCU en état de « sommeil » (basse consommation) durant une fraction de temps aussi élevée que possible. Nano-RK a déjà exploré la piste du fonctionnement *tickless* pour augmenter son efficacité énergétique (voir [Nano-RK Time Management API, 2015]).

Un des désavantages de RIOT, par rapport à TinyOS et Contiki, est sa plus

grande exigence en ressources matérielles, notamment concernant l'occupation mémoire. La pile réseau complète (de la couche physique à l'implantation 6LoWPAN en passant par les couches RPL et MAC / RDC) ne peut pas être compilée pour un matériel de type SkyMote / TelosB car la quantité de mémoire exigée dépasse celle disponible sur ces appareils. À l'heure actuelle, des noeuds limités comme ceux basés sur l'architecture MSP430 sont cantonnés sous RIOT au rôle de RFD, les rôles de FFD étant pour l'instant réservés aux matériels plus puissants (comme ceux basés sur des microcontrôleurs ARM).

Notons toutefois que, grâce à l'architecture modulaire du système, le noyau RIOT compilé avec les seules couches PHY et MAC / RDC reste très léger et occupe très peu de mémoire, étant ainsi parfaitement compatible même avec les appareils les plus limités.

En outre, la partie réseau du système ayant fait l'objet d'une réorganisation en profondeur (comme nous le verrons plus en détail ultérieurement en section 4.3 page 95), nous espérons qu'à l'avenir, il sera possible d'utiliser encore plus efficacement des appareils aux ressources limitées sous RIOT.

Notons également qu'outre la pile réseau intégrée de RIOT, la pile **OpenWSN** [Watteyne et al., 2012] a fait l'objet d'un portage sur le noyau RIOT. Ce dernier est donc l'une des deux plates-formes logicielles sélectionnées pour accueillir cette pile à hautes performances (l'autre étant le noyau FreeRTOS [Barry, 2006]).

3.3.6 Autres OS spécialisés

D'autres systèmes d'exploitation spécifiquement conçus pour les réseaux de capteurs sans-fil ont été développés, mais ceux-ci sont nettement moins utilisés, et souffrent de limitations nous ayant empêché d'envisager sérieusement leur utilisation.

SOS [Han et al., 2005] Le développement de ce système a été arrêté en novembre 2008. Ses auteurs recommandent explicitement sur leur site Web de « considérer l'utilisation d'alternatives activement maintenues. »

Lorien [Porter and Coulson, 2009] Si celui-ci est basé sur approche orientée composant intéressante, ce système ne semble pas d'une utilisation très répandue. Il n'est actuellement disponible que pour un seul type de matériel (TelosB / SkyMote) ce qui limite considérablement la portabilité que l'on est en droit d'attendre de l'utilisation d'un OS, donc son intérêt. En outre, son développement semble avoir nettement ralenti, la dernière version stable publiée par le projet Lorien datant de 2011, tandis que le dernier *commit* dans le dépôt SourceForge du projet (r46) remonte à janvier 2013.

Mantis [Abrach et al., 2003] Alors que ce projet prétend être *open source*, celui-ci n'a publié aucune version accessible sur son site SourceForge ; en outre, l'accès au dépôt source du projet (`http://mantis.cs.colorado.edu/viewcvs`) semble défaillant. Enfin, les dernières nouvelles officielles affichées sur la page Web principale du projet parlent d'une version bêta censée être publiée en 2007. Les

dernières publications scientifiques concernant MantisOS semblent également remonter à l'année 2007. Tous ces éléments nous font penser que ce projet est tombé à l'abandon...

LiteOS [Cao et al., 2008] Ce système offre des fonctionnalités très intéressantes, notamment la possibilité de mettre à jour le *firmware* des noeuds à la volée, depuis la connexion réseau sans-fil (mise à jour *"Over-the-Air"*, dont LiteOS semble avoir été le pionnier), ainsi que le support intégré d'un système de fichiers hiérarchique. Malheureusement, LiteOS est actuellement uniquement disponible pour les plates-formes matérielles IRIS et MicaZ, et nécessite l'emploi d'Atmel Studio pour son développement. Cela nuit gravement à la portabilité du système, vu que LiteOS semble très fortement lié à l'architecture de microcontrôleurs AVR.

MansOS [Strazdins et al., 2010] Ce système très récent offre de nombreuses fonctionnalités intéressantes, comme le multi-tâche préemptif (de façon optionnelle), une pile réseau intégrée, et un langage de script. Il est disponible sur deux architectures de microcontrôleurs : AVR et MSP430 (mais malheureusement, pas les architectures ARM dont la progression est constante). En outre, les capacités temps-réel du système semblent limitées : seuls des *timers* logiciels d'une granularité minimale de 1 milliseconde semblent disponibles.

NanoQplus [Kim et al., 2008a] Il s'agit d'un système dédié aux WSN, apparu semble-t-il en 2008, annonçant des fonctionnalités intéressantes comme le multitâche préemptif, une grande portabilité sur de nombreuses architectures de MCUs (allant des 8 bits aux 32 bits), et surtout un système de protection mémoire. L'article de 2008 le présentant [Kim et al., 2008a] semble prometteur, et un autre article de 2011 [Jeong et al., 2011] indique qu'une pile IPv6 ainsi que RPL y ont été portés.
Malheureusement, le site du projet (`http://dukemon.tistory.com/tag/NanoQplus`) est exclusivement en coréen, et seul un faible nombre d'articles écrits en Corée (le plus souvent en coréen) semblent y faire référence — les deux publications citées dans le présent paragraphe, certes en anglais, sont parues dans des journaux / évènements se voulant internationaux mais s'étant déroulés en Corée.
Il semble donc malheureusement difficile de se faire une idée précise de ce système, dont l'adoption hors de son pays d'origine semble actuellement extrêmement limitée.

OpenTag Il s'agit d'une pile réseau chargée d'implanter le protocole DASH7 (standard ISO/IEC 18000-7, spécialisé dans le RFID et incompatible avec le standard IEEE 802.15.4), fournie avec son propre système temps-réel minimaliste (noyau dédié). S'il s'agit *stricto sensu* d'un système fonctionnant sur des WSN, lesdits WSN sont très différents de ceux que nous étudions dans cette thèse — ils fonctionnent par exemple à la fréquence radio de 433 MHz réservée à DASH7. Leur développement a au départ été prévu pour des applications militaires. Il ne s'agit donc pas d'un système comparable à tous ceux que nous avons vu jusqu'à présent, et ne correspond absolument pas aux travaux de notre thèse.

3.3.7 OS temps-réel classiques / généralistes

Une autre possibilité utilisée par plusieurs constructeurs de *motes* est de recourir à des systèmes embarqués « classiques », lesquels offrent le plus souvent des fonctionnalités temps-réel.

3.3.7.1 FreeRTOS

La référence actuelle en matière de système d'exploitation temps-réel *open source* pour l'embarqué est **FreeRTOS** [Barry, 2006].

FreeRTOS est un projet mature (première version publiée en 2003), stable et très largement employé, notamment dans l'industrie.

Il a été porté sur plus de 30 architectures de microcontrôleurs différentes, c'est-à-dire quasiment toutes les architectures présentes sur le marché ; ce qui en fait le système le plus portable cité jusqu'ici dans le présent manuscrit, largement devant n'importe quel système dédié aux réseaux de capteurs sans-fil.

L'un des points forts majeurs de FreeRTOS est son excellente documentation : le code source est abondamment et consciencieusement commenté, et l'auteur publie deux manuels complets (en anglais uniquement) sur son système :

1. un manuel d'utilisation [Barry, 2010], complet et très didactique, tant sur l'utilisation du système lui-même que sur les notions de temps-réel en informatique et les conséquences de ces notions quant aux bonnes pratiques de programmation ;
2. un manuel de référence du système [Real Time Engineers Ltd., 2013], détaillant de façon exhaustive l'API et les options de configuration, avec de nombreux exemples de code (*"listings"*).

Ces deux ouvrages, rédigés avec soin et maintenus à jour, facilitent considérablement l'apprentissage et la programmation avec ce système. Il s'agit également d'une des sources de revenus du projet FreeRTOS, ces deux manuels étant vendus sous forme électronique sur le site Web du projet : cette documentation sur FreeRTOS n'est donc ***pas*** libre, contrairement au système lui-même qui est publié sous la licence GNU GPL.

Le code source du noyau en lui-même se compose uniquement de quatre fichiers source en langage C — dont un représente une fonction optionnelle (coroutines). Le reste du source du projet se compose des portages (couches d'abstraction) pour les nombreuses plates-formes matérielles et compilateurs supportés, et d'exemples. Ce système reste ainsi très léger quant aux ressources matérielles exigées (généralement moins de 10 Ko de code programme) ce qui le rend parfaitement adapté aux systèmes embarqués les plus limités.

La simplicité du noyau (quantité de code minimale) et les critères de qualité stricts imposés au code source contribuent à la robustesse de FreeRTOS.

Cela lui permet d'ailleurs de fournir une version certifiée pour les utilisations dans les systèmes embarqués critiques, nommée **SafeRTOS**. Contrairement au noyau FreeRTOS de base, SafeRTOS n'est ***pas*** un logiciel libre : l'achat d'une licence

payante est exigé, en échange de la certification (IEC 61508 : SIL 3) et d'un support technique avancé. Cette version spéciale, SafeRTOS, est une autre source de revenus du projet FreeRTOS (en plus de la vente de manuels citée plus haut).

Citons également l'existence d'**OpenRTOS**, qui est un clone absolu de FreeRTOS sous une licence ***propriétaire***, vendu à l'intention des industriels rebutés par les exigences de la GNU GPL, et fournissant également un support technique — et par la même occasion une troisième source de revenus au projet FreeRTOS.

Le noyau FreeRTOS fournit des fonctionnalités avancées telles que :

— le multi-tâche, préemptif par défaut, coopératif en option, via un ordonnanceur gérant les priorités ;
— un mode *"tickless"* optionnel ;
— des structures de données de base (listes, files) ;
— des mécanismes de communication entre tâches (*"Inter Process Communication"*) : sémaphores, *mutex* ;
— des *timers* logiciels ;
— et une gestion optionnelle de l'allocation de mémoire (*"heap"*) : trois mécanismes différents disponibles.

Cette compacité a toutefois un prix : FreeRTOS n'est en fait qu'un noyau, fournissant les fonctionnalités de base citées ci-dessus. Il est dépourvu de tout pilote matériel, y compris pour les périphériques de base intégrés aux microcontrôleurs (par exemple : les ports séries ou les GPIO) — ces pilotes de périphériques devant être développés et fournis en sus par les fournisseurs de matériels et / ou d'applications désireux de l'utiliser.

Par conséquent, il est également évident que FreeRTOS ne fournit aucune pile réseau, aussi rudimentaire soit-elle.

Notons qu'il existe des extensions réseau proposées en sus pour FreeRTOS, mais la plupart sont des composants logiciels propriétaires et à source secret (proposés dans le cadre du projet « FreeRTOS plus », un écosystème de logiciels basé sur FreeRTOS), ce qui ne peut convenir pour nos travaux de thèse.

Toutefois, cette situation pourrait changer, car la pile réseau avancée **OpenWSN** [Watteyne et al., 2012] a été développée en prenant pour l'un de ses noyaux de base FreeRTOS (l'autre noyau étant celui de RIOT OS comme nous l'avons vu plus haut). Si l'intégration d'OpenWSN et du noyau FreeRTOS se développe et est amenée à être largement adoptée, nous pourrons alors compter sur un nouveau système dédié aux réseaux de capteurs sans-fil à part entière, *open source* et très performant.

Certains constructeurs de capteurs sans-fil ont bâti leur propre système d'exploitation dédié en se basant sur FreeRTOS : on pourra notamment citer la société HiKoB et son projet OpenLab. Malheureusement, un tel projet est par définition intimement lié à un type de matériel donné, et fait donc perdre l'un des principaux avantages liés à l'utilisation d'un OS, à savoir la portabilité.

3.3.7.2 Autres OS temps-réel

Notons qu'il existe d'autres systèmes temps-réel embarqués et *open source*, tel qu'**Erika Entreprise**, largement utilisé dans l'industrie notamment automobile. Ce système, ayant un long historique (première version datant de 2002), et ayant reçu une certification (OSEK/VDX), a lui aussi été porté sur de nombreuses plates-formes, dont des *motes* comme la MicaZ. Toutefois, les WSN ne sont absolument pas la spécialité de ce système — il n'offre aucune pile réseau —, et s'il a un succès certain dans l'industrie, il est totalement ignoré dans le milieu de la recherche académique (nous n'avons trouvé aucune publication le mettant en œuvre).

De nombreux autres systèmes d'exploitation temps-réel existent, dotés d'une licence libre ou propriétaire — la plupart du temps destinés à l'embarqué —, mais à notre connaissance, aucun d'entre eux n'est employé, même de façon ponctuelle, dans le cadre des WSN.

FreeRTOS est actuellement l'OS temps-réel embarqué « généraliste » ayant largement le plus de succès comme plate-forme de recherche dans le domaine des réseaux de capteurs sans-fil.

3.3.8 Discussion : les systèmes d'exploitation dédiés

Depuis l'an 2000, plusieurs OS *open source* spécialisés dans les réseaux de capteurs sans-fil ont été développés : TinyOS en a été le pionnier. Toutefois, ses nombreuses et importantes limitations, ses paradigmes inhabituels et son langage spécifique inhabituel (nesC) en font désormais un système en perte d'influence.

La référence actuelle en ce domaine est Contiki : sa légèreté quant aux ressources matérielles demandées, ses nombreuses fonctionnalités, assurées par diverses innovations techniques significatives, l'importante communauté de développeurs qu'il a su réunir autour de lui, et désormais le support professionnel assuré par une société fondée par les créateurs du système (Thingsquare) en font le système incontournable et la référence dans le domaine des réseaux de capteurs sans-fil.

Celui-ci souffre toutefois de limitations — principalement liées aux choix techniques effectués lors de sa conception, notamment pour créer un système économe en ressources matérielles.

La communauté académique n'en est pas restée là, et a démarré plusieurs autres projets de systèmes dédiés aux réseaux de capteurs sans-fil. Si nombre d'entre eux semblent être tombés en désuétude, plusieurs projets ont montré des qualités dignes d'intérêt : citons notamment LiteOS, ayant développé la notion de mise à jour du *firmware* des noeuds à l'exécution via la communication sur le réseau (*"On the Air"*), et surtout Nano-RK et RIOT OS, qui sont à la fois des systèmes embarqués temps-réel ***et*** conçus pour les réseaux de capteurs sans-fil.

On citera également les OS embarqués « classiques », dont FreeRTOS qui est la référence actuelle en matière de système temps-réel *open source*. FreeRTOS, comme

RIOT, sont les deux noyaux systèmes sélectionnés pour accueillir la pile réseau avancée OpenWSN, développée pour implanter l'état de l'art en matière de réseaux de capteurs sans-fil (notamment les avancées de l'amendement 802.15.4e et la pile protocolaire 6TiSCH).

La liste des principaux OS utilisables pour les réseaux de capteurs sans-fil, avec leurs points forts et leurs faiblesses, peut ainsi être résumée dans les données de la table 3.1 page suivante.

Ainsi, si Contiki est à l'heure actuelle la référence en matière de système pour réseaux de capteurs sans-fil, les récentes avancées apportées par l'amendement 802.15.4e du protocole IEEE pourraient faire évoluer le paysage.

En effet, les insuffisances des fonctionnalités temps-réel de Contiki rendent très délicates les synchronisations temporelles — plus exactement : le déclenchement d'évènements à des moments extrêmement précis, jusqu'à quelques dizaines de microsecondes près — nécessaires au caractère *"slotté"* (TDMA et même FDMA) de la nouvelle couche MAC étendue.

Dans ces conditions, le domaine des plates-formes logicielles pour réseaux de capteurs sans-fil pourrait largement évoluer, et permettre à des systèmes comme RIOT OS et / ou le couple FreeRTOS / OpenWSN de s'imposer à l'avenir dans le paysage des WSN.

Pour les besoins de développement rapide, facile, permettant de passer facilement de l'émulation au test sur matériel de prototypage, une solution originale comme **OpenWiNo** [van den Bossche and Val, 2013], qui n'est pas un OS complet, mais un noyau ultra-léger offrant une pile protocolaire et des fonctionnalités de gestion de *timers*, pourrait devenir une alternative intéressante lorsqu'elle sera publiquement disponible.

3.4 Conclusion : protocoles MAC / RDC et OS spécialisés

L'analyse de l'état de l'art sur les protocoles MAC basés sur la couche physique du standard IEEE 802.15.4 d'une part, et d'autre part sur les systèmes d'exploitation dédiés aux réseaux de capteurs sans-fil, nous permet de faire le lien entre ces deux sujets.

Ainsi, nous pouvons déterminer que pour réussir à implanter efficacement des protocoles MAC avancés permettant d'optimiser à la fois la qualité de service « fonctionnelle » et la consommation d'énergie (voir 3.2.6 page 46), nous avons besoin de fonctionnalités temps-réel permettant de réagir efficacement aux évènements, notamment temporels — en d'autres termes, respecter des délais très stricts —, ces protocoles avancés nécessitant une synchronisation précise entre les différents noeuds communicants, et souvent un recours au multiplexage temporel (TDMA), comme iQueue-MAC, ou même la couche MAC du standard 802.15.4 en « mode *"beacon"* », *a fortiori* la couche MAC améliorée de l'extension 802.15.4e.

Nom	Noyau (capacités)	Exigences matérielles	Pile réseau	Fonctionnalités de base	spécifiques	Portabilité	Adoption
TinyOS	modulaire statique	minimales	oui (intégrée)	gestion asynchrone des E/S	langage spécifique (nesC), simili-« mode *tickless* »	moyenne (AVR, MSP430)	Pionnier, Répandu (mais en baisse)
Contiki OS	multitâche coopératif	très modérées	deux : uIPv6 ou Rime	gestion des E/S, *timers* logiciels, IPC (limitées)	noyau évènementiel, protothreads, reprogrammation à l'exécution	élevée (AVR, MSP430, PIC, etc.)	Très répandu (OS de référence)
Nano-RK	multitâche préemptif temps-réel	relativement modérées	oui (intégrée)	gestion des E/S, *timers* logiciels, IPC	gestion des erreurs fatales, mécanisme de réservation de ressources, mode *tickless*	assez faible (MSP430 mais surtout AVR)	Relativement répandu
RIOT OS	multitâche préemptif temps-réel	moyennes (modérées pour le seul noyau)	oui (intégrée), plus utilisation possible d'OpenWSN (projet externe)	gestion des E/S, *timers* logiciels, IPC, structures de données (listes...)	gestion des erreurs fatales, gestion avancée des timers matériels, reprogrammation à l'exécution en cours de développement, mode *tickless*	bonne (MSP430, ARM, AVR, x86)	Moyennement répandu (en expansion)
Lite OS	multitâche préemptif	modérées	oui (intégrée)	gestion des E/S, mutex	reprogrammation à l'exécution (pionnier), système de fichiers intégré, journalisation d'évènements	faible (AVR uniquement)	Relativement répandu
Mans OS	multitâche préemptif en option	relativement modérées	deux : intégrée et uIPv6	gestion des E/S, *timers* logiciels, mutex	gestion du GPS, reprogrammation à l'exécution, langage de script disponible	moyenne (AVR et MSP430)	Apparemment peu répandu
FreeRTOS	multitâche préemptif temps-réel (coopératif en option)	modérées	non (utilisation possible d'OpenWSN, projet externe)	*timers* logiciels, IPC, coroutines (option), structures de données (listes...)	noyau uniquement, excellente documentation, gestion mémoire, mode *tickless* optionnel	excellente (plus de 35 architectures matérielles)	Extrêmement répandu (généraliste, non dédié aux WSN)

TABLE 3.1 – *Principales plates-formes logicielles utilisables en 2015 dans le cadre des réseaux de capteurs sans-fil.*

C'est pourquoi nous chercherons, dans le chapitre 4 suivant, la meilleure plateforme logicielle (c-à-d. système d'exploitation) pour atteindre ces objectifs.

Chapitre 4

Plates-formes logicielles : évaluation, problèmes et améliorations

Une étape essentielle de nos travaux a consisté à chercher une plate-forme logicielle (c'est-à-dire un système d'exploitation) spécialisée dans les réseaux de capteurs sans-fil et présentant les caractéristiques adéquates pour nos travaux ; ces travaux consistant notamment à implanter et à évaluer les performances des protocoles réseaux avancés décrits dans la section 3.2.6 page 46 du présent manuscrit.

Suite à l'analyse de l'état de l'art sur les systèmes d'exploitation dédiés (cf. section 3.3 page 61), nous avons retenu deux plates-formes logicielles de travail : la référence actuelle, Contiki OS (décrit en section 3.3.3 page 65), et celle nous semblant la plus prometteuse du point du vue fonctionnel, RIOT OS (détaillée dans la section 3.3.5 page 68).

Cette tâche s'est révélée plus longue et ardue que nous ne l'avions imaginé *a priori*, les systèmes de référence établis de longue date ne convenant pas à nos besoins.

4.1 Contiki : développement et limitations

Notons que nous avons dès le départ écarté TinyOS, pour toutes les raisons décrites dans la section 3.3.2 page 63 qui lui est consacrée. Notre premier choix de plate-forme logicielle a donc été logiquement le système de référence actuel, Contiki OS.

Comme indiqué précédemment dans la section 3.3.3 page 65, les qualités proposées par ce système, notamment le support actif des communautés tant académique qu'industrielle, en faisait un choix de départ logique pour effectuer nos développements.

En outre, ce système est depuis 2011 fourni avec son propre protocole MAC, ContikiMAC (décrit en section 3.2.2.4 page 39), lequel est désormais le protocole utilisé par défaut par Contiki. Ce protocole est ainsi devenu le standard de fait dans

nombre de publications récentes dans le domaine des réseaux de capteurs sans-fil.

L'une des premières tâches à laquelle nous nous sommes attelés est ainsi d'implanter S-CoSenS (décrit dans la section 3.2.6.1 page 47 sur nos protocoles avancés) afin de pouvoir effectuer une comparaison équitable entre ce protocole et ContikiMAC. Nous avons préféré commencer par implanter ce protocole plutôt qu'iQueue-MAC, nettement plus performant mais aussi plus complexe à implanter.

L'intégration de S-CoSenS dans la pile réseau Contiki devait notamment nous permettre d'évaluer l'influence de toutes les couches de la pile réseau, et l'interaction de celles-ci avec chaque protocole MAC / RDC, afin d'obtenir des résultats les plus fidèles possibles à la réalité du déploiement d'un réseau de capteurs sans-fil en production.

Malheureusement, durant notre effort d'implantation du protocole S-CoSenS au sein de la pile réseau (*"netstack"*) de Contiki, nous avons fait face à de nombreux problèmes nuisant au développement à l'intérieur de ce système, tant au niveau général que dans le domaine plus spécifique de la pile réseau [Roussel and Song, 2013]. Nous allons dans la présente section décrire les problèmes rencontrés, en tentant — quand cela est possible — de proposer des solutions ou tout au moins des moyens de contournement.

4.1.1 Documentation minimaliste

Ceci est un problème récurrent avec Contiki : à l'exception du code source du système et des exemples d'applications fournis, il n'y a quasiment aucune documentation. Ceci concerne de façon générale l'ensemble de Contiki, y compris et notamment sa pile réseau.

On regrettera notamment l'absence totale de documents techniques de référence, où l'architecture générale du système, les choix de conception et d'implantation seraient expliqués ; une telle documentation de référence représente un outil essentiel pour réellement comprendre une plate-forme logicielle, et ainsi permettre aux développeurs débutants de devenir rapidement efficaces dans leur travail.

Il y a également très peu de documents d'introduction (« tutoriels ») : le seul document d'initiation officiel est la page *"get started"* du site Web du projet Contiki. Celle-ci montre comment télécharger et utiliser la distribution Linux dédiée au test du système, nommée *"Instant Contiki"*, pour effectuer rapidement des simulations de réseaux de capteurs sans-fil — grâce à l'utilisation du simulateur **Cooja** [Österlind et al., 2006] fourni par le projet Contiki — puis télécharger des programmes d'exemple sur du matériel (des *motes* de type Zolertia Z1). Aucun document n'est disponible pour montrer et (plus important encore) expliquer aux développeurs les nombreuses fonctionnalités de Cooja ; aucun document pour détailler comment programmer des applications avec Contiki ; aucune introduction signalant quelles sont les spécificités du développement embarqué sur des systèmes aussi contraints que les *motes* constituant les réseaux de capteurs sans-fil (par opposition au développement « classique » sur PC). Enfin, l'absence de documentation sur le fonctionnement interne de Contiki,

et sur les méthodes éventuelles pour adapter le système à ses propres besoins, est particulièrement regrettable pour qui souhaite mener des travaux concrets de recherche et de développement avec, et surtout ***dans*** ce système — or, de telles initiatives d'exploitation « avancée » sont appelées à être relativement nombreuses, étant donné le statut d'OS pour WSN de référence que possède actuellement Contiki.

En résumé, une telle absence de documentation raisonnablement accessible rend l'approche initiale de Contiki difficile pour les nouveaux développeurs, et contribue à leur imposer une courbe d'apprentissage relativement élevée.

La principale source de documentation, en matière de développement sous Contiki, est la *mailing-list* `"contiki-developers"`[1]. Toute personne souhaitant développer sur cette plate-forme logicielle (qu'il s'agisse d'applications ou au niveau du système) ne peut espérer atteindre un quelconque but sérieux sans souscrire à cette liste, et demander de l'aide et des renseignements à ses membres. Bien que cette liste soit une source riche d'informations, soit réactive, et en général bien disposée à l'égard des nouveaux venus, on peut difficilement considérer qu'elle remplace de façon satisfaisante le manque de documentation technique et de tutoriels.

Ajoutons également que, si de nombreux exemples d'applications conçues avec le système Contiki sont fournis avec le code source du système, il n'y a par contre aucun exemple de code destiné à s'insérer dans le système lui-même (comme par exemple un *plug-in* pour la pile réseau ou toute autre partie du cœur du système). Le développement au sein du système Contiki, par exemple pour l'adapter à ses besoins ou y rajouter des fonctionnalités, n'en est ainsi qu'encore plus difficile à apprendre et à maîtriser.

4.1.2 Limitations techniques

De nombreuses limitations du système Contiki ont été dictées par les contraintes fortes imposées par les appareils constituant les réseaux de capteurs sans-fil : ces noeuds sont en effet extrêmement limités quant à leur puissance de calcul et (surtout) leur espace mémoire disponible.

Nous étudierons dans cette section ces différentes limitations, certaines étant dues à des problèmes plus profonds, découlant directement de choix de conception, notamment concernant la pile réseau de Contiki.

4.1.2.1 Fonctionnalités manquantes : pilotes radio incomplets

La version stable de Contiki alors disponible au moment des présents travaux (version 2.7, datant de novembre 2013) disposait pour les émetteurs / récepteurs radio de pilotes dont l'API n'offrait que les fonctionnalités les plus basiques possibles :

— envoi et réception de trames,
— mise en fonction (*"on"*) et hors fonction (*"off"*) de la radio,
— et vérification de la disponibilité du médium radio (CCA : *Clear Channel Assessment*).

1. Adresse : `contiki-developers@lists.sourceforge.net`
URL de gestion : `https://lists.sourceforge.net/lists/listinfo/contiki-developers`

L'accès à toutes les autres fonctionnalités de l'émetteur / récepteur radio (comme le changement de canal / fréquence, la définition de la puissance d'émission, le changement des adresses des noeuds, etc.) nécessitait d'accéder directement aux registres spécifiques de la puce radio, et faisait ainsi perdre l'avantage de la portabilité offert par l'utilisation d'un OS, ainsi que le principe de séparation des couches de la pile réseau (l'accès à de telles fonctionnalités étant souvent nécessaire par exemple depuis la couche MAC).

Pour résoudre ce problème, nous avons imaginé une extension de l'API des pilotes radio, permettant d'accéder à ces fonctionnalités via un mécanisme standard et flexible de « capacités » (*"capabilities"*) génériques, le tout sans nuire à la compatibilité avec le code existant. Concrètement, nous avons proposé l'ajout à l'API des pilotes radio Contiki de trois fonctions permettant respectivement :

1. d'accéder à des constantes de configuration spécifiques à l'émetteur / récepteur radio voulu (par exemple : la puissance maximale d'émission en dB) ;
2. d'interroger
3. et de définir des paramètres de configuration — ce que nous avons appelé *"capabilities"* — impactant le comportement de la radio (par exemple : le canal radio employé, la ou les adresses, etc.) ;

ces capacités supplémentaires s'obtenant en n'ajoutant qu'une charge minimale de code : par l'ajout de trois pointeurs sur fonction à l'implantation d'un pilote radio Contiki.

Ces propositions ont été soumises, en tant que contributions, par le biais de deux *pull requests* dans le dépôt GitHub principal de Contiki [Roussel et al., 2013c] [Roussel et al., 2014a]. Aucune n'a été acceptée, mais peu de temps après, l'équipe de développement de Contiki a elle-même ouvert une nouvelle *pull request* ([Finne et al., 2014]) ayant effectivement abouti à une API radio étendue reprenant certaines des idées que nous avons soumises (comme le montre la remarque *"The extended radio API as shown below has been adapted based on the discussions in #519."* en [Finne et al., 2014]).

L'amélioration de l'API radio fait d'ailleurs partie des améliorations de Contiki 3.0 vantées par A. Dunkels sur son blog [Dunkels, 2015], en ces termes : *"The radio API has been updated to better match the way the radio duty cycling protocols use the radio. For example, the previous radio API lacked a clean way to set the radio channel, which now is part of the new API."*.

4.1.2.2 Pile réseau centrée sur un unique *"packetbuf"*

Une fonctionnalité unique de la pile protocolaire réseau de Contiki OS est d'être centrée autour d'un unique *buffer* — destiné à contenir la trame réseau en cours de traitement — nommé *"packetbuf"* selon la terminologie de Contiki. Ce *packetbuf* constitue ainsi la « zone de travail » incontournable pour les différentes couches de la pile réseau (comme nous allons le voir en section 4.1.2.4 page 84).

Ce choix de conception a bien sûr été fait pour épargner de la mémoire : sur les noeuds constituant les capteurs sans-fil, cette ressource est extrêmement précieuse car

limitée. Le code programme doit ainsi, pour les appareils les moins puissants, tenir dans quelques dizaines de kilo-octets de mémoire Flash, tandis que les données — dont font partie les trames radio émises et reçues — doivent se partager la place dans une RAM encore plus restreinte (dont la taille ne dépasse parfois pas 2 kilo-octets).

Malheureusement, ce choix — s'il est parfaitement compréhensible — entraîne plusieurs conséquences indésirables :

— ***Le risque de perte de trames.*** Ce *buffer* unique devant être utilisé à tout moment par toutes les couches de la pile réseau, toute erreur dans la synchronisation entre les différents composants de cette pile réseau peut aboutir à l'écrasement du contenu du *packetbuf*, provoquant ainsi une perte de données. Le risque de l'arrivée d'une trame (venant d'être reçue depuis le canal radio) pendant le traitement d'autres données par la pile réseau est particulièrement critique, étant donné que la réception de données depuis le réseau est, par nature, un évènement asynchrone et totalement imprévisible.

— ***L'impossibilité de gérer les files de trames de façon efficace.*** La conception de protocoles MAC performants implique de gérer des files de trames — tant en émission qu'en réception. Bien que Contiki fournisse de tels mécanismes (*"queuebuf"*, *"packetqueue"*), la conception de la pile réseau centrée sur un unique *packetbuf* implique d'incessantes copies entre ce *packetbuf* et ces files. Cela implique des pertes de temps, du gaspillage de puissance du processeur et même de mémoire : la possibilité de désigner une portion de la mémoire (par exemple : une trame donnée dans une file) en tant que zone de travail courante pour la pile réseau durant l'exécution, ou des files de pointeurs sur des zones mémoire, permettrait *in fine* d'utiliser moins de mémoire que le concept actuel de *packetbuf* unique. Notons également que les mécanismes de *"queuebuf"* et de *"packetqueue"* fournis sont eux-mêmes complexes et d'un usage contre-intuitif, car ceux-ci ont dû être conçus pour fonctionner avec ce *packetbuf* unique.

— ***La complexité de la pile réseau.*** La nécessité de gérer correctement cet unique *packetbuf* contribue en réalité à rendre le code de la pile réseau plus complexe, les différents éléments (couches) composant cette pile devant effectuer différentes opérations pouvant s'avérer délicates — comme le besoin d'une synchronisation rigoureuse, ou les copies incessantes depuis ou vers le *packetbuf* —, lesquelles opérations ne seraient pas nécessaires avec une conception plus flexible (c'est-à-dire : basées sur des files de trames).

4.1.2.3 Séparation des couches MAC et RDC

La pile réseau de Contiki est basée sur le principe de séparation des différentes couches. Ce principe est fort logique, et permet d'avoir des éléments de code séparés pour les pilotes des émetteurs / récepteurs radio, les protocoles de routage, les couches de transport, etc.

Dans le cadre de cette stratégie, les concepteurs de Contiki ont choisi de faire une distinction entre la couche **RDC (*Radio Duty Cycle*)** — dont le rôle est

de contrôler comment la radio est mise en activité et à l'arrêt durant les cycles de fonctionnement (*"duty cycles"*) d'un réseau, afin d'implanter une stratégie efficace d'économie d'énergie pour les noeuds dudit réseau — et la couche **MAC (*Medium Access Control*)** — qui, en théorie, se charge uniquement d'ordonnancer et de séquencer temporellement la transmission des trames sur le réseau.

En pratique, la plupart des protocoles MAC modernes gèrent simultanément ces deux aspects, qui sont intimement liés l'un à l'autre. Ces protocoles peuvent ainsi s'adapter dynamiquement pour maximiser l'efficacité du réseau (notion de Qualité de Service : QdS) ***et*** optimiser la consommation d'énergie, le tout en fonction du trafic réseau et de son évolution en cours d'exécution. X-MAC [Buettner et al., 2006], RI-MAC [Sun et al., 2008], S-CoSenS [Nefzi and Song, 2012] et iQueue-MAC [Zhuo et al., 2013] sont des exemples, parmi de nombreux autres, de protocoles jouant à la fois le rôle de couche MAC et de couche RDC.

Cette séparation entre ces deux couches est par conséquent artificielle, et ne fait qu'ajouter une complexité supplémentaire et inutile à l'implantation de ces protocoles, la distinction entre aspects MAC et RDC étant au mieux difficile, et parfois même impossible. Un exemple du côté artificiel de cette séparation est l'implantation du protocole ContikiMAC dans le système Contiki lui-même, ContikiMAC étant considéré comme un protocole RDC, qu'il est conseillé d'utiliser avec la couche `"nullMAC"` (qui n'est en fait qu'une « coquille vide » transférant sans aucun traitement les messages à la couche RDC sous-jacente) ce qui prouve implicitement que ContikiMAC joue bien les deux rôles à lui seul.

4.1.2.4 Complexité excessive de la pile réseau

La conséquence des deux points précédents (4.1.2.2 à 4.1.2.3 pages 82–83) est que la pile réseau de Contiki souffre d'une complexité déroutante.

Celle-ci inclut déjà un nombre très élevé de couches, pour les seules relevant du protocole IEEE 802.15.4 (couches basses). Par ordre « croissant », on dénombre notamment :

1) **RADIO :** le pilote « physique » contrôlant l'émetteur / récepteur radio ;
2) **FRAMER :** le générateur / analyseur de trames formatées pour un médium physique donné (généralement : "`framer_802154`" pour les trames au format IEEE 802.15.4) ;
3) **RDC :** la couche *Radio Duty Cycle* (voir section 4.1.2.3 page précédente) ;
4) **MAC :** l'implantation du protocole *Medium Access Control*, sans la gestion de l'alimentation de la radio (là encore, voir section 4.1.2.3 page précédente) ;
5) **NETWORK :** le lien avec les couches hautes de la pile réseau, qui peuvent consister en la pile uIP [Dunkels, 2003], la pile Rime simplifiée [Dunkels, 2007], etc.

Si en théorie, une telle conception peut permettre une plus grande flexibilité d'implantation, celle-ci augmente encore la complexité de la pile réseau, déjà très élevée en

raison de ses autres contraintes, à savoir : l'architecture basée sur un unique *packetbuf*, ainsi que la nécessité de fonctionner de façon efficace avec les autres éléments du système — tout spécialement les mécanismes d'évènements et de *protothreads* [Dunkels et al., 2006], qui permettent au noyau de Contiki d'offrir ses fonctionnalités de multitâche coopératif sur du matériel aux ressources très restreintes.

Enfin, le principe de segmentation des couches sur lequel repose, en théorie, la pile réseau de Contiki, n'a pas été réellement respecté lors de l'implantation du code : de nombreuses couches réseau font en effet appel les unes aux autres, ainsi qu'à d'autres éléments du système, sans structure évidente, formant un écheveau de dépendances rendant le système difficile à comprendre et, plus encore, à maintenir.

Un exemple de dépendance difficilement compréhensible est la dépendance de fichiers comme `'contikimac.c'` ou `'nullrdc.c'` — donc appartenant à la couche RDC — à des fichiers comme `'net/rime.h'` ou `'net/rime/rimestats.h'` — c'est à dire à la pile protocolaire Rime de plus haut niveau, dont l'utilisation n'est de surcroît pas supposée être systématique (uIP peut être utilisée à la place).

Un schéma résumant les dépendances entre les différentes couches de la pile réseau et le reste du système Contiki est présenté figure 4.1. Ce schéma montre clairement la grande complexité de l'ensemble.

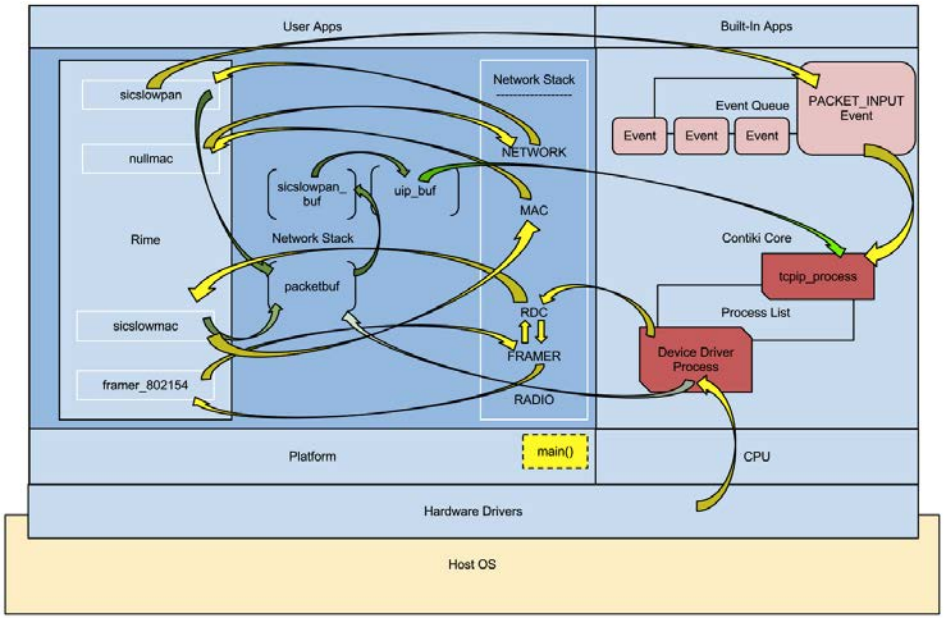

FIGURE 4.1 – *Représentation fonctionnelle des dépendances au sein de l'architecture de Contiki OS. (Source : [Udit, 2012])*

Cette complexité, combinée avec le manque de documentation (notamment de références techniques), fait de toute tentative de contribution au code de Contiki — et notamment de sa pile réseau — une tâche difficile, pénible et sujette à de nombreuses erreurs.

En outre, comme ces difficultés sont des conséquences directes des principes fondamentaux ayant présidé à la conception du système et de sa pile réseau, il est difficilement envisageable de corriger ces défauts sans devoir changer en profondeur l'architecture interne du système, ce qui impliquerait bien évidemment des incompatibilités majeures (et donc très probablement inacceptables pour la communauté des utilisateurs) avec le code existant. Par conséquent, le besoin d'une documentation technique précise, complète, détaillée et accessible est encore plus critique.

4.1.3 Fonctionnalités temps-réel insuffisantes

Pour implanter les protocoles avancés que nous avons décrit en section 3.2.6 page 46 du présent document, il est nécessaire de gérer les évènements système (comme les interruptions) avec réactivité — c'est-à-dire avec un délai de latence minimal — et flexibilité. Ces protocoles reposent en effet sur un *timing* précis pour assurer une synchronisation efficace entre les différents noeuds et autres appareils appartenant aux différents réseaux de capteurs sans-fil (PANs), permettant ainsi de faire fonctionner l'émetteur / récepteur radio des noeuds seulement quand cela est nécessaire. L'émetteur / récepteur radio est en effet le composant le plus gourmand en énergie dans ces noeuds, et le mettre hors fonction est l'un des principaux moyens d'augmenter la durée de vie de leur batterie.

Pour parvenir à une synchronisation temporelle suffisamment précise, nous avons besoin d'une plate-forme logicielle offrant des fonctionnalités temps-réel (c'est-à-dire garantissant que l'exécution d'une tâche donnée se fera dans un délai respectant une échéance maximale donnée) avec une granularité temporelle suffisamment fine.

Contiki, comme nous l'avons vu en section 3.3.3 page 65, n'est pas à la base un système temps-réel : le noyau est basé sur un ordonnanceur évènementiel, basé sur du multitâche coopératif. Cet ordonnanceur ne se déclenche qu'à intervalle fixe, prédéterminé par une constante de compilation. Sur les plates-formes que nous utilisons sous Contiki (les *motes* Sky/TelosB et Zolertia Z1), ce rythme de déclenchement de l'ordonnanceur est fixé à 128 Hz, ce qui correspond à un délai de traitement d'un évènement pouvant monter jusqu'à 8 millisecondes (8000 microsecondes), sachant que le traitement d'une interruption fait partie des « évènements » potentiels. Une granularité aussi large dans la réactivité du système est clairement un problème majeur, notamment pour l'implantation de protocoles MAC / RDC à hautes performances, sachant que la durée de transmission d'une trame 802.15.4 de taille maximale (127 octets) est d'environ 4 millisecondes, et que la durée de base d'une période de *backoff* CSMA / CA est de seulement 320 microsecondes.

Pour tenter de pallier ce problème, Contiki propose une fonctionnalité dédiée à la gestion des évènements en temps-réel, nommée `rtimer`. Ce mécanisme permet d'outrepasser l'ordonnanceur du noyau de Contiki, et d'utiliser un *timer* matériel (c'est-à-dire implanté comme périphérique physique dans le microcontrôleur au cœur du noeud) pour déclencher une fonction choisie par l'utilisateur. Malheureusement, ce mécanisme souffre de sévères limitations :

— Une seule et unique instance de `rtimer` est disponible pour tout le système ; par conséquent, une seule tâche en temps-réel peut être programmée ou exé-

cutée à chaque instant. Cette limitation rend la conception de programmes temps-réel un tant soit peu complexes — tel un protocole MAC / RDC, ou pire encore une pile réseau complète, devant gérer simultanément plusieurs délais précis — plus difficile à gérer et implémenter.

— De plus, il est dangereux, pour la stabilité du système, d'exécuter depuis une fonction déclenchée par `rtimer` la quasi-totalité des fonctions de base de Contiki (c'est-à-dire : le noyau, la pile réseau, etc.), même de façon indirecte, car le code composant le cœur de Contiki n'a pas été conçu pour gérer la préemption (techniquement parlant, ces fonctions ne sont pas « réentrantes »). Contiki est en effet basé sur le paradigme du multitâche coopératif, tandis que `rtimer` se comporte plutôt comme un mécanisme « indépendant », venant avec son propre paradigme. Seul un ensemble restreint de fonctions système définies comme *"interrupt-safe"* (par exemple : la fonction `process_poll()`) peuvent être appelées en toute sécurité depuis une « tâche » `rtimer`, l'utilisation de toute autre fonctionnalité de Contiki conduisant de façon quasi-certaine à un plantage ou à un comportement imprévisible du système. Cette restriction rend en pratique le développement d'extensions temps-réel au système Contiki (via l'emploi de `rtimer`) extrêmement difficile et limité.

