UNIVERSITÉ DE NGAOUNDÉRÉ
FACULTÉ DES SCIENCES
DÉPARTEMENT DE MATHÉMATIQUES ET INFORMATIQUE

THE UNIVERSITY OF NGAOUNDERE
FACULTY OF SCIENCE
DEPARTMENT OF MATHEMATICS AND COMPUTER SCIENCE

MASTER EN INGÉNIERIE INFORMATIQUE

PARCOURS : SYSTÈMES ET LOGICIELS EN ENVIRONNEMENTS DISTRIBUÉS

MÉMOIRE DE MASTER

Une approche de multithreading efficace dans les réseaux de capteurs sans fil

Présenté par :

WOHWE SAMBO DAMIEN

Matricule : 09A062FS

Sous l'encadrement de :

DR YENKE BLAISE OMER

Chargé de Cours

Université de Ngaoundéré

ANNÉE ACADÉMIQUE : 2014-2015

Dédicace

À

Ma feue mère NANASO DODO

Mon père SAMBO GAROUA

Remerciements

Béni soit **DIEU Tout Puissant** pour toutes les grâces reçues, ainsi que ma présence en ce jour.

Je tiens à exprimer ma plus vive reconnaissance au Dr YENKE Blaise Omer, pour m'avoir honoré de son encadrement et de son expertise lors de mes travaux ; sa très bonne humeur, ses conseils et sa simplicité de vie m'ont inspiré à surmonter les difficultés rencontrées lors de cette recherche. L'image gardée à la fin de mes travaux est celle d'un *père* bienveillant et ses apports majeurs me permettent de présenter ce mémoire.

Je remercie tous les acteurs de la Faculté des Sciences qui n'ont ménagé aucun effort pour notre formation. Grand MERCI, je le dis au Dr. TCHAKOUNTE Franklin, M. FOTSA David, Mme. ZONGO Minette et M. ARI Abba pour tous les conseils et les remarques apportés à ce travail.

Mes études à l'Université de Ngaoundéré m'ont donné l'occasion de rencontrer des personnes formidables et avec lesquelles il est plaisant de travailler, je pense à MATADAH Chanelle, SALDEU Arnaud, TALA Diane, TANZOUAK Joël et ZACKO Edinio.

Un grand merci à tous les membres de ma famille : NASER Adèle, MBESSO SAMBO Koué Brice, MAÏTAO-YANG SAMBO Marie-José, DOBA SAMBO Gaëtan, DANGWANG SAMBO Hermann, NENBA SAMBO Jean-Jacques et MAÏLAÏSSO SAMBO Verra qui ont toujours cru en moi et n'ont jamais cessé de m'encourager de près comme de loin.

Mes remerciements vont également à l'endroit de mes amis et proches qui m'ont toujours soutenu durant les moments difficiles, je parle entre autre de TOTOUM FOTSING Christian, MOULOUOM NDAM Samuel, KOUONG JAURES Casimir, NGUENANG TOUKAP Rodrigue, MVONDO MEBO Eric Carel, DOSSOL Manasse et MADAMA TCHINDEBE Nicole.

Une pensée forte à tous ceux dont j'ai oublié de mentionner le nom ci-dessus.

Table des Matières

Liste des Figures

Liste des Tableaux

Abréviations

6LoWPAN : IPv6 Low-power Wireless Personal Area Networks

CAN : Convertisseur Analogique - Numérique

CSMA : Carrier Sense Multiple Access

CPU : Central Processing Unit

MAC : Media Access Control

OS : Operating System

RAM : Random Access Memory

RCSF : Réseau de Capteurs Sans Fils

RDC : Radio Duty Cycling

RPL : Routing Protocol for Low-power and lossy networks

ROM : Read Only Memory

WSN : Wireless Sensor Network

Résumé

De nos jours, l'utilisation des Réseaux de Capteurs Sans Fil (RCSFs) est de plus en plus croissante du fait de leurs applications nombreuses et variées. La taille des RCSFs devenant plus conséquente, la communication entre les capteurs s'effectue à l'aide de communications multisauts grâce aux ondes radio. Toutefois, chaque noeud capteur étant limité par ses ressources, des systèmes d'exploitations dédiés aux RCSFs sont mis sur pied afin d'optimiser la gestion de ces ressources parmi lesquelles celle énergétique qui détermine la durée de vie d'un RCSF.

Dans ce sens, le système d'exploitation hybride Contiki utilisant la couche faible consommation Rime permet d'effectuer des envois multisauts à faible coût énergétique grâce à l'implémentation de processus légers ou protothreads. Ces derniers présentent un bon rapport efficacité/consommation pour les tâches monolithiques mais ne permettent pas une bonne gestion de plusieurs tâches. Afin de permettre le traitement multitâche, Contiki fournit aux utilisateurs un module de multithreading préemptif permettant la gestion de plusieurs threads, cependant, son utilisation induit une plus grande consommation énergétique. Afin de rendre possible un multithreading basse consommation dans les RCSFs, nous proposons une nouvelle approche efficace du multitâche à l'aide des protothreads qui fournit pratiquement une même efficacité que le module multithreading tout en observant des consommations minimales.

Mots clés : *Réseaux de Capteurs Sans Fil (RCSFs), Contiki, protothreads, multisauts, threads*.