Au final, et malgré l'introduction du mécanisme `rtimer`, qui est comme nous venons de le voir extrêmement limité, il est impossible de considérer Contiki comme un système temps-réel (même dans la définition la plus « large » du terme).

Une tentative d'implantation de S-CoSenS a malgré tout été réalisée sous Contiki, mais pour les raisons citées dans la présente section 4.1, il a été impossible de rendre celle-ci un tant soit peu fonctionnelle. Suite à cet échec, nous avons donc définitivement abandonné l'idée d'utiliser Contiki comme plate-forme logicielle pour nos travaux de thèse, la présence de fonctionnalités temps-réel étant indispensable pour pouvoir :

— synchroniser de façon suffisamment précise les différents noeuds des WSN ;
— ou même, ne serait-ce que pour respecter les durées des diverses périodes et donc implanter de façon satisfaisante nos protocoles MAC / RDC avancés.

4.2 RIOT OS : découverte et contributions

4.2.1 La plate-forme logicielle RIOT OS

Nous nous sommes alors intéressés à RIOT OS qui, comme indiqué dans la section 3.3.5 page 68, offre toutes les fonctionnalités dont nous avons besoin pour nos développements, notamment : un micro-noyau avec un ordonnanceur fonctionnant selon le paradigme du multitâche préemptif, ainsi que la possibilité d'utiliser les *timers* matériels du microcontrôleur, et ce avec une granularité au moins aussi fine que le permettent ces *timers* matériels.

Sur les premiers matériels que nous avons utilisés (TelosB et Z1), cette granularité est d'environ $30,5$ μs (la fréquence des *timers* étant comme nous l'avons vu de 32 KHz). Une telle granularité est tout à fait satisfaisante pour nos travaux.

RIOT OS a historiquement d'abord été développé — sous le nom de « Feuerware » — sur des appareils à architecture ARM aujourd'hui obsolètes (famille ARM7-TDMI), puis a été porté sur des microcontrôleurs plus récents (ARM Cortex-M) ainsi que d'autres architectures (MSP430 puis plus récemment Atmel AVR).

Toutefois, lorsque nous avons commencé à travailler avec RIOT (début 2014), le portage sur MSP430 n'était pas aussi bien débogué que le code ARM, et était souvent victime de plantages.

4.2.2 Contributions au projet RIOT OS

Nos contributions, lors de cette première partie de nos travaux de thèse sur RIOT OS, peuvent ainsi être résumées en les points suivants :

4.2.2.1 Gestion des erreurs fatales

Nous avons d'abord ***ajouté des fonctionnalités de déboguage au noyau de RIOT OS, et notamment un mécanisme destiné à prendre en charge les erreurs fatales*** [Roussel et al., 2014i].

Cet ajout s'est inspiré de la fonction noyau `panic()` bien connue dans le monde des systèmes Unix et dérivés. Cette fonction, en générale interne au noyau système, sert à gérer les erreurs fatales, en arrêtant le système, en essayant souvent de fournir le maximum d'informations concernant le problème survenu, et si possible en limitant les risques de dommages (notamment aux données stockées sur le système, voire dans certains rares cas au matériel). Cette procédure d'arrêt « en catastrophe » du système est nommée, dans le monde « unixien », ***"kernel panic"***.

Son équivalent dans les systèmes Microsoft (Windows) et autres (OS/2, iOS) est le *"Screen of Death"* — le plus souvent bleu ou noir (d'où l'acronyme BSOD).

Dans ces systèmes destinés à l'informatique « lourde » (des PC aux macro-ordinateurs), `panic()` est une fonction interne au noyau, car ce dernier s'appuie sur des dispositifs matériels de gestion de la mémoire (à savoir une **MMU** : unité de gestion de mémoire), rendant la majorité des programmes — nommément tous ceux ne s'exécutant pas en « mode privilégié » ou « espace mémoire noyau », c'est-à-dire normalement toutes les applications lancées par l'utilisateur — incapables d'endommager les structures de données et autres ressources vitales pour l'exécution du système. Ces programmes ne manipulent en effet, via le fonctionnement d'une MMU en mode « non privilégié » ou « utilisateur », que des adresses « logiques » (également parfois appelées adresses « linéaires » ou « virtuelles ») au sein d'un espace mémoire virtuel propre à chaque programme en cours d'exécution ; la manipulation réelle des adresses physiques sous-jacentes se faisant exclusivement au travers de la MMU et de ses stricts mécanismes de sécurité et de séparation des espaces mémoires, configurés par le noyau. Un programme en « mode utilisateur » est ainsi normalement incapable de corrompre tout espace mémoire différent du sien, que ce soit celui des autres applications, ou *a fortiori* celui du cœur du système. Une erreur fatale ne peut par conséquent normalement survenir que lorsqu'un problème survient dans le code

s'exécutant en « mode privilégié » (c-à-d : code du noyau lui-même, ou pilotes de périphériques).

Dans les systèmes embarqués, il n'y a pas de MMU : tout au plus une **MPU** (unité de protection mémoire) offrant une limitation d'accès à certaines zones mémoire selon le mode (privilégié ou non).

Malheureusement, toutes les architectures de processeurs embarqués n'offrent pas ce mécanisme de MPU : il n'existe pas dans les architectures AVR et MSP430, et n'est qu'optionnel pour les ARM (notamment Cortex-M), beaucoup de microcontrôleurs implantant cette dernière architecture font ainsi l'impasse sur ce mécanisme.

Il faut également préciser que les applications (souvent uniques) d'un système embarqué ayant régulièrement besoin d'accéder directement aux ressources matérielles — notamment pour des raisons d'efficacité — beaucoup d'entre elles sont, pour des raisons de simplicité, programmées pour fonctionner exclusivement en mode privilégié, rendant ainsi tout mécanisme de MPU inopérant.

Pour toutes ces raisons RIOT OS (du moins dans les versions `2013.08` à `2015.12` incluses, que nous avons utilisées dans le cadre de la présente thèse) n'utilise aucune MPU, même sur les architectures où un tel mécanisme est présent.

Ainsi, nous avons dès le départ clairement déterminé qu'une fonction de gestion des erreurs fatales ne saurait être un mécanisme interne au noyau de RIOT, mais devrait être une primitive système explicitement offerte par ce dernier à l'ensemble du code — noyau, pilotes et applications système. Cette fonction a été, après discussion avec le reste de l'équipe de développement, nommée `core_panic()`.

Nous avons voulu reproduire le maximum des possibilités offertes par les procédures équivalentes des systèmes pour l'informatique « classique » (PC et autres) : les systèmes « plantés » peuvent ainsi être « gelés » pour faciliter leur déboguage durant les phases de développement ; ou, en production, être au contraire immédiatement redémarrés, réduisant ainsi l'indisponibilité d'un appareil fonctionnant sous RIOT au minimum ; le choix entre les deux comportements se fait via une constante de compilation.

Cette contribution s'est vue décerner le titre de ***"PR of the month"*** pour le mois de février 2014 par le projet RIOT OS.

Cette contribution utilise une autre de nos contributions implantant une primitive de redémarrage (fonction `reboot()`) au noyau de RIOT pour le mode production [Roussel et al., 2014k] [Roussel et al., 2014l].

4.2.2.2 Portage sur une nouvelle plate-forme matérielle

Nous avons ***porté RIOT OS sur la Zolertia Z1*** [Roussel et al., 2014h], qui est rappelons-le une *mote* basée sur l'architecture MSP430, utilisée de façon industrielle (notamment avec Contiki).

Dans la plupart des systèmes d'exploitation visant la portabilité (comme par exemple Unix / Linux, ou les différentes déclinaisons de Microsoft Windows NT), la majorité du code du système est conçue de façon générique, c'est-à-dire avec

une structure et un fonctionnement indépendants de la plate-forme matérielle sous-jacente.

Le code dépendant de la plate-forme est réduit au minimum, et se charge d'offrir au reste (très majoritaire) du système une API prédéfinie lui permettant d'accéder au matériel tout en « ignorant » ces spécificités. L'ensemble restreint de ce code dépendant des plates-formes matérielles et cachant les disparités de ces dernières est nommé **couche d'abstraction matérielle** (**HAL :** *Hardware Abstraction Layer*).

RIOT OS, comme la plupart des systèmes portables, est conçu selon ce modèle employant une couche d'abstraction matérielle. Un portage de RIOT OS sur un nouveau matériel consiste donc à créer une variante de cette couche capable de gérer la nouvelle plate-forme matérielle visée.

Cette couche d'abstraction matérielle se charge notamment, sous RIOT OS, d'assurer le support des changements de contexte lors des basculements entre tâches (en support à l'ordonnanceur), l'implantation des fonctions concernant les *timers* ou les différents périphériques implantés au sein du microcontrôleur, ainsi que l'accès aux entrées / sorties (GPIO). Du point de vue de l'implantation, la couche d'abstraction matérielle peut être vue comme un ensemble de pilotes (*"drivers"*) chargés d'implanter l'API prédéfinie entre cette couche et le reste du système d'exploitation ; ces pilotes doivent en général lire et écrire les registres matériels de façon adéquate, en fonction des opérations demandées.

La couche d'abstraction matérielle de RIOT OS se divise fonctionnellement en plusieurs sous-parties :

1. Une partie « ***MCU*** » destinée à supporter les différentes architectures de microcontrôleurs visées. Cette partie prend en charge les fonctionnalités spécifiques de chaque architecture de microcontrôleur, elle regroupe ainsi le code commun entre plates-formes matérielles basées sur des microcontrôleurs similaires (ARM7, Cortex-M, MSP430, AVR...). Dans notre cas, le code commun destiné à supporter l'architecture MSP430 existait déjà : nous n'avons ainsi pas eu à nous occuper de cette partie MCU pour effectuer notre portage.

   Cette partie de la couche d'abstraction matérielle se charge notamment des opérations liées aux registres et mécanismes du cœur du microcontrôleur : par exemple de l'implantation des changements de contexte lors des basculements entre tâches.

2. Une partie « ***périphériques*** » regroupant des pilotes génériques pour des circuits intégrés communs, susceptibles d'être retrouvés dans plusieurs matériels différents : c'est le cas des émetteurs / récepteurs radio, des capteurs environnementaux divers (ex. : température, humidité, etc.). Nous n'avons pas non plus eu besoin pour ce travail de portage d'intervenir directement sur cette partie.

   Ces pilotes « génériques » sont ensuite exploités de façon adéquate par du code « de liaison » (*"glue"* en anglais) spécifique — impliquant souvent un bus géré par le microcontrôleur comme SPI ou I^2C — permettant leur mise en œuvre sur chaque plate-forme matérielle différente. Ce code de liaison est implanté dans la dernière partie de la couche d'abstraction matérielle décrite ci-dessous.

3. Une partie « ***plate-forme*** » regroupant les éléments véritablement spécifiques à une plate-forme matérielle donnée. Elle regroupe notamment les pilotes du matériel purement spécifique à une plate-forme donnée, ainsi que le code « de liaison » (*"glue"*) avec les parties plus génériques de la couche d'abstraction matérielle. C'est cette partie que nous avons créée pour le support spécifique de la plate-forme matérielle Zolertia Z1.

   Ce travail, comme nous l'avons dit ci-dessus, a principalement consisté à implanter des pilotes — notamment pour les *timers* matériels, le port série / USB (UART) et les GPIO — ainsi qu'une fonction d'initialisation adéquate, principalement en manipulant les registres adéquats des périphériques. Nous avons également créé le code de liaison nécessaire avec le pilote pré-existant pour l'émetteur / récepteur radio CC2420 qui équipe cette *mote*, via la configuration adéquate du bus SPI du microcontrôleur.

4.2.2.3 Déboguage des portages sur MSP430

Nous avons ***débogué les portages de RIOT OS sur MSP430*** — plus précisément : la portion spécifique de la couche d'abstraction matérielle de l'ordonnanceur chargée de traiter les changements de contexte lors des basculements entre tâches [Roussel et al., 2014s] [Roussel et al., 2014t] [Roussel et al., 2014v] — rendant ainsi RIOT robuste et prêt à l'usage en production sur les appareils basés sur l'architecture MSP430.

Pour effectuer un tel déboguage de façon efficace, il est nécessaire :

— Soit de disposer de la possibilité de faire appel à des fonctions de déboguage directement présentes dans le matériel, en général rendues accessibles via le protocole JTAG. La plupart des microcontrôleurs sur lesquels sont basées les *motes* offrent des broches spécifiques dédiées à ce mécanisme de déboguage ; il faut néanmoins que la *mote* elle-même rende ces signaux accessibles via un port quelconque, ce qui n'est pas évident. En outre, un adaptateur JTAG, presque toujours spécifique de l'architecture du microcontrôleur à déboguer, est nécessaire pour relier le matériel à déboguer et le PC / station de travail du développeur. Enfin, le protocole JTAG ne permet sauf exception que d'agir sur le microcontrôleur lui-même, pas d'avoir accès aux circuits annexes comme les émetteurs / récepteurs radio autonomes (tel le CC2420 de la Zolertia Z1).

— Soit de disposer d'un émulateur suffisamment fiable, et fidèle au fonctionnement réel du matériel sur lequel le programme à déboguer doit s'exécuter. Celui-ci peut alors offrir des fonctions de déboguage avancées, comparables à ce qu'offrent les environnement de développement pour PC. Un tel émulateur est MSPSim [Eriksson et al., 2009], fourni en standard avec le simulateur de réseaux de capteurs sans-fil Cooja [Österlind et al., 2006] du projet Contiki, lequel émule plusieurs *motes* basées sur l'architecture MSP430, notamment les TelosB /SkyMotes et — surtout — la Zolertia Z1.

Les problèmes qui touchaient les portages de RIOT OS sur MSP430 étaient mal reproductibles, car imprévisibles (bien que fréquents) : leur déclenchement n'ayant

pu être lié ni à l'utilisation d'une certaine fonctionnalité, ni au passage dans une portion spécifique du code de RIOT, ni à l'écoulement d'un certain délai. Dans ces conditions, il est très difficile de trouver la cause d'un problème, même en utilisant les meilleurs outils de déboguage.

Pour trouver la raison principale des plantages de RIOT OS sur MSP430, nous avons eu recours à la technique dite de **traçage des instructions :** cette technique consiste à enregistrer chaque instruction exécutée par le matériel lors du fonctionnement du code défectueux (soit pour toute la durée d'exécution, soit pour un intervalle de temps donné). Au final, l'utilisation de cette technique permet d'obtenir un *listing* assembleur précis, reproduisant fidèlement les instructions élémentaires exécutées lors du déroulement du programme.

Employer la technique de traçage avec un débogueur matériel nécessite l'emploi d'un adaptateur JTAG haut de gamme, disposant de la quantité de mémoire nécessaire pour enregistrer les instructions exécutées. De tels adaptateurs sont coûteux, et n'existent pas pour toutes les architectures : l'unique adaptateur JTAG MSP430 vendu par Texas Instruments ne dispose pas d'une telle fonctionnalité.

À l'inverse, ***le couple Cooja / MSPSim offre une telle fonctionnalité de traçage des instructions***. Nous avons donc utilisé cette possibilité, qui nous a finalement permis de découvrir la source de la principale cause d'instabilité sur MSP430, à savoir : l'inactivation trop tardive des interruptions pendant le changement de contexte de tâche, permettant à une interruption de stopper la sauvegarde du contexte de la tâche en cours d'arrêt, ce qui bien évidemment corrompait le plus souvent le contexte sauvegardé, provoquant le plantage du système lors de la réactivation de la tâche touchée. Cette réactivation intervenant de façon non déterministe, en fonction de la charge du système et de sa gestion par l'ordonnanceur, on comprend ainsi pourquoi ces plantages intervenaient sans raison *a priori* discernable. Nous avons donc, grâce aux fonctions avancées de déboguage et traçage de MSPSim, pu comprendre le principal problème et y apporter une solution. Cette solution consiste à corriger l'ordre des instructions dans la fonction `thread_yield()`, faisant partie de la partie de la couche d'abstraction matérielle de RIOT gérant l'architecture MSP430 : les interruptions sont désormais désactivées dès le début de la fonction, ***avant*** de commencer à sauvegarder le contexte du *thread* en cours de suspension, et ne sont réactivées (si nécessaire) qu'une fois le contexte du *thread* à relancer totalement restauré [Roussel et al., 2014s].

La correction de deux autres problèmes liés à la gestion des *timers* des MSP430 par cette même couche d'abstraction matérielle, plus simples à comprendre et à résoudre, a permis d'achever la fiabilisation des portages de RIOT OS :

1. la correction de l'implantation de la fonction `hwtimer_spin()`, en évitant que ne soit calculée une valeur-cible impossible à atteindre par le compteur du *timer* voulu [Roussel et al., 2014v] ;
2. l'abandon de la prise en charge de délais supérieurs à 2^{16} *"ticks"* de *timers* pour l'implantation du mécanisme `hwtimer` sur MSP430 [Roussel et al., 2014t].

Ces deux erreurs étaient dues à la différence de type entre les valeurs calculées par les implantations des fonctionnalités liées aux mécanismes temps-réel de `hwtimer` —

utilisant des entiers 32 bits — et les compteurs des *timers* matériels des MSP430 — qui ne sont que des entiers de 16 bits [2].

La possibilité de faire fonctionner RIOT sur MSP430 permet donc ***d'exécuter des applications basées sur ce système dans le simulateur Cooja / MSPSim. Cela facilite grandement la rapidité et l'efficacité du développement de telles applications.*** Nous venons notamment de le démontrer, le déboguage de la couche d'abstraction matérielle de RIOT pour les microcontrôleurs MSP430 aurait probablement été extrêmement difficile sans ses fonctionnalités de déboguage et notamment de traçage.

Au final, on voit qu'un émulateur ***de qualité*** peut se révèler un outil de développement et déboguage extrêmement précieux, en complément des débogueurs matériels, ou même en remplacement de ces derniers, lorsque ceux-ci sont trop coûteux ou — surtout ! — n'offrent pas les fonctionnalités nécessaires comme cela a été le cas ici. Bien évidemment, ***l'émulateur en question, pour remplir ce rôle, doit être très fidèle au fonctionnement du matériel émulé pour être utile***. Dans ce cas précis, MSPSim s'est révélé tout à fait à la hauteur quant à l'émulation du cœur du microcontrôleur MSP430 (au niveau duquel se produisait le bogue à corriger).

4.2.2.4 Contributions diverses

D'autres contributions ont été apportées, ayant eu un impact moindre sur l'avancée du projet RIOT OS. Celles-ci se divisent en deux groupes :

1. les contributions de ***déboguage*** :
   - réparation du mécanisme d'échange de données entre microcontrôleur et émetteur / récepteur radio sur Zolertia Z1, en corrigeant le test vérifiant la bonne émission d'un octet sur le bus SPI (retrait d'une négation logique erronée) [Roussel and Hahm, 2014b],
   - correction de diverses erreurs dans le pilote pour l'émetteur / récepteur radio CC2420 au niveau de l'implantation : correction d'une constante [Roussel and Hahm, 2014d], de la gestion des adresses et du PAN ID [Roussel et al., 2014b], de la gestion du bus SPI pour communiquer avec la radio (notamment la fonction `cc2420_do_send()`) [Roussel et al., 2014q], et dans la détermination du CCA (mauvaise gestion du bus SPI dans l'implantation de cette fonctionnalité) [Roussel et al., 2014r],
   - correction d'erreurs intermittentes concernant les *timers* matériels des MCU MSP430, notamment en déclarant certaines variables `volatile`s au sein de l'implantation [Roussel et al., 2014d],
   - fiabilisation des mécanismes de gestion des interruptions sur architecture MSP430, en ajoutant un « délai d'attente » (instruction `NOP`) après l'activation ou la désactivation générale des interruptions au niveau du microcon-

2. Ce type de problèmes et limitations a conduit au remplacement du mécanisme `hwtimer` par le module `xtimer`, de conception différente et de plus haut niveau (comme vu en section 3.3.5 page 68), et donc normalement moins vulnérable vis-à-vis de telles difficultés d'implantation.

trôleur (comme recommandé dans les manuels de TI), et en s'assurant que les interruptions sont bien activées avant l'entrée en mode basse consommation (pour pouvoir sortir de ce dernier) [Roussel et al., 2014e],
— fiabilisation de l'alignement du pointeur de pile (registre SR) lors de la création de threads [Roussel et al., 2014j],
— fiabilisation du processus de *flashage* des applications sur ARM Cortex-M3, en forçant un effacement de la mémoire et un *reset* avant le lancement du processus de *flashage* proprement dit [Roussel et al., 2014m],
— correction et la fiabilisation de la fonction d'attente active via l'utilisation des timers matériels (`hwtimer_spin()`), en modifiant l'implantation pour la rendre plus simple, compréhensible et fiable [Roussel et al., 2014u] ;

2. les contributions ***apportant des améliorations et fonctionnalités supplémentaires*** :
— amélioration de l'API et des fonctionnalités des pilotes des émetteurs / récepteurs radio [Roussel and Hahm, 2014a] [Roussel and Hahm, 2014c] [Roussel et al., 2014c] [Roussel et al., 2014g] [Roussel et al., 2014p],
— gestion des modes « basse énergie » des MCU MSP430 [Roussel et al., 2013a],
— réorganisation et simplification des fichiers d'entête (`xxx.h`) pour la partie MSP430 de la couche d'abstraction matérielle [Roussel et al., 2013b],
— augmentation du nombre d'instances de *timers* matériels (`hwtimers`) utilisables sur MSP430 [Roussel et al., 2014f],
— ajout d'une primitive de redémarrage (fonction `reboot()`) au noyau de RIOT (notamment employée par la gestion des erreurs fatales pour réactiver au plus vite un système planté en mode production, voir 4.2.2.1 page 88) [Roussel et al., 2014k] [Roussel et al., 2014l],
— définition d'attributs de fonction « portables » [Roussel et al., 2014n],
— possibilité pour un thread d'envoyer un message à lui-même (pour permettre par exemple les appels de fonction retardés — *"deferred procedure calls"* en anglais) [Roussel et al., 2014o],
— ajout d'une option pour la gestion du *"CCA threshold"* dans la nouvelle pile gnrc [Roussel et al., 2015a].

Notons que toutes ces contributions ont été revues et acceptées par l'équipe de développement de RIOT, qui les a intégrées dans le source *"master"* du système : elles font donc maintenant partie du système de base [3].

Grâce à toutes ces avancées, nous avons maintenant une plate-forme logicielle robuste et offrant les fonctionnalités nécessaires pour développer nos protocoles MAC / RDC à hautes performances — notamment ceux basés sur l'ordonnancement tem-

3. Les changements récents dans la pile réseau et la gestion des *timers* du système RIOT ont toutefois rendu obsolète le code apporté par certaines de ces contributions ; néanmoins, les fonctionnalités correspondantes ont en général été réimplantées d'une façon différente et adéquate dans les nouvelles versions de RIOT OS.

porel (TDMA). C'est pourquoi nous avons implanté et testé S-CoSenS sur RIOT OS lors des expériences que nous décrirons au chapitre 5 page 103.

4.2.3 Évolution du système RIOT OS

Depuis que lesdites expériences du chapitre 5 ont été menées, RIOT OS a continué à évoluer de façon rapide et profonde — et tout particulièrement sa pile réseau, qui a fait l'objet d'une refonte totale. Cette évolution étant potentiellement amenée à modifier, non seulement les résultats des expériences citées ci-dessus, mais aussi ceux des travaux présentés plus loin au chapitre 6 page 135 (et notamment à influer sur les problèmes détaillés dans ce dernier chapitre), nous allons maintenant procéder à une analyse détaillée et critique de cette nouvelle pile réseau.

4.3 La nouvelle pile réseau de RIOT : une analyse critique

Nous décrivons, dans les sections 4.1.2.1 page 81 puis 6.3.1 page 151, notre mécanisme de « capacités » permettant de profiter de façon portable des spécificités des émetteurs / récepteurs radio des *"motes"*, grâce à des fonctions génériques de gestion d'un ensemble varié et extensible de constantes et paramètres. Un mécanisme de ce type est désormais intégré dans Contiki 3.0 [Dunkels, 2015] (suite en partie et de façon indirecte à certaines de nos contributions). Une évolution similaire a eu lieu pour RIOT OS.

En effet, pendant que nous tentions d'effectuer nos travaux et subissions les problèmes techniques détaillés dans la section 6.3 page 151, RIOT OS a durant l'année 2015 subi une refonte de sa pile réseau — une pile réseau nouvelle génération, se voulant générique (« **gnrc** »), est apparue, sur laquelle nous allons nous pencher dans la présente section.

Compte-tenu du sujet de la présente thèse, nous nous focaliserons sur les couches les plus basses (physique / pilote radio, et MAC / RDC) de cette nouvelle pile.

4.3.1 Description

L'API des pilotes pour les émetteurs / récepteurs radio de la nouvelle pile « gnrc » a repris le principe de ce que nous avons appelé le mécanisme de « capacités », implantant pour les pilotes radio un ensemble, extrêmement réduit, de fonctions génériques, dont le détail est donné en table 4.1 page suivante. (Notons toutefois que nous n'avons pas pu participer à l'implantation de ces nouveaux pilotes, ni à la conception de la nouvelle pile « gnrc », étant monopolisés par nos propres travaux.)

Les « évènements » pour lesquels il est possible de définir des *"callbacks"* sont listés en table 4.2 page suivante. (Le déclenchement ou non de ces *"callbacks"* se fait en fonction de la valeur des options `NETOPT_*_*_IRQ` définies en table 4.3 page 97.)

Les différentes options pouvant être réglées par les fonctions `get()` et `set()` sont listées dans la table 4.3 page 97.

Nom	Rôle
`send_data()`	Envoie une trame via l'émetteur / récepteur radio
`add_event_callback()`	Ajoute une fonction *"callback"* pour la gestion des évènements remontés par le pilote radio donné
`rem_event_callback()`	Retire une fonction *"callback"* pour la gestion des évènements remontés par le pilote radio donné
`get()`	Lit la valeur courante d'une option de configuration d'un pilote radio donné
`set()`	Définit la valeur courante d'une option de configuration d'un pilote radio donné
`isr_event()`	Fonction *"callback"* pouvant être appelée par une couche supérieure de la pile réseau, pour signaler un évènement au pilote radio (cet « évènement » étant actuellement une valeur numérique totalement arbitraire)

TABLE 4.1 – *Liste des fonctions des pilotes radio nouvelle génération (« gnrc »).*

Nom	Signification
`NETDEV_EVENT_RX_STARTED`	Début de réception d'une trame (SFD)
`NETDEV_EVENT_RX_COMPLETE`	Réception d'une trame terminée : analyse possible du contenu
`NETDEV_EVENT_TX_STARTED`	Début de transmission d'une trame
`NETDEV_EVENT_TX_COMPLETE`	Fin de transmission d'une trame

TABLE 4.2 – *Liste des évènements radio interceptables (« gnrc »).*

Les fonctions présentées en table 4.1 signalent la survenue d'erreurs en retournant les valeurs d'erreurs (toujours négatives) listées en table 4.4 page 98.

(Notons que les tables 4.1 à 4.4 pages 96–98 représentent l'état d'avancement de la pile « gnrc » au moment où nous écrivons ces lignes — début septembre 2015. Cette pile étant toujours à l'heure actuelle l'objet d'un développement rapide et poussé, le contenu de ces tables ne sera sans doute plus exhaustif au moment où ce manuscrit de thèse sera lu.)

Le pilote radio amélioré, implantant notre concept de « capacités », dont nous discuterons en section 6.3.1 page 151 est donc déjà obsolète, et ne sera pas intégré au code de RIOT OS.

En effet, un nouveau type de pilote radio, nommé `gnrc_netdev_t`, a été créé ; toutes les fonctions de la nouvelle API « gnrc » travaillent avec des instances de ce type.

Un effort pour convertir tous les pilotes radio de RIOT à l'API de ce nouveau type `gnrc_netdev_t` est actuellement en cours.

Identifiant	**Signification**
NETOPT_CHANNEL	Canal (fréquence) utilisée par l'émetteur / récepteur radio
NETOPT_IS_CHANNEL_CLR	Vérifie si le médium radio est disponible (*"Clear Channel Assessment"*)
NETOPT_ADDRESS	Définit l'adresse « courte » (intra-PAN, 16 bits) de la radio
NETOPT_ADDRESS_LONG	Définit l'adresse « longue » (64 bits pour le 802.15.4) de la radio
NETOPT_ADDR_LEN	Taille de l'adresse longue de la radio (normalement 64 bits pour le 802.15.4)
NETOPT_SRC_LEN	Taille de l'adresse source à utiliser pour la radio
NETOPT_NID	Définit l'identifiant du réseau (PAN ID)
NETOPT_IPV6_IID	Définit l'adresse IPv6 de l'interface réseau (RFC 4291, section 2.5.1)
NETOPT_TX_POWER	Définit la puissance d'émission de la radio en dBm
NETOPT_MAX_PACKET_SIZE	Définit la taille maximale d'une trame (normalement 127 octets pour le 802.15.4)
NETOPT_PRELOADING	Active ou désactive la possibilité de pré-charger le *"buffer"* de transmission de la radio sans lancer immédiatement l'émission de la trame
NETOPT_PROMISCUOUSMODE	Active ou désactive le mode « moniteur » (*"promiscuous"*)
NETOPT_AUTOACK	Active ou désactive l'acquittement automatique des trames reçues par la radio
NETOPT_RETRANS	Nombre maximal de retransmissions en mode CSMA/CA
NETOPT_PROTO	Type de protocole (6LoWPAN, IPv6, TCP, UDP...) pour la couche voulue
NETOPT_STATE	État de la radio, à choisir parmi : — NETOPT_STATE_OFF : hors tension — NETOPT_STATE_SLEEP : mode inactif basse consommation — NETOPT_STATE_IDLE : en écoute (passive) — NETOPT_STATE_RX : réception en cours — NETOPT_STATE_TX : transmission en cours / déclenche l'émission d'une trame préchargée — NETOPT_STATE_RESET : provoque une réinitialisation de la puce radio
NETOPT_RAWMODE	Active ou désactive l'analyse automatique des trames reçues
NETOPT_RX_START_IRQ	Active ou désactive le déclenchement d'une interruption au début d'une réception (SFD)
NETOPT_RX_END_IRQ	Active ou désactive le déclenchement d'une interruption lors de la réception d'une trame complète
NETOPT_TX_START_IRQ	Active ou désactive le déclenchement d'une interruption au début d'une transmission
NETOPT_TX_END_IRQ	Active ou désactive le déclenchement d'une interruption à la fin de la transmission d'une trame
NETOPT_AUTOCCA	Active ou désactive la vérification automatique de la disponibilité du médium radio (*"Clear Channel Assessment"*) par l'émetteur /récepteur radio au début d'une transmission

TABLE 4.3 – *Liste des options — « capacités » potentielles — des émetteurs / récepteurs radio (« gnrc »).*

Nom	Signification
-ENODEV	Tentative d'accès à un pilote radio `gnrc_netdev_t` invalide (toutes les fonctions)
-ENOMSG	Tentative d'envoi d'une trame invalide (fonction `send_data()`)
-EOVERFLOW	Le *buffer* chargé de contenir la trame à envoyer, ou les données du paramètre à lire est trop petit (fonctions `send_data()` et `get()`)
-ENOBUFS	Le nombre maximal de fonctions *"callbacks"* pour un pilote radio donné est dépassé (fonction `add_event_callback()`)
-ENOENT	La fonction *"callback"* à retirer n'a pas été enregistrée auparavant (fonction `rem_event_callback()`)
-ENOTSUP	Option non supportée par ce pilote radio (fonctions `get()` et `set()`)
-ECANCELED	Erreur interne du pilote radio (fonctions `get()` et `set()`)
-EINVAL	Valeur invalide pour l'option transmise (fonction `set()`)

TABLE 4.4 – *Liste des erreurs potentiellement renvoyées (« gnrc »).*

4.3.2 Avantages

On peut voir ici que le travail de refonte de la pile a été poussé très loin, et que le jeu d'options réglables grâce à la nouvelle API est déjà très complet. La logique de la gestion générique des paramètres est poussée jusqu'au bout : seul l'envoi de trames et la réception d'évènements — ce qui inclut la réception de trames — coexistent encore avec la paire de fonctions `get()` et `set()` gérant les paramètres.

Nous avons avec ces nouveaux pilotes radio d'ores et déjà le mécanisme nécessaire pour compter les SFDs, que nous comptions utiliser pour améliorer S-CoSenS. La seule option que nous pourrions vouloir ajouter serait celle permettant de définir le seuil de distinction signal / bruit (*"CCA Threshold"*), par exemple via une option `NETOPT_CCA_THRESHOLD`. Un tel ajout sera probablement très facile à réaliser dans cette nouvelle API.

Le passage à la généricité améliore grandement la souplesse dans la gestion des émetteurs / récepteurs radio — le nombre d'options n'étant pas limité à celles déjà présentes, et pouvant s'étendre pour gérer autant de fonctionnalités que nécessaires, même les plus spécifiques.

Concernant la couche la plus basse — les pilotes radio — un nombre réduit de fonctions à implanter (six au total) devrait permettre d'espérer une simplification du code, réduisant ainsi mécaniquement le nombre de bogues potentiels, et permettant (en théorie) d'espérer des améliorations de performances. N'ayant pas eu le temps de tester nous-même cette nouvelle pile réseau pour les raisons citées plus haut, une perspective de travail intéressante à court terme est d'effectuer des essais pour démontrer les améliorations que celle-ci apporte.

Notons surtout que le développement de cette nouvelle pile « gnrc » est toujours

en cours : de nouvelles améliorations et optimisations seront donc encore sans doute ajoutées. L'enthousiasme et la compétence de la communauté du projet RIOT — communauté par ailleurs en expansion — permet d'attendre encore de nouveaux progrès.

4.3.3 Inconvénients et manques

Par rapport à l'API telle que nous l'avons imaginée et détaillée plus loin dans le présent manuscrit (section 6.3.1 page 151), on regrettera juste l'absence d'un mécanisme d'interrogation de constantes, qui peut aider le programmeur à éviter certaines erreurs par la programmation défensive.

L'erreur `-EINVAL` renvoyée par la fonction `set()` n'indique en effet pas au programmeur quelles valeurs sont autorisées pour quels paramètres.

Il ne s'agit toutefois nullement d'une fonctionnalité indispensable ou même majeure, le programmeur pouvant (devant ?) avoir recours à la documentation disponible avant de définir un (ou plusieurs) paramètre(s).

Nous ne voyons sinon aucune autre lacune au travail déjà effectué.

Le principal défaut, paradoxalement, de ce passage à la généricité, est celui de la difficulté de compréhension.

La généricité a én effet été poussée à un niveau extrême. Le traitement des trames et paquets ne se fait plus « directement » couche après couche (c-à-d. : le pilote radio passant la *payload* de la trame à la couche MAC, qui elle-même analyse et retire ses entêtes pour passer cette « nouvelle charge utile » réduite à la couche 3, etc).

La pile réseau est maintenant gérée par une *"registry"* (`gnrc_netreg`), auprès de laquelle doivent s'enregistrer tous les threads intéressés par les paquets / trames d'un certain protocole (par exemple : 6LoWPAN, TCP, etc. ; les trames « nues », tels que reçus par la radio étant désignés par le type `UNDEF`). La couche MAC ne fait pas exception et doit s'enregistrer pour pouvoir recevoir ces trames de « type indéfini ».

La couche MAC est d'ailleurs minimaliste, car seul existe — au moment où nous écrivons ces lignes — un protocole nommé `nomac`, qui comme son nom l'indique, n'est qu'un relai passif (n'effectuant aucun traitement) entre les couches supérieures de la pile réseau et le(s) pilote(s) radio.

En outre, cette *"registry"* travaille également uniquement avec des *threads*, en leur passant des messages. Au final, la complexité du code gérant la pile réseau risque, ironiquement, d'augmenter, avec la multiplication de *threads* différents se passant des messages pour tout traitement de données issues du réseau.

Le fonctionnement par passage de messages peut aussi potentiellement provoquer un retard dans le traitement d'un évènement, contrecarrant les avantages des fonctionnalités temps-réel avancées qui sont l'un des grands atouts de RIOT OS. Le système de passage de messages de RIOT est certes performant, et jouer sur la priorité des *threads* ayant un rôle critique est une solution logique ; mais la maîtrise de ces notions est clairement une compétence avancée, dont la subtilité n'est pas à la portée de tout développeur. La programmation rapide de petites applications

communicantes simples sous RIOT par des « amateurs éclairés » ou des débutants pourrait éventuellement devenir, avec l'emploi de cette pile « gnrc », plus difficile, et décourager des utilisateurs potentiels.

De plus, bien que ce modèle de *threads* et de messages soit conforme aux bonnes pratiques classiques de programmation, telles qu'elles sont appliquées au monde des PCs et autres ordinateurs « complets », nous restons ici sur du matériel très limité, notamment au niveau de la mémoire. Il faut espérer que cette pile réseau ne va pas poser de problèmes pour l'exploitation de RIOT sur les *motes* bas de gamme, comme celles à base de MSP430 ou d'AVR.

Notons toutefois l'existence des fonctions `add_event_callback()` et `rem_event_callback()`, suffisantes pour gérer la réception des trames physiques ; on peut ainsi imaginer la possibilité de voir apparaître une nouvelle pile réseau « allégée » pour les appareils les plus limités.

La pile « gnrc » complète est, d'un autre coté, certainement à son aise sur les microcontrôleurs haut-de-gamme type ARM Cortex-M — dont la puissance et l'espace mémoire ne cessent d'augmenter, cf. la sortie récente des Cortex-M7 —, où elle pourrra être mise à profit notamment pour le développement d'applications professionnelles / industrielles lourdes.

4.3.4 Discussion : la pile « gnrc »

Notre première impression sur cette nouvelle pile réseau « gnrc » est, malgré les défauts cités dans la précédente section 4.3.3, nettement positive.

Le travail effectué sur cette nouvelle pile est d'ores et déjà très complet et impressionnant. Nous n'avons, vu notre sujet de thèse, détaillé que les couches basses de cette pile « gnrc », mais celle-ci gère déjà les protocoles RPL, 6LoWPAN, IPv6, ICMPv6, et NDP [Narten et al., 2007], le tout en s'appuyant sur une liste commune de *"buffers"* pour les paquets / trames, permettant ainsi l'économie de mémoire et de recopies à répétition des données de ces paquets / trames au fil des couches de la pile réseau.

Au niveau des pilotes radio, le travail est déjà très complet (l'ajout d'une fonction d'interrogation de constantes comme nous l'avons suggéré ci-dessus serait un plus, mais n'est pas un élément indispensable).

La couche MAC, elle, est embryonnaire : elle est pour l'instant clairement le parent pauvre de cette nouvelle pile « gnrc ». Ici se trouve un champ largement ouvert aux futures contributions.

Toutefois, si cette nouvelle pile est fonctionnellement riche et prometteuse, elle risque aussi d'imposer des exigences matérielles auxquelles les appareils les plus limités sur lesquels tourne RIOT risquent de ne pas pouvoir répondre.

La structure des pilotes radio « gnrc » laissent toutefois entrevoir la possibilité d'une pile alternative allégée, destinée à ces *motes* bas-de-gamme. Une telle situation serait comparable à celle que connaît Contiki OS, qui à coté de uIPv6, dispose de Rime pour les applications plus légères.

Rappelons également, pour finir, qu'en guise de pile réseau alternative, OpenWSN fonctionne également sur le noyau de RIOT (outre celui de FreeRTOS). Les utilisateurs de RIOT ont donc déjà le choix quant à la pile réseau à employer pour leurs applications. Par conséquent, on peut sans doute raisonnablement envisager d'élargir encore ce choix.

4.4 Discussion : plates-formes logicielles, contributions et conclusions

Nous avons, dans le présent chapitre, étudié, critiqué et comparé les deux systèmes d'exploitation spécialisés dans les WSN que nous avons choisis comme plates-formes logicielles potentielles à l'issue de l'étude de l'état de l'art dans ce domaine (section 3.3 page 61).

Nous nous sommes particulièrement focalisés sur les piles réseau de ces derniers, qui ont toutes deux fait l'objet d'une analyse critique détaillée.

Ainsi, nous en avons dégagé ***les contributions suivantes*** :

— Nous avons d'abord — dans le chapitre 3 — ***passé en revue les OS*** classiquement utilisés dans le domaine des réseaux de capteurs sans-fil (comme TinyOS, entre autres — cf. section 3.3 page 61), analysé leurs faiblesses, et ***démontré pourquoi ceux-ci ne pouvaient servir de base au développement de protocoles MAC / RDC avancés***.

— Nous avons montré que ***Contiki OS, par sa conception, est mal adapté au développement de protocoles MAC /RDC à hautes performances, tout particulièrement à cause de fonctionnalités temps-réel extrêmement restreintes, ainsi que d'une pile réseau à la structure complexe et imposant de fortes contraintes***, cette structure s'expliquant toutefois par la nécessité de faire fonctionner Contiki OS sur du matériel aux ressources limitées.

— Nous avons ***étudié une plate-forme logicielle — RIOT OS — offrant toutes les fonctionnalités nécessaires à l'implantation de ces protocoles MAC / RDC à hautes performances***. Nous avons aussi et surtout ***contribué activement et à son évolution, son déboguage, et son portage sur une nouvelle plate-forme matérielle*** (la Zolertia Z1).

  Parmi ces contributions, la plupart sont totalement spécifiques à RIOT (corrections de bogues, portage sur Z1, améliorations de parties spécifiques de l'implantation).

  Toutefois, ***certaines de ces contributions — en particulier celle offrant un mécanisme de gestion des erreurs fatales [Roussel et al., 2014i]*** permettant de « geler » l'appareil en mode déboguage (pour faciliter une analyse *post mortem* du système), ou au contraire un redémarrage immédiat en mode production — peuvent avantageusement être adaptées à n'importe quel système d'exploitation ne disposant pas déjà de telles fonctions. On peut faire la même remarque pour ***la primitive système permettant le redémarrage immédiat par logiciel du système*** [Roussel et al., 2014k].

— Nous avons enfin effectué ***une analyse critique de la nouvelle pile réseau « gnrc » de RIOT, notamment de ses couches basses, en recensant ses avantages et ses inconvénients*** (le principal étant sa relative complexité et sa potentielle incompatibilité avec des matériels trop limités). ***Nous avons évoqué une possible solution à ce dernier problème, à savoir la création d'une pile alternative plus légère*** pour les appareils et applications plus modestes — solution déjà mise en œuvre par d'autres systèmes équivalents, en premier lieu Contiki.

On peut également citer ici ***notre contribution concernant la gestion des erreurs liées aux débordements mémoire***, abordée dans le chapitre suivant en section 5.3.5 page 120, consistant en une proposition mêlant plate-forme logicielle et compilateur(s), laquelle est susceptible de concerner ***tout système d'exploitation destiné au monde de l'embarqué***.

Notre plate-forme logicielle de référence, RIOT OS, étant maintenant clairement et définitivement choisie, améliorée et déboguée — en partie grâce à nos contributions —, nous allons maintenant commencer à détailler les travaux d'expérimentation effectués dans cette thèse. Nous commencerons par l'implantation sous RIOT OS du plus simple des protocoles à hautes performances décrit en section 3.2.6 page 46, S-CoSenS, dans le chapitre 5 suivant.

Chapitre 5

Évaluation et comparaison des implantations de deux protocoles MAC / RDC : ContikiMAC et S-CoSenS

Avant de détailler nos recherches et travaux sur les protocoles MAC / RDC, nous allons brièvement détailler le matériel que nous avons utilisé lors de nos premiers travaux.

5.1 Premières plates-formes matérielles

Nous avons employé, pour le début de ces travaux de thèse, des *motes* basées sur des microcontrôleurs d'architecture MSP430. Cette architecture, conçue par Texas Instruments, offre une consommation d'énergie très basse, un prix réduit, et de bonnes performances, grâce à une conception RISC 16 bits spécifique. Elle est très couramment utilisée dans les *motes* utilisées dans les réseaux de capteurs sans-fil. Elle est également supportée par le simulateur Cooja [Österlind et al., 2006] (par le biais de l'émulateur MSPSim [Eriksson et al., 2009] intégré à ce dernier), ce qui permet d'effectuer des simulations permettant de concevoir et de tester facilement des scénarios de réseaux sans-fil, notamment lorsque de nombreux noeuds sont impliqués.

Les matériels que nous avons utilisés dans un premier temps sont :

La TelosB [DataSheet TelosB, 2006]— également appelée **SkyMote** selon le constructeur d'origine.
Ce matériel a en effet la spécificité d'être une architecture libre, conçue par l'université de Berkeley, et est donc produite par plusieurs fabricants différents (Crossbow / Memsic, Sentilla, AdvanticSys, etc.).

Cet appareil est conçu autour :
— du microcontrôleur TI MSP430F1611 [DataSheet MSP430F1611, 2011] cadencé à 8 MHz, comportant 48 Ko de mémoire Flash (programme) et 10 Ko de RAM (données),
— de l'émetteur / récepteur radio TI ChipCon CC2420 [DataSheet CC2420, 2007],
— d'un port USB pour la communication notamment avec un PC,
— de divers capteurs intégrés (lumière, et humidité / température en option),
— d'une mémoire Flash externe au microcontrôleur (1 Mo),
— et d'une antenne intégrée (utilisation possible d'une antenne externe en option).

Cette *mote* est maintenant de conception ancienne, et plusieurs appareils concurrents sont apparus sur le même segment de marché.

La Zolertia Z1 [DataSheet Zolertia Z1, 2010]— est l'un de ces concurrents. Son constructeur, Zolertia / Advancare, l'a conçu comme une évolution plus puissante et (relativement) compatible de la TelosB. Il ne s'agit pas comme la TelosB d'un *design* matériel libre, Zolertia / Advancare est donc l'unique source de ce matériel.

Cet appareil est conçu autour :
— du microcontrôleur TI MSP430F2617 [DataSheet MSP430F2617, 2012] cadencé à 16 MHz, comportant 92 Ko de mémoire Flash (programme) et 8 Ko de RAM (données),
— de l'émetteur / récepteur radio TI ChipCon CC2420 [DataSheet CC2420, 2007]
— d'un port micro-USB pour la communication notamment avec un PC,
— de divers capteurs (accéléromètre 3D et température),
— de bus d'extension permettant la connexion de capteurs supplémentaires (« Phidgets »)
— d'une mémoire Flash externe au microcontrôleur (2 Mo),
— et d'une antenne intégrée (utilisation possible d'une antenne externe en option).

Ce matériel, bien que plus récent, est assez répandu, et utilisé industriellement (notamment sous Contiki).

Une version destinée au déboguage (notamment avec un adaptateur JTAG intégré) est disponible auprès de Zolertia / Advancare : Z1 Starter Platform [DataSheet Zolertia Z1 Starter Platform, 2011].

Notons qu'outre l'architecture des microcontrôleurs, ces deux matériels présentent de nombreuses similitudes : une consommation énergétique très faible (les deux appareils sont conçus pour être alimentés par des piles AAA classiques), la même puce radio (émettant sur la bande 2,4 GHz à 250 Kbps), et des performances de même ordre (bien que sensiblement supérieures pour la Z1).

Les deux appareils utilisent également un cristal d'une fréquence de 32768 Hz (cristal de montre à quartz) comme base de temps pour leurs *timers*, ce qui a toute son importance pour des utilisations en temps-réel comme nous le verrons plus bas.

5.2 Implantation de S-CoSenS sous RIOT OS : précision de la synchronisation entre noeuds

Comme nous l'expliquons en fin de section 4.1.3 page 87, nous n'avons pas réussi à implanter de façon satisfaisante le protocole S-CoSenS sous Contiki OS, à cause des limitations imposées par ce dernier. Nous avons alors fait le choix de RIOT OS comme plate-forme logicielle pour l'ensemble de nos travaux de thèse.

Une fois notre plate-forme logicielle de travail choisie, puis améliorée et déboguée notamment grâce nos contributions (détaillées en section 4.2 page 87), notre premier objectif a été de valider notre implantation de S-CoSenS sous RIOT, puis nous avons ensuite cherché à la comparer avec l'implantation standard de ContikiMAC sous Contiki OS.

Pour ces expérimentations, nous avons principalement utilisé, pour des raisons pratiques, le simulateur Cooja plutôt que des tests sur du matériel réel. Tous les noeuds simulés lors de nos expérimentations sont des Zolertia Z1.

Nous avons néanmoins effectué quelques tests de vérification sur du matériel (voir section 5.3.7 page 123) malheureusement légèrement différent de nos noeuds simulés, aucun matériel identique n'étant disponible (les noeuds matériellement les plus proches des Zolertia Z1 nous étant accessibles en quantité sur un *testbed* étant les WSN430 d'IoT-LAB, qui sont similaires aux TelosB / SkyMotes).

Nous avons ainsi programmé des applications de test sur ces *motes* virtuelles, et exécuté des simulations sur un PAN (*"Personal Area Network"*) constitué d'un noeud central dit « routeur » et de dix « noeuds-feuilles » (ou noeuds simples, ou noeuds terminaux). Ces dix noeuds simples envoient régulièrement des trames de données au routeur, qui les retransmet lui-même vers un concentrateur (*"sink"*) situé à proximité. La topologie de ce PAN virtuel est représentée figure 5.1 page suivante.

Dans ce PAN virtuel :

— les dix noeuds-feuilles sont tous à la portée les uns des autres, ***et*** à la portée du routeur ;
— le *"sink"* est à la portée du routeur, mais d'aucun autre noeud ;
— le routeur est à le seul noeud portée de tous les autres, sans exception.

Cette configuration a été volontairement mise en place pour assurer un trajet en deux étapes (*"two-hop transmission"*) pour les trames entre les noeuds-feuilles de départ et la destination (le *"sink"*).

Nos premiers tests ont fait fonctionner le routeur et les dix noeuds simples exclusivement sous le protocole S-CoSenS [Roussel et al., 2015c]. Le but de ces premiers essais était de vérifier la précision de la synchronisation entre les différents appareils composant le PAN.

Cooja simule ici un médium radio « idéal », sans bruit, affaiblissement ou autres problèmes survenant en réel. Tous les noeuds sont à la portée les uns des autres, sans zone d'ombre.

Ces tests ont clairement montré une excellente synchronisation entre les noeuds-feuilles et le routeur, grâce à la granularité temporelle très fine du système de gestion des évènements de RIOT OS (et plus particulièrement la possibilité d'utiliser direc-

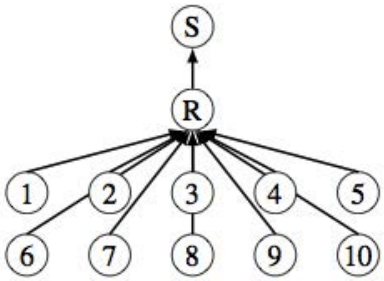

FIGURE 5.1 – *Schéma fonctionnel de notre PAN virtuel de test.*
(Légende : ***R*** *= noeud routeur ;* ***S*** *= "sink")*

tement les différents *timers* matériels disponibles). Cette synchronisation peut être observée sur la copie d'écran de notre simulation Cooja, visible en figure 5.2 page ci-contre [1]. Pour une meilleure lisibilité, la portion centrale de la fenêtre *"timeline"* (délimitée par un rectangle jaune épais) a été zoomée en figure 5.3 page 108

Sur la figure 5.3 page 108, les nombres situés à gauche représentent les identifiants numériques (ID) des *motes* : le routeur possède l'ID numéro 1, tandis que les noeuds-feuilles ont les IDs 2 à 11. Les barres grises représentent les périodes d'activité (écoute active ou transmission) de l'émetteur / récepteur radio d'une *mote* donnée. Les barres bleues représentent l'émission d'une trame, et les barres vertes la réception ***réussie*** d'une trame, tandis que les barres rouges représentent des collisions (quand plusieurs appareils émettent simultanément) et correspondent ainsi à la présence sur le canal radio de signaux indéchiffrables.

La figure 5.3 page 108 représente un court instant (environ 100 millisecondes), correspondant à la fin d'un cycle radio du routeur : les 20 premières millisecondes correspondent à la fin de la période de sommeil SP (voir le mode de fonctionnnement de S-CoSenS en section 3.2.6.1 page 47), et les 80 millisecondes suivantes représentent la période d'écoute WP, avant le début d'un nouveau cycle radio (la période de retransmission TP du routeur ayant été désactivée dans la simulation montrée sur ces copies d'écran pour une meilleure lisibilité).

Dans l'exemple de la figure 5.3 page 108, quatre noeuds ont des données à transmettre au routeur : les *motes* numéro 3, 5, 9 et 10 ; les autres noeuds (2, 4, 6, 7, 8 et 11) se préparent eux à transmettre une trame durant le cycle suivant.

À l'instant marqué par la première flèche jaune (en haut à gauche de la figure 5.3 page 108), la période de sommeil SP se termine et le routeur active son émetteur / récepteur radio pour entrer en période d'écoute WP. On notera que les quatre noeuds ayant des trames à émettre (3, 5, 9 et 10) activent également leur radio ***précisément*** au même instant (à un *tick* de *timer* soit environ 30 microsecondes près) : ceci grâce à la précision des mécanismes temps-réel de RIOT OS (basé sur les *timers* matériels), permettant aux différents noeuds de se synchroniser précisément sur les valeurs temporelles transmises par le routeur dans le précédent *beacon* de début de

1. **Note :** bien que le titre de la figure 5.2 page ci-contre mentionne Contiki (dont le projet est à l'origine du simulateur Cooja), les applications simulées fonctionnent bel et bien sous RIOT OS.

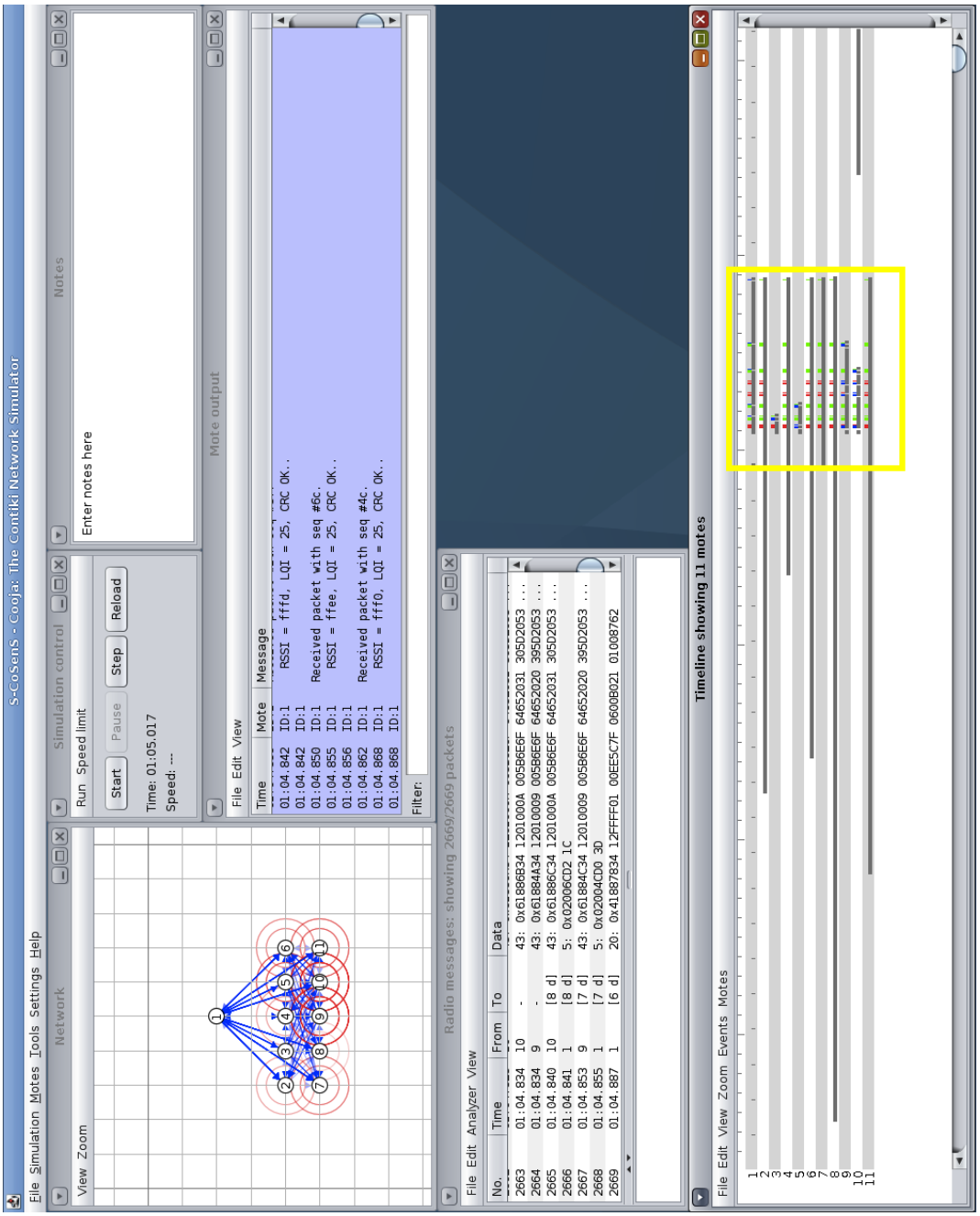

FIGURE 5.2 – *Copie d'écran d'une des simulations de notre premier jeu de tests sous Cooja.*

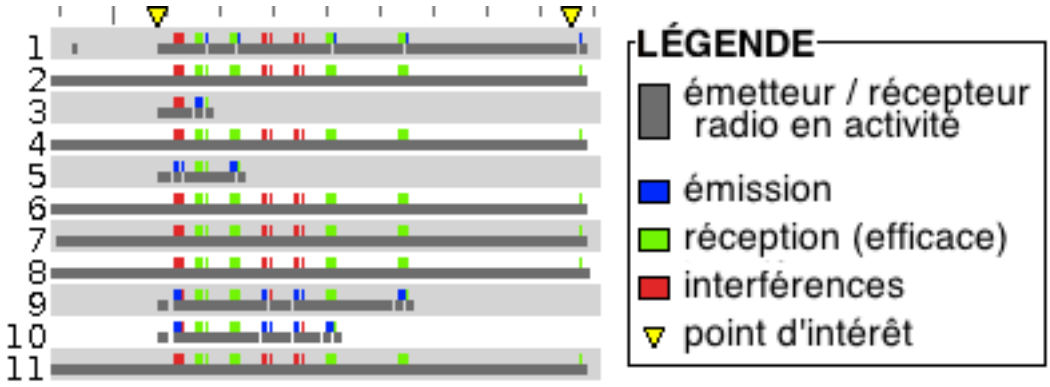

FIGURE 5.3 – *Zoom sur la partie centrale de la timeline de la figure 5.2 page précédente.*

cycle. Grâce à ce même mécanisme, les noeuds sont capables de passer leur émetteur / récepteur radio ***et*** leur microcontrôleur en mode basse consommation aux moments adéquats, le noyau de RIOT OS étant piloté par les interruptions.

Durant la période d'écoute, nous voyons également que plusieurs collisions se produisent ; celles-ci sont résolues par la méthode CSMA/CA (sur laquelle est basé S-CoSenS), laquelle oblige les *motes* à attendre un délai aléatoire avant de réémettre une trame en cas de conflit. Dans cet exemple, nos quatre noeuds-feuilles peuvent finalement transmettre leur trame au routeur dans cet ordre : 3 (après une première collision), 5 et 10 (après deux autres collisions), et finalement 9.

Notons que chaque fois que le routeur (ID numéro 1) reçoit une trame avec succès, un acquittement (ACK) est renvoyé à l'émetteur : cela correspond aux barres bleues très fines suivant chaque barre verte sur la ligne numéro 1.

Finalement, à l'instant marqué par la seconde flèche jaune (en haut à droite de la figure 5.3), la période d'écoute WP s'achève, et un nouveau cycle commence. En conséquence, le routeur émet un *beacon* contenant les durées calculées pour les différentes périodes constituant le nouveau cycle, et qui serviront à la synchronisation des dix noeuds-feuilles avec le routeur.

Il est à noter que les *beacon*, contrairement aux trames de données, ne font pas l'objet d'une transmission classique avec un ou plusieurs destinataire(s) désigné(s) (*"unicast"* ou *"multicast"*), mais sont émis pour réception par tous les noeuds à portée d'écoute : on dit que les *beacons* font l'objet d'une diffusion (*"broadcast"*). Par conséquent, contrairement aux trames de données, les *beacons* — étant diffusés — ne font pas l'objet d'un acquittement après réception.

Nous pouvons voir que tous les six noeuds attendant de transmettre une trame durant ce nouveau cycle (2, 4, 6, 7, 8 et 11) se mettent en sommeil (leur radio et leur microcontrôleur passent en mode basse consommation) juste après avoir reçu le *beacon* ; comme nous le voyons dans la figure 5.3, ils vont profiter de la précision des fonctionnalités temps réel de RIOT pour se réveiller précisément quand le routeur passera en période d'écoute WP et sera ainsi prêt à recevoir leurs trames.

5.3 Évaluation des performances : comparaison avec ContikiMAC

Après avoir ainsi validé (de façon théorique) la fiabilité de notre implantation de S-CoSenS sous RIOT, nous avons comme prévu lancé des simulations pour la comparer avec l'implantation standard de ContikiMAC sous Contiki OS [Roussel et al., 2015b].

5.3.1 Configuration des expériences

Nous avons continué à effectuer des simulations sur le même PAN virtuel montré en figure 5.1 page 106. Nous avons cette fois programmé des applications de test équivalentes sous Contiki (avec ContikiMAC) et RIOT OS (avec S-CoSenS), effectuant des scénarios similaires au premier : les dix noeuds-feuilles envoient des trames au routeur, qui les retransmet au *sink* jouant le rôle de destination finale des trames. Ces modalités de transfert sont statiquement programmées, aucun protocole de routage n'étant utilisé : nous testons ici uniquement les protocoles MAC / RDC, et non des piles complètes.

Notre second type de simulations a consisté à faire varier le trafic réseau, générant des charges allant d'un encombrement modéré à un trafic extrême. Nous avons ainsi cherché à tester le comportement de ces protocoles MAC / RDC face à des charges réseau élevées, largement supérieures au trafic habituellement traité dans les réseaux de capteurs sans-fil actuels.

Ces charges réseau sont générées par l'application de test s'exécutant sur les dix noeuds-feuilles. Elle a été programmée pour générer des trames 802.15.4 de grande taille : 90 octets de charge utile, ce qui correspond à une taille de trame physique de 110 octets. Les trames sont envoyées de façon à respecter un intervalle d'arrivée de trames (*"Packet Arrival Interval"* ou PAI) moyen, en envoyant les trames suivant un intervalle fixe modulé par un décalage aléatoire pouvant atteindre la moitié du délai voulu — l'intervalle entre l'envoi de deux trames consécutives est donc aléatoirement choisi entre $[0,5\,\text{PAI}\ ;\ 1,5\,\text{PAI}]$ — pour limiter les collisions entre noeuds.

Les différentes configurations sont décrites en table 5.1 : cette table donne le PAI moyen souhaité, le nombre moyen de trames émises chaque seconde par l'ensemble des dix noeuds-feuilles, et le débit réseau résultant attendu.

Configuration	PAI moyen	Trames/s	Débit attendu
Modéré	1500 ms	6,7	5867 bit/s
Élevé	1000 ms	10	8800 bit/s
Très élevé	500 ms	20	17600 bit/s
Extrême	100 ms	100	88000 bit/s

TABLE 5.1 – *Débits réseaux utiles programmés sur l'ensemble des noeuds-feuilles.*

En considérant la surcharge occasionnée par la couche MAC / RDC, le chemin de transmission en deux étapes — ce qui signifie que le noeud routeur, à un moment

donné, doit communiquer soit avec les noeuds-feuilles, soit avec le *sink* ; ce qui revient à diviser effectivement la bande passante du réseau par deux —, la grande taille de nos trames de données 802.15.4 (proche du maximum physique de 127 octets), nous nous attendons à ce que notre scénario « très élevé » soit proche du débit maximal effectif possible, et à ce que le scénario « extrême » soit largement au-délà de la saturation du médium radio.

5.3.2 Taux de réception des paquets

Nous avons utilisé le taux de réception des paquets (TRP, en anglais PRR : *Paquet Reception Rate*) au niveau du *"sink"* comme premier indicateur pour quantifier la qualité de service (QdS) obtenue par les protocoles testés.

Nous avons ainsi pu déterminer que le seul paramètre commun aux deux protocoles ayant une influence réelle sur le TRP est la longueur de la *subframe* sous S-CoSenS (cf. section 3.2.6.1 page 47), l'équivalent sous ContikiMAC étant l'intervalle de réveil (*check interval*, durée entre deux écoutes consécutives du médium radio, cf. section 3.2.2.4 page 39). Ces deux paramètres correspondent en fait, dans leurs protocoles respectifs, à la longueur du cycle de fonctionnement du protocole, et donc à la fréquence de renouvellement de ces cycles au cours d'une seconde.

Cet intervalle est configuré, au niveau de l'implantation, en changeant la fréquence à laquelle le médium radio est écouté : ce paramètre, dans le code source de Contiki, est une constante de compilation nommée `NETSTACK_CONF_RDC_CHANNEL_CHECK_RATE` donnée en Hertz. Sa valeur par défaut est de 8, ce qui signifie que ContikiMAC écoute cycliquement le médium radio pour détecter une activité huit fois par seconde.

S-CoSenS offre également un autre paramètre permettant de régler finement l'équilibre entre qualité de service et économie d'énergie (via la mise en sommeil de la radio) : il s'agit des valeurs WP_{min} et WP_{max} dont le rôle est de borner le poids relatif de la période d'attente WP au sein des cycles de S-CoSenS dans un intervalle choisi par l'utilisateur. Plus spécifiquement, l'augmentation de la valeur de WP_{min} permet de s'assurer que le TRP ne descendra jamais en dessous d'un certain niveau. Pour nos simulations, étant donné que nous souhaitons obtenir une QdS maximale, nous avons systématiquement défini $\mathrm{WP}_{max} = \mathrm{PAI}$ et $\mathrm{WP}_{min} = \frac{1}{2}\mathrm{PAI}$, PAI étant ici l'intervalle moyen d'arrivée des packets souhaité pour la simulation donnée.

Notons également que nous avons permis, sous Contiki, à la couche MAC CSMA / CA standard d'effectuer jusqu'à 7 tentatives de retransmissions pour une trame donnée, en définissant la constante de compilation `CSMA_CONF_MAX_MAC_TRANSMISSIONS` à 8 dans le code source de Contiki. Ceci a été fait pour mettre Contiki sur un pied d'égalité avec la configuration par défaut de S-CoSenS, dans laquelle une trame peut tenter d'être transmis jusqu'à 8 fois avant d'être abandonné. Attention toutefois : le terme de « réessai » n'a pas la même sémantique sous S-CoSenS — où un réessai signifie la réémission ***d'une instance*** d'une trame donnée — que sous ContikiMAC — où un réessai indique ***un nouveau cycle*** durant lequel la trame sera réémise un nombre de fois indéterminé. Cette différence de signification est la conséquence des

conceptions totalement différentes entre les deux protocoles, comme cela a été décrit dans la section 3.2 page 34 décrivant l'état de l'art en matière de protocoles MAC.

La configuration par défaut de 8 Hz / 125 ms pour le *channel check rate* ne donne que des résultats médiocres pour ContikiMAC, cette valeur n'ayant visiblement pas été prévue pour permettre à ce protocole de gérer les lourdes charges réseau. Dans un souci d'équité, nous avons successivement doublé ce paramètre à 16 Hz / 62 ms, puis à 32 Hz / 31 ms pour retenter nos tests de comparaison.

Les résultats que nous avons obtenus pour le TRP (PRR dans la légende des figures) sont montrés dans les figures 5.4 à 5.8 pages 111–113. Ces figures montrent le taux de paquets arrivés avec succès à leur destination, en fonction de la durée du cycle sous S-ConSenS, ou du *channel check rate* sous ContikiMAC. Ces résultats sont obtenus avec des simulations envoyant toujours un nombre total constant de trames : 500 trames par noeud-feuille, soit 5000 trames au total pour chaque simulation.

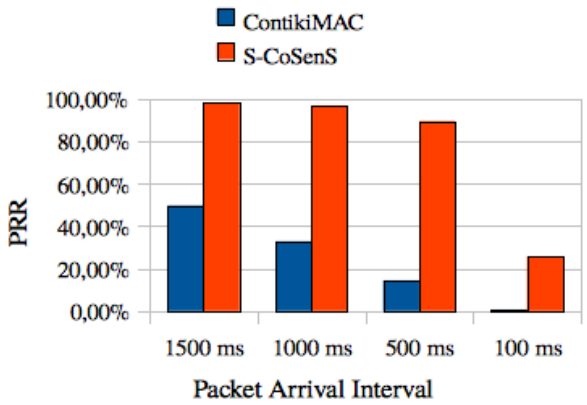

FIGURE 5.4 – *Résultats pour le taux de réception de paquets (PRR) pour un cycle MAC / RDC de 8 Hz / 125 ms.*

Les données représentées dans les figures 5.4 à 5.8 pages 111–113 nous montrent les faits suivants :

— Le TRP de ContikiMAC augmente de façon significative avec le *channel check rate*, c'est-à-dire : quand cet intervalle de *channel check rate* diminue, aboutissant à des cycles plus courts et des écoutes plus fréquentes du médium radio.

— Le TRP de S-CoSenS est toujours soit similaire, soit supérieur à celui de ContikiMAC, quel que soit le scénario observé.

— Le TRP de S-CoSenS reste très stable quelle que soit la valeur donnée à la durée du cycle (*subframe*) : cette stabilité est clairement visible en figure 5.7 page 113. À l'inverse, la figure 5.8 page 113 montre nettement que le TRP de ContikiMAC varie énormément en fonction du *channel check rate*, celui-ci devant être quadruplé (par rapport à sa valeur par défaut) pour obtenir des résultats comparables à S-CoSenS.

  Par conséquent, il est clair que les performances de S-CoSenS, en matière de

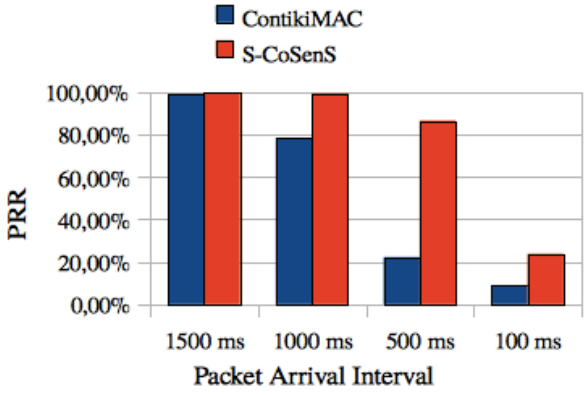

FIGURE 5.5 – *Résultats pour le taux de réception de paquets (PRR) pour un cycle MAC / RDC de 16 Hz / 62 ms.*

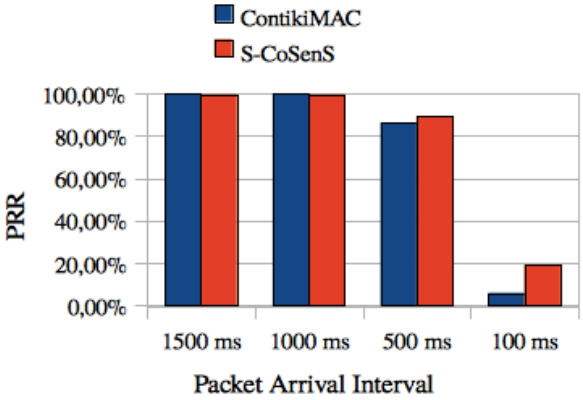

FIGURE 5.6 – *Résultats pour le taux de réception de paquets (PRR) pour un cycle MAC / RDC de 32 Hz / 31 ms.*

TRP, sont clairement plus prévisibles, et permettent de s'assurer que le TRP restera dans des valeurs correctes (> 80%) dans la plupart des situations — c'est-à-dire tant que le médium radio n'est pas complètement saturé comme dans le cas de notre scénario « extrême ».

L'analyse détaillée des résultats des simulations, notamment le nombre de retransmissions nécessaires pour chaque trame arrivé à destination, nous montre également que :

— avec ContikiMAC, les pertes de trames sont principalement dues au routeur ;
— avec S-CoSenS, le routeur ne perd jamais les trames qu'il a reçus, toutes les pertes ont lieu lors des transmissions entre les noeuds-feuilles et le routeur.

Cela n'est pas surprenant, car S-CoSenS réserve explicitement une période de transmission (TP) durant chaque cycle exécuté par le routeur, pour permettre à ce dernier de retransmettre les trames reçues dans les meilleures conditions possibles. ContikiMAC ne possède lui aucune fonctionnalité équivalente.

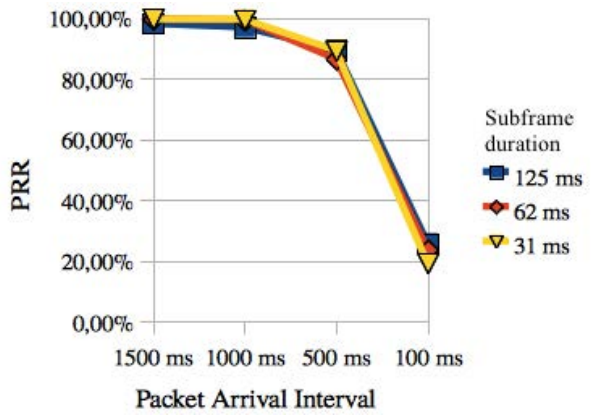

FIGURE 5.7 – *Résultats combinés pour le taux de réception de paquets (TRP) avec S-CoSenS, en fonction de la durée de la subframe.*

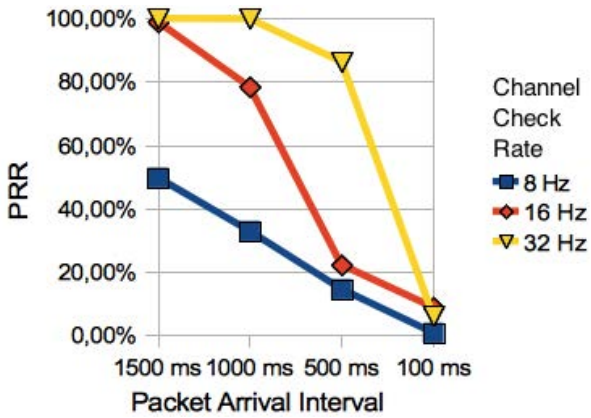

FIGURE 5.8 – *Résultats combinés pour le taux de réception de paquets (TRP) avec ContikiMAC, en fonction du channel check rate.*

Durant ces périodes de transmission, une « priorité probabilistique » est donnée par le protocole S-CoSenS aux routeurs, en leur accordant un intervalle de base inférieur pour le calcul des périodes de *backoff* (il s'agit du paramètre nommé ***macMinBE*** par le standard IEEE 802.15.4) par rapport aux autres noeuds : pour les routeurs $macMinBE = 2$, tandis que pour les autres noeuds $macMinBE = 3$. Durant leurs périodes de transmission, les routeurs sont ainsi privilégiés pour transmettre les trames présentes dans leur file d'envoi. Enfin et surtout, les noeuds-feuilles appartenant au même PAN qu'un routeur, et ayant reçu ses *beacons*, savent quand se déroule la période de réception (WP) dudit routeur, et quand celle-ci s'achève pour laisser la place à la période de transmission (TP). Les noeuds-feuilles appartenant au même PAN que le routeur s'abstiennent alors de toute transmission, remettant l'envoi des trames leur restant éventuellement à envoyer au cycle suivant.

On peut clairement comprendre qu'au contraire, sous une forte charge réseau, un routeur ContikiMAC va avoir des trames de données à recevoir quasiment à chaque

fois que celui-ci va sonder le médium radio (*"Channel Check"*) à chaque cycle. Ainsi, il va devenir de plus en plus difficile pour ce routeur de trouver un intervalle disponible pour l'envoi de ses propres trames au fur et à mesure que le trafic depuis les noeuds-feuilles augmente. C'est pourquoi au cours de tels scénarios, le noeud routeur devient clairement le point faible des PANs fonctionnant sous ContikiMAC ; tandis que S-CoSenS peut éviter ces difficultés grâce à sa conception, incluant une période de transmission (TP) réservée à l'expédition des trames en attente dans la file d'envoi des routeurs.

5.3.3 Délai total (« de bout-en-bout ») de transmission des trames

Nous avons également calculé le délai que prend chaque trame transmise avec succès pour aller de son noeud-feuille d'origine jusqu'au sink — c'est à dire la durée totale de son voyage en deux étapes (*"two-hop transmission"*). Ceci est le second indicateur de QdS que nous avons étudié lors de ces expériences.

Les résultats que nous avons obtenus sont montrés dans les figures 5.9 à 5.13 pages 114–116. Ces figures représentent les délais moyens nécessaires aux trames transmises avec succès pour effectuer leur voyage en deux étapes. Notons que les figures 5.9 à 5.11 pages 114–115, pour des raisons de lisibilité (tant la différence entre les résultats est énorme) montrent les délais sur un axe vertical en échelle ***logarithmique***.

Contrairement aux résultats de TRP, ces résultats sont obtenus avec des simulations où sont envoyés sans interruption des trames en nombre indéfini, suivant là encore un intervalle moyen précalculé de transmission entre deux trames consécutives. Le nombre de trames envoyées n'est donc ici plus constant ; mais la charge générée sur le réseau reste fixée de la même façon que pour les expériences sur le TRP.

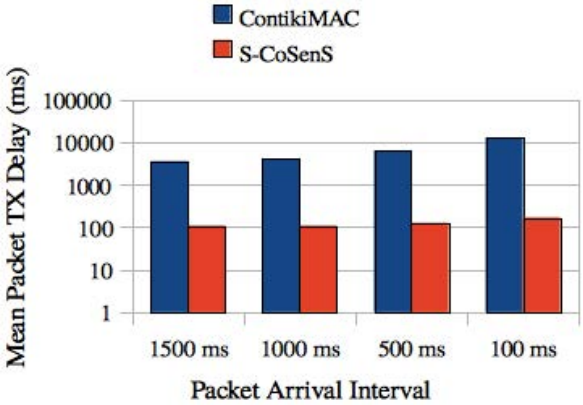

FIGURE 5.9 – *Résultats pour les délais moyens de transmission d'une trame pour un cycle MAC / RDC de 8 Hz / 125 ms.*

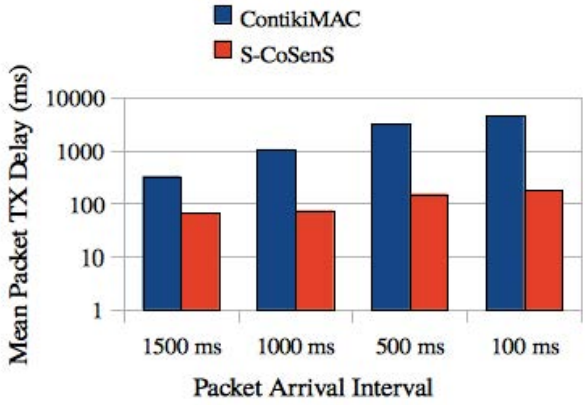

FIGURE 5.10 – *Résultats pour les délais moyens de transmission d'une trame pour un cycle MAC / RDC de 16 Hz / 62 ms.*

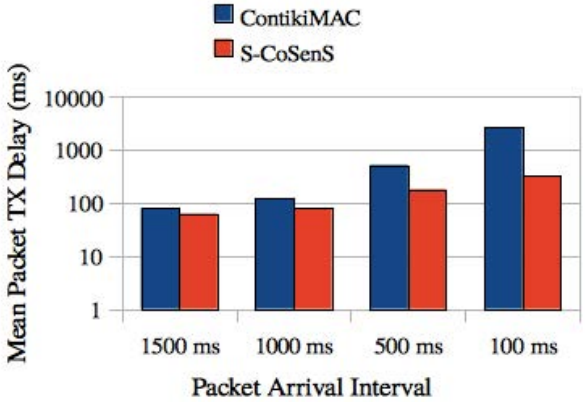

FIGURE 5.11 – *Résultats pour les délais moyens de transmission d'une trame pour un cycle MAC / RDC de 32 Hz / 31 ms.*

Ces résultats sur les délais de transmissions nous indiquent que :

— Les délais de transmission sous ContikiMAC diminuent avec l'augmentation du *channel check rate*, c'est-à-dire : quand les cycles ContikiMAC deviennent eux-mêmes plus courts.

— Les délais de transmission sous S-CoSenS sont toujours soit similaires, soit (dans la majorité des cas) nettement plus courts que ceux de ContikiMAC, quel que soit le scénario observé.

— Les délais de transmission sous S-CoSenS restent stables quelle que soit la valeur donnée à la durée du cycle (*subframe*) : cette stabilité est clairement visible en figure 5.12 page suivante. Dans la plupart des cas (c'est-à-dire quand le médium radio n'est pas saturé), S-CoSenS permet de s'assurer que les délais de transmission des trames resteront inférieurs à une limite raisonnable (toujours < 200 ms, y compris dans les cas les moins favorables).
À l'inverse, la figure 5.13 page suivante montre une instabilité énorme sous

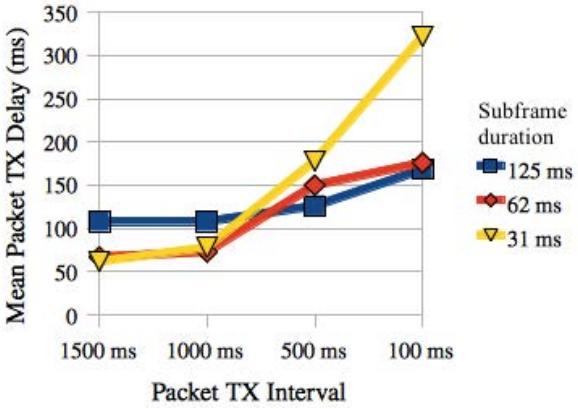

FIGURE 5.12 – *Résultats combinés pour les délais moyens de transmission d'une trame avec S-CoSenS, en fonction de la durée de la subframe.*

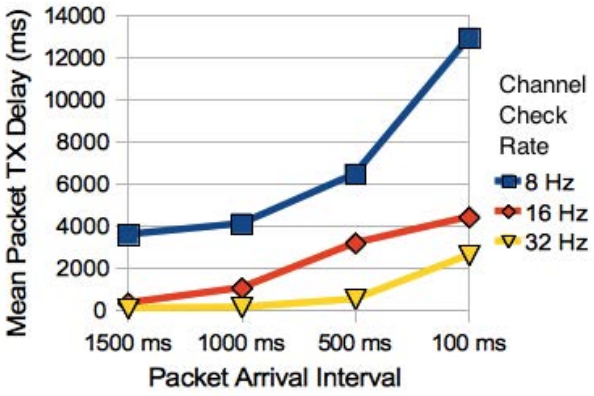

FIGURE 5.13 – *Résultats combinés pour les délais moyens de transmission d'une trame avec ContikiMAC, en fonction du channel check rate.*

ContikiMAC dans les délais de transmission en fonction des scénarios étudiés : avec le réglage par défaut (8 Hz), le délai de transmission des trames dépasse systématiquement et largement le seuil de la seconde (plus de 12 secondes pour transmettre une trame dans le cas le plus extrême).

On peut noter que ces trois observations (grande variabilité des résultats de ContikiMAC, résultats similaires ou meilleurs pour S-CoSenS, stabilité des résultats de S-CoSenS garantissant une QdS minimale) sont strictement similaires à celles notées pour le taux de réception de paquets.

L'énorme augmentation dans les délais de transmission de bout-en-bout pour ContikiMAC peut être interprétée comme un « excès de zèle ». Une trame peut être transmise au cours d'un nombre prédéfini d'essais (3 par défaut sous Contiki, 8 dans notre configuration par analogie avec S-CoSenS). Rappelons que pour le couple ContikiMAC – CSMA/CA, la transmission d'une trame peut être mise en attente

jusqu'à la disponibilité du médium radio, et qu'un « essai de transmission » consiste à envoyer répétitivement cette trame jusqu'à concurrence d'un cycle entier (durée calculée du *channel check rate*) : par conséquent, un délai énorme peut s'écouler avant que ce protocole ne finisse par abandonner la transmission d'une trame donnée.

Si une telle « obstination » peut (en théorie) améliorer le TRP en augmentant les chances de réception d'une trame par son destinataire, elle finit par allonger de façon considérable le délai moyen pour la transmission d'une trame de bout-en-bout. Au final, on peut s'interroger sur l'utilité de continuer à chercher à transmettre des trames ayant atteint un tel âge (dont les données risquent de devenir obsolètes), souvent au détriment de la transmission de trames plus récentes contenant des données plus actuelles donc plus utiles.

La configuration du nombre d'essais à tenter pour la transmission d'une trame donnés est — comme pour de nombreux autres paramètres — un équilibre optimal à trouver entre deux buts contradictoires : on souhaite atteindre un TRP maximal, tout en permettant la perte et l'élimination des trames trop âgées dont les données ont vraisemblablement perdu leur intérêt. Il apparaît que dans les différents scénarios que nous avons étudié, S-CoSenS permet d'obtenir un meilleur compromis entre ces deux objectifs que ContikiMAC.

5.3.4 Consommation d'énergie : *"Duty Cycles"*

Cooja peut facilement calculer les statistiques précises de *"duty cycle"* — ou pour être plus exact : la fraction de temps pendant laquelle les émetteurs / récepteurs radio sont actifs — pour chacune des *motes* virtuelles émulées. Ces statistiques sont un indicateur très significatif pour quantifier les consommations d'énergie dans un réseau de capteurs sans-fil, l'émetteur / récepteur radio étant, et de loin, le circuit le plus gourmand en énergie au sein d'une *mote*.

Les résultats obtenus pour les statistiques de *duty cycle* sont présentés dans les tables 5.2 à 5.3 page suivante. Notons qu'à cause des différences importantes de TRP pour les durées de cycle de 125 et 62 ms, seule la comparaison des résultats de *duty cycle* pour des cycles de 31 ms avait un sens ; ce sont donc les seuls résultats que nous présentons ici.

Dans ces deux tables, les lignes « activité radio » indiquent le pourcentage du temps de simulation durant lequel l'émetteur / récepteur accède activement au médium, et consomme donc de l'énergie. Parmi ce temps d'activité, on distingue :

- la fraction de temps durant laquelle le noeud émet avec succès des trames de données (lignes « transmission efficace ») ;
- la fraction de temps durant laquelle le noeud reçoit avec succès des trames de données (lignes « réception efficace ») ;
- la fraction de temps durant laquelle plusieurs noeuds émettent simultanément, conduisant à la présence de signaux indéchiffrables sur le médium radio (lignes « interférences », qui couvrent en fait toutes les émissions sans succès) ;
- le temps d'activité radio n'étant pas couvert par ces trois états précédents correspondent à des écoutes « à vide » du canal radio.

ContikiMAC : *Channel Check Interval* = 32 Hz			
	INTERVALLE D'ÉMISSION DES TRAMES		
	1500 ms	1000 ms	500 ms
ROUTEUR			
Activité radio	12,77%	17,85%	31,13%
Transmission efficace	2,64%	4,00%	6,88%
Réception efficace	2,78%	4,18%	8,24%
Interférences	0,16%	0,40%	2,64%
NOEUDS-FEUILLES (moyenne)			
Activité radio	5,61%	7,45%	14,27%
Transmission efficace	0,90%	1,48%	3,89%
Réception efficace	0,66%	1,03%	1,96%
Interférences	0,11%	0,21%	1,13%

TABLE 5.2 – *Statistiques de duty cycle pour ContikiMAC.*

S-CoSenS : Durée d'un cycle (*subframe*) = 31 ms			
	INTERVALLE D'ÉMISSION DES TRAMES		
	1500 ms	1000 ms	500 ms
ROUTEUR			
Activité radio	66,22%	67,56%	67,91%
Transmission efficace	4,96%	5,48%	8,43%
Réception efficace	3,84%	4,43%	8,90%
Interférences	1,04%	1,48%	8,39%
NOEUDS-FEUILLES (moyenne)			
Activité radio	6,15%	7,68%	40,04%
Transmission efficace	0,58%	0,70%	2,64%
Réception efficace	0,68%	0,93%	7,08%
Interférences	0,16%	0,22%	3,98%

TABLE 5.3 – *Statistiques de duty cycle pour S-CoSenS.*

Les statistiques de *duty cycle* présentées dans les tables 5.2 à 5.3 page ci-contre indiquent de façon évidente que ContikiMAC possède sur ce terrain un avantage très significatif. Les taux d'activité radio, que ce soit en émission, en réception, ou en attente, sont toujours plus faibles — c'est à dire meilleurs — pour ContikiMAC que pour S-CoSenS. Cela est particulièrement vrai pour l'activité radio globale sur les noeuds routeurs, où la différence en faveur de ContikiMAC est proprement spectaculaire. (Cette différence est surtout due à la nécessité de fixer WP_{min} à 50% pour estimer correctement le trafic radio — cf. section 5.4.1 page 125.) Ces constatations sont illustrées par les figures 5.14 à 5.15 de la présente page.