Abstract

Nowadays, the use of Wireless Sensor Networks (WSN) is increasingly growing due the fact that they are varied and have many applications. The size of WSNs becoming more consistent, communication between the sensors is done using multihop communications with the radio. However, each sensor node is limited by its resources then operating systems dedicated to WSNs are developed to optimize the management of these resources including the energy that determines the lifetime of a WSN.

In this sense, the hybrid operating system Contiki using low consumption layer called Rime can perform multihop sending with low energy cost thanks to the implementation of lightweight processes or protothreads. These processes have good ratio efficacy/consumption for monolithic tasks but do not allow good management of several tasks. To enable multitasking, Contiki provides to users a module of a preemptive multithreading which manage multiple threads, however, its use causes greater energy consumption. In order to make possible a low power multithreading in WSNs, we propose a new efficient approach to multitasking using protothreads that provides virtually the same efficiency as the multithreading module while observing minimum consumption.

Key words : *Wireless Sensors Networks (WSNs), Contiki, protothreads, multihop, threads*.

Introduction

Les avancées technologiques et techniques opérées dans le domaine des réseaux sans fil, de la microfabrication et de l'intégration des microprocesseurs ont fait naître une nouvelle génération de réseaux constitués d'entités à capacité de calcul et de mémoire limités capable d'obtenir des informations sur l'environnement appelées *capteurs* [1–5]. De plus, les progrès réalisés dans la microélectronique et la communication sans fil permettent de produire à des coûts raisonnables les composants d'un capteur, voyant l'expansion et la diversification des RCSFs [4, 6–9]. Cependant, la mise sur pied d'un RCSF s'effectue à travers un déploiement des capteurs suivant deux différents modèles [10] :

- Après une étude approfondie, chaque capteur a un emplacement précis dans le réseau.
- Déployés aléatoirement par voie aérienne (avion ou hélicoptère) [1, 2, 4], les capteurs sont introduits dans des terrains inaccessibles ou des zones à reliefs sensibles [11–13].

La taille des RCSFs étant toujours plus grande, le nombre de noeuds capteurs déployés sera d'autant plus considérable. Deux noeuds éloignés ne pouvant communiquer directement du fait des ressources limitées dans les capteurs, une information sera véhiculée via le réseau à travers les noeuds voisins intermédiaires. Ce mode de communication multisauts permet à un noeud capteur d'utiliser moins d'énergie relativement à une communication simple saut [7, 8, 14]. Les capteurs fonctionnant généralement à l'aide de batterie, la gestion énergétique va se classer comme la contrainte majeure dans la mise en place d'un RCSF [6, 12]. Dans l'optique d'une réduction de la consommation d'énergie dans les capteurs, des mécanismes d'envois multisauts sont élaborés où la rapidité de retransmission d'un capteur est vue comme un atout majeur dans cette quête de réduction [1, 2, 5, 6]. Afin d'optimiser une bonne utilisation de ces ressources limitées, les systèmes d'exploitation dédiés pour RCSFs ont vu le jour et chacun d'eux propose une approche

distincte [11]. Parmi ces systèmes figure le système Contiki [1] [15, 16]. Ce système hybride utilise la couche faible consommation Rime [17] pour les envois multisauts entre des noeuds voisins et implémente en plus des processus légers appelés protothreads [18, 19] qui ont une faible exigence mémoire (moins de 200 octets). Pour la gestion du multitâche, ces processus légers n'étant pas appropriés, Contiki fournit aux utilisateurs un module permettant le multithreading qui implémente des threads préemptifs malgré que son utilisation induit une forte utilisation de la mémoire par conséquent une plus grande consommation énergétique [18].

L'objectif principal de ce travail est la mise en place d'un nouveau paradigme du multithreading dans les RCSFs à travers l'utilisation des protothreads tout en se basant sur le fonctionnement des threads. L'idée est de développer un module qui permette un traitement multitâche, ce module devra être efficace tout en permettant une bonne consommation énergétique.

Afin d'atteindre l'objectif fixé, nous avons considéré un RCSF dans lequel un noeud central reçoit de certains noeuds plusieurs paquets à retransmettre vers d'autres noeuds différents. De par sa position, le noeud central sera l'unique moyen d'acheminer ces paquets vers leur destination car ne pouvant communiquer directement. Dans un premier temps, nous analysons les performances et la consommation d'énergie par le noeud central lorsque ce dernier utilise le mécanisme d'envoi multisauts par défaut ; dans un second temps, nous étudions dans les mêmes conditions l'envoi multisauts avec threads en utilisant la bibliothèque *mt.h* de Contiki OS. Sur la base de ces deux architectures, nous proposons une nouvelle architecture hybride plus efficace se situant entre ces deux architectures tout en observant de bonnes consommations énergétiques.

Le présent document s'organise comme suit : au chapitre 1, un parcours en revue des réseaux de capteurs sans fils est présenté ; le chapitre 2 présente les fonctionnalités du système d'exploitation Contiki ; des expérimentations sur l'envoi multisauts de la couche Rime sont effectuées au chapitre 3. Au chapitre 4, une proposition d'une architecture efficace pour le transfert multisauts est exposée. Le document s'achève par une conclusion et des perspectives.

1. Système hybride léger multitâche développé à la Swedish Institute of Computer Science en 2004

CHAPITRE 1

RÉSEAUX DE CAPTEURS SANS FIL

Dans ce chapitre la notion de Réseaux de Capteurs Sans Fil est présentée, ainsi que leur structure, leurs composants et les différentes applications des RCSFs. Les systèmes d'exploitation dédiés aux RCSFs y sont également présentés.