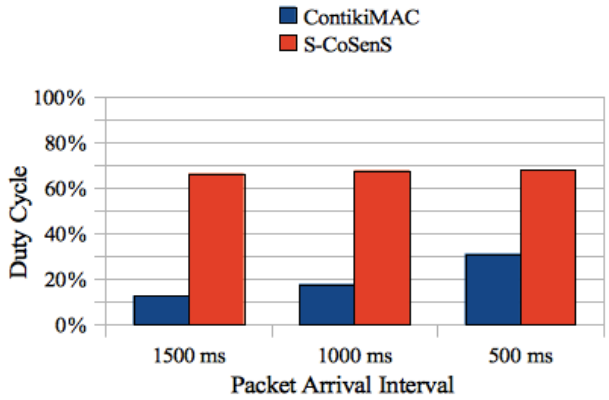

FIGURE 5.14 – *Comparaison de l'activité radio (duty cycle) sur les noeuds routeurs fonctionnant respectivement sous ContikiMAC et S-CoSenS, avec des subframes de 31 ms.*

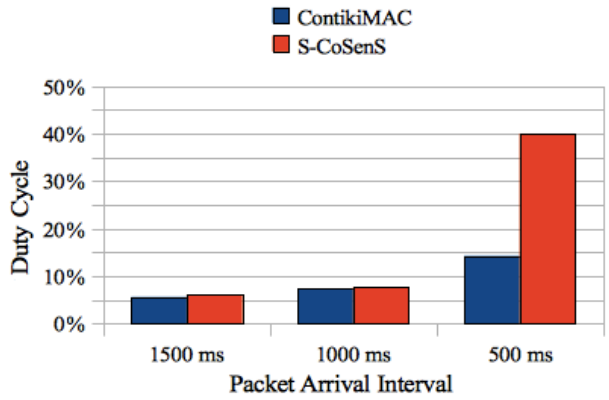

FIGURE 5.15 – *Comparaison de l'activité radio (duty cycle) sur les noeuds-feuilles fonctionnant respectivement sous ContikiMAC et S-CoSenS, avec des subframes de 31 ms.*

ContikiMAC prouve ainsi qu'il est en effet ***un protocole à très basse consommation d'énergie***.

La seule exception à cette constatation de supériorité générale de ContikiMAC en matière de *duty cycle* se situe dans les temps d'émission (lignes « transmission efficace ») : pour les noeuds-feuilles — mais ***pas*** pour les routeurs — les *duty cycles* sont systématiquement légèrement inférieurs (c-à-d. meilleurs) pour S-CoSenS. Cela s'explique aisément par la conception de ces deux protocoles : S-CoSenS est conçu — pour les transmissions entre noeuds-feuilles et routeur — sur le paradigme RI-LPP (cf. section 3.2.3 page 40), ce qui implique naturellement moins de tentatives de transmissions de trames de données que pour ContikiMAC qui est conçu sur le paradigme LPL (cf. section 3.2.2.4 page 39) — et plus spécifiquement sur l'envoi continu de trames de données entières par l'émetteur jusqu'à acquittement par le destinaire.

5.3.5 Stabilité et contraintes mémoire

Durant le développement et l'évaluation de S-CoSenS, nous avons fait face à plusieurs plantages, notamment lorsque nous avons tenté d'utiliser de longues files d'envoi de trames. Ces plantages étaient dûs aux débordements de la pile système qui empiétait sur l'espace mémoire de cette file : quand cette dernière était pleine, les données des trames pouvaient ainsi écraser le contenu de la pile système (lorsque ladite pile contenait une quantité de données dépassant son espace mémoire alloué), aboutissant bien évidemment à des erreurs logicielles fatales.

Les plantages observés avec cette implantation de S-CoSenS sous RIOT sont dûs à la quantité très limitée de RAM disponible pour les données (8 Ko) dans le microcontrôleur MSP430 autour duquel est bâtie la *mote* Zolertia Z1. Ceci, combiné à l'absence d'unité de protection mémoire (MPU) dans cette architecture matérielle, aboutit à la survenue de problèmes de corruption mémoire silencieux et indétectables en temps voulu.

Si l'utilisation de microcontrôleurs disposant d'une quantité de RAM plus conséquente peut limiter la survenue de tels problèmes, la détection systématique de ces derniers et leur traitement correct nécessite l'un des mécanismes suivants :

1. ***Un mécanisme matériel*** destiné à détecter les accès mémoire incorrects et à provoquer une exception au début de leur survenue, avant leur réalisation effective, prévenant ainsi leurs conséquences : au sein d'un microcontrôleur, ceci est le rôle d'une **MPU** (*Memory Protection Unit*). Certaines architectures (comme les ARM Cortex-M) proposent en option une telle fonctionnalité. Celle-ci nous semble indispensable pour la conception de systèmes destinés à offrir un niveau minimal de sûreté de fonctionnement (*"safe systems"*).

2. Pour les architectures dépourvues de MPU, ***un système de support logiciel*** dont le rôle est de surveiller les valeurs de registres critiques — tels que le pointeur de pile (**SP** : *Stack Pointer*) — ne serait-ce que pour s'assurer que ces valeurs restent comprises au sein d'intervalles autorisés.
   Un tel mécanisme peut être implanté ***au niveau d'un système d'exploitation*** (FreeRTOS et RIOT fournissent une telle option) ; mais ce mécanisme ne peut — sans le support matériel d'une MPU — que vérifier et constater la présence ou non d'une violation d'espace mémoire, et en aucun cas empêcher sa

survenue et ses conséquences. En cas de problème, la seule alternative possible reste d'arrêter le système, et de le faire éventuellement redémarrer « à froid » (*"hard reset"*).
Un moyen plus efficace serait ***d'ajouter automatiquement, au cours du processus de compilation, des séquences de code de vérification,*** notamment pour surveiller l'évolution de la valeur du registre SP lors de l'entrée dans un sous-programme ou — pour les environnements multitâches — lors d'un changement de contexte. Ces types d'évènements étant ceux augmentant la taille de la pile, ils correspondent donc aux moments où cette dernière risque de déborder de son espace réservé. Il deviendrait alors possible (comme avec une MPU) d'implanter un traitement spécifique préventif des erreurs mémoires, n'interrompant que les tâches ou routines fautives, sans devoir arrêter le système entier.

Pour nos présentes expérimentations, nous avons contourné le problème en réduisant simplement la profondeur de notre file d'envoi de trames, laissant ainsi une place suffisante à la pile système.

Notons enfin que ce problème n'est nullement spécifique à RIOT ni à aucun système d'exploitation embarqué, mais peut toucher tous les OS fonctionnant sur du matériel aux ressources limitées, et surtout dépourvu d'une MPU ou d'un autre mécanisme matériel de protection de la mémoire. Citons par exemple [Oikonomou and Phillips, 2011] où sont expliquées les difficultés de portage de Contiki sur une plate-forme matérielle courante et aux ressources limitées : on y parle clairement des problèmes de débordement de pile (voir par exemple la figure 1 de cette publication), et des optimisations qui ont été nécessaires pour éviter ce problème.

La stratégie de conception de systèmes extrêmement peu gourmands en ressources comme Tiny OS et Contiki est de pousser l'optimisation à l'extrême pour limiter ces problèmes au maximum (sans pour autant pouvoir garantir une fiabilité absolue).

5.3.6 Influence de l'optimisation des implantations

Lors de nos différents tests, nous avons constaté que les délais observés pour charger les trames dans le *buffer* d'envoi du composant radio CC2420 étaient toujours significativement plus longs sous RIOT OS que sous Contiki.

Nos investigations à ce sujet nous ont montré que ces importantes différences de délai étaient dues à la façon dont la communication sur le bus SPI — assurant le lien entre le microcontrôleur central et l'émetteur / récepteur radio dans la *mote* utilisée — est gérée par le système d'exploitation. Sous RIOT OS, lorsqu'un octet est transmis sur le bus SPI, le résultat de la transmission est toujours attendu, en consultant les *flags* indiquant le succès ou l'échec, avant de transmettre l'octet suivant. À l'inverse, Contiki utilise une procédure nommée *"fast SPI write"* envoyant un octet sur le bus SPI dès que ce dernier est disponible (c'est-à-dire que le registre d'envoi de l'interface microcontrôleur du bus SPI est disponible en écriture), sans

attendre de connaître le résultat de la transmission précédente (les *flags* indiquant le résultat d'une transmission prenant un certain temps pour se mettre à jour).

Si la solution utilisée par Contiki est clairement plus rapide, elle est également peu sûre car incapable de détecter correctement un problème pouvant survenir sur le bus SPI ; l'implantation de RIOT est au contraire sûre, mais plus lente.

Il y a clairement ici un compromis entre optimisation des performances et fiabilité. Pour le chargement du *buffer* d'envoi de l'émetteur / récepteur radio CC2420, cela dépend en fait moins de l'OS lui-même que des choix d'implantation pour le pilote du bus SPI.

Système / Pilote SPI	Délai chargement	
RIOT OS / Standard (Sûr)	89 ticks	2716 μs
RIOT OS / *"Fast SPI write"*	36 ticks	1099 μs
Contiki / *"Fast SPI write"*	14 ticks	213 μs

TABLE 5.4 – *Delais observés pour le chargement d'une trame de 110 octets dans le buffer d'envoi de la radio CC2420 d'une mote Zolertia Z1, en fonction de l'OS et de l'implantation du pilote SPI.*

Nous avons implanté l'optimisation *"fast write"* du pilote SPI sous RIOT, et avons mesuré les différences de rapidité entre les deux versions du pilote sous RIOT, ainsi qu'avec le pilote standard de Contiki. Les résultats sont montrés en table 5.4.

Les délais dans la table 5.4 sont donnés dans la deuxième colonne en *"ticks"* de `rtimer/hwtimer` [2], les deux s'incrémentant à la fréquence de 32768 Hz. Le *"tick"* est donc une unité de temps fixe égale à environ 30,5 microsecondes. Les valeurs correspondantes en microsecondes sont données en troisième colonne pour plus de facilité.

Nous pouvons quoi qu'il en soit noter que la pénalité imposée à S-CoSenS par le pilote SPI standard de RIOT ne l'a pas empêché d'obtenir des performances supérieures en matière de QdS (tous les tests faits avec RIOT décrits dans la présente section ayant été exécutés avec le pilote SPI standard du système).

Notons par contre que, contre toute attente, les délais de chargement obtenus par simulation sous Cooja / MSPSim, montrés en table 5.4 diffèrent très largement de ceux obtenus sur une *mote* Zolertia Z1 réelle. Nous reviendrons en détail sur ce phénomène dans la section 6.1 page 136 du prochain chapitre.

Ce phénomène étonnant nous a en outre poussé à tenter de reproduire, au moins partiellement, nos expériences sur matériel réel. C'est le sujet de la section 5.3.7 suivante.

2. Rappelons que `rtimer` est le mécanisme de gestion de *timer* matériel en temps réel de Contiki OS, dont nous avons décrit les limitations en section 4.1.3 page 86 ; `hwtimer` est le mécanisme équivalent, plus performant, que nous avons utilisé sous RIOT OS. Cependant, comme nous l'avons vu en section 3.3.5 page 68, `hwtimer` a depuis été remplacé par le module `xtimer`.

5.3.7 Comparaison avec essais sur matériel

En nous basant sur les résultats obtenus dans la présente section 5.3, nous avons constaté que S-CoSenS avait l'avantage sur ContikiMAC concernant les critères de QdS (TRP et délais de transmission). Inversement, ContikiMAC est supérieur à S-CoSenS concernant les *duty cycles* (et donc la consommation d'énergie).

Les précédentes expériences ont bien sûr leurs limitations, la principale d'entre elles étant qu'il s'agit uniquement de simulations. Cela peut entraîner des inexactitudes et, à l'extrême, des conclusions erronées, comme nous le verrons ultérieurement en section 6.1 page 136.

Pour tenter de valider ces résultats de simulations, nous avons effectué quelques tests sur un ensemble de noeuds WSN430 d'IoT-LAB. IoT-LAB [FIT IoT-LAB, 2008] est un banc d'essai matériel permettant d'effectuer des tests à distance sur des milliers de *motes* de différents types ; nous en reparlerons plus largement au chapitre 6 page 135.

Les *motes* WSN430 que nous avons utilisées sont similaires aux TelosB / Sky-Motes (elles sont notamment basées sur le MSP430F1611), et non à la Zolertia Z1 (construite autour d'un MSP430F2617, cf. section 5.1 page 103) ; les conditions d'expérimentation ne sont donc pas strictement similaires, mais sont destinées à nous donner une idée générale de la validité de nos simulations.

Nous avons repris un PAN dont la structure est similaire à celle présentée en figure 5.1 page 106. Bien évidemment, il s'agit désormais dans ces nouveaux tests de matériel réel.

Nous nous sommes limités à deux jeux d'essais, consistant à tester l'implantation de S-CoSenS sous RIOT, avec deux durées de *subframe* :

— une durée de *subframe* égale à 125 ms, correspondant à la configuration de nos premiers jeux de simulation ;

— une durée de *subframe* égale à 100 ms — donc intermédiaire entre les deux premières valeurs simulées — pour essayer d'avoir une idée de l'évolution en allant vers des valeurs de *subframe* inférieures, comme pour les simulations.

Les résultats que nous avons obtenus avec nos expériences sur matériel réel pour les taux de réception de paquets sont présentés en figure 5.16 page suivante.

On voit que les courbes présentées sont tout à fait similaires à celles obtenues par simulation, telles que montrées dans la figure 5.7 page 113, et semblent donc valider globalement nos travaux et conclusions pour le TRP de notre protocole.

Concernant les délais de transmission, les résultats des expériences sur matériel sont visibles en figure 5.17 page suivante.

Notons ici que les délais, pour les simulations, étaient calculés très précisément grâce à la fonction de relevé temporel des trames circulant sur le média radio virtuel offerte par Cooja (fenêtre *"Radio messages"* et ses données exportables sous forme de fichier CSV). Les motes WSN430 d'IoT-LAB ne possédant pas de *"sniffers"* de trames, les délais obtenus par ces tests sur matériel sont calculés à partir de traces (*"logs"*) de déboguage émises par les noeuds utilisés lors de ces tests. Ces opérations

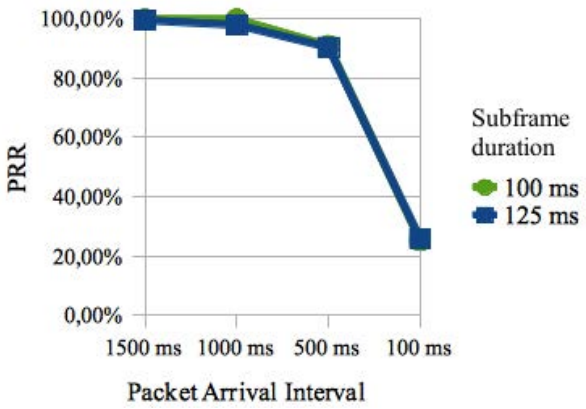

FIGURE 5.16 – *Résultats obtenus pour le taux de réception de paquets (TRP) avec S-CoSenS, en fonction de la durée de la subframe, sur des motes IoT-LAB WSN430.*

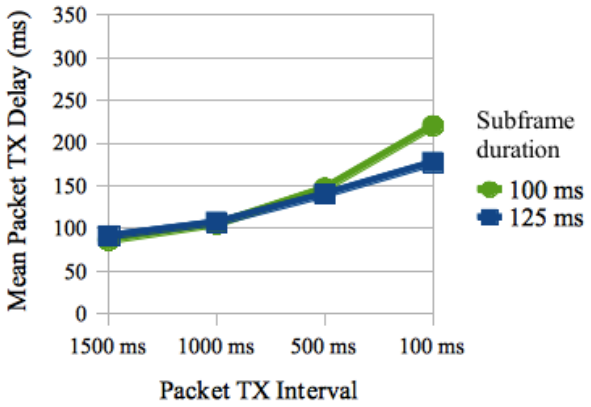

FIGURE 5.17 – *Résultats obtenus pour les délais moyens de transmission d'une trame avec S-CoSenS, en fonction de la durée de la subframe, sur des motes IoT-LAB WSN430.*

d'écriture de traces (fonctions de la famille `*printf()`) prenant elles-mêmes un certain temps d'exécution, cela représente un biais supplémentaire entre ces expériences sur matériel et nos simulations (en plus de la différence de microcontrôleur).

Cela dit, les courbes nous montrent là encore des valeurs du même ordre que celles obtenues par simulation en figure 5.12 page 116. Nos travaux et conclusions pour les délais de transmissions de notre protocole semblent donc eux aussi valides.

Nous n'avons par contre aucune donnée de *"duty cycle"* issue de ces expériences sur matériel à fournir, les motes WSN430 étant comme nous l'avons dit dépourvues de l'instrumentation adéquate — alors que Cooja fournit une fonction de calcul automatique de ces *"duty cycles"* via sa fenêtre *"Timeline"*. Nous ne pouvons donc valider ou invalider nos données simulées de *"duty cycle"* via les présents jeux d'expériences sur matériel.

5.4 Améliorations potentielles des protocoles MAC /RDC

5.4.1 Propositions d'améliorations algorithmiques

Là encore, il s'agit de trouver un équilibre entre les deux objectifs contradictoires que sont l'obtention d'une QdS maximale et de *duty cycles* minimaux. Par conception, ContikiMAC est clairement plus orienté vers le second objectif que S-CoSenS, même s'il est possible de diminuer la valeur du paramètre WP_{min} pour pousser notre protocole à se rapprocher du même objectif. Cela est toutefois contraire à notre politique consistant à donner la priorité absolue à la QdS.

Nous avons également, au cours de ces expériences, été amenés à réfléchir sur les possibilités d'amélioration et d'optimisation de notre protocole S-CoSenS, puis plus généralement de tout protocole MAC / RDC à haute performance :

1. En premier lieu, l'observation de ContikiMAC et de son fonctionnement nous a poussé à ***tenter d'implanter un mécanisme similaire à l'optimisation "phase lock" de ContikiMAC (discutée en section 3.2.2.4 page 39)*** afin d'obtenir des *duty cycles* inférieurs — c-à-d. meilleurs — pour les noeuds-feuilles fonctionnant sous S-CoSenS.

   Les résultats de nos essais de simulation avec une telle optimisation se sont révélés relativement concluants sous certaines conditions, comme expliqué dans la section 5.4.2 suivante.

   Plus qu'avec S-CoSenS, ***cette optimisation peut surtout être mise à profit pour améliorer tout protocole asynchrone***, pour ajuster au mieux les moments d'activation de la radio pour tenter d'émettre (protocoles LPL) ou recevoir (protocoles LPP) des trames, ***si le protocole en question possède une durée de cycle constante (ou au minimum prévisible).***

2. Une autre amélioration que nous avons imaginée pour S-CoSenS est la suivante : au lieu de compter le nombre de paquets retransmis par le routeur durant les cycles précédents pour évaluer le trafic radio et calculer la durée des périodes d'écoute WP (cf. section 3.2.6.1 page 47), nous souhaitons désormais nous servir du nombre de **SFD** [3] entendus durant les périodes d'écoute précédentes pour mieux évaluer le trafic réseau passant et donc calculer plus justement les durées des périodes d'écoute suivantes.

   Comme nous l'avons vu dans la section 3.2.6.1 page 47 de l'état de l'art, S-CoSenS (dans sa version actuelle) partage ses *"subframes"* de longueur fixe entre une période d'écoute WP et une période de sommeil SP. Le calcul de la durée de WP est actuellement effectué sur la base du nombre de trames reçues avec succès durant les périodes d'écoutes des cycles précédents (via une moyenne glissante).

3. **SFD : *"Start of Frame Delimiter"***; il s'agit d'un signal radio (de contenu constant, prédéfini par la couche physique du standard IEEE 802.15.4) envoyé sur le médium comme marqueur pour signaler le début de l'émission d'une trame, et permettre la synchronisation des récepteurs radio pour la bonne réception de cette nouvelle trame.

Ce mécanisme est efficace en théorie, mais ne tient pas compte du bruit. Un réseau sans-fil peut être parasité par de nombreux facteurs (interférences, collisions, ou même trames destinées à d'autres noeuds surtout si plusieurs PANs différents sont présents à proximité).

En présence d'un tel bruit, les trames destinées à un noeud N vont être « noyées » dans la masse de signaux sans intérêt pour ce noeud N. Par conséquent, avec le mode de calcul actuel, le noeud N, recevant peu de trames utiles, va diminuer rapidement sa période d'écoute WP (passant ainsi la majorité de ses cycles en période de sommeil SP). Il arrivera alors, dans un milieu fortement bruité, que le noeud N manque ainsi des trames lui étant destinées. C'est pour cela que nous avons dû fixer WP_{min} à 50% sur les routeurs S-CoSenS lors de nos tests (cf. section 5.3.4 page 117) afin d'assurer une QdS suffisante. (Nous donnons priorité absolue à la QdS : voir la conclusion en section 5.5 page 132).

Nous nous proposons, pour calculer la durée de WP, de ne plus compter le nombre de trames reçues avec succès, mais de compter le nombre de SFD reçus, c'est-à-dire le nombre de trames détectées sur le médium radio. Un tel mode de calcul permet de tenir compte de tout le trafic « normal » sans pour autant prendre en compte le « bruit brut » (interférences, signaux non conformes au protocole IEEE 802.15.4) ; cela devrait ainsi permettre à un noeud N d'éviter de manquer des trames lui étant destinées, pour cause de WP trop courte. Le défaut potentiel de cette technique est d'augmenter la durée de WP, donc mécaniquement de diminuer celle de SP, c'est-à-dire d'augmenter la consommation d'énergie du noeud, peut-être inutilement.

La contribution attendue de ces tests sur matériel réel est donc de ***comparer les deux modes de calcul de la durée de WP — nombre de trames reçues, et nombre de SFD détectés — et de déterminer par l'expérience lequel représente le meilleur compromis*** entre QdS (nombre de trames reçues ou manquées) et consommation d'énergie (durée optimale ou non de la période de sommeil SP).

Notons également que ***le recours aux SFD pour évaluer l'intensité du trafic réseau peut être mis à profit pour améliorer tout protocole ayant besoin de connaître cette intensité — ou même pouvant faire usage de cette information — pour adapter dynamiquement son fonctionnement***. ContikiMAC pourrait par exemple s'en servir pour faire varier de façon dynamique son *Channel Check Interval* en fonction du trafic, l'augmentation de la fréquence de ce paramètre permettant visiblement de mieux traiter les charges réseau intenses.

3. Nous avons également eu l'idée de combiner nos tests avec une ***évaluation de l'influence du seuil de réception de l'émetteur / récepteur radio (“CCA Threshold”)***, telle qu'étudiée en détail dans une publication de 2013 [Sha et al., 2013] — proposant un protocole nommé **AEDP** (*“Adaptive Energy Detection Protocol”*) permettant d'améliorer la fiabilité des liens réseaux face au bruit présent sur le médium radio —, pour une éventuelle inclusion de la gestion du *“CCA Threshold”* dans la pile réseau de notre plate-forme logicielle.

Le protocole AEDP décrit en [Sha et al., 2013] se penche sur le problème du bruit, et de l'activité inutile des émetteurs / récepteurs radio qui en découle.

Son principe est, pour résumer, d'adapter le seuil de sensibilité en dBm de la radio (*"CCA Threshold"*) en fonction de deux variables : le nombre d'envois nécessaires d'une trame donnée pour que celle-ci puisse être transmise avec succès (variable ETX), et le nombre d'activations de la radio nécessaires pour pouvoir détecter l'arrivée d'un signal utile (variable WR). Le protocole AEDP consiste à ajuster le *"CCA Threshold"* (ici nommé variable T) tout en gardant ETX en dessous d'une valeur maximale d'essais pour réussir à envoyer une trame (constante $ETX_{threshold}$) — ce qui implique de diminuer T —, ***et*** en évitant que WR ne dépasse une valeur maximale correspondant au nombre estimé nécessaire d'activations de la radio pour une réception correcte (constante $WR_{threshold}$) — ce qui implique de d'augmenter T.
Pour résumer, le but du protocole AEDP est de trouver la valeur minimale de T telle que $ETX < ETX_{threshold}$ et $WR < WR_{threshold}$. On voit qu'il s'agit, une fois de plus, d'un compromis entre QdS (taux de réception de trames et débit de données : ETX) et consommation minimale d'énergie (WR).

Ce protocole a au départ été conçu pour les protocoles LPL, dont le principe est rappelons-le de se réveiller à intervalles (plus ou moins) réguliers pour vérifier si une trame leur est destinée — cf. section 3.2.2 page 37 de l'état de l'art —, et a été testé par ses auteurs sur TinyOS et Contiki.

Bien que S-CoSenS ne repose pas principalement sur LPL, le principe consistant à jouer sur le *"CCA Threshold"* pour améliorer le rapport signal / bruit (**SNR** : *Signal/Noise Ratio*) est intéressant et peut être complémentaire, pour S-CoSenS, de l'essai de la méthode du comptage des SFD pour optimiser le rapport QdS / consommation d'énergie.

On peut raisonnablement penser que ***l'adaptation dynamique du "CCA Threshold" pour améliorer le rapport signal / bruit peut être mise à profit par n'importe quel protocole MAC / RDC — voire même plus directement au niveau des pilotes radio eux-mêmes — pour améliorer les performances et l'efficacité des transmissions sur les réseaux sans-fil***.

5.4.2 Évaluation de ces propositions : difficultés et inadéquation des simulations

Il est naturel, pour tenter une première évaluation des trois propositions d'améliorations algorithmiques détaillées dans la section 5.4.1 précédente, de songer à effectuer des simulations utilisant des scénarios adéquats (notamment avec le *framework* Cooja / MSPSim que nous avons déjà largement utilisé).

Toutefois, il s'avère que la mise en place de tels scénarios pour effectuer des simulations / émulations valides est problématique, pour les raisons suivantes :

1. Concernant ***l'implantation d'un mécanisme similaire à l'optimisation "phase lock" de ContikiMAC***, notre but est d'obtenir des données de *"duty*

cycle" afin de vérifier si des améliorations significatives — c.-à-d. des valeurs d'activité radio significativement inférieures — peuvent être obtenues. Or, comme nous l'avons évoqué en section 5.3.6 page 121, et comme nous l'étudierons en détail en section 6.1 au chapitre suivant, les résultats (notamment de *"duty cycle"*) obtenus à l'aide du *"framework"* Cooja / MSPSim manquent de fiabilité. ***Les conclusions sur les éventuelles améliorations obtenues par l'implantation du mécanisme de "phase lock" via une étude faite par simulations / émulations ne seront donc pas robustes, et pourront facilement être remises en question***.

2. Concernant ***l'étude de l'influence des SFD sur le calcul de la durée de la période d'écoute WP de S-CoSenS***, le matériel émulé par Cooja / MSPSim — à savoir les *motes* Sky / TelosB tout comme les Z1 — ne permet pas de recevoir l'interruption correspondant aux SFD (début de réception d'une trame) de façon simple : ces différentes *motes* relient en effet cette interruption à une entrée (*"capture"*) d'un des *timers* du microcontrôleur central. Il faudrait pour la détecter effectuer des modifications complexes et non portables au mécanisme de gestion des *timers* de RIOT, ce qui nuirait probablement à la fiabilité de ce dernier. Une étude mettant en place une version ainsi « altérée » de RIOT, d'une qualité insuffisante, n'aurait que peu d'intérêt car ne pouvant fournir une implantation robuste et pérenne. Il existe des *motes* permettant de recevoir et gérer facilement (du moins au niveau physique) les interruptions liées aux SFD, mais toutes celles que nous connaissons sont des plates-formes matérielles basées sur l'architecture ARM, pour laquelle aucun émulateur efficace n'est à l'heure actuelle reconnu comme fiable, disponible et compatible avec Cooja. ***Il nous est donc impossible d'émuler — du moins avec le "framework" Cooja / MSPSim que nous avons jusqu'ici utilisé — un matériel permettant de détecter les SFD de manière fiable et efficace.***
3. Concernant ***l'étude de l'influence du rapport signal / bruit (SNR) sur l'optimisation de la qualité de service et la consommation d'énergie***, une telle étude implique — par nature — le médium radio de façon centrale. Or, il est bien connu que la simulation du médium radio est le point faible de toutes les études par simulation des réseaux sans-fil (qu'il s'agisse des WSN comme des réseaux plus « classiques » comme le WiFi). En effet, comme nous l'avons vu notamment au chapitre 2, le médium radio est par nature peu fiable, sujet à la diffusion et présente des variations fortes et rapides — le plus souvent imprévisibles — en fonction du temps et de l'espace. ***Des études du SNR par simulation sont donc vouées à avoir une fiabilité, et donc un intérêt, extrêmement limités.***

Nous voyons donc que sur nos trois propositions d'améliorations algorithmiques, les deux dernières ne peuvent pas être étudiées de façon assez fiable par simulation / émulation, et nécessiteront donc des études sur des réseaux réels. Nous avons fourni une contribution mettant en place la fonctionnalité de gestion du *"CCA Threshold"* [Roussel et al., 2015a] (nécessaire par exemple au protocole AEDP), qui pourra ainsi faire l'objet de travaux ultérieurs idoines.

Concernant la première amélioration ("phase lock"), une étude par simulation, si elle n'offre pas une fiabilité satisfaisante, est néanmoins possible pour tenter de se faire une première idée grossière de l'efficacité d'une telle optimisation (qui demandera bien sûr par la suite confirmation par des tests sur matériel réel). Nous avons donc exécuté un jeu de simulations, dans cette optique bien limitée d'exploration préliminaire.

Nous avons repris le scénario d'émission de trames entre 10 noeuds-feuilles, un routeur et un *"sink"*, décrit et utilisé en section 5.3 page 109. Après avoir modifié notre implantation de S-CoSenS pour tenter de synchroniser les noeuds-feuilles sur l'émission des *beacons* du routeur, nous avons relancé une série de tests et obtenu les résultats de *"duty cycle"* montrés en table 5.5. La figure 5.18 résume l'influence constatée du *"phase lock"* au niveau de l'activité radio totale.

S-CoSenS avec *"phase lock"* : Durée d'un cycle (*subframe*) = 31 ms			
INTERVALLE D'ÉMISSION DES TRAMES	1500 ms	1000 ms	500 ms
NOEUDS-FEUILLES (moyenne)			
Activité radio (protocole standard)	6,15%	7,68%	40,04%
Activité radio (avec *"phase lock"*)	5,76%	7,03%	39,90%
Transmission efficace	0,58%	0,70%	2,64%
Réception efficace	0,68%	0,93%	7,08%
Interférences	0,16%	0,22%	3,98%

TABLE 5.5 – *Statistiques de duty cycle pour S-CoSenS avec optimisation de type "phase-lock" au niveau des noeuds-feuilles.*

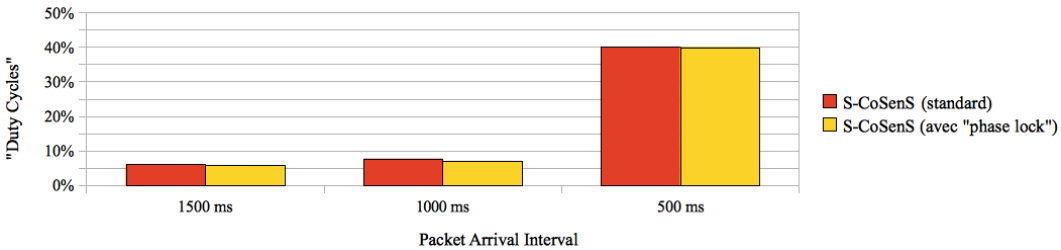

FIGURE 5.18 – *Influence du mécanisme de "phase lock" sur l'activité radio (duty cycle) induite par S-CoSenS, avec des subframes de 31 ms.*

On remarque que seules les données pour les noeuds-feuilles sont présentées : cette optimisation ne s'applique pas au routeur, du fait de la conception même du protocole S-CoSenS.

On voit également que les valeurs de « transmission efficace », « réception efficace », et « interférences » sont identiques à celles de la table 5.3 page 118. Notre optimisation tente en effet de réduire le temps d'écoute « à vide » des noeuds-feuilles, lors de l'attente de réception des *"beacons"*. Seules les valeurs de la ligne « activité radio » varient donc par rapport aux résultats du protocole S-CoSenS standard.

Ces valeurs « d'activité radio » totale sont effectivement moins élevées par rapport au protocole S-CoSenS standard, mais la baisse de ces valeurs reste relativement limitée, tout spécialement lors des forts trafics réseau : à un intervalle d'émission des trames de 500 ms, la baisse de valeur d'activité radio est à peine perceptible. Ceci est logique : dans une telle configuration, la majorité de l'activité radio a lieu lors de la période d'écoute du routeur (WP), et les nombreuses collisions sont responsables de l'activité radio intense nécessaire pour pouvoir réussir à faire passer les trames (nombreux réessais).

Avec des débits plus modérés (une trame toute les 1000 et 1500 ms), la baisse est plus sensible, et permet de se rapprocher des (et même dans un cas descendre sous les) valeurs d'activité radio constatées pour les noeuds-feuilles sous ContikiMAC. Néanmoins, la baisse de valeur d'activité radio due à la modification de S-CoSenS reste limitée : étant donné que S-CoSenS ne nécessite pas de répéter continuellement l'envoi des trames jusqu'à réception par le destinataire (contrairement à ContikiMAC), on peut être déçu de ne pas repasser systématiquement et surtout significativement sous les valeurs d'activité radio de ContikiMAC.

Nous avons retenté nos simulations, pour évaluer les *"duty cycles"* de ContikiMAC et S-CoSenS avec optimisation de type *"phase-lock"*, cette fois avec une durée de cycle plus longue. Les résultats sont visibles respectivement dans les tables 5.6 et 5.7, dans lesquelles nous n'avons reporté que les valeurs des lignes « Activité radio » des noeuds-feuilles, étant donné que nous avons vu que les autres valeurs n'étaient pas affectées par cette optimisation. (Rappelons que dans cette configuration — cycles de 125 ms —, S-CoSenS obtient des résultats largement meilleurs en termes de qualité de service que ContikiMAC, comme vu dans les sections 5.3.2 à 5.3.3 pages 110–114.)

Ces résultats sont également illustrés par la figure 5.19 page ci-contre.

ContikiMAC : *Channel Check Interval* = 8 Hz			
INTERVALLE D'ÉMISSION DES TRAMES	1500 ms	1000 ms	500 ms
NOEUDS-FEUILLES (moyenne)			
Activité radio	3,38%	3,87%	4,79%

TABLE 5.6 – *Statistiques de duty cycle pour ContikiMAC avec un cycle long : CCI de 8 Hz (valeur par défaut), soit un cycle de 125 ms.*

S-CoSenS : Durée d'un cycle (*subframe*) = 125 ms			
INTERVALLE D'ÉMISSION DES TRAMES	1500 ms	1000 ms	500 ms
NOEUDS-FEUILLES (moyenne)			
Activité radio (protocole standard)	9,02%	11,45%	27,37%
Activité radio (avec *"phase lock"*)	4,61%	5,82%	17,26%

TABLE 5.7 – *Statistiques de duty cycle pour S-CoSenS au niveau des noeuds-feuilles, avec une subframe longue de 125 ms.*

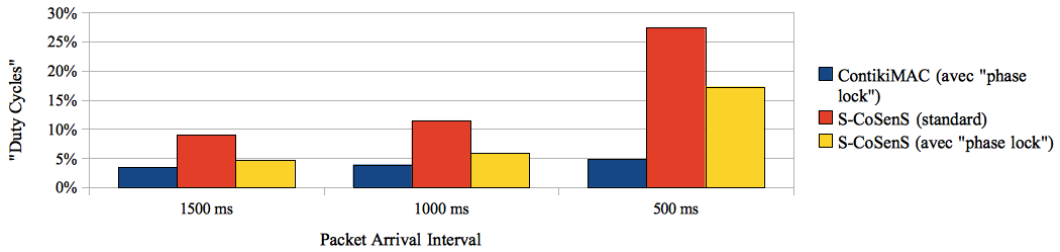

FIGURE 5.19 – *Influence du mécanisme de "phase lock" sur l'activité radio (duty cycle) induite par S-CoSenS et ContikiMAC, avec respectivement des subframes de 125 ms et des CCI de 8 Hz.*

On voit ici qu'avec une durée de *subframe* plus longue, le taux d'activité radio sous S-CoSenS baisse de façon significative avec le mécanisme de *"phase-lock"* : de l'ordre d'un tiers d'activité en moins avec un trafic réseau élevé (intervalle d'émission des trames de 500 ms), et une réduction de moitié de l'activité radio pour les charges réseau plus modérées (intervalle d'émission des trames égal ou supérieur à une seconde). Ce résultat est très intéressant, compte-tenu de la supériorité de S-CoSenS du point de vue de la QdS dans cette situation.

Par contre, on voit que là encore, même si la baisse du pourcentage d'activité radio devient importante au niveau de S-CoSenS avec une durée de *subframe* plus longue, ContikiMAC garde un *duty cycle* nettement inférieur.

Nous expliquons la relative importance des valeurs de *duty cycle* de S-CoSenS — par rapport à ContikiMAC — de la façon suivante : outre la *"subframe"* de longueur connue à l'avance, S-CoSenS intègre dans son cycle une phase de transmission dont la durée reste inconnue jusqu'à sa fin, puisque cette durée dépend du nombre de trames à transmettre et du temps nécessaire pour les transmettre (éventuelles interférences et collisions). Ainsi, S-CoSenS a toujours des cycles de durée variable, tandis que l'intervalle entre deux cycles — c.-à-d. deux CCA consécutifs — est toujours constant pour ContikiMAC quelles que soient les circonstances (revoir les sections 3.2.2.4 page 39 et 3.2.6.1 page 47 pour le détail du fonctionnement de ces deux protocoles).

Il est clair qu'un mécanisme du type *"phase-lock"*, qui retient la durée de cycle d'un récepteur pour optimiser les temps d'activation des émetteurs, fonctionnera bien mieux avec un protocole à durée de cycle constante comme ContikiMAC qu'avec un protocole auto-adaptatif comme S-CoSenS ayant des cycles de durée variable et difficilement prévisible.

Ainsi, nous pouvons penser que :

— ***l'implantation d'un mécanisme de type "phase-lock" se justifie pour S-CoSenS lorsqu'il est configuré avec une durée de cycle (i.e. : "subframe") suffisamment longue, là où les baisses d'activité radio sont significatives, et de surcroît l'avantage de S-CoSenS en termes de qualité de service est flagrant*** (revoir sections 5.3.2 à 5.3.3 pages 110–114) ;

— par contre, ***la complexité supplémentaire introduite par l'ajout d'un mécanisme de type "phase-lock" au sein d'un protocole à durée de cycle variable comme S-CoSenS ne se justifie guère lors de cycles de fonctionnement ("subframes") courts, surtout en présence de trafics réseau intenses, au vu des résultats forcément limités qu'apportera dans de telles conditions un tel mécanisme*** (à cause notamment de la grande variabilité des durées de cycles) ;
— ***un mécanisme de type "phase-lock" devrait être assez avantageux dans tous les cas pour envisager son implantation dans les protocoles MAC / RDC à durée de cycle fixe et / ou connue à l'avance***, à condition bien sûr que la conception du protocole le permette.

5.5 Discussion : évaluation et comparaison de protocoles MAC / RDC, contributions et conclusion

Nos travaux ont permis de tester et comparer deux implantations complètes et fonctionnelles de deux protocoles MAC / RDC, ContikiMAC et S-CoSenS. Ces travaux ont consisté en des scénarios simulant de lourdes charges réseau, dans le but d'observer la capacité de ces protocoles à monter en charge, puis à les pousser dans leurs limites ultimes de fonctionnement — tout en sachant qu'approcher la limite théorique de 250 Kbit/s de bande passante pour le médium radio physique défini par le standard IEEE 802.15.4 est impossible, à cause des délais imposés par le standard lui-même (SIFS, LIFS), des limites matérielles des émetteurs / récepteurs radio, et des délais occasionnés par les différentes couches des piles réseau (et notamment les couches MAC et RDC).

Nous avons, durant l'implantation de S-CoSenS sur RIOT OS, surmonté de nombreux problèmes dont nous avons discuté dans le présent chapitre.

Ce faisant, nous avons ***fourni les contributions suivantes*** :

— Nous avons d'abord ***prouvé que Cooja (le simulateur de WSN du projet Contiki) n'était nullement limité à cette plate-forme logicielle***. Nous l'avons employé avec succès pour tester des applications conçues sous RIOT OS, et ***nous sommes convaincus que celui-ci peut servir à la simulation de WSN fonctionnant avec n'importe quelle plate-forme logicielle, ou même sans (programmation "bare metal")***, et ce grâce au recours à des émulateurs pour la gestion des *motes* virtuelles.
— Nous avons montré que des ***fonctionnalités temps-réel sont nécessaires*** pour implanter des protocoles MAC / RDC novateurs et efficaces. Plus précisément : une résolution temporelle à fine granularité (nécessairement inférieure à la milliseconde, une résolution de l'ordre de 30 microsecondes étant tout à fait satisfaisante), et la disponibilité de plusieurs *timers* offrant cette fine granularité temporelle — l'usage d'au moins un *timer* matériel étant en général nécessaire pour atteindre un tel but.
— Nous avons montré que ***notre implantation du protocole S-CoSenS présente un avantage en termes de QdS*** (TRP et délais de transmissions)

particulièrement en présence d'un trafic réseau intense, tandis que ContikiMAC (dans son implantation standard sous Contiki) est largement supérieur en terme de *duty cycles*.

— Nous avons envisagé des ***contre-mesures pour gérer les erreurs liées à des problèmes de corruption mémoire*** (comme l'écrasement par la pile système d'autres données, à cause de l'espace mémoire très limité sur les *motes* de base). Notamment — sur les architectures matérielles dépourvues de circuit MPU dédié — la nécessité de surveiller régulièrement la pile système, ***de préférence par l'insertion automatique de code de vérification, durant le processus de compilation,*** à des endroits stratégiques du chemin de code (appels de sous-programmes, interruptions, etc.).

— Nous avons ***évalué l'influence des optimisations des implantations sur les performances***, notamment dans la gestion du bus SPI, selon le traitement des transmissions d'une façon sûre ou au contraire optimisée : le choix de l'une ou l'autre des solutions impacte la vitesse de chargement des trames dans l'émetteur / récepteur radio d'un facteur 2 à 3, cette accélération se faisant au détriment de la fiabilité. Notons toutefois que cette différence ne semble pas avoir changé significativement les résultats globaux de nos comparaisons.

— Nous avons enfin ***proposé des idées d'améliorations algorithmiques au protocole S-CoSenS*** (liées à l'optimisation des periodes de réveil de l'émetteur / récepteur radio, à l'estimation du trafic réseau et à l'optimisation du SNR), ***et à notre sens facilement généralisables à de nombreux protocoles MAC / RDC***, si ce n'est — pour la dernière de ces trois améliorations — à toutes les couches basses des piles réseau des plates-formes logicielles destinées aux WSN.

En conclusion, nous avons constaté qu'il est essentiel pour un protocole MAC / RDC d'être capable de traiter efficacement une lourde charge réseau (ou des pointes de trafic), tout en gardant un *duty cycle* minimal durant les périodes de faible trafic.

Ces deux objectifs contradictoires doivent faire l'objet d'un compromis, dont dépend le choix d'un protocole MAC / RDC et éventuellement la configuration de celui-ci.

Notre conviction est toutefois qu'un WSN est ***d'abord*** déployé pour communiquer des informations, et doit donc ***en priorité*** transmettre le trafic réseau quand celui-ci apparaît. L'économie d'énergie est le ***deuxième*** objectif principal, pour allonger la durée de fonctionnement et / ou de vie des noeuds du WSN, mais ne doit jamais — selon nous — se faire au détriment de la qualité de service « fonctionnelle ».