1.1 Généralités sur les RCSFs

Dans cette section, nous présentons les différents éléments qui constituent un capteur, les contraintes liées aux RCSFs et ses applications.

1.1.1 Architecture d'un capteur

Un RCSF se compose d'un ensemble de noeuds capteurs interconnectés par des communications sans fil [7]. Ces noeuds sont organisés en champs [1], où chaque noeud est capable de collecter, traiter et transférer les données vers un ou plusieurs *puits* [2] (Figure 1.1) à travers un mécanisme d'envoi multisauts [4]. Afin d'effectuer toutes ces tâches, un noeud capteur se dote de plusieurs éléments ou modules. La Figure 1.2 présente l'architecture matérielle d'un capteur avec ses différents modules. Chacun de ces modules effectue une tâche particulière d'acquisition, de traitement, de transmission de données ou de gestion de l'énergie [1, 12]. Plusieurs types de capteurs sont utilisés de nos jours, le Tableau 1.1 en présente les plus utilisés.

1. ou *sensor fields* pour champs de capteurs
2. ou *gateway* ou *sink* est le noeud passerelle

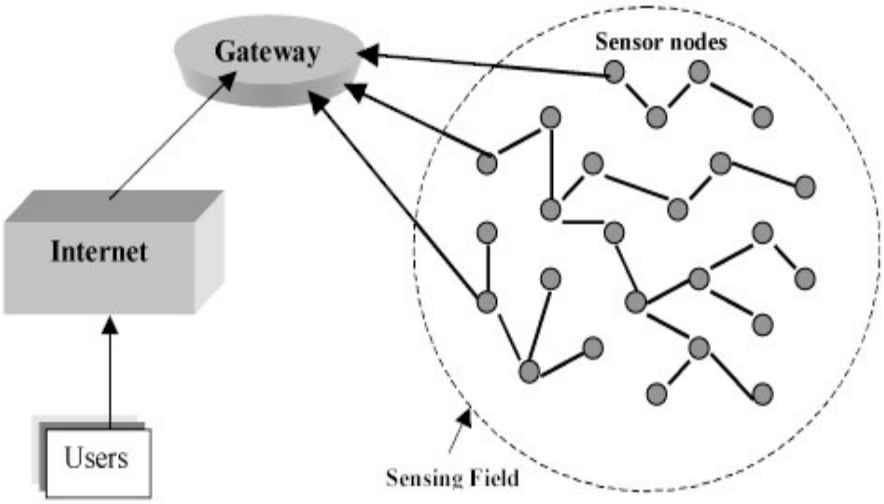


FIGURE 1.1 – Exemple de RCSF [20]

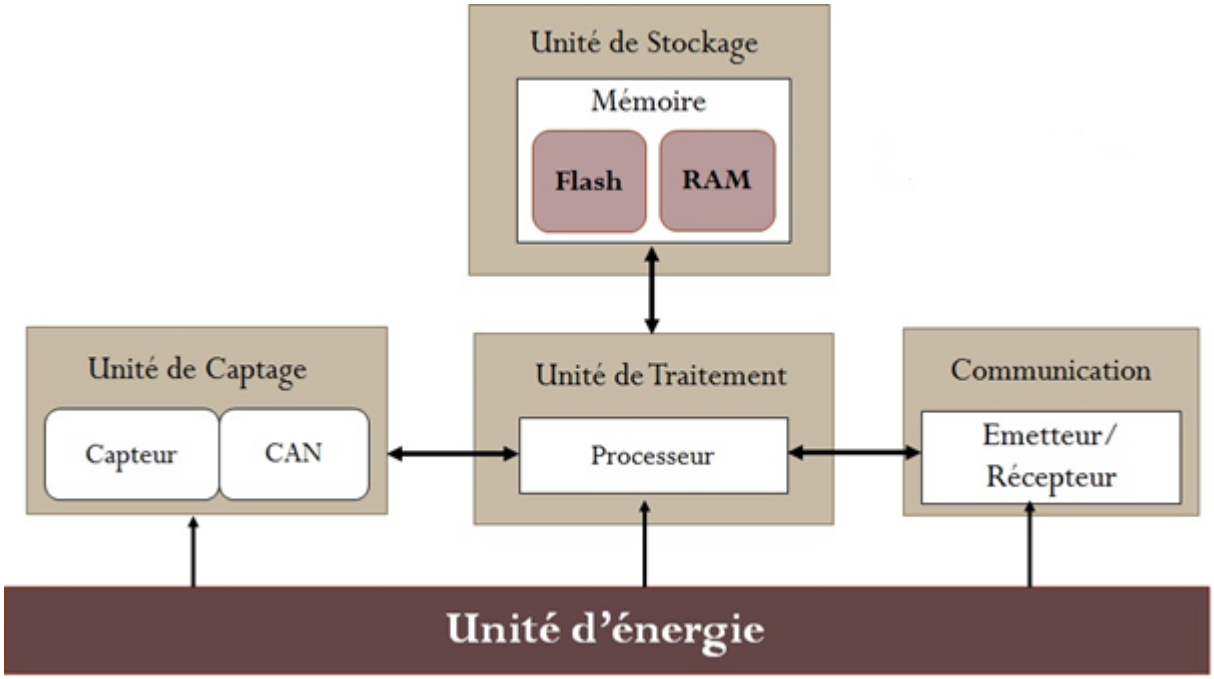


FIGURE 1.2 – Architecture matérielle d'un capteur [13]

TABLEAU 1.1 – Quelques capteurs sans fil [14].

Identification	**Mica2Dot (MPR500CA)**	**Mica2 (MPR400CB)**	**Tmote Sky**
Microcontrôleur	**ATmega128L**	**ATmega128L**	**MSP430F**
Architecture	8-Bit	8-Bit	16-Bit
Fréquence	4MHz	7.3728MHz	8MHz
Mémoire de données (RAM)	4Ko	4Ko	10Ko
Mémoire Flash	512Ko	512Ko	1024Ko

L'Unité de captage

Elle permet au capteur d'effectuer une capture ou mesurer les données physiques à partir d'un objet cible. Cette unité se compose de deux sous unités : le récepteur ou capteur qui reconnait la grandeur physique à capter et le transducteur qui permet la conversion du signal analogique reçu en signal numérique. Il peut arriver qu'un capteur dispose de plusieurs unités de captage [13].

L'Unité de traitement

C'est le coeur du capteur et permet de recueillir des données de l'unité de captage ou d'autres capteurs, traite ces données et décide quand et où les envoyer [21]. Les processeurs utilisés dans un capteur incluent les *Digital Signal Processors* [3](DSP), les *Application Specific Integrated* Circuit [4](ASIC), les *Field Programmable Gate Array* [5](FPGA) et le microcontrôleur [7]. Ce dernier, de par sa flexibilité, son bon prix et sa faible consommation énergétique est le plus utilisé pour les capteurs [9, 21, 22]. L'unité de traitement fonctionne à l'aide d'un système d'exploitation spécialement conçu pour les micro-capteurs [13].

L'unité de communication

Elle est responsable de toutes les émissions et réceptions de données dans le capteur et utilise un support de communication sans fil pour les RCSFs, soit entre autre la Radio-Fréquence (RF), le laser ou l'infrarouge [9]. Ainsi concevoir une telle unité de type RF (qui est la plus efficace en termes de consommation énergétique) est un défi car pour qu'un noeud ait une portée de communi-

3. Microprocesseur conçus pour résoudre efficacement les opérations mathématiques complexes
4. Circuit intégré pouvant être facilement créé pour une application spécifique)
5. Similaire aux ASIC mais plus flexible

cation suffisamment grande, il est nécessaire d'utiliser un signal assez puissant et donc une énergie consommée plus importantes [1]. L'alternative consistant à utiliser de longues antennes n'est pas possible à cause de la taille réduite des micro-capteurs et de leur zone de déploiement [5].

L'unité de stockage

Cette unité inclut la mémoire pour l'exécution des programmes (RAM) dont les instructions sont exécutées par le microcontrôleur. Cette mémoire consomme la majeure partie de l'énergie allouée au microcontrôleur, c'est pourquoi on lui adjoint souvent une mémoire flash (ROM) moins coûteuse en énergie qui contient le système d'exploitation [1].

La source d'énergie

Cette unité permet d'alimenter les différents composants du capteur. Elle est généralement représentée par une batterie [23]. Cette ressource est limitée et généralement non-remplaçable étant donné la petite taille du capteur [6]. Tout ceci fait de l'énergie la ressource la plus précieuse d'un RCSF car elle influe directement sur la durée de vie des capteurs et donc du réseau entier [4, 5].

Un noeud capteur dispose en général de plusieurs modules qui interviennent lors de son fonctionnement. Toutefois, la mise sur pied d'un RCSF se fait suivant certaines contraintes.

1.1.2 Contraintes de conception des RCSFs

Plusieurs facteurs tels que la consommation d'énergie, la tolérance aux pannes, les coûts de production, l'environnement ou la topologie du réseau doivent être pris en compte lors de la conception d'un RCSF [1].

La consommation d'énergie

La consommation en énergie est l'une des problématiques majeures dans les réseaux de capteurs. En effet, la recharge des sources d'énergie est souvent trop coûteuse et parfois impossible [6]. Il faut donc que les capteurs économisent au maximum l'énergie afin de pouvoir fonctionner plus longtemps car de par leur taille, les capteurs sont limités en énergie (moins de 1.2V) [1]. Les

RCSFs fonctionnant selon un mode de routage multisauts, chaque noeud du réseau joue un rôle important dans la transmission de données.

La tolérance aux pannes

Les noeuds peuvent être sujet à des pannes dues à leur fabrication car étant des produits de série bon marché [3], il peut donc y avoir des capteurs qui sont défectueux ou en manque d'énergie. Les interactions externes (chocs, interférences) peuvent aussi être la cause des dysfonctionnements. Afin que les pannes n'affectent pas la tâche première du réseau, il faut évaluer la capacité du réseau à fonctionner sans interruption [1].

Les coûts de production

Les RCSFs sont en partie composés d'un très grand nombre de noeuds. Le prix d'un noeud doit être accessible afin de mettre sur pieds de plus grands RCSFs ainsi couvrir une plus grande zone. [4].

Les contraintes matérielles

La principale contrainte matérielle est la taille du capteur qui influe directement sur la capacité de la batterie. Le capteur doit avoir soit une petite taille afin qu'il s'adapte dans différents environnements (forte chaleur, humidité) ; soit très résistant vu qu'il est souvent déployé dans des environnements hostiles [1, 4].