C'est pourquoi nous pensons qu'un protocole MAC / RDC idéal est un protocole dont le "duty cycle" doit varier dynamiquement en fonction du trafic, certes pour permettre d'optimiser la consommation d'énergie, mais jamais au prix de la perte de données à transmettre (sauf bien sûr dans les cas de saturation du médium radio).

Pour finir, rappelons que lors de nos tests concernant l'influence de l'optimisation des implantations sur les performances (section 5.3.6 page 121), nous avions observé des incohérences dans les valeurs obtenues entre simulation et exécution sur matériel

réel, tout particulièrement en ce qui concerne la plate-forme matérielle Zolertia Z1. Comme nous l'avions alors dit, nous allons maintenant étudier ce phénomène en détail dans la section 6.1 du chapitre suivant.

Chapitre 6

Validation des expérimentations sur plates-formes réelles

Le présent chapitre, dont le but est de valider les expériences effectués jusqu'ici dans le cadre de la présente thèse, notamment au précédent chapitre 5, va consister en deux parties logiquement liées et consécutives :

1. Comme nous l'avons signalé en fin de section 5.3.6 page 121, nous avons constaté des incohérences entre nos simulations / émulations effectuées sous Cooja, et des tests similaires effectués sur matériel réel (ce qui nous avait déjà poussé à faire quelques premiers tests limités en section 5.3.7 page 123).

   Ce phénomène d'incohérence va être étudié en détail dans la section 6.1 suivante. Nous y verrons l'ampleur du problème, ses causes probables, ses conséquences potentielles et, au final, nous devrons ainsi reconnaître les limitations rendant les simulations effectuées sous Cooja inadaptées aux travaux d'évaluation de performances — notamment liées aux temps et délais — de projets basés sur les WSN.

2. Partant de cette dernière conclusion, le reste de ce chapitre sera consacré à des travaux de validation, consistant à reproduire nos expérimentations simulées sur matériel réel, et plus précisément sur un *testbed* offrant un nombre suffisant de noeuds de WSN, dotés de fonctionnalités d'instrumentation suffisantes pour recueillir suffisamment facilement les données que Cooja permet d'obtenir sans effort. La section 6.2 page 148 décrira les travaux de validation ainsi prévus, puis la section 6.3 page 151 détaillera les problèmes techniques nous ayant rapidement empêché de mener à bien cette campagne de tests de validation, fournissant autant que possible de détails techniques dans l'espoir de permettre la poursuite de ces travaux et leur réussite dans un effort, qui sera forcément ultérieur à la présente thèse.

Ce chapitre sera enfin bien évidemment terminé par une section de discussion et conclusion : section 6.4 page 164.

6.1 Motivation : limitations et inexactitudes des simulations

Cooja, le simulateur de réseaux de capteurs sans-fil du projet Contiki (utilisant MSPSim pour l'émulation des noeuds MSP430 des réseaux), est un outil très largement utilisé pour le développement et le déboguage, mais aussi pour l'évaluation des performances des projets basés sur les WSN.

Comme nous l'avons dit en section 5.3.6 page 121, nous avons observé des incohérences temporelles entre simulations avec Cooja / MSPSim (dans leurs versions fournies avec Contiki 2.7) et des essais similaires effectués ensuite sur du matériel réel.

La table 6.1 reprend les données de la table 5.4 page 122, en montrant cette fois les différences constatées entre délais obtenus par simulation et délais obtenus par expérimentation sur matériel réel (les valeurs sont toutes données en *"ticks"* de *timers* matériels représentant environ 30,5 μs). Ces différences de délai, extrêmement élevées, n'ont *a priori* aucune raison rationnelle d'apparaître.

Système / Pilote SPI	Simu. Cooja	Test matériel
RIOT OS / Standard (Sûr)	89 ticks	32 ticks
RIOT OS / *"Fast SPI write"*	36 ticks	14 ticks
Contiki / *"Fast SPI write"*	14 ticks	7 ticks

Table 6.1 – *Delais observés par le chargement d'un paquet de 110 octets dans le buffer d'envoi de la radio CC2420 d'une mote Zolertia Z1, en fonction de l'OS et de l'implantation du pilote SPI, par simulation et par expérimentation sur matériel.*

De telles imprécisions pourraient remettre en cause l'utilisation de Cooja / MSPSim comme outil d'évaluation, du moins pour les paramètres liés aux performances temporelles [Roussel et al., 2016].

Dans ce chapitre, après un bref rappel sur Cooja et MSPSim, nous détaillerons les résultats de nos investigations, ainsi que les risques potentiels liés à ce problème, et nous tenterons de donner des pistes pour le régler ou du moins éviter ses conséquences.

6.1.1 Rappels sur Cooja et MSPSim

Fourni par le projet Contiki [Dunkels et al., 2004], le simulateur **Cooja** [Österlind et al., 2006] est devenu un outil très largement employé dans le domaine des réseaux de capteurs sans-fil (WSN). La communauté académique l'emploie notamment de façon intensive pour effectuer des simulations de réseaux allant de différentes tailles — de quelques noeuds formant un unique PAN comme dans nos expériences décrites en sections 5.2 page 105 et 5.3 page 109, à plusieurs centaines de *motes* divisées en de nombreux PANs interconnectés —, lesquelles simulations permettent de développer, déboguer et évaluer les performances des projets basés sur l'une des technologie des WSN.

Le recours à de telles simulations est tout spécialement utile pour effectuer des expérimentations sur des réseaux étendus et complexes, contenant de nombreuses *motes* ; réseaux qu'il serait long, difficile et coûteux de mettre en oeuvre avec du matériel réel.

Cooja est en lui-même une application écrite avec le langage et la plate-forme logicielle Java. Cette application offre trois fonctionnalités principales :

1. Une interface utilsateur graphique (GUI) — basée sur le jeu de composants graphiques standard de Java (*"Swing"*) — permettant de concevoir, d'exécuter et d'analyser des simulations de WSN.
2. La simulation du médium radio servant de support aux communications sans-fil des réseaux de *motes*.
   Plusieurs modèles sont possibles, du médium « idéal », au médium simulant (aléatoirement) des interférences et autres pertes de signal, en passant par un médium simulant uniquement un affaiblissement du signal proportionnel à la distance à l'émetteur (modèle par défaut). Tous ces modèles incluent pour chaque noeud la gestion d'une « portée utile » ainsi que d'une « zone d'interférences », les deux étant configurables par l'utilisateur.
3. Un *framework* logiciel extensible, permettant l'intégration d'outils externes amenant des fonctionnalités additionnelles à l'application Cooja.

C'est cette dernière fonctionnalité qui permet à Cooja d'émuler réellement les différentes *motes* constituant les WSN. En effet, Cooja embarque et utilise — grâce à ce mécanisme — des programmes dédiés capables d'effectuer l'émulation au cycle près (*"cycle-exact emulation"*) des circuits intégrés avec lesquels sont contruites les *motes* : microcontrôleurs (MCUs), émetteurs / récepteurs radio, etc.

Cette capacité d'émulation est l'un des principaux, sinon ***le*** principal point fort de Cooja : grâce à l'émulation des *motes*, des simulations extrêmement précises, de bas niveau et à granularité très fine sont possibles. C'est cet avantage décisif qui a sans aucun doute permis à Cooja de devenir l'outil de référence, tout spécialement pour le déboguage et l'évaluation des applications tournant sur des noeuds de réseaux de capteurs sans fil — applications tournant à l'heure actuelle très souvent sous Contiki OS. Mais l'émulation précise du matériel permet de faire tourner des programmes sous n'importe quelle plate-forme logicielle, comme le prouvent les tests sous RIOT OS que nous avons détaillés au chapitre 5.

Les versions actuelles de Cooja embarquent et utilisent deux émulateurs différents : **Avrora** [Titzer et al., 2005] pour l'émulation des appareils basés sur l'architecture Atmel AVR, et **MSPSim** [Eriksson et al., 2009] pour l'émulation des appareils basés sur l'architecture MSP430 de Texas Instruments.

De ces deux émulateurs, MSPSim est actuellement le plus utilisé dans les simulations Cooja décrites dans la littérature, étant donné que les *motes* basées sur des microcontrôleurs MSP430 sont les plus fréquemment rencontrées et employées : on citera notamment la famille TelosB / Skymote, ou la plate-forme Zolerta Z1, que nous avons décrites en section 5.1 page 103, et qui font partie des plates-formes matérielles supportées officiellement par Contiki OS.

Ces rappels étant faits, nous allons maintenant dans la prochaine section 6.1.2 détailler les inexactitudes que nous avons constatées, à travers un ensemble étendu de comparaisons entre simulations et expérimentations sur matériel réel.

6.1.2 Inexactitudes des délais sous MSPSim

En effectuant nos propres simulations avec des réseaux virtuels de *motes* basées sur la technologie MSP430 (comme celles décrites en sections 5.2 page 105 et 5.3 page 109), nous avons noté des valeurs incohérentes pour les délais en comparaison avec divers essais de transmissions faits sur du matériel réel. Plus précisément, nous avons constaté que nos simulations présentaient de façon inexpliquée des délais différents durant la transmission de paquets sur le médium radio (TX), par rapport aux durées constatées durant la transmission de paquets de même taille effectuée sur des *motes* physiques. Nous avons alors mené nos investigations sur ce problème, et fait les découvertes suivantes.

Le problème réside dans l'émulation d'appareils à microcontrôleur MSP430 équipés d'émetteurs / récepteurs radio (c'est-à-dire des *motes*) par le logiciel MSPSim.

Les différences de *timing* concernent une opération particulière : le chargement d'un paquet de données dans le buffer de transmission (TX) de l'émetteur / récepteur radio virtuel. MSPSim, lorsqu'il émule une *mote*, effectue ce chargement de *buffer* à une vitesse différente du matériel réel.

Par conséquent, nous avons écrit un programme simple de test, dont le seul rôle est d'envoyer des paquets de données de différentes tailles, lesquelles tailles sont choisies parmi :

— une taille ***modérée***, avec une charge utile (*"payload"*) de 30 octets ;
— une taille ***moyenne***, avec une charge utile de 60 octets ;
— une ***grande*** taille, avec une charge utile de 110 octets — c'est-à-dire proche de la taille maximale de 127 octets des paquets au standard IEEE 802.15.4.

Notons également que la taille des paquets physiques réellement transmis sur le médium radio comporte 11 octets supplémentaires (entêtes et somme de contrôle) en sus de leur charge utile.

Ce programme de test envoie consécutivement 50 paquets de la taille choisie, à la vitesse de 1 paquet par seconde, à chaque exécution. La valeur mesurée est la durée — ou « délai » — de chargement de chacun de ces paquets dans le *buffer* d'émission de l'émetteur / récepteur radio. Nous avons ainsi calculé la moyenne et l'écart-type pour tous les groupes de 50 paquets correspondant aux différentes configurations testées.

De plus, nous avons pour chaque configuration testée exécuté plusieurs fois notre programme, afin de nous assurer de la stabilité des résultats obtenus.

Ce programme a été compilé et exécuté sur les deux plates-formes suivantes (déjà décrites auparavant en section 5.1 page 103) :

— la famille bien connue des *motes* TelosB / SkyMote, bâtie autour d'un microcontrôleur MSP430F1611 ;
— la *mote* Zolertia Z1, plus récente, construite autour d'un microcontrôleur MSP430F2617.

Ces deux plates-formes utilisent le même émetteur / récepteur radio TI CC2420.

Ces valeurs sont données pour les deux systèmes orientés WSN que nous avons utilisés chapitre 5 : le système de référence Contiki OS, et le plus récent RIOT OS.

Comme nous l'avons déjà fait lors de nos simulations précédentes, deux types de pilotes pour la gestion du bus SPI ont également été étudiés, le choix du pilote influant largement sur les délais observés (revoir la section 5.3.6 page 121) :

— Un modèle de pilote SPI dit « sûr », attendant que chaque octet transmis soit validé par l'interface du bus SPI avant d'envoyer le suivant ; ce type de pilote est employé en standard sous RIOT OS.

— Un modèle de pilote SPI dit *"fast write"*, où un octet est écrit chaque fois que le registre d'envoi de l'interface du bus SPI est vide, sans attendre le résultat (réussite ou échec) pour l'octet précédemment transmis ; ce type de pilote est employé sous Contiki.

Pour compléter les résultats obtenus avec les pilotes SPI par défaut de chacun des deux systèmes, nous avons implanté à titre d'essai un pilote SPI *"fast write"* sous RIOT OS, nous donnant ainsi une configuration supplémentaire pour effectuer nos tests.

Les résultats des comparaisons entre l'exécution de notre programme, d'une part sur des *motes* simulées sous Cooja / MSPSim, et d'autre part sur du matériel réel, sont détaillés en table 6.2 page suivante pour les noeuds TelosB / SkyMote, et en table 6.3 page 141 pour les noeuds Zolertia Z1.

Il est à noter que dans ces deux tables, les valeurs des délais sont données en *"ticks"* de *timers* matériels s'incrémentant à une fréquence de 32768 Hz pour les deux plates-formes matérielles testées ici. Le *tick* correspond donc, dans ces tables, à une unité fixe de temps égale à environ 30,5 microsecondes.

Les résultats avec les configurations « standard » montrent que :

— Pour la plate-forme matérielle Z1, les résultats sont tout bonnement catastrophiques : la différence entre valeurs expérimentales et valeurs simulées représente une surestimation allant de 100% jusqu'à presque 200% du délai réel. Des valeurs tellement surestimées sont de toute évidence inutilisables dans le cadre de l'évaluation de performances.

— Pour les plates-formes TelosB/SkyMote, les différences sont largement moins importantes, mais restent toutefois au-dessus de 10% pour Contiki, et même 15% pour la configuration sous RIOT OS, ce qui n'est pas négligeable pour des travaux d'évaluations de performances précis. De plus, ces différences peuvent aller dans les deux directions — c'est à dire : une surestimation comme une sous-estimation des valeurs réelles —, ce qui rend l'inexactitude sur le *timing* des simulations assez difficile à prédire et estimer sans effectuer de comparaison avec des expérimentations sur du matériel réel.

Avec la configuration RIOT OS modifiée (*"fast SPI"*), nous pouvons voir que :

— Il n'y a absolument aucune amélioration pour la plate-forme Z1, les délais obtenus par simulation montrant toujours une surestimation d'environ 170% par rapport à la valeur réelle.

Résultats avec Contiki OS

Taille paquet	Simulation Cooja		Expérience matérielle		Différence moyenne		
	Moyenne	Écart-type	Moyenne	Écart-type			
Modérée	6.4 *ticks*	0.50 *ticks*	7.2 *ticks*	0.40 *ticks*	–0.8 *ticks*	(≈ 24 μsec.)	≈ 11% val. exp.
Moyenne	10.7 *ticks*	0.46 *ticks*	12.7 *ticks*	0.48 *ticks*	–2.0 *ticks*	(≈ 60 μsec.)	≈ 15% val. exp.
Grande	18.0 *ticks*	0.00 *ticks*	20.6 *ticks*	0.49 *ticks*	–2.6 *ticks*	(≈ 80 μsec.)	≈ 13% val. exp.

Résultats avec RIOT OS (pilote SPI standard « sûr »)

Taille paquet	Simulation Cooja		Expérience matérielle		Différence moyenne		
	Moyenne	Écart-type	Moyenne	Écart-type			
Modérée	58.0 *ticks*	0.00 *ticks*	50.3 *ticks*	0.46 *ticks*	7.7 *ticks*	(≈ 235 μsec.)	≈ 15% val. exp.
Moyenne	85.2 *ticks*	0.39 *ticks*	73.6 *ticks*	0.50 *ticks*	11.6 *ticks*	(≈ 355 μsec.)	≈ 16% val. exp.
Grande	131.2 *ticks*	0.39 *ticks*	111.5 *ticks*	0.51 *ticks*	19.7 *ticks*	(≈ 601 μsec.)	≈ 18% val. exp.

Résultats avec RIOT OS (pilote SPI *"fast writes"*)

Taille paquet	Simulation Cooja		Expérience matérielle		Différence moyenne		
	Moyenne	Écart-type	Moyenne	Écart-type			
Modérée	39.2 *ticks*	0.39 *ticks*	38.4 *ticks*	0.49 *ticks*	0.8 *ticks*	(≈ 24 μsec.)	≈ 2% val. exp.
Moyenne	53.2 *ticks*	0.39 *ticks*	52.8 *ticks*	0.40 *ticks*	0.4 *ticks*	(≈ 12 μsec.)	≈ 1% val. exp.
Grande	76.2 *ticks*	0.39 *ticks*	75.2 *ticks*	0.39 *ticks*	1.0 *ticks*	(≈ 31 μsec.)	≈ 1% val. exp.

TABLE 6.2 – *Delais observés pour le chargement d'un paquet dans le buffer d'envoi du CC2420 d'une mote SkyMote/TelosB, avec différentes configurations logicielles.*

Résultats avec Contiki OS

Taille paquet	Simulation Cooja		Expérience matérielle		Différence moyenne		
	Moyenne	Écart-type	Moyenne	Écart-type			
Modérée	5.0 *ticks*	0.14 *ticks*	2.3 *ticks*	0.44 *ticks*	2.8 *ticks*	($\approx$ 84 μsec.)	$\approx$ 122% val. exp.
Moyenne	8.9 *ticks*	0.27 *ticks*	4.2 *ticks*	0.37 *ticks*	4.8 *ticks*	($\approx$ 145 μsec.)	$\approx$ 114% val. exp.
Grande	14.0 *ticks*	0.14 *ticks*	7.2 *ticks*	0.39 *ticks*	6.8 *ticks*	($\approx$ 209 μsec.)	$\approx$ 95% val. exp.

Résultats avec RIOT OS (pilote SPI standard « sûr »)

Taille paquet	Simulation Cooja		Expérience matérielle		Différence moyenne		
	Moyenne	Écart-type	Moyenne	Écart-type			
Modérée	46.0 *ticks*	0.00 *ticks*	16.2 *ticks*	0.39 *ticks*	29.8 *ticks*	($\approx$ 910 μsec.)	$\approx$ 184% val. exp.
Moyenne	69.0 *ticks*	0.00 *ticks*	24.2 *ticks*	0.39 *ticks*	44.8 *ticks*	($\approx$ 1368 μsec.)	$\approx$ 185% val. exp.
Grande	106.8 *ticks*	0.39 *ticks*	38.0 *ticks*	0.00 *ticks*	68.8 *ticks*	($\approx$ 2100 μsec.)	$\approx$ 181% val. exp.

Résultats avec RIOT OS (pilote SPI *"fast writes"*)

Taille paquet	Simulation Cooja		Expérience matérielle		Différence moyenne		
	Moyenne	Écart-type	Moyenne	Écart-type			
Modérée	27.0 *ticks*	0.00 *ticks*	10.0 *ticks*	0.00 *ticks*	17.0 *ticks*	($\approx$ 519 μsec.)	$\approx$ 170% val. exp.
Moyenne	35.0 *ticks*	0.00 *ticks*	13.2 *ticks*	0.39 *ticks*	21.8 *ticks*	($\approx$ 665 μsec.)	$\approx$ 166% val. exp.
Grande	49.0 *ticks*	0.00 *ticks*	18.2 *ticks*	0.39 *ticks*	30.8 *ticks*	($\approx$ 941 μsec.)	$\approx$ 170% val. exp.

TABLE 6.3 – *Delais observés pour le chargement d'un paquet dans le buffer d'envoi du CC2420 d'une mote Zolertia Z1, avec différentes configurations logicielles.*

— Pour les plates-formes TelosB/SkyMote, la situation s'améliore, aboutissant à l'obtention de résultats de simulation relativement corrects (2% ou moins d'inexactitude).

Ceci nous donne des pistes quant à la cause de ce problème de délais incorrectement estimés : puisqu'il dépend largement de la plate-forme matérielle simulée mais ***aussi*** de la méthode utilisée pour écrire sur le bus SPI, il n'est probablement pas (principalement) dû à l'émulation de l'émetteur / récepteur radio CC2420, mais plutôt à l'estimation du temps pris par les transferts sur le bus SPI, estimation faite au niveau de l'émulation du microcontrôleur. Il est évident que l'émulation du MSP430F2617 (le MCU autour duquel est bâtie la Zolertia Z1) surestime très largement les délais liés au bus SPI, sans doute à cause d'une mauvaise calibration, tandis que l'émulation du MSP430F1611 (celui des TelosB / SkyMotes) est nettement mieux calibrée de ce point de vue.

Nous voyons donc que, quel que soit l'environnement logiciel utilisé, nous pouvons émettre les observations suivantes concernant l'inexactitude des délais simulés pour l'opération de chargement du *buffer* d'envoi de la radio :

— Pour la plate-forme matérielle Zolertia Z1, l'erreur est tout simplement énorme, le délai de chargement simulé est toujours 2 à 3 fois supérieur au délai réel sur matériel, quelle que soit la configuration logicielle (OS et pilote SPI) employée.

— Pour la plate-forme matérielle TelosB / SkyMote, l'inexactitude est largement moins importante en valeur absolue, mais plus difficile à prédire, puisque pouvant aller dans les deux sens : sous-estimation ou surestimation du délai réel sur matériel. En outre, cette inexactitude peut aller jusqu'à 15% de la valeur réelle sous Contiki OS (et même au-delà sous RIOT OS), ce qui n'est pas négligeable pour des travaux d'évaluation nécessitant une grande précision.

Tout ceci rend les résultats temporels obtenus avec les simulations Cooja / MSP-Sim inexploitables par manque de fiabilité pour des travaux d'évaluations de performances, tout particulièrement sur Zolertia Z1, mais également dans une moindre mesure sur la famille TelosB / SkyMote, deux plates-formes d'expérimentation très populaires auprès de la communauté scientifique spécialisée dans le domaine des réseaux de capteurs sans-fil.

Le temps de chargement d'un paquet dans le *buffer* d'envoi de la radio n'étant qu'une étape dans le processus complet d'envoi d'un paquet de données — l'autre partie étant l'émission proprement dite des signaux correspondant au paquet sur le médium par l'émetteur radio — nous avons voulu estimer le poids relatif des inexactitudes que nous avons constatées jusqu'ici dans le temps total employé pour les émissions radio.

En calculant le poids relatif du chargement du *buffer* d'envoi dans la durée totale de transmission d'un paquet de données, nous avons constaté que ce délai de chargement représente toujours au moins 10% — et peut monter jusqu'à plus de 50% — de la durée totale d'envoi d'un paquet, tout spécialement en exécutant un OS différent de Contiki. (Nous n'avons dans ces tests d'importance relative des durées utilisé que les pilotes standard de chaque OS, c'est-à-dire seulement le pilote SPI « sûr », plus

lent, sous RIOT OS). Les résultats détaillés de nos estimations de poids relatif des durées de chaque étape dans la transmission d'un paquet sont disponibles en table 6.4 page suivante.

Il est à noter que, dans cette table 6.4, les délais d'émission (TX) sont calculés en utilisant le débit indiqué par le standard IEEE 802.15.4 (250 Kbit/s, soit 32 μsecondes par octet), en tenant compte des 11 octets d'entête et somme de contrôle systématiquement rajoutés à la charge utile du paquet.

De telles inexactitudes dans l'évaluation des durées de transmission sont de toute évidence condamnées à avoir des conséquences sérieuses sur la fiabilité des évaluations de performances faites via des simulations sous Cooja / MSPSim.

En outre, puisque le problème nous semble lié à l'émulation des microcontrôleurs effectuée par le logiciel MSPSim, nous nous attendons à ce que la vaste majorité des *motes* et autres cartes d'évaluation basées sur des microcontrôleurs d'architecture MSP430, lorsqu'elles sont émulées par la version de Cooja / MSPSim fournie avec Contiki 2.7 (dernière *release* stable disponible du système au moment où nous écrivons ces lignes), soient également impactées — à des degrés plus ou moins intenses — par ce problème, visiblement dépendant de la calibration temporelle de l'émulateur effectuée pour chaque microcontrôleur différent.

En conséquence, nous pensons que les utilisateurs de tels appareils devraient sérieusement envisager d'effectuer leurs propres tests de comparaison avec du matériel réel — lorsque cela est possible — pour évaluer les possibles inexactitudes frappant leurs simulations et l'impact éventuel de ces inexactitudes sur leur travaux.

6.1.3 Risques et conséquences potentiels

Nous avons fait des recherches dans la littérature consacrée aux réseaux de capteurs sans-fil, et nous avons trouvé de nombreuses publications récentes — c'est-à-dire : publiées en 2014 ou ultérieurement — basées, directement ou indirectement, sur des simulations Cooja / MSPSim pour évaluer les performances temporelles de projets liés à des WSN.

Parmi ces publications, nous pouvons constater que la plupart d'entre elles utilisent des noeuds virtuels TelosB / SkyMote comme [Marques and Ricardo, 2014], [Gaddour et al., 2014], [Amoretti et al., 2014], [Djamaa et al., 2014], [Ancillotti et al., 2014a], [de Oliveira et al., 2014], [Ancillotti et al., 2014b] et [Felemban et al., 2014]. D'autres utilisent des *motes* basées sur des microcontrôleurs MSP430 différents — par exemple : la *mote* EXP5438 pour [Seo et al., 2015], ou la WisMote pour [Antonini et al., 2014] — microntrôleurs et *motes* dont la sensibilité au problème d'inexactitude discuté ici nous est totalement inconnue.

Nous avons également remarqué une publication basée à la fois sur des simulations ***et*** des résultats expérimentaux sur matériel réel : il s'agit de [Wu et al., 2015], laquelle présente en outre des résultats purements numériques produits avec MATLAB. Nous supposons que de tels articles, (beaucoup) plus rares, sont naturellement largement moins exposés à des erreurs liées aux inexactitudes de simulation.

Quasiment toutes les publications que nous avons vues dans cette section sont

P.-F. matérielle	Système	Taille paquet	Chargement	Émission (TX)	Délai total	Chargement / Total
SkyMote/TelosB	Contiki	Modérée	196 μsec.	1312 μsec.	1508 μsec.	13%
SkyMote/TelosB	Contiki	Moyenne	327 μsec.	2272 μsec.	2599 μsec.	13%
SkyMote/TelosB	Contiki	Grande	549 μsec.	3872 μsec.	4421 μsec.	12%
SkyMote/TelosB	RIOT OS	Modérée	1770 μsec.	1312 μsec.	3082 μsec.	57%
SkyMote/TelosB	RIOT OS	Moyenne	2599 μsec.	2272 μsec.	4871 μsec.	53%
SkyMote/TelosB	RIOT OS	Grande	4003 μsec.	3872 μsec.	7875 μsec.	51%
Zolertia Z1	Contiki	Modérée	153 μsec.	1312 μsec.	1465 μsec.	10%
Zolertia Z1	Contiki	Moyenne	272 μsec.	2272 μsec.	2544 μsec.	11%
Zolertia Z1	Contiki	Grande	428 μsec.	3872 μsec.	4300 μsec.	10%
Zolertia Z1	RIOT OS	Modérée	1404 μsec.	1312 μsec.	2716 μsec.	52%
Zolertia Z1	RIOT OS	Medium	2106 μsec.	2272 μsec.	4378 μsec.	48%
Zolertia Z1	RIOT OS	Grande	3260 μsec.	3872 μsec.	7132 μsec.	46%

TABLE 6.4 – *Poids relatif du chargement du buffer d'envoi de l'émetteur / récepteur radio dans le temps de transmission d'un paquet.*

liées aux couches hautes de la pile réseau des capteurs sans-fil, notamment aux protocoles de routage — comme [Marques and Ricardo, 2014], [Gaddour et al., 2014], [Ancillotti et al., 2014a] et [Ancillotti et al., 2014b] — ou à des protocoles de niveau applicatif — par exemple : [Amoretti et al., 2014], [Djamaa et al., 2014], [Felemban et al., 2014] et [Seo et al., 2015]. Certains articles présentent même des piles réseau alternatives : c'est le cas de [de Oliveira et al., 2014].

Impact sur nos propres simulations

Les plupart des résultats de nos expérimentations antérieures, concernant l'implantation de S-CoSenS sous RIOT OS que nous avons présentée au chapitre 5 du présent manuscrit, ont été obtenus par des simulations Cooja / MSPSim, qui plus est sur la plate-forme matérielle Zolertia Z1, qui comme nous venons de le voir est la plus gravement touchée par ces problèmes de mauvaise calibration et d'inexactitude des délais. Dans ces conditions, il est légitime et même indispensable de s'interroger sur la validité des conclusions que nous avons tirées de ces travaux.

Le seul moyen de valider ou d'invalider de façon définitive ces résultats serait (comme nous l'avons signalé plus haut) de refaire ces tests sur du matériel réel. Nous n'avons pas eu la possibilité de refaire de façon exhaustive tous ces tests sur le matériel adéquat — des *motes* Z1 réelles, qui plus est instrumentées pour obtenir les résultats voulus avec assez de précision.

Nous allons donc tenter, par des raisonnements logiques, d'évaluer la fiabilité des principales conclusions que nous avons émises dans la section 5.3 page 109 :

Taux de Réception de Paquets (TRP / PRR). Les résultats de nos simulations montrent un avantage constant de S-CoSenS / RIOT par rapport à ContikiMAC. Nous voyons au regard des résultats des tables 6.2 page 140 et 6.3 page 141 que les différences entre simulations et expériences sur matériel réel restent modérées sur plate-forme TelosB / SkyMote, tandis que ces différences sont très élevées sur plate-forme Z1, ces différences étant toujours en défaveur de S-CoSenS / RIOT. On peut donc raisonnablement en conclure qu'une correction de ces inexactitudes ne ferait que renforcer l'avantage de S-CoSenS /RIOT par rapport à ContikiMAC / Contiki en matière de TRP.

Délais de transmissions de paquets de bout-en-bout. Nous pouvons ici encore suivre un raisonnement similaire : une correction de ces inexactitudes ne ferait que renforcer l'avantage de S-CoSenS /RIOT par rapport à ContikiMAC / Contiki en matière de délais de transmission.

En résumé, concernant les critères de Qualité de Service (QdS), incluant taux de réception des paquets et délais de transmission, nous pensons que nos conclusions ne sont pas invalidées par le problème posé par MSPSim. Les quelques tests effectués sur du matériel (certes légèrement différent) en section 5.3.7 page 123 tendent à conforter notre confiance dans cette partie de nos travaux.

Consommation énergétique *"Duty Cycles"*. Les simulations ont ici montré un indiscutable avantage de ContikiMAC sur S-CoSenS / RIOT. Nous ignorons

totalement si les surestimations constatées en table 6.3 page 141 seraient de nature à changer la conclusion. Notons que ContikiMAC est un protocole LPL explicitement conçu pour garder un *duty cycle* minimal, surtout dans sa configuration par défaut.

En matière de consommation énergétique ("duty cycles"), nous sommes incapables de déduire l'impact des inexactitudes occasionnées par MSPSim sans expérimentations adéquates sur matériel pour trancher. Nos conclusions en matière de duty cycles ne peuvent pour le moment ni être validées ni être rejetées. Nous avons ici atteint les limites du « pouvoir prédictif » des simulations en matière de fiabilité et de robustesse.

Stabilité et contraintes mémoire. Ces conclusions ne sont en rien liées à un facteur temporel, et ne sauraient donc être remises en cause par ces inexactitudes de simulation.

Influence de l'optimisation des implantations. Nous avons vu dans le présent chapitre, dans les tables 6.2 page 140 et 6.3 page 141, que l'implantation du pilote SPI influait bien sur la rapidité de transmission — le pilote « sûr » étant plus lent que le pilote *"fast write"* dans tous les cas. Les inexactitudes de MSPSim ne remettent pas en cause cette conclusion.

Expériences sur matériel. Par définition, ces données ne peuvent être influencées par un quelconque simulateur / émulateur.

Limitations et améliorations potentielles. Les conclusions d'ordre général sur le compromis nécessaire entre QdS et *"duty cycle"*, ainsi que la possibilité d'optimiser S-CoSenS en y implantant un mécanisme comparable au *"phase lock"* de ContikiMAC, ne peuvent en aucune façon être invalidées par des questions d'implantation telles que le problème lié à MSPSim.

Les autres conclusions présentées dans les sections 5.3.5 page 120 « Stabilité et contraintes mémoire », 5.3.6 page 121 « Influence de l'optimisation des implantations », 5.3.7 page 123 « Expériences sur matériel » et 5.4 page 125 « Améliorations potentielles des protocoles MAC / RDC » ne peuvent nullement être remises en cause par le problème d'inexactitudes de MSPSim.

Au final, nous pensons que malgré les inexactitudes temporelles dues à l'émulation du bus SPI par MSPSim, les conclusions que nous avons tirées de nos travaux présentés en section 5.3 page 109 restent exactes, à l'exception du paragraphe sur les *"duty cycles"* dont nous ne pouvons garantir la validité sans nouveaux tests sur du matériel réel proposant des fonctions d'instrumentation adéquates.

6.1.4 Discussion : limites des simulations et émulations

Les tests effectués lors des travaux de la présente section prouvent de façon claire que l'émulation des *motes* basées sur MSP430 souffre, dans le *framework* Cooja / MSPSim d'inexactitudes concernant l'évaluation d'au moins certains délais, remttant

en cause au moins partiellement la fiabilité des travaux effectués avec les simulations issues l'utilsation de cet outil. Ce problème d'inexactitudes est particulièrement intense concernant la plate-forme Zolertia Z1, moins concernant la famille TelosB / SkyMote, et est susceptible de toucher tout autre matériel de type MSP430 émulé par MSPSim. Nous pensons que ce problème est du à une mauvaise calibration des délais d'émulation de certaines opérations (tout spécialement le chargement de données dans le *buffer* d'envoi de l'émetteur / récepteur radio) dans la programmation de l'émulateur MSPSim.

Nous avons brièvement énuméré une liste de plusieurs publications récentes dans le domaine des réseaux de capteurs sans-fil dont les résultats sont potentiellement impactés négativement par ce problème, ces publications reposant (au moins en partie) sur des simulations Cooja / MSPSim pour évaluer leur travail. La validité de telles publications peut être remise en cause par le problème que nous venons de décrire, notamment quand des résultats liés au temps obtenus par simulation sont utilisés.

Cette remise en cause de la fiabilité des résultats de simulation touche évidemment de plein fouet les travaux antérieurs que nous avons décrits dans le présent manuscrit au chapitre 5, et nous avons tenté d'analyser logiquement dans quelle mesure la validité de nos conclusions était potentiellement faussée par ce problème d'inexactitudes temporelles.

Jusqu'à ce qu'une correction soit effectuée et publiée pour régler ce problème de *timing* dans l'émulateur MSPSim, nous pensons que le seul moyen d'obtenir des résultats fiables concernant les évaluations liées au temps est de procéder à des tests sur du matériel réel, ce qui éliminera tout biais potentiel pouvant être introduit par des erreurs ou des inexactitudes dues aux simulations et émulations.

Bien entendu, de tels tests matériels sont bien plus difficiles, longs et coûteux à mettre en place, exécuter et analyser. C'est ce qui nous a empêché de faire de tels tests matériels exhaustifs pour valider complètement les résultats des simulations que nous avons détaillées en section 5.3 page 109 du présent manuscrit de thèse. Quelques tests limités sur matériel ont malgré tout été effectués en section 5.3.7 page 123, et semblent partiellement valider nos simulations du point de vue des critères de qualité de service.

Notons néanmoins que de nombreuses publications dans le domaine des réseaux de capteurs sans-fil, y compris des publications très récentes, font usage de simulations faites avec le *framework* Cooja / MSPSim. Cela est une preuve éclatante de la grande utilité de ce logiciel. C'est pourquoi corriger le problème que nous décrivons dans le présent chapitre dès que possible est, nous le croyons, d'une très grande importance.

Insistons également sur le fait que si le problème décrit ici remet en cause l'utilité de Cooja / MSPSim en tant qu'outil d'évaluation de performances — tant que les inexactitudes dans les calibrations des délais ne sont pas corrigées —, cela ne nuit en rien aux autres applications de cet outil, notamment la possibilité de développer et déboguer bien plus facilement des logiciels destinés à fonctionner sur des réseaux

de capteurs sans-fil, grâce aux fonctionnalités avancées d'émulation de ce *framework* logiciel. Ceci a été clairement démontré à la section 4.2.2.3 page 91 du présent manuscrit, où son apport s'est révélé crucial pour la compréhension et la résolution de plantages récurrents de RIOT qui rendaient ce système instable sur architecture MSP430.

En outre, nous pouvons rappeler que toute simulation d'un réseau sans-fil est, par nature, « condamnée » à présenter des inexactitudes vis-à-vis de la réalité, en particulier à cause de la simulation du médium radio et de la diffusion des ondes via celui-ci — car il s'agit là d'un phénomène physique complexe, ayant notamment la particularité de subir des variations spatiales et temporelles nombreuses et quasiment imprévisibles, et par conséquent extrêmement difficile à modéliser correctement et efficacement.

Cela renforce encore l'utilité, et même la nécessité, de recourir à des tests sur matériel réel pour valider le bon fonctionnement d'un projet basé sur une technologie de communication sans-fil — comme les WSN. Nous allons donc maintenant tout naturellement porter notre attention sur notre propre campagne de tests sur matériel, dans la prochaine section 6.2.

6.2 Validation sur matériel : moyens et objectifs

Comme expliqué en introduction du présent chapitre, nous abordons maintenant la deuxième moitié « logique » de ce chapitre : les (tentatives de) tests de validation sur matériel, travaux qui se sont logiquement imposés à nous compte tenu de la conclusion de la précédente section 6.1. En effet, nous venons de constater que les résultats obtenus par simulation n'offrent pas toujours une fiabilité suffisante pour effectuer des travaux d'évaluations de performances suffisamment précis et robustes.

La présente section va donc (comme dit précédemment) décrire les tests que nous avons conçus et prévus afin de pallier le problème de manque de fiabilité des simulations, et valider nos précédents travaux.

6.2.1 Besoins en instrumentation

Pour pouvoir recueillir les données nécessaires à l'évaluation des performances des protocoles testés, nous avons besoin :

- de recueillir les sorties console (`"*printf()"`) issues des noeuds, si possible avec horodatage et identifiant du noeud émetteur pour chaque message ;
- de connaître précisément les moments ou les trames 802.15.4 sont émises sur le médium radio, ***et*** reçues de ce même médium par les émetteurs / récepteurs des différentes *motes* impliquées durant une expérience ;
- le contenu des trames échangées sur le medium radio ;
- la consommation énergétique exacte des différents noeuds au cours de l'expérience.

Au cours du chapitre 5, le *framework* Cooja / MSPSim utilisé pour nos simulations nous fournissait les trois premiers éléments sans effort — grâce à ses fenêtres

"Mote output" et *"Radio Messages"* — mais était lui-même incapable de nous fournir le quatrième élément, autrement que via le marqueur, très imparfait, des *"duty cycle"* de chaque noeud émulé.

Lors des premières comparaisons avec des essais sur matériel réel (en section 5.3.7 page 123), le matériel employé — des motes IoT-LAB de type WSN430 (décrites en section 6.2.2 suivante) — n'offrait pas l'instrumentation adéquate pour les trois derniers éléments, comme nous l'avions alors signalé dans la section sus-citée.

On peut donc déjà constater que trouver un matériel disposant de toutes les fonctionnalités d'instrumentation requises n'a rien de trivial.

6.2.2 Choix de plate-forme matérielle

Ne disposant pas dans notre équipe du matériel adéquat, et surtout offrant ne serait-ce que d'une partie des fonctionnalités d'instrumentation souhaitées, nous avons donc cherché à utiliser un banc d'essai matériel disposant d'un nombre suffisant de *motes* instrumentées et prêtes à l'emploi pour effectuer nos tests.

Nous nous sommes ainsi tournés vers la plate-forme IoT-LAB [FIT IoT-LAB, 2008], offrant des milliers de motes de différents types, disponible pour mener des tests à distance, et ouverte à tous les expérimentateurs, académiques ou industriels. Nous avons participé au *workshop* d'inauguration de la branche grenobloise cette plate-forme, durant lequel nous avons effectué une présentation [Roussel, 2014].

IoT-LAB offre plusieurs types de *motes* disponibles pour mener des expériences, dont deux sont susceptibles de nous intéresser :

Les noeuds WSN430 (v1.4) dont les caractéristiques sont similaires à celles des *motes* TelosB/SkyMote (se reporter à la section 5.1 page 103 pour un détail de ces caractéristiques). Celles-ci ne disposent en outre que de carctéristiques d'instrumentation basiques, insuffisantes pour nos besoins comme nous l'avons déjà vu.

Les noeuds M3 : beaucoup plus performants que les *motes* sur lesquelles nous avons travaillé jusqu'ici, ils sont basés sur les composants suivants :

- un microcontrôleur STM32F103REY [DataSheet STM32F103RE, 2011], basé sur une architecture ARM 32 bits (Cortex-M3), cadencé à 72 MHz, comportant 512 Ko de mémoire Flash (programme) et 64 Ko de RAM (données),
- un émetteur / récepteur radio Atmel AT86RF231 [DataSheet AT86RF231, 2009],
- divers capteurs (lumière, pression atmosphérique, température, accéléromètre, magnétomètre et gyroscope 3D),
- une mémoire Flash externe au microcontrôleur (16 Mo),
- toute une architecture d'instrumentation, permettant notamment de tracer les sorties consoles, les messages radio émis et reçus, ainsi que l'énergie consommée par chaque noeud, le tout avec horodatage (ce qui correspond à nos besoins en instrumentation exprimée en section 6.2.1 précédente),

— et une antenne intégrée.

Les performances très largement supérieures de ces noeuds M3, ainsi que la possibilité — théorique — de gérer plus facilement les interruptions en provenance de l'émetteur / récepteur radio (ce qui faciliterait de futures expériences destinées à tester nos idées d'amélioration algorithmiques décrites en section 5.4 page 125), nous ont poussé à choisir ce type de noeuds — à la place du matériel basé sur l'architecture MSP430 que nous utilisions jusqu'alors — comme plate-forme matérielle pour effectuer nos expérimentations suivantes.

6.2.3 Travaux de validations prévus et contributions attendues

Pour étayer et compléter les précédentes expériences de cette thèse, reposant principalement sur des simulations / émulations, nous avons prévu plusieurs travaux expérimentaux de validation à réaliser sur matériel réel :

1. Nous voulons d'abord grâce à ces tests pouvoir ***valider notre implantation de S-CoSenS***, implantation déjà décrite et testée par simulation dans la section 5.2 page 105 du présent manuscrit.
2. Nous souhaitons ensuite pouvoir ***confirmer (ou infirmer) de façon claire et indiscutable les conclusions que nous avons tirées des simulations en section 5.3 page 109, notamment dans nos comparaisons avec ContikiMAC***.

   Pour ces deux premiers tests, nous réutilisons naturellement la configuration utilisée lors de la simulation, dont la structure est montrée en figure 5.1 page 106. Bien évidemment, ce PAN n'est ici plus virtuel, mais composé de matériel bien réel.

   (Rappelons que si nous étions confiants sur la supériorité de notre implantation en matière de critères de QdS — taux de réception de paquets, et délais de transmission —, les imprécisions dans les évaluations temporelles du simulateur Cooja / MSPSim ne nous permettaient pas de nous prononcer sur la question des *"duty cycles"*, c'est à dire de la consommation énergétique[1] imputable aux couches MAC.)
3. Nous désirons ensuite effectuer des ***tests de grande envergure, impliquant plusieurs PANs ayant chacun leur routeur, lesdits routeurs jouant précisément leur rôle de relai des trames transmises vers une destination finale (le "sink")***. La topologie de l'expérimentation étendue est représentée figure 6.1 page suivante : comme on peut le voir, elle comporte 40 noeuds-feuilles, 5 routeurs et enfin le *"sink"*, ce qui en fait un réseau de taille et de complexité élevée, comparable — par exemple — à la totalité de l'installation que l'on pourrait retrouver dans une application de domotique comme « l'appartement intelligent » du LORIA / INRIA Nancy Grand-Est [LORIA / INRIA Nancy Grand-Est, 2009].

1. Revoir les section 2.5.2.2 page 22 et 5.3.4 page 117 pour des explications sur le lien entre consommation énergétique et *"duty cycle"*.

La contribution attendue est ici ***d'évaluer les performances de nos protocoles MAC / RDC— S-CoSenS, mais aussi iQueue-MAC — dans des conditions de complexité comparables à celles d'un logement « intelligent » réel,*** richement équipé en capteurs sans-fil à vocations diverses (domotique, assistance à la personne, etc.)