La topologie du réseau

Le déploiement d'un RCSF nécessite une maintenance de sa topologie en raison de la forte densité de noeuds dans la zone à observer [7]. Ainsi ces noeuds doivent être capables d'adapter leur fonctionnement afin de maintenir la topologie. La maintenance de la topologie d'un RCSF consiste en trois phases [1, 4] :

– Déploiement : Les noeuds sont soit répartis de manière prédéfinie soit de manière aléatoire, dans ce cas, ceux-ci s'organisent de manière autonome.

– Post-déploiement : Durant la phase d'exploitation, la topologie du réseau peut être soumise à des changements dus à des modifications de la position des noeuds ou bien à des pannes.
– Redéploiement : L'ajout de nouveaux capteurs dans un réseau existant implique aussi une remise à jour de la topologie.

Les médias de transmission

Dans un RCSF, les noeuds sont généralement reliés par une architecture sans fil. Pour permettre des opérations sur tout le réseau, le média de transmission doit avoir une même norme [7]. La conception d'un RCSF passe par le choix d'une technologie de communication sans fil adapté, le Tableau 1.2 ci-dessous présente les différentes normes sans fil rencontrées.

TABLEAU 1.2 – Technologies de communication sans fil [14].

Technologie	Fréquence	Débit maximal
IEEE 802.11a	5GHz	54Mb/s
IEEE 802.11b	2.4GHz	11Mb/s
IEEE 802.11g	2.4GHz	54Mb/s
IEEE 802.15.1 (Bluetooth)	2.4GHz	1Mb/s
IEEE 802.15.4 (ZigBee)	2.4GHz	250Kb/s
IrDA	Infrarouge	4Mb/s
HDR UWB (Hotspot)	3.1-10.6GHz	100-500Mb/s
LDR/MDR UWB	3.1-10.6GHz	<1Mb/s

La mise sur pieds d'un RCSF passe par la définition d'un ensemble de contraintes. La gestion de ces contraintes donne aux RCSFs des applications variées.

1.1.3 Applications des RCSFs

Le concept de RCSF se résume à une équation simple [6] :

DETECTION + CPU + RADIO = MILLIERS D'APPLICATIONS POTENTIELLES.

Les applications des RCSFs étant très variées (Figure 1.3), elles pourront se classer de la surveillance environnementale en passant par les soins de santé jusqu'à l'automatisation industrielle et la surveillance militaire [5].

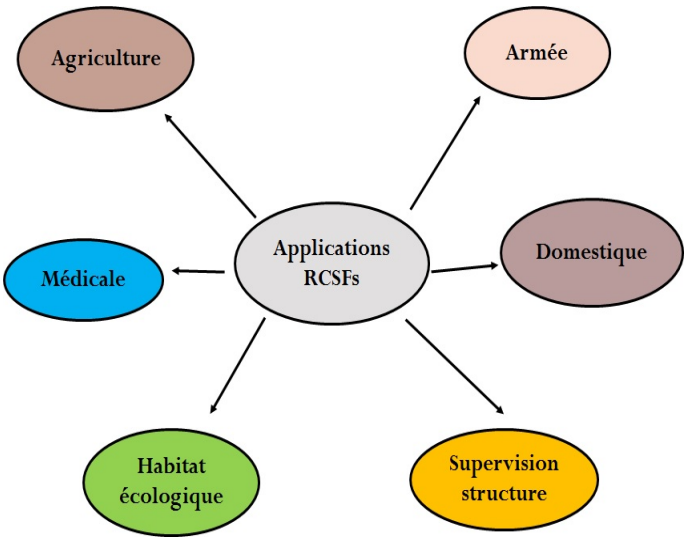


FIGURE 1.3 – Applications des RCSFs

Agriculture

Des noeuds peuvent être incorporés dans la terre et ensuite questionner le RCSF sur l'état du champ par conséquent déterminé par exemple les secteurs les plus secs afin de les arroser en priorité [4]. On peut aussi imaginer équiper des zones importantes de cultures tout en contrôlant la croissance de celles-ci.

Applications militaires

Comme beaucoup d'autres technologies de l'information, les RCSFs proviennent principalement de la recherche militaire [1]. La capacité de déploiement rapide, d'auto-organisation ainsi que la tolérance aux pannes des RCSFs forment dans le domaine militaire un système de *C4ISRT*[6] [12]. La reconnaissance d'ennemis, du terrain et du champ de combat permet de guider les forces alliées en zone hostile et rends pas possible la détection d'attaques chimique, biologique ou nucléaire.

Applications médicales

Certaines applications médicales fournissent une interface pour la surveillance, le diagnostic de patients [12]. C'est le cas par exemple du *Cardio Pad* (Figure 1.4) testé au *Centre Hospitalier Universitaire (CHU) de Yaoundé* et à *l'hôpital de Mbankomo*, et permet à toute personne de me-

6. Command, Control, Communications, Computing, Intelligence, Surveillance, Reconnaissance and Targeting

FIGURE 1.4 – Le Cardio Pad [24]

surer ses données physiologiques cardiaques. Cette tablette fait transiter les fréquences cardiaques du malade (à l'aide d'un terminal capteur) vers des serveurs, où elles sont stockées et traitées avant d'être transmises au cardiologue via le *Global System for Mobile communications* (GSM). Le spécialiste détient aussi un Cardio Pad, sur lequel il reçoit les résultats, les interprète et prescrit une médication. Une encyclopédie de maladies cardio-vasculaires est intégrée à l'appareil pour aider le cardiologue à faire son diagnostic. On peut également parler du projet *STAR* (Système Télé Assistant Réparti) qui consiste à concevoir une plateforme dédiée à la surveillance de personnes souffrant d'arythmies cardiaques. Ainsi la personne est équipée d'un capteur sans fil capable d'acquérir et analyser les signaux électrocardiogrammes en temps réel [14].

Applications domestiques

les RCSFs permet l'apparition d'habitation intelligente capable de gérer de façon autonome sa température, taux d'humidité, son éclairage et la sécurité. Ainsi, en plaçant, à différents points stratégiques, des détecteurs de mouvements, on peut prévenir des intrusions sans avoir à recourir à de coûteux dispositifs de surveillance vidéo [4].

Supervision de l'habitat écologique

Les études scientifiques des habitats écologiques (animaux, végétaux, micro-organismes) sont traditionnellement effectuées grâce à des activités sur le terrain par des enquêteurs. Un problème majeur dans ces études provient de ce qui est parfois appelé "l'effet de l'observateur" qui vient fausser les observations. Des RCSFs promettent une nouvelle approche écologique d'observation

à distance. C'est le cas du déploiement expérimental pour la surveillance de l'habitat de *l'océanite culblanc* [7] déroulé dans le golfe du *Maine* (Amérique du Nord) [25].

Supervision de structures

Une autre classe d'applications pour RCSFs concerne le suivi de l'état des structures civiles. Ces structures peuvent être des bâtiments, des ponts et des routes [14].

Afin de permettre la conception d'applications pour RCSF, la nécessité d'une interface entre ces applications et la partie logicielle se fait savoir : c'est le système d'exploitation.

1.2 Systèmes d'exploitation pour RCSF

Cette section présente les systèmes d'exploitation pour RCSFs. Ainsi, un éclaircissement sera fait sur ces systèmes puis certains d'entre eux seront présentés.

1.2.1 Définition et propriétés

Un système d'exploitation dans un RCSF est une couche mince de logiciel qui réside logiquement dans le matériel du noeud et fournit des abstractions de programmation de base aux développeurs d'applications. Sa tâche principale est de permettre aux applications une interaction avec les ressources matérielles. Ses fonctionnalités sont [7] :

- la gestion de la mémoire ;
- la gestion de l'alimentation ;
- la mise en réseau ;
- proposer un ensemble d'environnements de programmation et des outils (interpréteurs de commandes, éditeurs, compilateurs, débogueurs) pour permettre aux utilisateurs de développer, déboguer et exécuter leur propre programme ;

Traditionnellement, les systèmes d'exploitation sont classés comme systèmes *monotâche/multitâche* et *mono-utilisateur/ multi-utilisateur*. Un système d'exploitation *monotâche* traite les tâches

7. Ou *Oceanodroma leucorhoa* est un oiseau de mer

une par une, par contre celui *multitâche* peut exécuter plusieurs tâches simultanément. Toutefois, les Systèmes *multitâche* nécessitent une grande quantité de mémoire pour gérer les états des différentes tâches, mais ils permettent aux tâches de grande complexité de s'exécuter en parallèle. Par exemple, dans un noeud de capteurs sans fil, l'unité de traitement peut interagir avec l'unité de communication tout en agrégeant les données qui arrivent à partir de l'unité de détection. Un système *multitâche* est le meilleur candidat pour ce type d'environnement. Toutefois, en raison des ressources *limitées*, les frais généraux de traitement simultané peuvent ne pas être abordable [7]. Dans un système *monotâche*, où une tâche est exécutée à un moment, celles-ci doivent avoir une courte durée.

Le choix d'un système d'exploitation pour RCSF passe par plusieurs facteurs.

1.2.2 Facteurs de choix d'un système d'exploitation pour RCSF

Les facteurs suivants permettent le choix d'un RCSF [7].

Le type de données

Dans les RCSFs, la communication entre les différents éléments est vitale. Ces éléments communiquent entre eux pour diverses raisons telles que l'échange de données, l'attribution de fonctionnalités, et la signalisation. Les données complexes consomment plus de ressources, tandis que celles simples sont économes en ressources mais ont des capacités limités. Presque tous les systèmes d'exploitation existants dans les RCSFs chargent les types de données natifs du langage de programmation **C** ou toute autre variante.

L'ordonnancement

L'ordonnancement des tâches est l'une des fonctions de base d'un OS. L'efficacité de l'OS passe par la capacité des tâches à être organisée et exécutée suivant une priorité. D'une manière générale, il existe deux mécanismes de planification : *l'ordonnancement à base de files d'attente* et le *roundrobin* (ou tourniquet). Dans une planification basée sur les files d'attente, les nouvelles tâches sont stockées temporairement dans une file d'attente et exécutées en série selon un ordre prédéfini. Certains systèmes permettent de préciser les niveaux de priorité des tâches afin qu'elles

puissent avoir la priorité sur d'autres.

L'ordonnancement basé sur les files d'attente se classe en *First-In First-Out* (FIFO) et file d'attente triée.

Dans un schéma FIFO, les tâches sont traitées en fonction de leur arrivée : une tâche qui arrive en premier sera exécutée en premier. Dans un système de file d'attente triée, les tâches dans une file d'attente sont classifiées selon certains critères (durée d'exécution prévue).