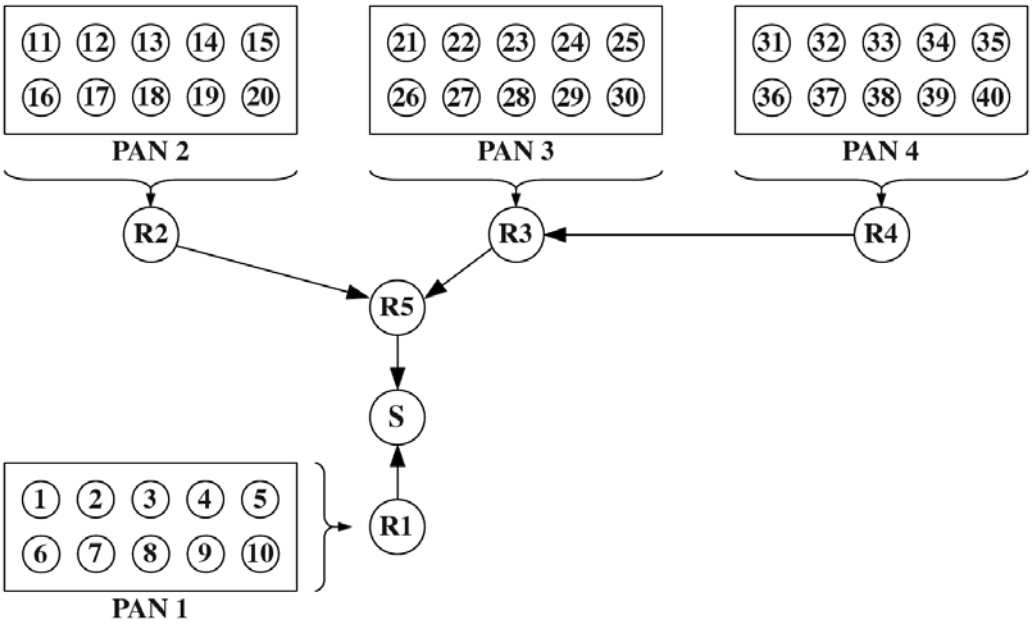

FIGURE 6.1 – *Schéma fonctionnel de notre seconde configuration de test. (Cette topologie est reprise de la publication [Zhuo et al., 2013] en figure 15.)*

Concernant le choix des plates-formes, nous avons, comme dit plus haut, trouvé une opportunité intéressante dans le choix des noeuds IoT-LAB M3 comme nouvelle plate-forme matérielle, notamment pour les performances nettement supérieures, ainsi que la possibilité de réagir (via des interruptions) à la détection de SFD par la puce radio, et surtout les fonctionnalités d'instrumentation avancées (consommation énergétique, *"sniffer"* de trames radio).

Pour la plate-forme logicielle, nous avons tout naturellement porté notre choix sur RIOT OS, auquel nous avons déjà participé de façon active, et qui nous a donné satisfaction dans nos précédents tests par simulation.

6.3 Travaux et mise en œuvre

6.3.1 Amélioration du pilote radio : proposition d'API étendue

Afin de commencer nos travaux d'expérimentation sur matériel décrits dans la section 6.2.3 précédente, il nous a d'abord fallu nous occuper des couches basses de notre plate-forme logicielle, RIOT OS.

Notre premier travail a concerné le pilote RIOT pour l'émetteur / récepteur radio des noeuds IoT-LAB M3 — qui pour rappel n'est pas le TI CC2420 comme dans les matériels que nous avions utilisé auparavant, mais l'Atmel AT86RF231, de conception totalement différente.

Le pilote présent dans RIOT OS au début des présents travaux avec les noeuds M3 était minimaliste : il ne permettait que d'envoyer et recevoir des trames, changer d'adresses et de canal. Il ne permettait ni d'allumer ni d'éteindre la puce radio, ni de traiter la plupart des exceptions lancées par cette dernière, et plusieurs bogues empêchaient le fonctionnement correct lors de l'appel à certains modes avancés (comme l'acquittement automatique des trames reçues). Nous avons donc dû procéder à des travaux de déboguage et de complétion du pilote de l'époque.

De plus, nous y avons ajouté les fonctionnalités nécessaires aux améliorations algorithmiques dont nous avons parlé par ailleurs dans la section 5.4 page 125, à savoir : le comptage des SFD détectés, et la capacité de connaître et définir le seuil (en dBm) à partir duquel l'émetteur / réceptur radio distingue bruit et signal radio utile (en anglais : *"CCA Threshold"*) ; le tout en prévision de futures expériences de recherche exploratoire, une fois les travaux de validation présentés dans la section 6.2.3 précédente accomplis.

Nous avons pour cela utilisé l'API de « capacités » dont nous avons déjà donné le principe en section 4.1.2.1 page 81. Elle se compose dans le cas présent de trois fonctions supplémentaires dans l'API du pilote radio, détaillées en table 6.5.

Nom	**Rôle**
`get_config_const()`	Lit la valeur d'une constante liée à la configuration d'une radio donnée ; il s'agit en général d'une limite dictée par le matériel (par exemple : valeur maximale de la puissance d'émission en dBm, ou canaux / fréquences radio accessibles)
`get_config_param()`	Lit la valeur courante d'une option de configuration d'un pilote radio donné (par exemple : nombre de SFD comptés, ou sensibilité radio en dBm)
`set_config_param()`	Définit la valeur courante d'une option de configuration d'un pilote radio donné

TABLE 6.5 – *Liste des fonctions de l'API complémentaires de « capacités ».*

Nous avons pour nos travaux rajouté seulement deux paramètres, listés en table 6.6 page suivante. Notons qu'un nombre arbitraire de constantes et de paramètres aurait parfaitement pu être rajouté selon les besoins ; nous nous sommes toutefois limités à ce moment-là au minimum d'ajouts pour gagner du temps.

Ces fonctions renvoient la valeur voulue (pour les deux fonctions `get_*`), ou une valeur d'erreur parmi celles citées en table 6.7 page ci-contre.

Une fois le pilote RIOT OS pour l'émetteur / récepteur radio des noeuds IoT-LAB M3 prêt, nous avons commencé nos expériences. Ce pilote n'est par contre pas

Nom	Signification
PARAM_SFD_COUNT	Nombre de SFD comptés par le récepteur radio depuis son démarrage, ou le dernier *reset* ; ce paramètre est en effet remis à zéro pour toute écriture (la valeur passée à `set_config_param()` est en effet ignorée)
PARAM_CCA_THRESHOLD	Seuil de sensibilité de la radio en dBm, alias *"CCA Threshold"*; la puce radio considère qu'en dessous de cette intensité, le médium est libre (bruit résiduel) ; au-delà de cette valeur, le canal radio est considéré comme occupé par un signal

TABLE 6.6 – *Liste des paramètres rajoutés par l'API complémentaires de « capacités ».*

Nom	Signification
ERR_UNKNOWN_CONST	Constante inconnue / sans signification (par exemple : CONST_WIND_SENSE)
ERR_UNKNOWN_PARAM	Paramètre inconnu / sans signification (par exemple : PARAM_CAPTAIN_AGE)
ERR_UNAVAILABLE_CONST	Constante non supportée par la radio voulue (par exemple : CONST_MIN_CHANNEL, constante indiquant le canal minimal disponible, sur une radio conçue pour une fréquence unique)
ERR_UNAVAILABLE_PARAM	Paramètre non supporté par la radio voulue (par exemple : PARAM_CHANNEL, paramètre de choix du canal, sur une radio conçue pour une fréquence unique)
ERR_READ_ONLY_PARAM	Tentative d'écriture d'un paramètre en lecture seule (par exemple : PARAM_CURRENT_ENERGY, qui représenterait l'énergie courante sur le médium radio en dBm)
ERR_WRITE_ONLY_PARAM	Tentative de lecture d'un paramètre en écriture seule (par exemple : PARAM_PERFORM_RESET, paramètre « fictif » qui réinitialiserait l'émetteur / recepteur radio)

TABLE 6.7 – *Liste des erreurs pouvant être retournées par les fonctions de l'API complémentaires de « capacités ».*

compatible avec la nouvelle pile réseau RIOT qui a été présentée dans un chapitre précédent en section 4.3 page 95 (celle-ci utilise toutefois déjà un mécanisme équivalent, comme nous l'avons vu). Certaines des idées implantées dans le présent pilote sont en outre susceptibles, comme nous l'avons également mentionné dans la section sus-citée, d'être rajoutées (contribuées) à l'API de cette nouvelle pile « gnrc ».

Nous avons d'abord réédité notre test de synchronisations entre noeuds du PAN (cf. section 5.2 page 105), qui n'a posé aucun problème : la précision des synchro-

nisations entre *motes* est toujours excellente (au niveau de quelques dizaines de microsecondes près). Le premier des travaux décrits en section 6.2.3 page 150 est donc une réussite.

Nous avons donc ensuite tenté de rééditer notre expérience d'évaluation de performances (cf. section 5.3 page 109), le second de nos travaux prévus. De nombreux problèmes techniques complexes et imprévus ont alors interrompu l'avancée de nos travaux.

Nous allons lister ces problèmes nous ayant empêché d'effectuer les autres travaux prévus en section 6.2.3 page 150, expliquer la cause de ceux que nous avons pu résoudre, et donner les éléments que nous possédons pour tenter de permettre, à l'avenir, la résolution de ceux qui restent présents.

6.3.2 Problèmes techniques rencontrés

6.3.2.1 Remarques techniques préliminaires

Pour permettre de bien comprendre la description des problèmes que nous allons énumérer, nous allons donner ici quelques détails techniques d'importance concernant le noeud IoT-LAB M3 — et surtout ses principaux composants : le MCU, l'émetteur / récepteur radio, et leur connexion.

Le MCU STM32F103REY. Plusieurs détails techniques sont à retenir dans le cas qui nous intéresse :

— Ce microcontrôleur, bien qu'étant d'architecture ARM, ne possède ***pas*** de MPU (unité de protection mémoire matérielle) : l'architecture Cortex-M3 considère en effet la MPU comme une option, que n'a pas retenue ST-Microelectronics pour ce modèle.

— Comme tous les Cortex-M3, ce MCU possède un NVIC (*Nested Vectored Interrupt Controller*), un contrôleur d'interruptions avancé permettant de choisir la priorité des interruptions, le niveau de masquage, et bien d'autres paramètres.

— Ce MCU possède de nombreuses GPIO, dont certaines assument également (selon le mode de configuration) des fonctions annexes : on citera notamment les liaisons SPI, I^2C, et des UART.

— Rappelons enfin son espace mémoire : 512 Ko de mémoire Flash pour le programme (*"firmware"*) et 64 Ko de RAM pour les données, ce qui est très confortable.

La radio AT86RF231. Du point de vue du programmeur, cet émetteur / récepteur radio est une machine à états finis, chaque état représentant une « fonction » de la radio. Le graphe des états de cette radio est représenté en figure 6.2 page ci-contre.

Le passage d'un état à l'autre se fait en écrivant dans un registre de contrôle de la radio (`TRX_STATE`), l'état courant étant lisible dans un registre d'état (`TRX_STATUS`). Il s'agit, avec le registre indiquant les interruptions radio levées (`IRQ_STATUS`), des trois principaux registres avec lesquels travaille le programmeur, l'ensemble (assez

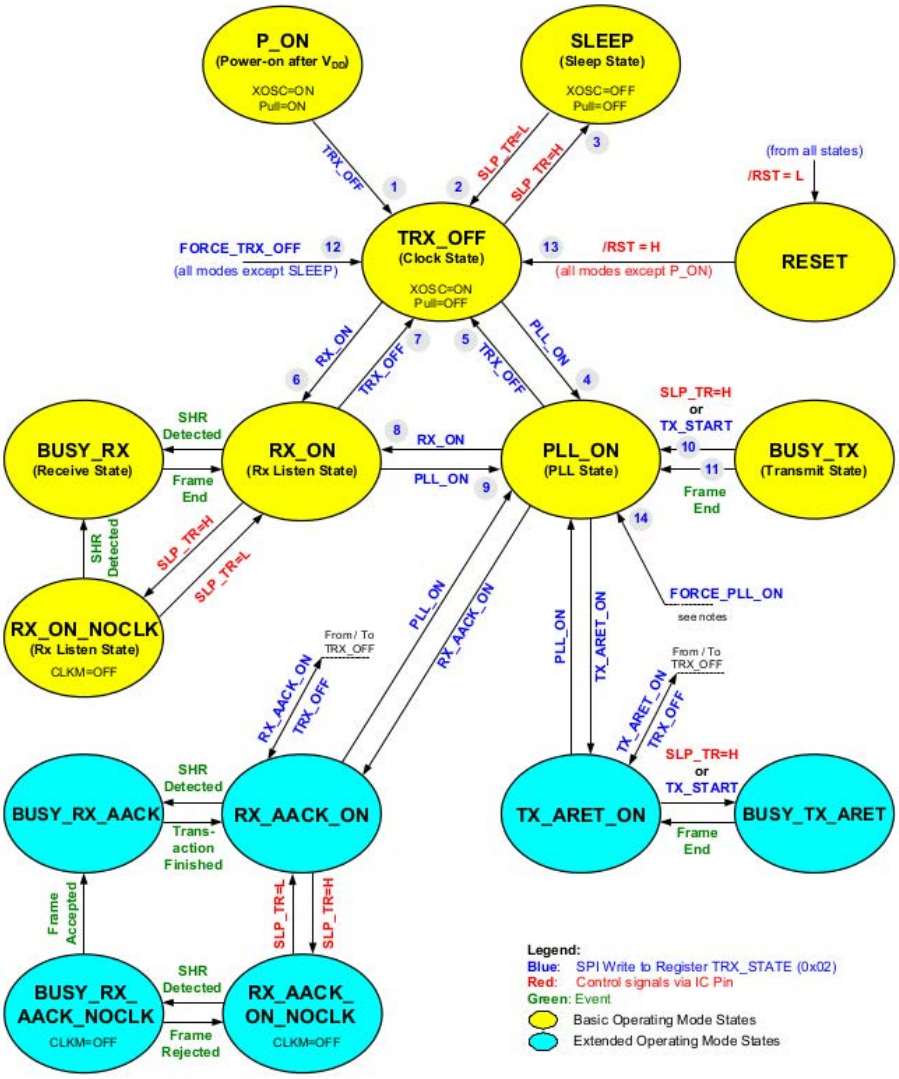

FIGURE 6.2 – *Diagramme complet de la machine à états finis de l'AT86RF231. (Source : [DataSheet AT86RF231, 2009] figure 7-8.)*

conséquent) des autres registres servant à la configuration plus « stable » de la radio (adresse(s), puissance d'émission, etc.) et étant donc relativement peu lus ou modifiés.

Pour nos travaux, nous avons exploité le mode dit « étendu » de cette radio, offrant la gestion automatique de la procédure CSMA / CA pour l'émission (mode `TX_ARET`) et l'acquittement automatique des trames reçus (mode `RX_AACK`). Ces modes sont colorés en bleu dans la figure 6.2.

Notons également que cette radio peut être mise en mode « sommeil » (`SLEEP`) à très basse consommation, mais que la conception du noeud IoT-LAB M3 ne permet

pas la mise hors-tension totale (*"off"*) de cette puce.

Le mode `TRX_OFF` est lui un état où la puce est active, mais n'accède pas au médium radio ni en émission ni en réception (mode *"idle"*).

Limitation importante à signaler : l'AT86RF231 ne possède qu'un ***unique "buffer" pour l'émission et la réception***, comme indiqué dans le chapitre 4 de [DataSheet AT86RF231, 2009] (contrairement à d'autres radios comme la TI CC2420). Des écrasements de données sont donc en théorie possibles si une trame arrive alors qu'une trame à émettre est en cours de chargement dans le *"buffer"*; la radio peut dans ce cas lancer une interruption d'alerte (`TRX_UR`, voir ci-dessous).

La connexion entre MCU et radio. Le lien entre ces deux circuits principaux du noeud IoT-LAB M3 est schématisé en figure 6.3.

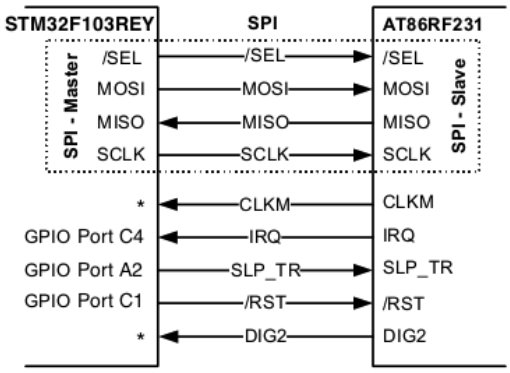

FIGURE 6.3 – *Schéma logique de l'interface entre MCU et AT86RF231 sur une mote IoT-LAB M3. (D'après [DataSheet AT86RF231, 2009] figure 6-1.)*

Les points importants à retenir de cette liaison sont :

— Toutes les communications avec la radio, c'est à dire la lecture et l'écriture de ses registres, pour configurer et surtout passer d'un état à l'autre en fonction des besoins, se fait par la liaison SPI.
  Pour le MCU, il s'agit de la première liaison SPI (correspondant aux broches 4 à 7 du port GPIO A en mode « fonction alternative », alias AFIO).
  Cette liaison SPI est entourée en pointillés dans la figure 6.3, fonctionne de façon standard, et dans notre cas, est gérée par le pilote générique dédié de RIOT OS.

— Deux broches GPIO en sortie (du point de vue du MCU) permettent de déclencher des opérations spéciales :
  > La broche `RST` (active à l'état bas), permet de réinitialiser la radio, par exemple en cas de problème. Elle ramène la puce dans l'état `RESET`, qui transitionne ensuite normalement vers l'état `TRX_OFF` automatiquement (cf. figure 6.2 page précédente).
  > Pour le MCU, il s'agit de la broche 1 du port GPIO C.

> La broche SLP_TR possède deux fonctions, en fonction de l'état courant de la radio :

1. En mode TRX_OFF, passer SLP_TR à l'état haut met la radio en mode sommeil (SLEEP) ; dans ce mode sommeil, passer SLP_TR à l'état bas réactive la radio et la ramène en mode TRX_OFF.
2. En mode PLL_ON ou TX_ARET_ON passer SLP_TR à l'état haut déclenche l'émission sur le canal radio du contenu du *"buffer"*.

Pour le MCU, il s'agit de la broche 2 du port GPIO A.

— Une broche GPIO en entrée (du point de vue du MCU), IRQ, active à l'état haut, permet de signaler au microcontrôleur que la radio a déclenché une interruption. La broche ne permet pas de savoir de quel type d'interruption il s'agit : il est nécessaire de lire (via la liaison SPI) un registre (IRQ_STATUS) pour connaître la ou les interruption(s) déclenchée(s). Notons qu'un autre registre (IRQ_MASK) permet d'activer ou désactiver (masquer) le déclenchement de chaque type d'interruption. Les types d'interruptions émissibles par l'AT86RF231 sont listées dans la table 6.8.
Pour le MCU, il s'agit de la broche 4 du port GPIO C.

— Nous n'utilisons pas les autres broches de l'interface (CLKM, DIG2) ; leur état est toujours ignoré.

Identifiant	**No. bit**	**Description**
BAT_LOW	7	Indique une chute de tension dans l'alimentation de la radio
TRX_UR	6	Indique une violation d'accès (écrasement) du *"buffer"*
AMI	5	Si le filtrage d'adresses sources est activé, indique qu'une trame provenant d'une source autorisée à été détecté
CCA_ED_DONE	4	Si la radio était en mode sommeil ou reset, indique son arrivée en mode TRX_OFF, c-à-d. sa disponibilité ; sinon, indique qu'une procédure de CCA (*"Clear Channel Assessment"*) est achevée
TRX_END	3	Indique soit la fin de la réception d'une trame (disponible pour analyse dans le *"buffer"*), soit la fin de la transmission de la trame présente dans le *"buffer"*
RX_START	2	Indique le début de la réception d'une trame (c-à-d. qu'un SFD a été détecté)
PLL_UNLOCK	1	Indique le déverrouillage de la PLL (condition anormale)
PLL_LOCK	0	Indique le verrouillage de la PLL (c-à-d. que la radio est prête à accéder au médium radio)

TABLE 6.8 – *Types d'interruptions de l'AT86RF231. (Source : [DataSheet AT86RF231, 2009] table 6-9.)*

6.3.2.2 Détail des problèmes rencontrés

— Des plantages dûs au débordement de la pile système ont eu lieu, exactement de la même façon que sur les Zolertia Z1, comme nous l'avons expliqué dans la section 5.3.5 page 120. La grande quantité de RAM présente dans ce MCU Cortex-M3 nous a permis de facilement contourner le problème en augmentant la taille de la pile à une valeur très élevée (10 Ko). On peut toutefois regretter l'absence de MPU dans ce microcontrôleur, étant donné que l'architecture ARM permet d'implanter un tel mécanisme, qui permet de prévenir efficacement ce type de problèmes.

— La figure 6.2 page 155 montre que l'émetteur / récepteur radio AT86RF231 n'autorise qu'un ensemble bien défini de transitions autorisées (représentées par des flèches sur la figure sus-citée). En outre, certaines de ces transitions ne se font pas de façon immédiate, mais nécessitent un certain délai d'exécution, durant lequel il ne faut pas tenter de communiquer avec la puce radio via la liaison SPI, par exemple pour consulter un registre. Ces délais sont (tout comme l'ensemble des transitions autorisées) précisés dans [DataSheet AT86RF231, 2009].
Par contre, ***la même [DataSheet AT86RF231, 2009] ne signale pas explicitement que le non-respect de ces règles peut provoquer la perte du contenu des registres de l'émetteur / récepteur radio, plongeant ce dernier dans un état de blocage dont seul un RESET peut le sortir***. Cela n'est pas le cas d'autres émetteurs / récepteurs radio, lequels en cas de mauvaise utilisation signalent une erreur, ou ignorent « silencieusement » la manipulation incorrecte, tout en restant capables de continuer à fonctionner correctement.
La version initiale (préexistante) du pilote que nous avons dû déboguer effectuait des transitions « interdites », ou interagissait avec la puce radio sans respecter les délais d'attente prescrits ; nous avons donc fait face à des blocages fréquents de la puce radio, jusqu'à ce que nous ayons réparé ces erreurs.
Nos corrections ont rendu ce problème beaucoup plus rare. Toutefois, nous continuons à subir de temps en temps ce problème de « remise à zéro » de la puce radio, sans explication logique à l'heure actuelle.

— Lors de l'émission de trames, il arrive parfois que la puce radio n'arrive plus à « terminer » sa transmission, c'est-à-dire : revenir de l'état `BUSY_TX_ARET` à l'état `TX_ARET_ON`, changement d'état normalement automatique une fois l'émission terminée. Or, l'état `BUSY_TX_ARET` correspond à l'émission d'une trame en respectant la procédure CSMA / CA[3], laquelle comporte un nombre maximal d'essais et des *"timeouts"* pour éviter de telles impasses. Nous ne nous expliquons donc pas la raison de ces blocages... Un RESET de la puce radio est nécessaire pour sortir de ces situations, qui ne sont heureusement pas systématiques, mais très gênantes, d'autant plus qu'elles sont actuellement incompréhensibles.

3. Revoir dans l'état de l'art la section 3.1.2.1 page 30 et la figure 3.3 page 32 si nécessaire.

— Le problème le plus gênant, car survenant lui ***systématiquement***, est le fait que nos noeuds deviennent subitement « sourds » : plus précisément, de façon imprévisible, et apparemment aléatoire, ils deviennent incapables de recevoir des trames. Ce problème peut apparaître très tôt durant les expériences, ou survenir après de longues minutes d'expérimentation. Le MCU n'est pourtant pas planté, car les interruptions temporelles (du *"timer"* matériel) gérant la génération des trames se produisent toujours.
Nous avons découvert que l'interruption chargée de gérer les évènements en provenance de la puce radio — y compris donc les réceptions de trames — ne se déclenchait plus, causant cette « surdité ».

Nous remarquons que :

> On note une tendance à avoir des noeuds qui fonctionnent mieux que les autres, mais la raison en est inconnue.
La disposition physique du PAN de noeuds M3 que nous avons utilisé (correspondant à la structure logique décrite en 5.1 page 106) est présentée en figure 6.4 page suivante, les résultats correspondant à nos différents essais étant décrits par un échantillon représentatif montré en table 6.9 page suivante.
Cette table montre, comme l'indique sa légende, le nombre de trames envoyées avec succès, sur 500 générés au total, par chaque noeud-feuille de ce premier PAN de test composé de noeuds IoT-LAB M3.
Au vu de la faible cadence d'envoi — 1 trame / noeud / seconde — il ne devrait y avoir quasiment aucun échec.
Ces données montrent que la distance entre noeuds et routeur n'a aucune influence sur ce manque de fiabilité : par exemple, le noeud **224** réussit toujours ses envois, et fait donc mieux que le noeud **225** parfois défaillant alors que ce dernier est le plus proche du routeur. Même remarque pour les noeuds **216**, **217** et **218**, le premier (le plus éloigné du routeur) arrivant par intermittence à réussir ses envois de 500 trames, alors que les deux autres, plus proches, échouent de façon systématique.

> Aucune amélioration notable n'est détectable en changeant de matériel (c-à-d. de groupe de noeuds pour constituer le PAN de test).

> Ce problème survient après un délai apparemment aléatoire (il peut survenir immédiatement, ou après l'envoi des trois-quarts des trames prévus).

> Le blocage des interruptions radio rencontré n'est pas dû à une non-sortie (*"deadlock"*) du gestionnaire d'interruptions. Des tests avec affichage à l'entrée et à la sortie de ce gestionnaire d'interruptions radio ont été effectués : aucune entrée sans sortie n'a jamais été repérée.

> Il n'est pas provoqué par un « chevauchement » d'interruptions : des tests avec blocage des autres interruptions pendant le traitement de l'interruption en cours ont été effectués : aucune amélioration constatée.

> La table des vecteurs d'interruptions est en mémoire Flash : il est donc impossible que le problème soit lié à un écrasement de pointeurs dû par exemple à un débordement de pile (cette dernière piste ayant en outre déjà

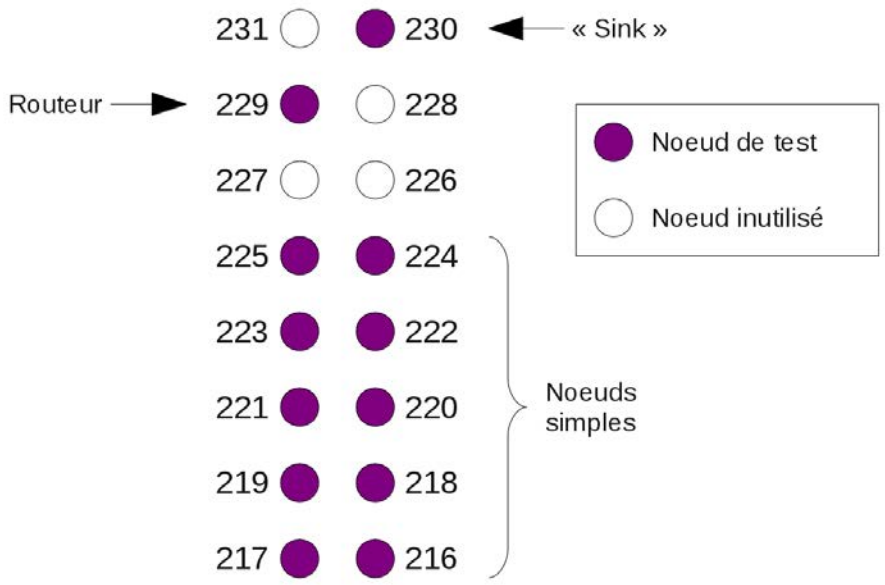

FIGURE 6.4 – *Disposition physique du PAN de test employé sur IoT-LAB (noeuds de type M3).*

	Noeuds									
Essai	**216**	**217**	**218**	**219**	**220**	**221**	**222**	**223**	**224**	**225**
I	19	0	0	24	129	287	500	35	500	500
II	81	0	0	500	500	500	500	211	500	10
III	500	4	0	500	500	102	17	205	500	500
IV	116	0	0	328	500	500	120	114	500	500
V	49	4	61	500	139	500	97	500	500	91
VI	17	6	0	500	500	500	68	66	500	63
VII	383	14	0	8	500	500	271	80	500	77
VIII	500	3	0	500	87	500	105	209	500	209
IX	500	21	0	500	17	500	26	478	500	314
X	500	0	0	500	500	500	89	88	500	500
XI	109	1	0	500	500	500	112	107	500	500
XII	500	0	0	208	500	500	66	66	500	500
XIII	500	27	0	23	22	500	7	500	500	500
XIV	25	2	23	208	500	500	500	312	500	9
XV	500	0	0	500	309	500	500	33	500	10

TABLE 6.9 – *Nombre de trames envoyées avec succès par chaque noeud-feuille, sur 500 envoyés.*

été traitée par une extension majeure de la taille de la pile système).

> Le problème semble toucher les microcontrôleurs (MCU) des noeuds, la broche `IRQ/C4` des noeuds « sourds » étant positionnée à 1 en fin d'expérience, tandis que sur les noeuds « sains », cette broche est revenue à 0 (puisqu'il n'y a plus rien à recevoir).
Pourtant ***aucune différence n'est visible (au bit près) entre la configuration physique des MCU des noeuds réussissant à envoyer tous leurs trames et ceux « devenant sourds »***. Sont pourtant vérifiés systématiquement, lors de nos comparaisons entre noeuds atteints de « surdité » et noeuds arrivant à recevoir, les registres physiques :

1. de l'arbre d'horloge, qui alimente bien tous les circuits voulus (GPIO, NVIC, EXTI, CPU) ;
2. des GPIO, notamment du port C4 (qui reçoit la broche `IRQ` de la radio), lequel est bien positionné comme entrée standard ;
3. du contrôleur d'interruptions : l'interruption correspondant à la broche GPIO C4 (EXTI4) est bien activée, sur front montant, avec une priorité suffisamment élevée pour être relayée par le NVIC ;
4. et de l'unité centrale (CPU) elle-même, c-à-d. : le cœur ARM est bien en mode « privilégié » (nécessaire à l'exécution des gestionnaires d'interruptions), et le masquage global de toutes les interruptions n'est pas actif.

> L'état de la radio a été affiché régulièrement durant l'éxécution (à chaque abandon de l'envoi d'une trame pour cause de file d'envoi pleine) : cette puce radio, lors de périodes de « surdité », est dans une configuration correcte (mode écoute actif, adresses et canal radio corrects, etc.). Ce problème n'est donc pas lié à celui de la « perte de configuration » touchant parfois les puces radio.

> Plus la fréquence d'envoi des trames augmente, plus la fiabilité diminue :
* impossible d'avoir 10 noeuds-feuilles fiables simultanément à 1 trame / seconde chacun ;
* impossible d'avoir 2 noeuds-feuilles fiables simultanément à 5 trames / seconde chacun ;
* impossible d'avoir un seul noeud-feuille fiable à environ 7 ou 8 trames / seconde.

Cela est étrange, car comme dit au point précédent, la configuration de la puce radio ne montre dans ces cas-là aucune anomalie.
Plus encore, l'AT86RF231 dispose d'une fonctionnalité étendue nommée *"High Data Rate Modes"* (voir [DataSheet AT86RF231, 2009] section 11.3), lui permettant d'émettre et de recevoir à un débit pouvant monter jusqu'à 2000 Kb/s (***huit fois*** le débit nominal IEEE 802.15.4) au prix d'une sensibilité diminuée (-89 dBm au lieu de -101 dBm en mode IEEE 802.15.4 standard). Même si nous n'utilisons absolument pas ces modes à haut débit, il semble difficilement crédible qu'un composant, capable de dépasser

aussi largement les limites de débit du standard, se bloque lorsque cette limite officielle de base de 250 Kb/s est approchée...

6.3.2.3 Résumé : les problèmes rencontrés et leur traitement

Nous avons recherché les causes de ce problème aussi bien au niveau logiciel (plus précisément : dans notre code), qu'au niveau de la radio ou du microcontrôleur lui-même. Mais comme nous l'avons dit, nous avons vérifié la configuration de la puce radio de façon poussée sans résultat. Concernant le MCU, nous avons examiné la configuration des entrées / sorties (GPIOs et AFIOs), des interruptions et évènements (NVIC et EXTI), de l'arbre d'horloge (RCC) [Manuel STM32F10x, 2011], et du cœur ARM Cortex-M3 lui-même [Manuel STMCortexM3, 2013], sans trouver d'explication logique quant à cette désactivation de l'interruption liée à la radio, ni aucune piste valide quant à sa ou ses cause(s). Nous avons également longuement et soigneusement testé et modifié notre code, en vain.

À l'heure actuelle, trois possibilités nous viennent à l'esprit pour expliquer ce phénomène :

— Nous pouvons nous demander si une défaillance brève mais marquée dans l'alimentation des noeuds (en anglais : *"brownout"*) pourrait provoquer ces problèmes. Certains éléments font penser à une telle hypothèse, mais nous n'avons aucun moyen de la vérifier, ce MCU ne comportant pas de mécanisme de détection de *"brownout"* comme en ont d'autres microcontrôleurs — ni même un mécanisme basique de contrôle de la tension d'alimentation comme la radio AT86RF231 avec son interruption `BAT_LOW`.
Notons toutefois que la baisse de fiabilité corrélée au débit des trames, quel que soit le nombre de noeuds, semble contredire cette hypothèse ; sauf si tous les noeuds employés lors de nos expériences dépendent de la même alimentation.

— Nous pouvons soupçonner un problème dans le mécanisme de gestion des interruptions dans le noyau de RIOT. Mais dans ce cas, pourquoi l'interruption liée à la radio est-elle la seule touchée ? Nous savons que les timers matériels (qui correspondent à un autre type d'interruption) restent eux fonctionnels sur tous les noeuds, y compris ceux touchés par le problème. En outre, après avoir signalé le problème sur les listes de diffusion de RIOT, aucun autre utilisateur ne semble rencontrer la même situation.

— Nous pouvons penser qu'un autre circuit appartenant au microcontrôleur, outre ceux dont nous avons vérifié les registres matériels, est parfois mal configuré. Toutefois, vu l'étendue des sous-systèmes du MCU dont nous avons vérifié les registres, nous ne voyons absolument pas quelle autre partie du microcontrôleur serait susceptible de provoquer de telles anomalies.

Comme on le voit, aucune de ces trois hypothèses n'est vérifiable facilement, ni suffisante à elle seule pour pouvoir expliquer toutes les anomalies constatées.

Nous n'avons en résumé aucune piste solide quant à la cause des problèmes que nous rencontrons, ni par conséquent du moyen de les résoudre ou même de les contourner dans le cadre de cette thèse.

6.3.3 Situation actuelle

Nos dernières expériences sur les noeuds IoT-LAB M3 ont toutes échoué — impossible d'avoir des résultats fiables — à cause de ce problème de « surdité » des noeuds.

Ce problème étant comme nous l'avons dit systématique, aucune expérience n'arrive à son terme sans erreur : tenter de les multiplier dans le but d'obtenir assez de résultats réussis sur le long terme est inutile.

En outre, d'autres problèmes rencontrés surviennent encore de façon sporadique :

— perte de sa configuration par le transmetteur radio, ce qui oblige à redémarrer le noeud touché ;

— problème de non-sortie du mode de transmission d'un trame.

Là également, nous ignorons les causes de ces anomalies.

Quelques discussions avec les ingénieurs du projet IoT-LAB nous ont appris que la plate-forme logicielle de référence sur les noeuds IoT-LAB M3 n'est pas RIOT OS, mais OpenLab d'HiKoB (la société ayant conçu ces matériels), ce qui explique que peu d'utilisateurs d'IoT-LAB soient concernés par ce problème, et / ou en mesure de nous aider.

Comme indiqué dans la section 6.3.2.3 précédente, nous n'avons, malgré nos nombreux efforts, actuellement trouvé aucune piste technique significative pour expliquer, contourner ou résoudre ces problèmes.

Ce problème a vraisemblablement une cause complexe (voire plusieurs causes multiples), tout comme celui qui touchait le portage de RIOT OS sur MSP430 [Roussel et al., 2014v] [Roussel et al., 2014s] [Roussel et al., 2014t]. Toutefois, nous sommes ici face à un appareillage d'un niveau de complexité bien supérieur à celui d'un noyau MSP430.

Corriger le ou les bogues sous-jacent(s) qui provoque(nt) les problèmes auxquels nous faisons face est évidemment un travail de longue haleine, dont la durée peut difficilement être estimée précisément — mais se compterait probablement en mois, pour reprendre l'exemple antérieur de RIOT sur MSP430. Il s'agira en tout cas d'un travail d'ingénierie de haut niveau, nécessitant clairement du personnel compétent et formé, sinon aux WSN, du moins aux systèmes embarqués.

Pour toutes ces raisons, nous n'avons donc pu réaliser que le premier des travaux envisagés en section 6.2.3 page 150 en temps voulu.

Nous espérons qu'un travail ultérieur à cette thèse permettra de résoudre ces problèmes — ou de passer outre, par exemple en employant une autre plate-forme matérielle ; à moins que la nouvelle pile réseau de RIOT, que nous avons abordé dans la section 4.3, ne permette, même partiellement, de régler ces difficultés — et en tirera les bénéfices et contributions pour l'instant manquants que nous citons en section 6.2.3 page 150.

Nous souhaitons également que les informations techniques et pistes de réflexion que nous avons compilées dans la présente section contribuent à la découverte d'une future solution à ces différents problèmes techniques.

Enfin, notons que des travaux sont d'ores et déjà en cours pour tenter de porter nos protocoles MAC / RDC avancés sur la nouvelle pile « gnrc » de RIOT OS, afin de pouvoir tester dès que possible si cette dernière permet d'apporter des avancées concernant ces difficultés (voir section 7.3.1 page 171 sur les perspectives à court terme).

6.4 Discussion : validation des expérimentations, contributions et conclusion

Nous avons dans le présent chapitre ***apporté les contributions suivantes*** :

— Nous avons clairement montré que ***l'émulation effectuée par le logiciel MSPSim souffre d'un sérieux problème d'inexactitudes temporelles concernant l'accès au bus SPI, probablement dû à une mauvaise calibration des délais pour l'émulation des MCUs***. Ce problème impacte les communications entre les microcontrôleurs et les émetteurs / récepteurs radio — et par conséquent, les opérations liées à la radio — dans les *motes* constituant les réseaux de capteurs sans fil.

— Nous avons ***décrit la gravité du problème avec des résultats d'expérimentations détaillées*** — gravité qui se trouve être sérieuse, tout spécialement concernant l'émulation de la plate-forme matérielle Zolertia Z1.

— Nous avons ***fourni des pistes sérieuses quant aux causes de ce problème***, et par conséquent délimité une zone raisonnable d'investigation à traiter pour parvenir à une correction des erreurs sous-jacentes et à une résolution de ces difficultés.

— Nous avons ***proposé une extension générique de l'API des pilotes radio (couche 1) des piles réseau des plates-formes logicielles spécialisées dans les WSN***. Si notre première implantation décrite dans ce chapitre ne pourra pas être réutilisée, cette idée à d'ores et déjà été reprise et implantée par RIOT OS (comme nous l'avons vu précédemment en section 4.3 page 95) et Contiki dans sa version 3.0 (partiellement et indirectement grâce à nos contributions). Nous sommes en outre convaincus que ***cette API générique, à la fois simple et performante, peut être adaptée à toute plate-forme logicielle, et même à toute pile réseau, pour améliorer l'interface de ses pilotes radio*** (interface entre la couche 1 et les couches supérieures) dans le domaine des WSN, et certainement même dans un cadre encore bien plus général : on peut comparer notre concept à la primitive `ioctl()` classique dans le monde Linux/Unix/POSIX).

— Nous avons ***proposé une liste de travaux devant permettre de valider nos précédentes expériences avec S-CoSenS, puis tester la « montée en charge » de ce dernier sur un réseau étendu reproduisant assez fidèlement le niveau de complexité d'un « logement intelligent »***, qui représente le principal domaine d'application du contexte de la présente thèse (le projet LAR).

— Malheureusement, des problèmes techniques que nous n'avons pu résoudre nous ont empêché de mener les dits travaux à leur terme. Nous avons alors ***fourni le maximum de détail techniques concernant les problèmes recontrés et les plates-formes sous-jacentes, dans l'espoir de faciliter la résolution de ces difficultés et le succès des travaux que nous avions prévus lors d'un travail ultérieur à la présente thèse***. Notons que nos investigations nous font penser que ces problèmes sont complexes, potentiellement provoqués par plusieurs causes distinctes, et que leur résolution représentera un travail d'ingénierie délicat et donc potentiellement long à accomplir.

Ce chapitre aura au final démontré la fragilité des résultats d'expériences obtenus par simulation / émulation : si ceux-ci sont bien plus faciles et moins coûteux à obtenir — surtout en quantité et avec les éléments de précision désirés —, ils sont toujours susceptibles d'être faussés par des inexactitudes ou des erreurs dans les logiciels de simulation ou d'émulation, comme nous venons de le voir pour le *framework* Cooja / MSPSim. Le recours abondant à ces techniques de simulation / émulation pour évaluer des travaux dans la littérature pousse à se poser des questions quand à la validité de nombreuses publications.

Soulignons néanmoins que ces outils de simulation et d'émulation sont par contre de formidables outils de développement et de déboguage, simplifiant souvent, et parfois considérablement, ces deux tâches. Ces fonctionnalités suffisent à elles seules à justifier l'existence et le développement continu de tels outils logiciels.

L'évaluation fiable du fonctionnement de réseaux sans-fil, et en particulier de projets conçus sur la technologie des WSN, nécessite *a contrario* le recours à des tests sur matériel réel, comme nous pensons l'avoir démontré ici. La mise en place de tels tests n'est en aucun cas triviale : elle est longue, coûteuse, et présente des difficultés techniques nombreuses et souvent difficiles à résoudre — plus que l'on ne l'imagine en général au premier abord.

Des efforts ont été mis en place pour faciliter l'exécution de tels tests, comme le *testbed* Iot-LAB, que nous avons longuement utilisé et décrit dans le présent chapitre. Malgré tout, nous avons pu voir que malgré les moyens importants déployés par ce projet — notamment en matériels et compétences humaines (ingénieurs du projet réactifs et disponibles pour aider les utilisateurs) — nous n'avons pu surmonter les problèmes techniques qui ont interrompu nos travaux de validation. Les difficultés qui entravent les tests de validation d'expérimentation sur matériel concernant les WSN sont donc bien réelles, importantes, et ne sauraient être sous-estimées. Cela explique aussi sans doute aussi pourquoi les validations par simulation / émulation sont encore souvent employées : par défaut.

Au final, ce chapitre, s'il n'a pas, comme nous l'escomptions, validé nos précédentes expérimentations sur S-CoSenS, se révèle être riche en enseignements, parfois désagréables à constater, et représente — nous l'espérons — une source de renseignements qui permettra la poursuite de nos travaux de validation dans un travail

ultérieur, et au mieux fournira explications et solutions pour faciliter d'autres futurs travaux d'expérimentations de WSN sur matériel réel. Il d'agit donc là de perspectives à court mais aussi peut-être à plus long terme, que nous allons détailler dans le chapitre 7 suivant, après les conclusions générales de ce manuscrit.

Chapitre 7

Conclusions et perspectives

7.1 Conclusions générales

Les réseaux de capteurs sans-fil (WSN), et par extension de l'Internet des Objets (IoT), constituent un domaine de recherche et de développement très actif ; ceci s'accompagne du développement d'applications toujours plus nombreuses de ces réseaux dans des domaines sans cesse plus variés. Ce développement promet encore de s'accélérer avec l'évolution des technologies et la montée en puissance des noeuds (*"motes"*) constituant ces réseaux.

Ces réseaux ont des objectifs contradictoires : la qualité de service (QdS) d'une part, et d'autre part l'économie d'énergie pour prolonger la durée de vie des *"motes"* et surtout de leur batteries. Pour atteindre ces objectifs contradictoires, de nombreux efforts de recherche ont été menés au niveau de la couche MAC (niveau 2 OSI) de la pile réseau, comme nous l'avons vu dans la première partie du chapitre 3 sur l'état de l'art. Le standard IEEE 802.15.4, sur lequel repose la plupart des WSN actuels, propose lui-même ses propres couches MAC (avec leurs limitations), et une extension du standard— 802.15.4e — se consacre à amener un nouveau protocole MAC nettement plus performant et complexe.