Le *roundrobin* est une technique d'ordonnancement en temps partagé dans lequel plusieurs tâches peuvent être traitées simultanément.

Pile

Elle est utilisée pour stocker temporairement des données dans la mémoire en les empilant les unes sur les autres sur la base du *Last-In First-Out* (LIFO).
Dans un OS multithreadé, chaque thread nécessite sa propre pile mémoire, c'est l'une des raisons pour lesquelles les systèmes d'exploitation multitâches sont coûteux dans les RCSFs.

Multithreading

Dans un OS monotâche et non-préemptif, les tâches sont monolithiques et il y a qu'un seul thread d'exécution. Dans un environnement multitâche, une tâche peut être divisée en plusieurs morceaux logiques qui peuvent être programmés indépendamment les uns des autres et exécutés simultanément. De même, des tâches multiples provenant de différentes sources peuvent être exécutées simultanément dans plusieurs threads. Les avantages d'un système d'exploitation multitâche sont :

- Les tâches ne bloquent pas d'autres tâches ; ceci est particulièrement important pour faire face aux tâches relatives des systèmes d'Entrée/Sortie (E/S).
- Les tâches de courte durée peuvent être exécutées ainsi que celles de longue durée.

Paradigme d'exécution : Thread-Based vs Event-Based

Dans les réseaux de capteurs sans fil deux paradigmes d'exécution sont possibles : celui basé sur les threads (*Threads-based*) et celui basé sur les événements (*Event-based*).

Les OS Threads-based utilisent plusieurs threads de contrôle au sein d'un seul programme, de cette façon, un thread bloqué par un dispositif E/S peut être suspendu tandis que d'autres tâches sont exécutées.

Dans la programmation Event-based, on retrouve des événements et le gestionnaire d'événements. Le gestionnaire d'événements est prévenu lorsqu'un un événement se produit. Le noyau appelle généralement le gestionnaire d'événements lorsque des événements se produisent. Un événement est traité jusqu'à achèvement.

Le choix d'un système d'exploitation pour RCSF se fait donc en fonctions des facteurs ci-dessus, toutefois, plusieurs systèmes d'exploitation pour RCSF existent à ce jour.

1.2.3 Quelques OS dédiés pour RCSF

Les possibilités matérielles des capteurs étant limitées en mémoire, en puissance de calcul et en autonomie énergétique (batterie), les systèmes d'exploitation pour RCSF doivent avoir une « faible empreinte mémoire » lors de leurs embarquement dans la mémoire flash des capteurs [26]. Les OS les plus utilisés sont : Mantis OS [27], TinyOS [28] et Contiki [15].

Mantis OS [27]

MANTIS (MultimodAl system for NeTworks of In-situ wireless Sensors) est un système multitâche préemptif basé sur le partage temporel. Sa configuration de base s'adapte aux applications de réseaux de capteurs. Il se constitue du noyau, d'un ordonnanceur et d'une pile de protocole réseau et a une empreinte mémoire inférieure à 1 Kilo octet [14]. Ainsi, les ressources logicielles et matérielles du système sont ainsi affectées temporellement entre les différents processus en cours. Ce fonctionnement fait en sorte qu'un programme ne puisse pas être mis en attente ou bloquer le système indéfiniment. Ainsi, les tâches de longue durée ne peuvent pas accaparer les ressources du système au-delà d'un certain temps au détriment d'une autre dont l'exécution est critique. Le système MANTIS utilise un ordonnanceur prenant en compte la consommation énergétique et comprend un mécanisme de programmation à distance des capteurs sans fil. Il a également la possibilité de passer en mode veille lorsqu'aucun traitement n'est à réaliser. Son ordonnanceur quant

à lui alloue dynamiquement une zone mémoire pour stocker les registres du processeur et la pile pour chaque thread et leur nombre maximal de thread est réglable lors de la compilation (la valeur par défaut est 12). Chaque tâche du système d'exploitation devant être pris en charge peut être implémentée en utilisant la norme **C** comme un thread. Dans la mise en oeuvre de MANTIS, la tâche de traitement de paquet a une priorité plus élevée que la tâche de détection.

TinyOS [28]

Contrairement à MANTIS, TinyOS dispose d'un mode *faible puissance* qui permet d'avoir des consommations énergétiques meilleures que celles de MANTIS. C'est un système d'exploitation open source développé et suivi par l'Université de Berkeley. Son fonctionnement est évènementiel et il devient actif lorsqu'un événement se produit (par exemple la réception d'un message), dans le cas contraire les noeuds capteurs sont en état de veille. Ce processus permet de mieux économiser les ressources énergétiques des capteurs.