Nous avons également vu que ces WSN disposaient de systèmes d'exploitation dédiés. Ces plates-formes logicielles spécialisées ont fait l'objet d'une revue critique lors de la seconde partie du chapitre 3.

La comparaison de ces deux états de l'art nous montre que les nombreux et divers protocoles MAC issus de travaux de recherche n'ont que très rarement fait l'objet d'implantations dans les systèmes d'exploitations dédiés (ContikiMAC étant la seule exception notable). La problématique de cette thèse a donc consisté à faire avancer l'implantation effective de protocoles MAC innovants et performants dans ces OS dédiés, afin d'améliorer le facteur limitant que représentent, à l'heure actuelle, les couches basses des piles réseau pour capteurs sans-fil.

Du chapitre 3, nous avons tiré la conclusion que l'implantation de protocoles MAC hautes performances nécessitait des fonctionnalités de la part de la plateforme logicielle fournissant la pile réseau : il s'agit principalement de fonctionnalités temps-réel, permettant de respecter des délais très stricts pour nos implantations,

ce qui est nécessaire pour une synchronisation suffisamment précise entre noeuds communicants, laquelle est cruciale pour les protocoles récents basés en particulier sur le multiplexage temporel (TDMA). Nous avons constaté que les plates-formes logicielles les plus employées et connues (Tiny OS et Contiki) n'étaient pas en mesure de fournir de telles fonctionnalités ; cela explique notamment pourquoi ces systèmes ne proposent que des protocoles MAC basés sur la contention (CSMA), car ils sont dans l'incapacité de supporter des protocoles basés sur des paradigmes plus exigeants en respect de contraintes temporelles.

Les travaux de notre thèse nous ont ainsi amené à rechercher une plate-forme logicielle adéquate au chapitre 4. Nous avons étudié deux plates-formes spécialisées : Contiki et RIOT OS. Ce chapitre décrit notre point de vue sur les forces et faiblesses de ces dernières, et tout particulièrement une analyse critique de leurs piles réseau. Nous avons alors choisi la plate-forme fournissant les fonctionnalités nécessaires. Notre choix s'est porté sur RIOT OS, qui parmi de nombreux autres avantages, se montre accueillant et ouvert aux contributions extérieures. Tout en contribuant activement à son déboguage, son évolution et son portage sur le matériel que nous utilisions à l'époque, nous avons ainsi pu implanter avec succès notre premier protocole, S-CoSenS, sous RIOT.

Une fois cette implantation de S-CoSenS prête, nous avons au chapitre 5 d'abord testé son bon fonctionnement, puis comparé ses performances avec ContikiMAC, principalement via des tests effectués par des simulations sous Cooja (le simulateur de réseaux de capteurs sans-fil du projet Contiki), mais aussi quelques tests limités sur matériel. Il ressort de ces tests que notre implantation de S-CoSenS sous RIOT offre une meilleure QdS que ContikiMAC dans son implantation standard sous Contiki, tandis que ce dernier — si l'on se fie à l'indicateur certes imparfait que sont les *"duty cycles"* — obtient clairement de meilleurs résultats concernant les économies d'énergie. Nous avons conclu de ces tests que comme nous le pensions, des fonctionnalités temps-réel sont nécessaires pour implanter des protocoles MAC novateurs et efficaces. Nous avons également proposé des pistes pour faire face aux problèmes de corruption mémoire en l'absence de mécanismes matériels de protection mémoire, ainsi que plusieurs techniques susceptibles d'améliorer la robustesse des protocoles MAC / RDC, grâce à une meilleure évaluation du trafic réseau courant, et une adaptation dynamique du rapport signal / bruit (SNR).

La supériorité affichée du protocole S-CoSenS en matière de QdS, au détriment du *"duty cycle"*, correspond à notre approche : nous pensons en effet que la première priorité d'un WSN est de transmettre correctement les données qui lui sont confiées (QdS), l'économie d'énergie étant une seconde priorité, qui ne doit pas nuire à la première. Nos tests montrent que le protocole S-CoSenS assure bien mieux le bon ordre de ces priorités que ContikiMAC, spécialement quand le trafic réseau est intense.

En testant l'influence de l'implantation des pilotes SPI des OS dédiés sur les performances de communication, nous avons été amenés à découvrir des inexactitudes importantes, au niveau temporel, dans les simulations effectuées par Cooja. Nous avons alors étudié ce problème, et avons détaillé nos découvertes au début du chapitre 6. Nous avons déterminé que le problème venait de l'émulateur MSPSim, que

Cooja utilise pour émuler les matériels basés sur l'architecture MSP430. Le problème vient visiblement d'un problème de calibration des délais d'émulation du bus SPI, l'inexactitude de calibration variant visiblement selon le microcontrôleur émulé. Il est par exemple clair que si l'erreur de calibration reste modérée pour le MSP430F1611 équipant entre autres la TelosB / Skymote, elle devient extrêmement gênante pour le MSP430F2617 équipant la Zolertia Z1.

Notre conclusion est que tant que ces erreurs de calibration ne sont pas corrigées, les tests sur matériels sont seuls à même de permettre des évaluations de performances fiables (approche empirique). Cooja reste quoi qu'il en soit un outil très utile, notamment pour faciliter le dévoppement et le déboguage d'applications sur capteurs sans-fil.

La suite de ce même chapitre détaille les travaux que nous avions prévus d'effectuer sur du matériel réel pour valider nos expériences précédentes concernant S-CoSenS, puis tester sa montée en charge, notamment en mettant en oeuvre un réseau comportant de nombreux noeuds et divisé en différents sous-réseaux, reproduisant ainsi plus fidèlement un « logement intelligent » richement pourvu en appareils susceptibles d'appartenir à un WSN. Nous avons choisi d'effectuer les dits travaux sur le *testbed* IoT-LAB, offrant l'accès à de très nombreuses *motes*, dont les plus récentes (IoT-LAB M3) offrent une puissance confortable et des possibilités d'instrumentation riches et avancées.

Malheureusement, seule la première partie de ces travaux a pu être menée à terme, des problèmes techniques nombreux et complexes nous ayant empêché de terminer leur réalisation. Nous avons toutefois validé notre idée d'amélioration générique de l'API des pilotes radio, idée reprise par Contiki et RIOT OS dans sa nouvelle pile réseau. Nous n'avons au final pas pu réaliser et implanter toutes nos idées de tests et d'optimisations, mais celles-ci peuvent servir de contributions pour des travaux ultérieurs.

7.2 Contributions de la thèse : résumé

Au final, les principales contributions de cette thèse sont les suivantes :

— ***L'analyse des différentes plates-formes logicielles (systèmes d'exploitation) dédiées aux réseaux de capteurs sans-fil, et de leurs fonctionnalités*** (section 3.3 page 61 de l'état de l'art).

— ***De nombreuses contributions techniques amenées au projet RIOT OS*** (section 4.2 page 87) : notamment l'ajout d'un mécanisme de gestion des erreurs fatales, le portage du système sur la *mote* Zolertia Z1, et la résolution d'un problème majeur qui empêchait RIOT OS de fonctionner de façon fiable sur les systèmes à architecture MSP430. À cela peuvent s'ajouter de nombreuses autres contributions mineures au projet RIOT, ainsi qu'une contribution partielle et indirecte au projet Contiki (cf. section 4.1.2.1 page 81).

— ***Nous nous sommes penchés sur la nouvelle pile réseau (« gnrc ») de RIOT, montré ses possibilités et les améliorations qu'elle apporte, ainsi que ses défauts*** (section 4.3 page 95). Nous sommes confiants sur le

fait qu'une fois ces derniers corrigés ou contournés [1], RIOT OS deviendra une plate-forme encore plus performante et pratique pour les réseaux de capteurs sans-fil.

— ***L'affirmation de l'utilité — et même de la nécessité — de fonctionnalités temps-réel avancées pour implanter des protocoles MAC / RDC évolués***, offrant à la fois une qualité de service maximale et une consommation d'énergie optimale (les *"duty cycles"* étant les marqueurs nous ayant été les plus accessibles, bien que très imparfaits, pour appréhender cette consommation énergétique). Une première démonstration en a été faite avec l'implantation de S-CoSenS sous RIOT OS, et une comparaison avec ContikiMAC donnant des résultats encourageants (en partie confirmée par des tests sur matériel), ***particulièrement les bons résultats de S-CoSenS en termes de QdS en présence d'un trafic réseau intense***. Cela correspond à l'approche que nous avons revendiquée en conclusion du chapitre 5 (section 5.5 page 132).

— ***Plusieurs perspectives d'amélioration des couches basses des piles réseau des plates-formes spécialisées dans les capteurs sans-fil***, idées dont la description et les bénéfices / contributions attendus sont décrits en section 5.4 page 125.

— ***La découverte et l'analyse d'un problème d'inexactitudes sévères au niveau temporel — délais de communication entre microcontrôleur et émetteur / récepteur radio — dans le simulateur de réseaux de capteurs sans-fil Cooja*** (ou plutôt : dans l'émulateur MSPSim sur lequel il s'appuie). Nous avons fourni des pistes sérieuses quant à la cause de cette anomalie — mauvaises calibrations des délais d'émulation, variables selon le MCU — dans l'espoir de faciliter sa future correction. Nous avons enfin tenté d'analyser le retentissement de ce problème sur la validité des travaux d'évaluation de performances réalisés grâce à cet outil — y compris la validité de nos propres travaux de simulation précédents. Cette contribution correspond à la section 6.1 page 136 constituant la première partie du chapitre 6.

— ***La validation de nos expériences réalisées par simulation / émulation grâce à des tests sur du matériel réel disposant des capacités d'instrumentation adéquates***, tests décrits en 6.2.3 page 150. Malheureusement, seul le premier de ces travaux a pu être réalisé à cause des problèmes techniques décrits en section 6.3.2 page 154. ***Nous avons alors donné tous les détails techniques en notre possession, souhaitant aider à la résolution — dans des travaux ultérieurs — de ces difficultés*** dont les causes, visiblement complexes, restent actuellement pour nous inconnues et incompréhensibles. Ceci constitue la seconde et dernière partie du chapitre 6.

1. Notamment par la conception d'une pile alternative plus légère, dédiée aux appareils très limités et / ou aux applications nécessitant des réactions extrêmement rapides aux évènements réseau (sans passer par une pile complète et complexe).

7.3 Perspectives

7.3.1 Perspectives à court terme

Les perspectives à court terme consisteraient bien évidemment à terminer les travaux prévus en section 6.2.3 page 150 du présent manuscrit, pour confirmer (ou non) de façon indiscutable la validité des expériences effectuées dans la présente thèse, pour lesquelles nous avons majoritairement eu recours aux techniques de simulation / émulation.

Nous nous efforçons actuellement, avec le soutien actif de l'équipe de développement de RIOT OS, d'adapter S-CoSenS à la nouvelle pile « gnrc » de ce système, dans l'espoir d'obtenir ainsi des avancées, ou au moins des informations complémentaires, concernant les problèmes qui nous ont empêché de mener à leur terme nos travaux de validation.

Dans un second temps, afin de compléter les solutions à la problématique posée par la présente thèse, et d'en tirer de nombreux bénéfices et contributions, il serait fort utile de procéder aux tests des améliorations décrites en section 5.4 page 125. Nous pensons notamment à :

— L'étude de l'influence des SFD sur S-CoSenS ;
— L'étude plus largement de l'influence du SNR (comme le fait le protocole AEDP présenté dans la section sus-citée) sur l'amélioration de la couche MAC des piles réseau.

Dans un troisième temps, nous pensons qu'il serait extrêmement profitable de s'intéresser à l'implantation du protocole iQueue-MAC (voir section 3.2.6.2 page 51 dans l'état de l'art), dont nous n'avons pas pu nous occuper durant la présente thèse. Il s'agit de l'un des protocoles MAC / RDC parmi les plus performants disponibles actuellement, et l'un de nos objectifs initiaux était notamment de mesurer ses performances en matière de consommation d'énergie, grâce à l'instrumentation spécifique des noeuds d'IoT-LAB.

Plus précisement, il se trouve qu'iQueue-MAC a déjà été testé sur matériel [Zhuo et al., 2013], mais en dehors du cadre d'une plate-forme logicielle. La contribution attendue de ces tests sur matériel est donc de déterminer clairement ***l'influence d'une plate-forme comme RIOT OS, multitâche et pourvue de fonctionnalités temps-réel avancées, sur les performances d'un protocole évolué comme iQueue-MAC.*** L'implantation du protocole iQueue-MAC, plus avancé, au sein du système RIOT OS, et son intégration dans la nouvelle pile réseau « gnrc » serait ainsi une contribution majeure, permettant d'offrir à cette plate-forme logicielle un protocole MAC / RDC très performant au sein de sa pile réseau.

Nous travaillons actuellement avec l'un des auteurs de ce protocole, S. Zhuo, dans le cadre d'une collaboration pour tenter d'implanter et de tester nos protocoles avancés — S-CoSenS et iQueue-MAC — sur RIOT et sa nouvelle pile gnrc, en développant et testant directement sur une des plates-formes matérielles de référence de RIOT OS : la carte d'évaluation Atmel SAMR21 Xplained Pro, contenant un MCU avec émetteur / récepteur radio 802.15.4 intégré. Cette collaboration, déjà évoquée

à la fin de la section 6.3.3 page 163, avance progressivement, l'une des premières tâches en cours de réalisation est de faire fonctionner la couche MAC IEEE 802.15.4 de base (sans *"beacon"*). Différents tests sur matériel ont déjà permis à S. Zhuo de valider le fonctionnement du module `csma_sender` que nous avons proposé pour la pile gnrc de RIOT (PR #4178), comme le montre la capture d'écran d'un *"sniffer"* de paquets ayant capté une transmission entre deux cartes SAMR21 en figure 7.1. Quelques problèmes subsistent pour faire fonctionner la couche MAC 802.15.4 que nous avons proposée (PR #4184), mais ce premier objectif semble *a priori* réalisable à court terme (dans les semaines qui viennent). L'implantation de S-CoSenS, et *a fortiori* iQueue-MAC, qui sont des protocoles bien plus avancés et complexes, demandera sans doute un délai plus long.

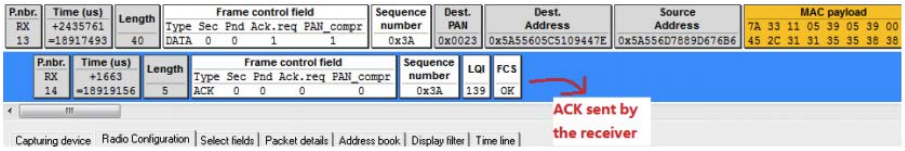

FIGURE 7.1 – *Capture d'écran d'un "sniffer" de paquets Zigbee / 802.15.4 lors de l'exécution de RIOT avec le module* `csma_sender`*. (Crédit : Shuguo Zhuo, travail en cours.)*

En outre, il serait intéressant d'étudier le comportement de ce protocole sur un réseau de topologie étendue (comme nous comptions le faire pour parachever la validation de S-CoSenS). Tester l'influence de la variation du *"CCA Threshold"* (c'est-à-dire du SNR) sur ce protocole serait sans doute également un test susceptible de fournir une contribution intéressante.

Enfin, un dernier test d'un grand intérêt serait notamment ***la comparaison d'iQueue-MAC avec la nouvelle couche MAC du standard amendé IEEE 802.15.4e*** ; cette dernière fait en effet appel au FDMA (changement dynamique de canal, *"channel hopping"*), ce que ne fait pas iQueue-MAC, qui se propose plutôt d'utiliser un unique canal de la façon la plus optimale possible. Le changement de fréquence étant une opération nécessitant un certain délai (potentiellement long) durant lequel l'émetteur / récepteur radio est indisponible, ***on peut légitimement se demander lequel de ces deux protocoles MAC / RDC est le plus performant en situation réelle. Une réponse claire à cette question serait une contribution d'importance pour l'évolution des couches basses des piles réseaux des WSN.***

Tous ces tests seraient bien évidemment à réaliser, dans l'idéal, sur du matériel en quantité suffisante, pourvu des fonctionnalités d'instrumentation. L'une des plates-formes matérielles répondant à ces exigences techniques serait le *testbed* IoT-LAB. Cependant, comme il l'a été dit en section 6.3.3 page 163, résoudre les problèmes techniques que nous avons rencontrés sur la plate-forme IoT-LAB, et notamment ses noeuds M3, sera certainement un travail d'ingénierie et de déboguage complexe, dont

il est difficile d'estimer la durée (probablement plusieurs mois si ces difficultés sont dues à plusieurs causes combinées).

7.3.2 Perspectives à long terme

Les perspectives à long terme, en matière de réseaux de capteurs sans-fil, sont immenses et semblent n'avoir comme limite que l'imagination.

Concernant les couches basses des piles réseau des plates-formes logicielles spécialisées dans les capteurs sans-fil, leur optimisation et surtout leur fiabilisation devrait améliorer considérablement la qualité des liaisons radio entre les *motes* de ces réseaux, comme cette thèse a — nous espérons — commencé à le montrer.

Au cours de nos travaux, nous avons remarqué que les plates-formes logicielles mais aussi et surtout matérielles semblent parfois peu adaptées, et souffrent de limites pénalisantes. Si nous avons travaillé à l'amélioration des plates-formes logicielles, nous voyons les limitations suivantes aux plates-formes matérielles, lesquelles s'avèrent — nous le pensons — inadaptées pour le développement futur de réseaux de capteurs sans-fil à hautes performances :

— Les limitations des microcontrôleurs (MCU). En la matière, le facteur réellement limitant n'est pas la puissance de calcul, mais l'espace mémoire, tout spécialement la RAM permettant de stocker les données, souvent disponible en quantités trop faibles (10 Ko ou moins).
L'apparition de microcontrôleurs plus puissants (comme les Cortex-M présents dans certains noeuds IoT-LAB) améliorent cet état de fait, mais au prix d'une complexité nettement plus élevée — donc une programmation et un déboguage plus délicats — que les architectures plus « classiques » comme les MSP430 ou les AVR. En outre, le gain en puissance de calcul apporté par ces nouveaux MCUs n'est souvent pas nécessaire, surtout pour les noeuds-feuilles destinés surtout à interagir avec leur environnement physique et non à faire des calculs complexes.

— La séparation des émetteurs / récepteurs radio du MCU central : le programmeur d'une *mote* se retrouve souvent à devoir programmer deux puces principales, seul le MCU étant accessible au déboguage avancé (en général via une liaison JTAG) ; la radio n'étant elle accessible qu'indirectement — souvent par le bus SPI —, il est donc difficile de connaître précisément son état interne en cas de problème complexe impliquant directement cette puce radio. (C'est l'une des difficultés que nous avons rencontré au cours de nos travaux de déboguage que nous avons détaillé en section 6.3.2.2 page 158).
Bien sûr, un émetteur / récepteur radio séparé offre par contre l'avantage d'être facilement interchangeable.

En fait, on peut réaliser qu'il n'existe encore que très peu de matériel électronique ***spécifiquement*** dédié aux capteurs sans-fil, les MCUs étant des composants existant de longue date et souvent dédiés à des applications embarquées aux besoins bien différents de ceux de nos réseaux de capteurs sans-fil.

La situation commence à changer doucement, avec l'apparition de MCUs avec radio intégrée, comportant en outre une quantité de RAM plus importante, mieux adaptée aux travaux de développement sur capteurs sans-fil. Un exemple récent est la famille Atmel AVR ATmegaRFR2 [DataSheet ATmegaRFR2, 2014] offrant, outre une radio intégrée comparable à celle équipant les noeuds IoT-LAB M3, un espace mémoire allant jusqu'à 256 Ko de mémoire Flash et 32 Ko de RAM, ce qui devient comparable aux MCUs plus complexes de type ARM Cortex-M3. Outre cette mémoire accrue tout à fait bienvenue, la présence de la radio directement sur le MCU rend les registres, et donc le fonctionnement de cette radio, directement accessible aux sessions de déboguage JTAG, facilitant grandement le travail du programmeur d'applications pour réseaux de capteurs sans-fil (du moins pour les noeuds n'ayant besoin que de cet unique émetteur / récepteur radio intégré).

Une solution matérielle commençant à être mise en œuvre est d'utiliser des circuits logiques programmables — des **FPGA** (*"Field-Programmable Gate Array"*) — permettant de développer des plates-formes matérielles parfaitement adaptées au rôle de capteur sans-fil : avec le choix de l'architecture processeur, de la quantité de mémoire, voire même de l'émetteur / récepteur radio (si la technologie le permet) optimaux pour une application donnée. Les FPGA devenant au cours du temps de plus en plus vastes et de moins en moins chers, on peut ainsi imaginer pouvoir développer un *"System-on-Chip"* sur mesure pour chaque type de réseaux de capteurs sans-fil ou même pour chaque application, tout en rendant la programmation et le déboguage plus aisés ; d'autant plus que les FPGA modernes offrent la possibilité de modifier[2] à la volée leur logique matérielle, permettant de changer d'architecture matérielle comme on change de programme sur un microcontrôleur classique.

Cela rejoint en tout cas les notions de SDN (*"Software-Defined Network"*) et de *"Software-Defined Radio"*. De même, certaines équipes de recherches et entreprises travaillent déjà sur des FPGA dans des buts similaires. On peut notamment citer le projet NetFPGA [Lockwood et al., 2007] déjà mature et largement utilisé[3].

Peut-être s'agit-il là de l'avenir des réseaux de capteurs sans-fil, voire même des systèmes embarqués dans leur ensemble ?

2. On parle alors de ***reconfiguration*** plutôt que de reprogrammation, car il s'agit de matériel et non de logiciel.

3. Site Web : `http://netfpga.org/`

Annexe A

Publications et Réalisations

Publications

Conférences / *workshops*

— Moutie Chehaider, Kévin Roussel, Ye-Qiong Song. « Interopérabilité des réseaux de capteurs hétérogènes dans un appartement intelligent » *9èmes journées francophones Mobilité et Ubiquité*, UbiMob 2013. Nancy, France. Juin 2013.

— Kévin Roussel. « Implementing a real-time MAC protocol under RIOT OS : running on Zolertia Z1 motes » In *Workshop Internet of Things / Equipex FIT IoT-LAB*, INRIA Grenoble Rhone-Alpes, Montbonnot, France. Novembre 2014. (présentation sans actes.)

— Kévin Roussel, Ye-Qiong Song, Olivier Zendra. « RIOT OS Paves the Way for Implementation of High-performance MAC Protocols » In *Proceedings of the 4th International Conference on Sensor Networks*, SensorNets 2015, pages 5–14. ESEO, Angers, France. Février 2015.

— Kévin Roussel, Ye-Qiong Song, Olivier Zendra. « Using Cooja for WSN Simulations : Some New Uses and Limits ». In *Proceedings of the 2016 International Conference on Embedded Wireless Systems and Networks* — EWSN '16 ; *workshop NextMote*, pages 319–324. *Technische Universität*, Graz, Autriche. Février 2016.

Rapports de recherche INRIA (HAL)

— Kévin Roussel, Ye-Qiong Song. « A critical analysis of Contiki's network stack for integrating new MAC protocols ». 13 pages. Décembre 2013.
Rapport de recherche INRIA n° 8776 *(CRI-Nancy Grand Est)*.
ID hal-01202542.

— Kévin Roussel, Ye-Qiong Song, Olivier Zendra. « Practical Lessons Learned through Implementation and Performance Evaluation of Two MAC/RDC Protocols on WSN OS » 25 pages. Mars 2015.
Rapport de recherche INRIA n° 8777 *(CRI-Nancy Grand Est)*.
ID hal-01202664.

Réalisations techniques (*"Pull Requests"* sur GitHub)

Pour Contiki OS

Ces deux contributions ont été refusées (mais réutilisées en partie et indirectement pour une amélioration ultérieure de l'API radio de Contiki : PR #617 de Niklas Finne et al, intégrée au code de Contiki OS en juin 2014).

— Kévin Roussel, George Oikonomou, Mariano Alvira, David Kopf, Valentin Sawadski, Joakim Eriksson, Peter A. Bigot, Adam Dunkels. PR #192 : « Extended radio drivers API ». (2013–2014).

— Kévin Roussel, Mariano Alvira, Joakim Eriksson, George Oikonomou, Oliver Schmidt, Adam Dunkels, Nicolas Tsiftes. PR #519 : « Radio api extension ». (2014).

Pour RIOT OS

Toutes les contributions citées ci-dessous dans la présente section ont été acceptées et ont été intégrées (*"merged PRs"*) à la branche principale du code de RIOT OS.

— Kévin Roussel, Ludwig Knüpfer, Oliver Hahm, Thomas Eichinger. PR #408 : « Simplify msp430 headers » (2013).

— Kévin Roussel, Oliver Hahm, Ludwig Knüpfer. PR #459 : « Msp430 lpm freq » (2013–2014).

— Kévin Roussel, Oliver Hahm, Kaspar Schleiser, Ludwig Knüpfer, Christian Mehlis, René Kijewski. PR #685 : « Panic » (2014).
Best PR award of the month (Février 2014).

— Kévin Roussel, René Kijewski, Oliver Hahm, Ludwig Knüpfer, Christian Mehlis. PR #687 : « Add a `reboot()` function to `kernel.h` definitions » (2014).

— Kévin Roussel, René Kijewski, Kaspar Schleiser. PR #689 : « Portable definition of function attributes » (2014).

— Kévin Roussel, René Kijewski, Oliver Hahm, Ludwig Knüpfer. PR #724 : « Reboot » (2014).

— Kévin Roussel, René Kijewski, Oliver Hahm. PR #881 : « Ensure that stack pointer is correcty aligned during thread creation on MSP430 » (2014).

— Kévin Roussel, Oliver Hahm, Thomas Eichinger. PR #882 : « CC2420 radio tranceiver's driver fixes » (2014).

— Kévin Roussel, Oliver Hahm, Ludwig Knüpfer, Christian Mehlis, Thomas Eichinger, Hauke Petersen. PR #893 : « Zolertia Z1 port for RIOT OS » (2014).

— Kévin Roussel, Ludwig Knüpfer, Thomas Eichinger, Oliver Hahm, René Kijewski. PR #915 : « Add a standard way to query CCA status on CC2420 transceiver » (2014).

— Kévin Roussel, Oliver Hahm, Ludwig Knüpfer, Thomas Eichinger, Martine Lenders, Hauke Petersen. PR #925 : « Proposal for common 802.15.4 radio driver API definition » (2014).

— Kévin Roussel, Oliver Hahm. PR #954 : « Fix for CC2420 radio driver for TelosB » (2014).
— Kévin Roussel, Oliver Hahm, Ludwig Knüpfer. PR #957 : « Handle race conditions preventing MSP430 timers to be set correctly » (2014).
— Kévin Roussel, René Kijewski, Kaspar Schleiser, Ludwig Knüpfer, Oliver Hahm, Christian Mehlis. PR #970 : « core : Add the ability to send a message to the current thread's message queue » (2014).
— Kévin Roussel, Ludwig Knüpfer, René Kijewski, Oliver Hahm, Christian Mehlis, Kaspar Schleiser. PR #1002 : « Enhance implementation of `hwtimer_spin()` » (2014).
— Kévin Roussel, Oliver Hahm, Ludwig Knüpfer. PR #1113 : « Use Timer B on MSP430 architecture » (2014).
— Kévin Roussel, Oliver Hahm. PR #1211 : « Completing low-level radio driver definition » (2014).
— Kévin Roussel, Oliver Hahm, Thomas Eichinger, Christian Mehlis. PR #1223 : « Modify & extend CC2420 driver to comply with API described in `radio_driver.h` » (2014).
— Kévin Roussel, Oliver Hahm. PR #1239 : « Add a missing constant in `'radio_tx_status_t'` enum » (2014).
— Kévin Roussel, René Kijewski, Hauke Petersen, Ludwig Knüpfer, Christian Mehlis, Oliver Hahm. PR #1380 : « Reset ARM Cortex-M3 MCUs before flashing » (2014).
— Kévin Roussel, Ludwig Knüpfer, Oliver Hahm. PR #1383 : « Fix a design error in `cc2420_do_send()` function » (2014).
— Kévin Roussel, Ludwig Knüpfer, Oliver Hahm. PR #1385 : « Fix a nasty race condition in CCA determination on CC2420 » (2014).
— Kévin Roussel, Oliver Hahm. PR #1388 : « boards/z1 : fix `cc2420_txrx` function in CC2420 driver HAL » (2014).
— Kévin Roussel, Ludwig Knüpfer, Hauke Petersen, Kaspar Schleiser. PR #1617 : « Ensure `hwtimer_spin()` won't wait for an unreachable stop counter value » (2014).
— Kévin Roussel, Ludwig Knüpfer, Oliver Hahm, Hinnerk van Bruinehsen. PR #1618 : « Fix `thread_yield()` on MSP430 platforms » (2014).
— Kévin Roussel, Ludwig Knüpfer, Oliver Hahm, Hinnerk van Bruinehsen, Martine Lenders. PR #1619 : « Only use 16 significative bits for MSP430 `hwtimers` » (2014).
— Kévin Roussel, Oliver Hahm, Ludwig Knüpfer. PR #2214 : « Msp430 misc interrupt-related fixes » (2014).
— Kévin Roussel, Jonas Remmert, Oliver Hahm PR #4138 : « Add a `netopt` for getting and setting CCA threshold » (2015).

Bibliographie

[Abrach et al., 2003] Abrach, H., Bhatti, S., Carlson, J., Dai, H., Rose, J., Sheth, A., Shucker, B., Deng, J., and Han, R. (2003). Mantis : System Support for Multimodal Networks of In-Situ Sensors. In *2nd ACM International Workshop on Wireless Sensor Networks and Applications*, WSNA 2003, pages 50–59. ACM.

[Agha et al., 2009] Agha, K. A., Bertin, M.-H., Dang, T., Guitton, A., Minet, P., J.-B., T. V., and Viollet (2009). Which Wireless Technology for Industrial Wireless Sensor Networks ? The Development of OCARI Technology. *Industrial Electronics, IEEE Transactions on*, 56(10) :4266–4278.

[Akyildiz and Vuran, 2010] Akyildiz, I. F. and Vuran, M. C. (2010). *Wireless Sensor Networks*. Advanced Texts in Communications and Networking. John Wiley & Sons.

[Alemdar and Ersoy, 2010] Alemdar, H. and Ersoy, C. (2010). Wireless sensor networks for healthcare : A survey. *Computer Networks*, 54(15) :2688–2710.

[Amoretti et al., 2014] Amoretti, M., Alphand, O., Ferrari, G., Rousseau, F., and Duda, A. (2014). DINAS : A Distributed Naming Service for All-IP Wireless Sensor Networks. In *WCNC 2014*, pages 2781–2786.

[Ancillotti et al., 2014a] Ancillotti, E., Bruno, R., and Conti, M. (2014a). Reliable Data Delivery With the IETF Routing Protocol for Low-Power and Lossy Networks. *Industrial Informatics, IEEE Transactions on*, 10(3) :1864–1877.

[Ancillotti et al., 2014b] Ancillotti, E., Bruno, R., Conti, M., Mingozzi, E., and Vallati, C. (2014b). Trickle-L^2 : Lightweight Link Quality Estimation through Trickle in RPL Networks. In *WoWMoM 2014*, pages 1–9.

[Antonini et al., 2014] Antonini, M., Cirani, S., Ferrari, G., Medagliani, P., Picone, M., and Veltri, L. (2014). Lightweight Multicast Forwarding for Service Discovery in Low-Power IoT Networks. In *SoftCOM 2014*, pages 133–138.

[Barry, 2006] Barry, R. (2006). FreeRTOS—a free RTOS for small embedded real time systems. `http://www.freertos.org/`.

[Barry, 2010] Barry, R. (2010). *Using the FreeRTOS^{TM}Real Time Kernel*. Real Time Engineers Ltd., first edition.

[Basha et al., 2008] Basha, E. A., Ravela, S., and Rus, D. (2008). Model-based Monitoring for Early Warning Flood Detection. In *Proceedings of the 6th ACM Conference on Embedded Network Sensor Systems*, SenSys '08, pages 295–308. ACM.

[Buettner et al., 2006] Buettner, M., Yee, G. V., Anderson, E., and Han, R. (2006). X-MAC : A Short Preamble MAC Protocol for Duty-cycled Wireless Sensor Networks. In *Proceedings of the 4th International Conference on Embedded Networked Sensor Systems*, SenSys '06, pages 307–320. ACM.

[Buevich et al., 2013] Buevich, M., Rajagopal, N., and Rowe, A. (2013). Hardware Assisted Clock Synchronization for Real-Time Sensor Networks. In *34th IEEE Real-Time Systems Symposium*, RTSS '13, pages 268–277. IEEE.

[Cao et al., 2008] Cao, Q., Abdelzaher, T., Stankovic, J., and He, T. (2008). the LiteOS Operating System : Towards Unix-Like Abstractions for Wireless Sensor Networks. In *Proceedings of the 7th International Conference on Information Processing in Sensor Networks*, IPSN '08, pages 233–244. IEEE Computer Society. `http://www.liteos.net/`.

[Chatterjea et al., 2004] Chatterjea, S., Hoesel, L. F. W. V., and Havinga, P. J. M. (2004). AI-LMAC : An Adaptive, Information-centric and Lightweight MAC Protocol for Wireless Sensor Networks. In *Proceedings of the Intelligent Sensors, Sensor Networks and Information Processing Conference, 2004*, pages 381–388. IEEE.

[Chehaider et al., 2013] Chehaider, M., Roussel, K., and Song, Y.-Q. (2013). Interopérabilité des réseaux de capteurs hétérogènes dans un appartement intelligent. In *9èmes journées francophones Mobilité et Ubiquité*, UbiMob 2013.

[Comment ça marche QoS, 2014] Comment ça marche QoS (2014). Comment ça marche – QoS - Qualité de service. [Online : Web site]. Available at : `http://www.commentcamarche.net/contents/532-qos-qualite-de-service`.

[Dam and Langendoen, 2003] Dam, T. V. and Langendoen, K. (2003). An Adaptive Energy-efficient MAC Protocol for Wireless Sensor Networks. In *Proceedings of the 1st International Conference on Embedded Networked Sensor Systems*, SenSys '03, pages 171–180. ACM.

[Dargie and Poellabauer, 2010] Dargie, W. W. and Poellabauer, C. (2010). *Fundamentals of Wireless Sensor Networks : Theory and Practice*. Wireless Communications and Mobile Computing. John Wiley & Sons.

[DataSheet AT86RF231, 2009] DataSheet AT86RF231 (2009). *AT86RF231, Low Power 2.4 GHz Transceiver for ZigBee, IEEE 802.15.4, 6LoWPAN, RF4CE, SP100, WirelessHART, and ISM Applications*. Atmel, 8111C–MCU Wireless–09/09 edition.

[DataSheet ATmega128L, 2011] DataSheet ATmega128L (2011). *ATmega128/128L, 8-bit Atmel Microcontroller with 128 KBytes In-System Programmable Flash*. Atmel, Rev. 2467X–AVR–06/11 edition.

[DataSheet ATmega128RFA1, 2014] DataSheet ATmega128RFA1 (2014). *ATmega128RFA1, 8-bit AVR Microcontroller with Low Power 2.4 GHz Transceiver for ZigBee and IEEE 802.15.4*. Atmel, 8266F–MCU Wireless–09/14 edition.

[DataSheet ATmegaRFR2, 2014] DataSheet ATmegaRFR2 (2014). *ATmega256/128/64RFR2, 8-bit AVR Microcontroller with Low Power 2.4 GHz*

Transceiver for ZigBee and IEEE 802.15.4. Atmel, 8393C–MCU Wireless–09/14 edition.

[DataSheet CC2420, 2007] DataSheet CC2420 (2007). *CC2420 2.4 GHz IEEE 802.15.4 / ZigBee-ready RF Transceiver*. Chipcon Products from Texas Instruments, SWRS041B edition.

[DataSheet MicaZ, 2007] DataSheet MicaZ (2007). *MSP430F261x, MSP430F241x, Mixed Signal Microcontroller*. Crossbow, Document Part Number : 6020-0060-04 Rev A edition.

[DataSheet MSP430F1611, 2011] DataSheet MSP430F1611 (2011). *MSP430F15x, MSP430F16x, MSP430F161x Mixed Signal Microcontroller*. Texas Instruments, SLAS368G edition.

[DataSheet MSP430F2617, 2012] DataSheet MSP430F2617 (2012). *MSP430F261x, MSP430F241x, Mixed Signal Microcontroller*. Texas Instruments, SLAS541K edition.

[DataSheet STM32F103RE, 2011] DataSheet STM32F103RE (2011). *STM32F103xC, STM32F103xD, STM103xE Datasheet*. ST Microelectronics, Doc ID 14611 Rev 8 edition.

[DataSheet TelosB, 2006] DataSheet TelosB (2006). *TelosB Mote Platform*. Crossbow, Document Part Number : 6020-0094-01 Rev B edition.

[DataSheet Zolertia Z1, 2010] DataSheet Zolertia Z1 (2010). *Z1 Datasheet*. Zolertia, trademark of Advancare, S.L., v1.1 edition.

[DataSheet Zolertia Z1 Starter Platform, 2011] DataSheet Zolertia Z1 Starter Platform (2011). *Z1 Starter Platform Datasheet*. Zolertia, trademark of Advancare, S.L., v1.2 edition.

[de Oliveira et al., 2014] de Oliveira, B. T., Margi, C. B., and Gabriel, L. B. (2014). TinySDN : Enabling Multiple Controllers for Software-Defined Wireless Sensor Networks. In *LATINCOM 2014*, pages 1–6.

[Djamaa et al., 2014] Djamaa, B., Richardson, M., Aouf, N., and Walters, B. (2014). Towards Efficient Distributed Service Discovery in Low-Power and Lossy Networks. *Wireless Networks*, 20(8) :2437–2453.

[Dujovne et al., 2014] Dujovne, D., Watteyne, T., Vilajosana, X., and Thubert, P. (2014). 6TiSCH : deterministic IP-enabled industrial internet (of things). *Communications Magazine, IEEE*, 52(12) :36–41.

[Dunkels, 2003] Dunkels, A. (2003). Full TCP/IP for 8-bit architectures. In *Proceedings of the 1st International Conference on Mobile Systems, Applications and Services*, MobiSys '03, pages 85–98. ACM.

[Dunkels, 2007] Dunkels, A. (2007). Rime — a lightweight layered communication stack for sensor networks. In *EWSN, Poster/Demo session*.

[Dunkels, 2011] Dunkels, A. (2011). The ContikiMAC Radio Duty Cycling Protocol. Technical Report T2011 :13, Swedish Institute of Computer Science.

[Dunkels, 2015] Dunkels, A. (2015). Contiki 3.0 Released, New Hardware from Texas Instruments, Zolertia. [Online : blog of the Contiki OS project]. Available at : `http://contiki-os.blogspot.fr/2015/08/contiki-30-released-new-hardware-from.html`, See the "Internal Changes" section, 2nd paragraph.

[Dunkels et al., 2004] Dunkels, A., Grönvall, B., and Voigt, T. (2004). Contiki – a Lightweight and Flexible Operating System for Tiny Networked Sensors. In *IEEE 29th Conference on Local Computer Networks*, LCN '04, pages 455–462. IEEE Computer Society. `http://www.contiki-os.org/`.

[Dunkels et al., 2006] Dunkels, A., Schmidt, O., Voigt, T., and Ali, M. (2006). Protothreads : Simplifying Event-Driven Programming of Memory-Constrained Embedded Systems. In *Proceedings of the 4th International Conference on Embedded Networked Sensor Systems*, SenSys '06, pages 29–42. ACM.

[El-Hoiydi and Decotignie, 2004] El-Hoiydi, A. and Decotignie, J.-D. (2004). WiseMAC : an ultra low power MAC protocol for the downlink of infrastructure wireless sensor networks. In *Proceedings of the Ninth International Symposium on Computers and Communications*, volume 1 of *ISCC 2004*, pages 244–251. IEEE.

[Eriksson et al., 2009] Eriksson, J., Österlind, F., Finne, N., Tsiftes, N., Dunkels, A., Voigt, T., Sauter, R., and Marrón, J. (2009). COOJA/MSPSim : Interoperability Testing for Wireless Sensor Networks. In *Proceedings of the 2nd International Conference on Simulation Tools and Techniques*, Simutools '09, pages 27 :1–27 :7. ICST.

[Eswaran et al., 2005] Eswaran, A., Rowe, A., and Rajkumar, R. (2005). Nano-RK : an energy-aware resource-centric RTOS for sensor networks. In *Proceedings of the 26th IEEE International Real-Time Systems Symposium*, RTSS '05, page 265. IEEE. `http://nanork.org/`.

[Felemban et al., 2014] Felemban, E., Sheikh, A., and Manzoor, M. (2014). Improving Response Time in Time Critical Visual Sensor Network Applications. *Ad Hoc Networks*, 23 :65–79.

[Finne et al., 2014] Finne, N., Tsiftes, N., Oikonomou, G., Eriksson, J., Quattlebaum, R., and Dunkels, A. (2014). Extended radio API with support for setting channel, pan id, addressing modes, etc (#617). [Online : GitHub PR]. Available at : `https://github.com/contiki-os/contiki/pull/617`.

[FireFly3, 2012] FireFly3 (2012). FireFly Version 3. [Online : Web Site]. Available at : `http://www.nanork.org/projects/nanork/wiki/FireFly3`.

[FIT IoT-LAB, 2008] FIT IoT-LAB (lancé sous le nom de « Senslab » en 2008). IoT-LAB : a vary large scale open testbed (formerly named "Senslab"). [Online : Web Site and Online Testbed]. Available at : `https://iot-lab.info/`.

[FreeRTOS Timer Resolution, 2012] FreeRTOS Timer Resolution (2012). FreeRTOS Support Archive—Tick Rate vs Performance of RTOS. [Online : Forum]. `http://www.freertos.org/FreeRTOS_Support_Forum_Archive/May_2012/freertos_Tick_Rate_vs_Performance_of_RTOS_5283865.html`.

[Gaddour et al., 2014] Gaddour, O., Koubaa, A., Rangarajan, R., Cheikhrouhou, O., Tovar, E., and Abid, M. (2014). Co-RPL : RPL Routing for Mobile Low Power Wireless Sensor Networks Using Corona Mechanism. In *SIES 2014*, pages 200–209.

[Gay et al., 2003] Gay, D., Levis, P., von Behren, R., Welsh, M., Brewer, E., and Culler, D. (2003). The nesC Language : A Holistic Approach to Networked Embedded Systems. In *Proceedings of the ACM SIGPLAN 2003 Conference on Programming Language Design and Implementation*, PLDI '03, pages 1–11. ACM.

[Hahm et al., 2013] Hahm, O., Baccelli, E., Günes, M., Wählisch, M., and Schmidt, T. C. (2013). RIOT OS : Towards an OS for the Internet of Things. In *INFOCOM 2013, Poster Session*. `http://www.riot-os.org/`.

[Han et al., 2005] Han, C.-C., Kumar, R., Shea, R., Kohler, E., and Srivastava, M. (2005). SOS : A Dynamic Operating System for Sensor Nodes. In *Proceedings of the 3rd International Conference on Mobile Systems, Applications, and Services*, MobiSys '05, pages 163–176. ACM. `https://projects.nesl.ucla.edu/public/sos-2x/doc/`.

[Hoesel and Havinga, 2004] Hoesel, L. F. W. V. and Havinga, P. J. M. (2004). A Lightweight Medium Access Protocol (LMAC) for Wireless Sensor Networks : Reducing Preamble Transmissions and Transceiver State Switches. In *First International Conference on Networked Sensing Systems*, page 15. Society of Instrument and Control Engineers (SICE).

[Huang et al., 2013] Huang, P., Xiao, L., Soltani, S., Mutka, M. W., and Xi, N. (2013). The Evolution of MAC Protocols in Wireless Sensor Networks : A Survey. *Communications Surveys & Tutorials, IEEE*, 15(1) :101–120.

[IEEE 802.15.4, 2011] IEEE 802.15.4 (2011). IEEE Standard for Local and metropolitan area networks–Part 15.4 : Low-Rate Wireless Personal Area Networks (LR-WPANs). *IEEE Std 802.15.4-2011 (Revision of IEEE Std 802.15.4-2006)*, pages 1–314.

[IEEE 802.15.4e, 2012] IEEE 802.15.4e (2012). IEEE Standard for Local and metropolitan area networks–Part 15.4 : Low-Rate Wireless Personal Area Networks (LR-WPANs) amendment 1 : MAC sublayer. *IEEE Std 802.15.4e-2012 (Amendment to IEEE Std 802.15.4-2011)*, pages 1–225.

[Jeong et al., 2011] Jeong, J. S., Kim, J. S., and Mah, P. S. (2011). Design and implementation of low power wireless IPv6 routing for NanoQplus. In *Proceedings of the 13th International Conference on Advanced Communication Technology*, ICACT 2011, pages 966–971.

[Khssibi, 2015] Khssibi, S. (2015). *Utilisation des réseaux de capteurs de canne pour les applications de surveillance de personnes*. PhD thesis, Université de Toulouse-le-Mirail.

[Kim et al., 2007] Kim, S., Pakzad, S., Culler, D., Demmel, J., Fenves, G., Glaser, S., and Turon, M. (2007). Health Monitoring of Civil Infrastructures Using Wireless Sensor Networks. In *Proceedings of the 6th International Symposium on Information Processing in Sensor Networks*, IPSN '07, pages 254–263. ACM.

[Kim et al., 2008a] Kim, S. C., Kim, H. Y., Song, J. K., Yu, M. S., and Mah, P. S. (2008a). NanoQplus : A Multi-Threaded Operating System with Memory Protection Mechanism for WSNs. *Proceedings of the 1st China-Korea WSN Workshop (CKWSN)*, 20(08).

[Kim et al., 2008b] Kim, Y., Schmid, T., Charbiwala, Z. M., Friedman, J., and Srivastava, M. B. (2008b). NAWMS : Nonintrusive Autonomous Water Monitoring System. In *Proceedings of the 6th ACM Conference on Embedded Network Sensor Systems*, SenSys '08, pages 309–322. ACM.

[Knaian, 2000] Knaian, A. N. (2000). *A wireless sensor network for smart roadbeds and intelligent transportation systems*. PhD thesis, MIT Media Lab.

[Krishnamurthy et al., 2005] Krishnamurthy, L., Adler, R., Buonadonna, P., Chhabra, J., Flanigan, M., Kushalnagar, N., Nachman, L., and Yarvis, M. (2005). Design and Deployment of Industrial Sensor Networks : Experiences from a Semiconductor Plant and the North Sea. In *Proceedings of the 3rd International Conference on Embedded Networked Sensor Systems*, SenSys '05, pages 64–75. ACM.

[*US Department of Energy*, 2011] *US Department of Energy* (2011). Artificial Retina Project. `http://artificialretina.energy.gov`.

[Levis et al., 2003] Levis, P., Lee, N., Welsh, M., and Culler, D. (2003). TOSSIM : Accurate and Scalable Simulation of Entire TinyOS Applications. In *Proceedings of the 1st International Conference on Embedded Networked Sensor Systems*, SenSys '03, pages 126–137. ACM.

[Levis et al., 2005] Levis, P., Madden, S., Polastre, J., Szewczyk, R., Whitehouse, K., Woo, A., Gay, D., Hill, J., Welsh, M., Brewer, E., and Culler, D. (2005). TinyOS : An Operating System for Sensor Networks. In *Ambient Intelligence*, pages 115–148. Springer Berlin Heidelberg. `http://www.tinyos.net/`.

[Lockwood et al., 2007] Lockwood, J. W., McKeown, N., Watson., G., Gibb, G., Hartke, P., Naous, J., Raghuraman, R., and Luo, J. (2007). NetFPGA–An Open Platform for Gigabit-Rate Network Switching and Routing. In *IEEE International Conference on Microelectronic Systems Education, 2007*, MSE '07, pages 160–161. IEEE.

[LORIA / INRIA Nancy Grand-Est, 2009] LORIA / INRIA Nancy Grand-Est (depuis 2009). Informatique Située — Plate-forme « Appartement intelligent pour l'assistance à la personne ». [Research Facility]. Web page available at : `http://infositu.loria.fr/`.

[Manuel STM32F10x, 2011] Manuel STM32F10x (2011). *RM0008 Reference manual — STM32F101xx, STM32F102xx, STM32F103xx, STM32F105xx and STM32F107xx advanced ARM-based 32-bit MCUs*. ST Microelectronics, Doc ID 13902 Rev 14 edition.

[Manuel STMCortexM3, 2013] Manuel STMCortexM3 (2013). *PM0056 Programming manual — STM32F10xxx/20xxx/21xxx/L1xxx Cortex-M3 programming manual*. ST Microelectronics, Doc ID 15491 Rev 5 edition.

[Marques and Ricardo, 2014] Marques, B. and Ricardo, M. (2014). Improving the Energy Efficiency of WSN by Using Application-Layer Topologies to Constrain RPL-Defined Routing Trees. In *MED-HOC-NET 2014*, pages 126–133.

[Mulligan, 2007] Mulligan, G. (2007). The 6LoWPAN Architecture. In *Proceedings of the 4th Workshop on Embedded Networked Sensors*, EmNets '07, pages 78–82. ACM.

[Nano-RK Time Management API, 2015] Nano-RK Time Management API (version of 2015). Nano-RK Time Management API. [Online : Wiki Page]. `http://nanork.org/projects/nanork/wiki/Nrk-api-time-management/`.

[Narten et al., 2007] Narten, T., Nordmark, E., Simpson, E., and Soliman, H. (2007). Neighbor Discovery for IP version 6. RFC 4861, Internet Engineering Task Force (IETF).

[Nefzi, 2011] Nefzi, B. (2011). *Mécanismes auto-adaptatifs pour la gestion de la Qualité de Service dans les réseaux de capteurs sans fil.* PhD thesis, Networking and Internet Architecture. Institut National Polytechnique de Lorraine (INPL).

[Nefzi and Song, 2010] Nefzi, B. and Song, Y.-Q. (2010). CoSenS : a Collecting and Sending Burst Scheme for Performance Improvement of IEEE 802.15.4. In *IEEE 35th Conference on Local Computer Networks*, LCN '10, pages 172–175. IEEE Computer Society.

[Nefzi and Song, 2012] Nefzi, B. and Song, Y.-Q. (2012). QoS for Wireless Sensor Networks : Enabling Service Differentiation at the MAC Sub-layer Using CoSenS. *Ad Hoc Networks*, 10(4) :680–695.

[Oikonomou and Phillips, 2011] Oikonomou, G. and Phillips, I. (2011). Experiences from porting the Contiki operating system to a popular hardware platform. In *International Conference on Distributed Computing in Sensor Systems and Workshops*, DCOSS 2011, pages 1–6. IEEE.

[Österlind et al., 2006] Österlind, F., Dunkels, A., Eriksson, J., Finne, N., and Voigt, T. (2006). Cross-Level Sensor Network Simulation with Cooja. In *IEEE 31st Conference on Local Computer Networks*, LCN '06, pages 641–648. IEEE Computer Society.

[Österlind et al., 2012] Österlind, F., Mottola, L., Voigt, T., Tsiftes, N., and Dunkels, A. (2012). Strawman : Resolving Collisions in Bursty Low-power Wireless Networks. In *Proceedings of the 11th International Conference on Information Processing in Sensor Networks*, IPSN '12, pages 161–172. ACM.

[Palattella et al., 2014] Palattella, M., Thubert, P., Vilajosana, X., Watteyne, T., Wang, Q., and Engel, T. (2014). 6TiSCH Wireless Industrial Networks : Determinism Meets IPv6. In *Internet of Things*, volume 9 of *Smart Sensors, Measurement and Instrumentation*, pages 111–141. Springer International Publishing.

[Papadopoulos et al., 2014] Papadopoulos, G. Z., Beaudaux, J., Gallais, A., and Noel, T. (2014). T-AAD : Lightweight traffic auto-adaptations for low-power MAC protocols. In *13th Annual Mediterranean Ad Hoc Networking Workshop (MED-HOC-NET), 2014*, pages 79–86.

[Papadopoulos et al., 2015a] Papadopoulos, G. Z., Gallais, A., Schreiner, G., and Noel, T. (2015a). Live Adaptations of Low-power MAC Protocols. In *Proceedings of the 2015 Workshop on Wireless of the Students, by the Students, & for the Students*, pages 38–40. ACM.

[Papadopoulos et al., 2015b] Papadopoulos, G. Z., Kotsiou, V., Gallais, A., Chatzimisios, P., and Noel, T. (2015b). Wireless Medium Access Control under Mobility and Bursty Traffic Assumptions in WSNs. *Mobile Networks and Applications*, 20(5) :649–660.

[Polastre et al., 2004] Polastre, J., Hill, J., and Culler, D. (2004). Versatile Low Power Media Access for Wireless Sensor Networks. In *Proceedings of the 2nd International Conference on Embedded Networked Sensor Systems*, SenSys '04, pages 95–107. ACM.

[Porter and Coulson, 2009] Porter, B. and Coulson, G. (2009). Lorien : a Pure Dynamic Component-Based Operating System for Wireless Sensor Networks. In *Proceedings of the 4th International Workshop on Middleware Tools, Services and Run-Time Support for Sensor Networks*, MidSens '09, pages 7–12. ACM. `http://lorienos.sourceforge.net/`.

[Real Time Engineers Ltd., 2013] Real Time Engineers Ltd., editor (2013). *Using the FreeRTOS^{TM}Real Time Kernel — API Functions and Configuration Options*. Real Time Engineers Ltd., version 1.4.1 edition.

[Roussel, 2014] Roussel, K. (2014). Implemeting a real-time MAC protocol under RIOT OS : running on Zolertia Z1 motes. In *Workshop Internet of Things / Equipex FIT IoT-LAB*. INRIA Grenoble Rhone-Alpes.

[Roussel et al., 2014a] Roussel, K., Alvira, M., Eriksson, J., Oikonomou, G., Schmidt, O., Dunkels, A., and Tsiftes, N. (2014a). Radio API extension (#519). [Online : GitHub PR]. Available at : `https://github.com/contiki-os/contiki/pull/519`.

[Roussel and Hahm, 2014a] Roussel, K. and Hahm, O. (2014a). Add a missing constant in 'radio_tx_status_t' enum (#1239). [Online : GitHub PR]. Available at : `https://github.com/RIOT-OS/RIOT/pull/1239`.

[Roussel and Hahm, 2014b] Roussel, K. and Hahm, O. (2014b). boards/z1 : fix cc2420_txrx function in CC2420 driver HAL (#1388). [Online : GitHub PR]. Available at : `https://github.com/RIOT-OS/RIOT/pull/1388`.

[Roussel and Hahm, 2014c] Roussel, K. and Hahm, O. (2014c). Completing low-level radio driver definition (#1211). [Online : GitHub PR]. Available at : `https://github.com/RIOT-OS/RIOT/pull/1211`.

[Roussel and Hahm, 2014d] Roussel, K. and Hahm, O. (2014d). Fix for CC2420 radio driver for TelosB (#954). [Online : GitHub PR]. Available at : `https://github.com/RIOT-OS/RIOT/pull/954`.

[Roussel et al., 2014b] Roussel, K., Hahm, O., and Eichinger, T. (2014b). CC2420 radio transceiver's driver fixes (#882). [Online : GitHub PR]. Available at : `https://github.com/RIOT-OS/RIOT/pull/882`.

[Roussel et al., 2014c] Roussel, K., Hahm, O., Eichinger, T., and Mehlis, C. (2014c). Modify & extend CC2420 driver to comply with API described in 'radio_driver.h' (#1223). [Online : GitHub PR]. Available at : https://github.com/RIOT-OS/RIOT/pull/1223.

[Roussel et al., 2013a] Roussel, K., Hahm, O., and Knüpfer, L. (2013a). Msp430 lpm freq (#459). [Online : GitHub PR]. Available at : https://github.com/RIOT-OS/RIOT/pull/459.

[Roussel et al., 2014d] Roussel, K., Hahm, O., and Knüpfer, L. (2014d). Handle race conditions preventing MSP430 timers to be set correctly (#957). [Online : GitHub PR]. Available at : https://github.com/RIOT-OS/RIOT/pull/957.

[Roussel et al., 2014e] Roussel, K., Hahm, O., and Knüpfer, L. (2014e). Msp430 misc interrupt-related fixes (#2214). [Online : GitHub PR]. Available at : https://github.com/RIOT-OS/RIOT/pull/2214.

[Roussel et al., 2014f] Roussel, K., Hahm, O., and Knüpfer, L. (2014f). Use Timer B on MSP430 architecture (#1113). [Online : GitHub PR]. Available at : https://github.com/RIOT-OS/RIOT/pull/1113.

[Roussel et al., 2014g] Roussel, K., Hahm, O., Knüpfer, L., Eichinger, T., and Lenders, M. (2014g). Proposal for common 802.15.4 radio driver API definition (#925). [Online : GitHub PR]. Available at : https://github.com/RIOT-OS/RIOT/pull/925.

[Roussel et al., 2014h] Roussel, K., Hahm, O., Knüpfer, L., Mehlis, C., Eichinger, T., and Petersen, H. (2014h). Zolertia Z1 port for RIOT OS (#893). [Online : GitHub PR]. Available at : https://github.com/RIOT-OS/RIOT/pull/893.

[Roussel et al., 2014i] Roussel, K., Hahm, O., Schleiser, K., Knüpfer, L., Mehlis, C., and Kijewski, R. (2014i). Panic (#685). [Online : GitHub PR]. Available at : https://github.com/RIOT-OS/RIOT/pull/685.

[Roussel et al., 2014j] Roussel, K., Kijewski, R., and Hahm, O. (2014j). Ensure that stack pointer is correctly aligned during thread creation on MSP430 (#881). [Online : GitHub PR]. Available at : https://github.com/RIOT-OS/RIOT/pull/881.

[Roussel et al., 2014k] Roussel, K., Kijewski, R., Hahm, O., and Knüpfer, L. (2014k). Reboot (#724). [Online : GitHub PR]. Available at : https://github.com/RIOT-OS/RIOT/pull/724.

[Roussel et al., 2014l] Roussel, K., Kijewski, R., Hahm, O., Knüpfer, L., and Mehlis, C. (2014l). Add a reboot() function to kernel.h definitions (#687). [Online : GitHub PR]. Available at : https://github.com/RIOT-OS/RIOT/pull/687.

[Roussel et al., 2014m] Roussel, K., Kijewski, R., Petersen, H., Knüpfer, L., Mehlis, C., and Hahm, O. (2014m). Reset ARM Cortex-M3 MCUs before flashing (#1380). [Online : GitHub PR]. Available at : https://github.com/RIOT-OS/RIOT/pull/1380.

[Roussel et al., 2014n] Roussel, K., Kijewski, R., and Schleiser, K. (2014n). Portable definition of function attributes (#689). [Online : GitHub PR]. Available at : https://github.com/RIOT-OS/RIOT/pull/689.

[Roussel et al., 2014o] Roussel, K., Kijewski, R., Schleiser, K., Knüpfer, L., Hahm, O., and Mehlis, C. (2014o). core : Add the ability to send a message to the current thread's message queue (#970). [Online : GitHub PR]. Available at : `https://github.com/RIOT-OS/RIOT/pull/970`.

[Roussel et al., 2014p] Roussel, K., Knüpfer, L., Eichinger, T., Hahm, O., and Kijewski, R. (2014p). Add a standard way to query CCA status on CC2420 transceiver (#915). [Online : GitHub PR]. Available at : `https://github.com/RIOT-OS/RIOT/pull/915`.

[Roussel et al., 2014q] Roussel, K., Knüpfer, L., and Hahm, O. (2014q). Fix a design error in cc2420_do_send() function (#1383). [Online : GitHub PR]. Available at : `https://github.com/RIOT-OS/RIOT/pull/1383`.

[Roussel et al., 2014r] Roussel, K., Knüpfer, L., and Hahm, O. (2014r). Fix a nasty race condition in CCA determination on CC2420 (#1385). [Online : GitHub PR]. Available at : `https://github.com/RIOT-OS/RIOT/pull/1385`.

[Roussel et al., 2013b] Roussel, K., Knüpfer, L., Hahm, O., and Eichinger, T. (2013b). Simplify msp430 headers (#408). [Online : GitHub PR]. Available at : `https://github.com/RIOT-OS/RIOT/pull/408`.

[Roussel et al., 2014s] Roussel, K., Knüpfer, L., Hahm, O., and van Bruinehsen, H. (2014s). Fix thread_yield() on MSP430 platforms (#1618). [Online : GitHub PR]. Available at : `https://github.com/RIOT-OS/RIOT/pull/1618`.

[Roussel et al., 2014t] Roussel, K., Knüpfer, L., Hahm, O., van Bruinehsen, H., and Lenders, M. (2014t). Only use 16 significative bits for MSP430 hwtimers (#1619). [Online : GitHub PR]. Available at : `https://github.com/RIOT-OS/RIOT/pull/1619`.

[Roussel et al., 2014u] Roussel, K., Knüpfer, L., Kijewski, R., Hahm, O., Mehlis, C., and Schleiser, K. (2014u). Enhance implementation of hwtimer_spin() (#1002). [Online : GitHub PR]. Available at : `https://github.com/RIOT-OS/RIOT/pull/1002`.

[Roussel et al., 2014v] Roussel, K., Knüpfer, L., Petersen, H., and Schleiser, K. (2014v). Ensure hwtimer_spin() won't wait for an unreachable stop counter value (#1617). [Online : GitHub PR]. Available at : `https://github.com/RIOT-OS/RIOT/pull/1617`.

[Roussel et al., 2013c] Roussel, K., Oikonomou, G., Alvira, M., Kopf, D., Sawadski, V., Eriksson, J., Bigot, P. A., and Dunkels, A. (2013c). Extended radio drivers API (#192). [Online : GitHub PR]. Available at : `https://github.com/contiki-os/contiki/pull/192`.

[Roussel et al., 2015a] Roussel, K., Remmert, J., and Hahm, O. (2015a). Add a netopt for getting and setting CCA threshold (#4138). [Online : GitHub PR]. Available at : `https://github.com/RIOT-OS/RIOT/pull/4138`.

[Roussel and Song, 2013] Roussel, K. and Song, Y.-Q. (2013). A critical analysis of Contiki's network stack for integrating new MAC protocols. Rapport de recherche RR-8776, INRIA. ID hal-01202542.

[Roussel et al., 2015b] Roussel, K., Song, Y.-Q., and Zendra, O. (2015b). Lessons Learned through Implementation and Performance Comparison of Two MAC/RDC Protocols on Different WSN OS. Rapport de recherche RR-8777, INRIA. ID hal-01202664.

[Roussel et al., 2015c] Roussel, K., Song, Y.-Q., and Zendra, O. (2015c). RIOT OS Paves the Way for Implementation of High-performance MAC Protocols. In *Proceedings of the 4th International Conference on Sensor Networks*, SensorNets 2015, pages 5–14. SCITEPRESS.

[Roussel et al., 2016] Roussel, K., Song, Y.-Q., and Zendra, O. (2016). Using Cooja for WSN Simulations : Some New Uses and Limits. In *Proceedings of the 2016 International Conference on Embedded Wireless Systems and Networks*, EWSN '16, pages 319–324. ACM / Junction Publishing.

[Schwiebert et al., 2001] Schwiebert, L., Gupta, S. K. S., and Weinmann, J. (2001). Research Challenges in Wireless Networks of Biomedical Sensors. In *Proceedings of the 7th Annual International Conference on Mobile Computing and Networking*, MobiCom '01, pages 151–165. ACM.

[Seo et al., 2015] Seo, S.-H., Won, J., Sultana, S., and Bertino, E. (2015). Effective Key Management in Dynamic Wireless Sensor Networks. *Information Forensics and Security, IEEE Transactions on*, 10(2) :371–383.

[Sha et al., 2013] Sha, M., Hackmann, G., and Lu, C. (2013). Energy-Efficient Low Power Listening for Wireless Sensor Networks in Noisy Environments. In *Proceedings of the 12th international conference on Information processing in sensor networks*, IPSN '13, pages 277–288. ACM.

[Song, 2013] Song, Y.-Q. (2012–2013). Protocoles MAC à « duty-cycle » et (un peu de) routage pour les réseaux de capteurs sans-fil. Université de Lorraine — ENSEM. Cours « Systèmes communicants contraints ».

[Stoianov et al., 2007] Stoianov, I., Nachman, L., Madden, S., Tokmouline, T., and Csail, M. (2007). PIPENET : A Wireless Sensor Network for Pipeline Monitoring. In *Proceedings of the 6th International Symposium on Information Processing in Sensor Networks*, IPSN '07, pages 264–273. ACM.

[Strazdins et al., 2010] Strazdins, G., Elsts, A., and Selavo, L. (2010). MansOS : Easy to Use, Portable and Resource Efficient Operating System for Networked Sensor Systems. In *Proceedings of the 8th ACM Conference on Embedded Networked Sensor Systems*, Sensys '10, pages 427–428. ACM. `http://mansos.edi.lv/`.

[Sun et al., 2008] Sun, Y., Gurewitz, O., and Johnson, D. B. (2008). RI-MAC : A Receiver-initiated Asynchronous Duty Cycle MAC Protocol for Dynamic Traffic Loads in Wireless Sensor Networks. In *Proceedings of the 6th International Conference on Embedded Networked Sensor Systems*, SenSys '08, pages 1–14. ACM.

[Thubert et al., 2013] Thubert, P., Watteyne, T., Palattella, M., Vilajosana, X., and Wang, Q. (2013). IETF 6TSCH : Combining IPv6 Connectivity with Industrial Performance. In *Seventh International Conference on Innovative Mobile and Internet Services in Ubiquitous Computing*, IMIS 2013, pages 541–546.

[Titzer et al., 2005] Titzer, B. L., Lee, D., and Palsberg, J. (2005). Avrora : Scalable Sensor Network Simulation with Precise Timing. In *Proceedings of the 4th International Symposium on Information Processing in Sensor Networks*, IPSN '05. IEEE Press. Article no. 67.

[Udit, 2012] Udit, J. (2012). Jeremy's blog – Contiki OS article. [Online : personal blog of Jeremy Udit]. Available at : `http://jeremyudit.blogspot.fr/2012/08/contiki-os.html`.

[van den Bossche and Val, 2013] van den Bossche, A. and Val, T. (2013). WiNo : une plateforme d'émulation et de prototypage rapide pour l'ingénierie des protocoles en réseaux de capteurs sans fil. In *9èmes journées francophones Mobilité et Ubiquité*, UbiMob 2013.

[Wang et al., 2008] Wang, M.-M., Cao, J.-N., Li, J., and Dasi, S. K. (2008). Middleware for Wireless Sensor Networks : A Survey. *Journal of Computer Science and Technology*, 23(3) :305–326.

[Watteyne et al., 2012] Watteyne, T., Vilajosana, X., Kerkez, B., Chraim, F., Weekly, K., Wang, Q., Glaser, S., and Pister, K. (2012). OpenWSN : a standards-based low-power wireless development environment. *Transactions on Emerging Telecommunications Technologies*, 23(5) :480–493.

[Werner-Allen et al., 2006] Werner-Allen, G., Lorincz, K., Ruiz, M., Marcillo, O., Johnson, J., Lees, J., and Welsh, M. (2006). Deploying a wireless sensor network on an active volcano. *IEEE Internet Computing*, 10(2) :18–25.

[Wu et al., 2015] Wu, X., Brown, K., and Sreenan, C. (2015). Contact Probing Mechanisms for Opportunistic Sensor Data Collection. *The Computer Journal*.

[Ye et al., 2002] Ye, W., Heidemann, J., and Estrin, D. (2002). An Energy-Efficient MAC Protocol for Wireless Sensor Networks. In *Proceedings of the 21st Annual Joint Conference of the IEEE Computer and Communication Societies*, volume 3 of *INFOCOM 2002*, pages 1567–1576. IEEE.

[Zhang et al., 2004] Zhang, P., Sadler, C. M., Lyon, S. A., and Martonosi, M. (2004). Hardware Design Experiences in ZebraNet. In *Proceedings of the 2nd International Conference on Embedded Networked Sensor Systems*, SenSys '04, pages 227–238. ACM.

[Zhuo et al., 2013] Zhuo, S., Wang, Z., Song, Y.-Q., Wang, Z., and Almeida, L. (2013). iQueue-MAC : A Traffic Adaptive duty-cycled MAC Protocol with Dynamic Slot Allocation. In *IEEE 10th Conference on Sensor, Mesh, and Ad Hoc Communications and Networks*, SECON 2013, pages 95–103. IEEE Communications Society.

Glossaire

0–9

6LoWPAN *"IPv6 over Low-Power Wireless Personal Area Network"*
Acronyme anglo-saxon, désignant une version « allégée » du protocole IP version 6, adaptée aux réseaux de capteurs sans-fil, notamment aux ressources limitées des matériels (nœuds) les composant.

802.15.4
Terme raccourci désignant le standard IEEE 802.15.4 (et en général ses différentes annexes et extensions). Il définit une technologie sur laquelle repose de nombreux réseaux de capteurs sans-fil, notamment ceux auxquels les travaux de la présente thèse se sont intéressés.

A

Actuator
Terme anglo-saxon, désignant en français un **actionneur** capable d'influer sur un phénomène physique environnemental. La présence de tels actionneurs sur un nœud équipe d'un émetteur / récepteur radio est ce qui le fait désigner sous le terme de « actionneur sans-fil » (*"Wireless Actuator"* en anglais). Les nœuds équipés d'actionneurs sont sauf exception également équipés de capteurs, ce qui fait que le terme de « capteur sans-fil » (*"Wireless Sensor"*) reste presque toujours utilisé, même en présence d'actionneurs.

API *"Application Programming Interface"*
Acronyme anglo-saxon, se traduisant en français par **« Interface de Programmation d'Applications »**. Désigne l'ensemble des fonctions, types, classes, etc., offerts aux développeurs pour construire leurs applications au sein d'un environnement logiciel particulier (par exemple : système d'exploitation, ou bibliothèque offrant un jeu de fonctionnalités).

B

BAN *"Body Area Network"*
Acronyme anglo-saxon, se traduisant en français par **« Réseau d'Étendue**

Corporelle ». Désigne un type particulier de PAN, conçu pour regrouper des capteurs et actionneurs sur le corps d'un individu. Ce terme possède — notamment dans le cadre de la présente thèse — une forte connotation médicale et d'aide à la personne (malade, agée, dépendante, etc.).

Beacon
Terme anglo-saxon, se traduisant en français par « **balise** ». Désigne une trame ne contenant pas de données applicatives (*"payload"*), mais servant à la synchronisation des différents nœuds d'un réseau de capteurs sans-fil. De telles « balises » ou *"beacons"* sont notamment utilisées par les protocoles MAC / RDC S-CoSenS et le protocole standard IEEE 802.15.4 (dans un de ses deux modes de fonctionnement).

C

CCA *"Clear Channel Assessment"*
Acronyme anglo-saxon, désignant la procédure consistant à vérifier la disponibilité du médium physique, juste avant l'envoi d'un message.

Contiki (OS)
L'une des plates-formes logicielles spécialisées parmi les plus utilisées dans le domaine des WSN, étant actuellement la référence de fait dans le domaine. Elle a fait l'objet de plusieurs travaux et études dans le présente thèse.

ContikiMAC
Protocole MAC / RDC, conçu par le créateur de Contiki OS, et désormais utilisé par défaut avec ce dernier depuis plusieurs années. Il s'agit d'un protocole asynchrone de type LPL, ayant fait l'objet de plusieurs optimisations conceptuelles et implantatoires. Compte-tenu de l'influence majeure de Contiki OS dans le domaine des WSN, ce protocole est devenu une référence par défaut dans le domaine, face à laquelle il est courant de tester les nouveaux protocoles MAC /RDC.

CoSenS
Protocole MAC, conçu au sein de notre équipe de recherche (Madynes). Est la base et le précurseur de S-CoSenS, lequel en est une extension majeure ajoutant la gestion du RDC.

CPU *"Central Processing Unit"*
Acronyme anglo-saxon, désignant un **cœur de microprocesseur**, autour duquel (ou desquels) est construit tout système informatique.

Ces cœurs CPU peuvent être associés avec divers périphériques au sein d'un même circuit, comme un microcontrôleur.

CSMA/CA *"Carrier Sense Multiple Access with Collision Avoidance"*
Acronyme anglo-saxon, désignant une procédure, basée sur la contention, destinée à gérer l'envoi correct d'un paquet de données sur le médium radio.

À noter que cette procédure, utilisée par la couche MAC standard du standard IEEE 802.15.4, est différente d'une procedure homonyme utilisée dans le standard WiFi (IEEE 802.11).

D

Driver
Terme anglo-saxon, se traduisant en français par « **pilote** ». Élement logiciel, en général de bas niveau, permettant d'accéder aux fonctionnalités d'un matériel donné (comme un périphérique).

E

ED *"Energy Detection"*
Acronyme anglo-saxon, se traduisant en français par « **Détection d'énergie** ». Mesure chiffrée de la puissance du signal sur le médium radio, notamment par rapport au bruit de fond. Sert souvent à déterminer la disponibilité ou non de ce médium, lors d'une procédure de CCA.

F

FDMA *"Frequency Division Multiple Access"*
Acronyme anglo-saxon, désignant une procédure, basée sur le multiplexage fréquentiel, destinée à gérer l'envoi correct d'un paquet de données sur le médium radio.

Employée par de nombreux protocoles MAC alternatifs, conçus par la communauté académique ou industrielle, mais aussi par l'amendement 802.15.4e du standard IEEE.

FFD *"Full Function Device"*
Acronyme anglo-saxon, issu du standard IEEE 802.15.4, désignant un matériel suffisamment puissant pour jouer le rôle de coordinateur de PAN — et par extension, bien souvent de routeur dans les couches supérieures de la pile protocolaire, voire même assurer d'autres fonctions de plus haut niveau comme par exemple des opérations de calcul. Ce terme est opposé à celui de RFD.

FPGA *"Field Programmable Gate Array"*
Acronyme anglo-saxon, désignant un circuit intégré dont l'organisation interne, et donc les fonctionnalités, peuvent être programmées et reprogrammées arbitrairement selon les besoins. Cela en fait de formidables outils de prototypage matériel, mais aussi, avec l'augmentation de leur puissance et la baisse de leur prix, de futures alternatives potentielles aux circuits « dédiés » (comme les MCU par exemple) pour construire des systèmes électroniques et informatiques, comme les nœuds de réseaux de capteurs sans-fil.

G

GPIO *"General Purpose Input/Output"*
Acronyme anglo-saxon, désignant une entrée / sortie à vocation généraliste. Dans notre contexte, il s'agit d'une broche de MCU jouant ce rôle d'entrée / sortie librement utilisable et programmable.

H

HAL *"Hardware Abstraction Layer"*
Acronyme anglo-saxon, se traduisant en français par « **Couche d'Abstraction Matérielle** ». Désigne le regroupement des éléments logiciels dépendants du matériel (en général les pilotes ou *"drivers"*) dans une couche clairement séparée, permettant au reste du logiciel — en général un système d'exploitation, ou un exécutif comparable — d'être conçu de façon générique et donc plus facilement portable.

I

I²C *"Inter-Integrated Circuit"*
Acronyme anglo-saxon, désignant une technologie de bus série, reliant des circuits au sein d'un même appareil électronique (parfois même au sein d'un même microcontrôleur).

IoT *"Internet of Things"*
Acronyme anglo-saxon, se traduisant en français par « **Internet des Objets** ». Ce terme désigne la possibilité des réseaux de capteurs sans-fil de s'interconnecter entre eux et avec Internet, par l'utilisation du protocole IP (*"Internet Protocol"*), notamment en version 6 (IPv6), comme couche 3 de leur pile protocolaire.

J

JTAG *"Joint Test Action Group"*
Acronyme anglo-saxon, désignant la norme IEEE 1149.1. Désigne (par abus de langage) dans le présent manuscrit, la possibilité d'effectuer du déboguage au sein du matériel lui-même. Dans notre contexte, il s'agit de déboguer les programmes téléchargés au sein des nœuds de réseaux de capteurs sans-fil ***lors*** de leur exécution, ce qui facilité grandement la compréhension et la résolution de bogues et autres problèmes.

L

LAR *"Living Assistant Robot"*
Acronyme désignant le projet au sein duquel s'est déroulé cette thèse.

LPL *"Low-Power Listening"*
Acronyme anglo-saxon, désignant une famille de protocoles MAC / RDC aynchrones, basés sur la contention, où l'initiative de la transmission des données est laissée aux nœuds émetteurs.

LPP *"Low-Power Probing"*
Acronyme anglo-saxon, désignant une famille de protocoles MAC / RDC aynchrones, basés sur la contention, où l'initiative de la transmission des données est laissée aux nœuds receveurs, via des mécanismes de transmissions régulières de « balises » afin de mieux économiser l'énergie et exploiter la bande passante du médium radio.

LQI *"Link Quality Indicator"*
Acronyme anglo-saxon, se traduisant en français par **« Qualité de Liaison »**. Mesure chiffrée de la qualité de transmission sur un médium, dans le contexte qui nous intéresse le médium radio. Il s'agit d'un indicateur de qualité de service.

M

MAC *"Medium Access Control"*
Acronyme anglo-saxon, se traduisant en français par **« Contrôle d'Accès au Médium »**. Représente la couche 2 de la pile protocolaire dans le modèle réseau OSI (ou précisément, la partie la plus basse de cette couche 2).

Cette couche est chargée de contrôler l'accès au médium du réseau, dans notre cas le médium radio, afin de s'assurer de la bonne qualité des transmissions, et notamment de l'absence de collisions. À cette fin, de nombreux protocoles ont été développés pour implanter cette couche 2 : les protocoles MAC. Certains sont issus de standards comme IEEE 802.15.4, d'autres sont le résultat de recherches académiques et/ou industrielles.

MCU *"MicroController Unit"*
Acronyme anglo-saxon, désignant un **Microcontrôleur**, c'est-à-dire un circuit regroupant processeur central et plusieurs périphériques de base intégrés, comme des *timers*, des entrées-sorties avec gestion de bus de communication intégrée, des convertisseurs entre signaux analogiques et digitaux, etc.

De tels circuits sont utilisés massivement dans le domaine de l'informatique embarquée, ainsi que dans les noeuds de WSN, dont ils constituent le « cœur ».

Mote
Terme anglo-saxon désignant — dans le cadre du présent manuscrit de thèse — un nœud d'un réseau de capteurs sans-fil.

MPU *"Memory Protection Unit"*
Acronyme anglo-saxon, désignant une **unité de protection mémoire**. Il s'agit d'un circuit dédié, optionnellement intégré à un microcontrôleur, afin de réserver certaines parties de la mémoire à certaines parties de code, notamment les parties de code s'exécutant dans un « mode privilégié » (parfois aussi appelé « superviseur », « système », etc.) que le microcontrôleur est susceptible de fournir.

La présence d'une MPU dans un microcontrôleur facilite ainsi la détection des erreurs liées à la mémoire (débordements, accès anormaux, etc.), et aide grandement à la réalisation de systèmes fiables et robustes.

O

OS *"Operating System"*
Acronyme anglo-saxon, se traduisant en français par **« Système d'Exploitation »**.

P

PAN *"Personal Area Network"*
Acronyme anglo-saxon, se traduisant en français par **« Réseau d'Étendue Personnel »**. Désigne en général un réseau de capteurs sans-fil élémentaire, regroupé autour de et contrôlé par un nœud coordinateur spécifique.

PHY « Couche PHYsique »
Diminutif désignant la couche 1 de la pile protocolaire dans le modèle réseau OSI, qui consiste en général en les pilotes — *"drivers"* — des émetteurs / récepteurs radio, dans le cadre des réseaux sans-fil auquel se consacre la présente thèse.

PRR *"Packet Reception Rate"*
Acronyme anglo-saxon, se traduisant en français par **« Taux de Réception de Paquets »**. Désigne la fraction de paquets transmise avec succès de leur émetteur à leur destinataire final. Il s'agit bien évidemment d'un critère essentiel de QdS.

Q

QdS **« Qualité de Service »**
Acronyme francophone, traduction de l'anglo-saxon *"Quality of Service"*.

QoS *"Quality of Service'*
Acronyme anglo-saxon, se traduisant en français par **« Qualité de Service »**.

R

RDC *"Radio Duty Cycle"*
Acronyme anglo-saxon, se traduisant en français par « **Cycle d'Activité de la Radio** ». Désigne la stratégie d'activation / désactivation cyclique de l'émetteur / récepteur radio d'un nœud de réseau de capteurs sans-fil, afin d'économiser au mieux l'énergie de ce nœud (la radio étant le circuit le plus gourmand en énergie dans de tels appareils), tout en assurant la bonne transmission des informations sur le réseau (c'est-à-dire que la radio doit rester active aux moments nécessaires). Cette stratégie dait l'objets de protocoles, en général intégrés aux protocoles MAC, lesquels deviennent alors des protocoles MAC /RDC — assurant ces deux fonctions liées et complémentaires.

Par extension, *"duty cycle"* désigne également la fraction cyclique d'activation de la radio (c-à-d., pour simplifier, le pourcentage d'activité radio) résultant de l'exécution d'un protocole RDC donné.

RFD *"Reduced Function Device"*
Acronyme anglo-saxon, issu du standard IEEE 802.15.4, désignant un matériel aux ressources limitées, incapable de jouer le rôle de coordinateur de PAN — et par conséquent, limité à celui de « noeud-feuille », se contentant en général d'interagir avec l'environnement physique avec ses capteurs et (éventuellement) ses actionneurs, et de transmettre les données physiques ainsi reçues sur le réseau, à l'exclusion de toute tâche (plus) complexe. Ce terme est opposé à celui de FFD.

RIOT OS
L'une des plates-formes logicielles parmi les plus avancées dans le domaine des WSN. Faisant l'objet d'un développement et d'une diffusion rapide, elle est au centre de nombreux travaux et études dans le présente thèse.

RSSI *"Received Signal Strength Indicator"*
Acronyme anglo-saxon, désignant la puissance efficace du signal radio lors d'une transmission donnée, notamment par rapport au bruit de fond ambiant. Il s'agit d'un indicateur important de qualité de service.

S

S-CoSenS
Protocole MAC / RDC, conçu au sein de notre équipe de recherche (Madynes). Dérivé évolué de CoSenS, son implantation et son amélioration, au sein de plates-formes logicielles adaptées (comme RIOT OS) ont constitué le cœur des travaux effectués au sein de la présente thèse. Il semble notamment offir des propriétés de QdS supérieures à celles de ContikiMAC, notamment en présence d'un trafic réseau intense.

SoC *"System-on-Chip"*
Acronyme anglo-saxon, se traduisant en français par « **Système sur une**

puce ». Désigne un système informatique complet regroupé sur un seul circuit intégré. Par rapport à un microcontrôleur, la logique est poussée à son maximum, le SoC étant normalement à lui seul un matériel informatique complet et auto-suffisant, là où le microcontrôleur est plutôt destiné à être associé à d'autres circuits ou éléments annexes pour former un autre appareil (système informatique, ou appareil industriel, électroménager, etc.)

Sensor
Terme anglo-saxon, désignant en français un **capteur** capable de mesurer un phénomène physique de l'environnement. La présence de tels capteurs sur un nœud équipé d'un émetteur / récepteur radio est ce qui le fait désigner sous le terme de « capteur sans-fil » (*"Wireless Sensor"* en anglais).

SNR *"Signal/Noise Ratio"*
Acronyme anglo-saxon, se traduisant en français par **« Rapport Signal sur Bruit »**. Désigne la force d'un signal utile par rapport au bruit ambiant (interférences) sur un médium radio donné. Ce paramètre physique a donc bien évidemment une influence primordiale sur la QdS. La nature même du médium radio rend ce paramètre SNR extrêmement variable dans l'espace et le temps, ce qui rend le dit médium peu fiable, ***et*** difficile à simuler par logiciel.

SPI *"Serial Peripheral Interface"*
Acronyme anglo-saxon, désignant une technologie de bus série, reliant des circuits au sein d'un même appareil électronique (parfois même au sein d'un même microcontrôleur). Souvent employé au sein des nœuds de réseaux de capteurs sans-fil pour relier le microcontrôleur central à l'émetteur / récepteur radio.

T

TDMA *"Time Division Multiple Access"*
Acronyme anglo-saxon, désignant une procédure, basée sur le multiplexage temporel, destinée à gérer l'envoi correct d'un paquet de données sur le médium radio.

Employée par de nombreux protocoles MAC alternatifs, conçus par la communauté académique ou industrielle.

Transceiver
Terme anglo-saxon, diminutif de *"Radio Transceiver"*, désignant en français un **émetteur / récepteur radio**.

TRP « Taux de Réception de Paquets »
Acronyme francophone, se traduisant en anglais par *"Packet Reception Rate"*. Désigne la fraction de paquets transmise avec succès de leur émetteur à leur destinataire final. Il s'agit bien évidemment d'un critère essentiel de QdS.

W

WAN *"Wide Area Network"*
Acronyme anglo-saxon, se traduisant en français par « Réseau de Vaste Étendue ». Désigne en général l'Internet.

WSN *"Wireless Sensor Network"*
Acronyme anglo-saxon, se traduisant en français par **« Réseau de Capteurs Sans-Fil »**.

Résumé

Dans le domaine des réseaux de capteurs sans-fil (dits « WSN »), les piles réseau spécialisées constituent un domaine de recherche très actif depuis maintenant de nombreuses années. Toutefois, beaucoup de ces études, notamment concernant les couches basses de ces piles réseau, n'ont pas dépassé le stade de la théorie. Leurs implantations n'ont sauf exception pas fait l'objet d'efforts poussés ou systématiques, surtout dans le cadre des systèmes d'exploitation spécialisés. Nous nous proposons donc, dans cette thèse, de nous focaliser sur l'analyse des interactions entre les protocoles des couches basses et les plates-formes logicielles dédiées, et de les optimiser, notamment au niveau de l'implantation.

Nous passons d'abord en revue et évaluons les différents systèmes d'exploitation spécialisés, et choisissons celui offrant les fonctionnalités, notamment temps-réel, que nous estimons nécessaires pour implanter des protocoles MAC / RDC novateurs et performants.

Nous entreprenons ensuite un effort d'étude, d'amélioration et d'optimisation de ces couches basses des piles spécialisées, et montrons, avec une implantation concrète d'un de nos protocoles MAC / RDC avancés, que nous pouvons amener des progrès notables dans la qualité de service (QdS) des WSN, notamment avec un trafic réseau intense.

Nous examinons en outre des inexactitudes inattendues dans les simulations / émulations effectuées par Cooja / MSPSim, et analysons les problèmes de fiabilité posés par l'utilisation de cet outil pour effectuer des évaluations de performances, notamment temporelles, de WSN.

Nous proposons enfin de nouvelles pistes pour de futures améliorations et optimisations de ces couches basses des piles réseau spécialisées, afin d'améliorer encore la fiabilité, les performances et la consommation énergétique des WSN.

Mots-clés: réseaux de capteurs sans-fil (WSN), systèmes d'exploitation, temps-réel, protocoles MAC, protocoles RDC, *"Radio Duty Cycle"*, qualité de service (QdS), énergie

Abstract

In the field of wireless sensors networks (WSN), specialized network stacks have been a very active research field for many years. However, most of this research, especially on lower layers of the network stacks, did not go beyond theory. Their implementations have generally not been the subject of deep or systematic effort, especially within the framework of dedicated operating systems. We thus propose, in this thesis, to focus on interaction analysis between lower layers' protocols and dedicated software platforms, and to optimize them, especially at the implementation level.

We first review and evaluate the various dedicated operating systems, and choose the one offering the necessary features to implement efficient and innovative MAC/RDC protocols.

We then study, improve and optimize these lower layers of specialized stacks, and show, with an actual implementation of one of our advanced MAC/RDC protocols, that we can bring significant improvements in the quality of service (QoS) of WSNs, especially under heavy network traffic.

We also report inaccuracies in Cooja/MSPSim simulations/emulations, and analyze the reliability issues caused by the use of this tool for performing evaluations (especially time-related) of WSNs.

We finally propose some new leads for future enhancements and optimizations of the lower layers of these specialized network stacks, in order to further improve the liability, performances and energy consumption of WSNs.

Keywords: wireless sensor networks (WSN), operating systems, real-time, MAC protocols, Radio Duty Cycle protocols, Quality of Service (QoS), energy