Le langage de programmation pour la conception de ce système est **NesC** qui est une variante du langage C et propose une architecture basée sur des composants, permettant de réduire considérablement la taille mémoire du système et de ses applications. Chaque composant correspond à un élément matériel (LEDs, timer) et peut être réutilisé dans différentes applications. Ces applications sont des ensembles de composants associés dans un but précis. L'implémentation de composants s'effectue en déclarant des tâches [8], des commandes [9] ou des évènements [10].
L'ordonnanceur de TinyOS est au cœur de la gestion des tâches et des évènements (Figure 1.5) du

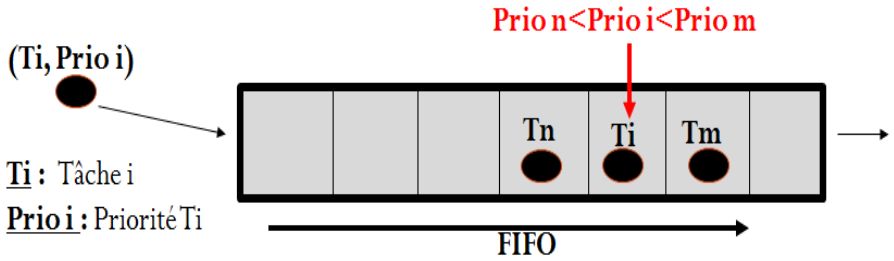


FIGURE 1.5 – Fonctionnement de l'ordonnanceur de TinyOS

8. Travail de longue durée.
9. Correspond à l'exécution d'une fonctionnalité précise dans un autre composant.
10. L'équivalent logiciel est une interruption matérielle.

système et permet :

- 2 niveaux de priorité (bas pour les tâches, haut pour les évènements)
- 1 file d'attente FIFO (disposant d'une capacité de 7 tâches)

Contiki [15]

Tout comme TinyOS, le système Contiki permet une faible consommation énergétique, en plus, il est flexible et permet l'évolution d'un RCSF [29]. Contiki est un système hybride multitâche et son noyau utilise des processus légers, qui peuvent être des applications ou des services, c'est-à-dire un processus proposant des fonctionnalités à une ou plusieurs applications.

Le système Contiki est divisé en deux parties : une partie noyau basse et une autre réservée aux programmes, telle qu'illustrer à la Figure 1.6.

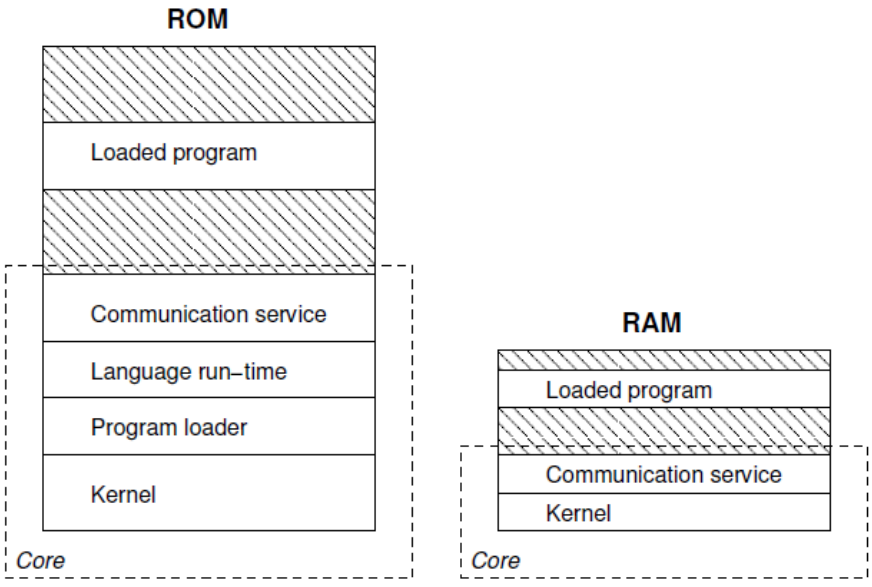


FIGURE 1.6 – Partitionnement Contiki [15]

Contiki permet une faible consommation énergétique en utilisant les protothreads et implémente 02 piles réseaux parmi lesquelles la pile faible consommation *Rime* [17]. De plus, grâce à une politique open source, à un noyau développé en C et à un environnement de simulation intégré, le système Contiki caractérisé généralement de *complet* permet une prise en main rapide et sera utilisé dans la suite de ce document.

Autres OS

En dehors des systèmes MANTIS, TinyOS et Contiki, d'autres OS pour RCSFs existent. Cependant, ces systèmes ne permettent pas une meilleure consommation énergétique que MANTIS, TinyOS et Contiki [26]. Il s'agit de : SOS [30], RETOS [31], LiteOS [32], Nano-RK [33], Nano-Qplus [26], PAVENET OS [26].

1.3 Conclusion

Dans ce chapitre, nous avons présenté les RCSFs ainsi que les contraintes liées à leur conception. Ces derniers présentent de nombreux domaines d'applications allant de l'usage domestique jusqu'à la surveillance militaire en passant par une utilisation médicale innovante. Nous avons également les systèmes d'exploitation pour RCSF, leurs fonctionnalités ainsi que leurs caractéristiques. Plusieurs OS sont présentés parmi lesquels : TinyOS, MOS et Contiki.

L'existence de plusieurs systèmes d'exploitation dédiés aux RCSFs permet d'avoir un large choix lors de la sélection d'un OS. Toutefois, parmi ces OS, un système hybride, polyvalent, multitâche et flexible avec une faible consommation se démarque du lot : *Contiki*. Le chapitre suivant présente plus en détail ce système.

CHAPITRE 2

CONTIKI OS : UN SYSTÈME D'EXPLOITATION HYBRIDE POUR RCSF

Dans ce chapitre, le système d'exploitation pour RCSF multitâche Contiki est présenté en détail à travers deux parties. La première, décrit le fonctionnement de Contiki, la gestion d'événements et de processus grâce à son ordonnanceur ; la seconde partie quant à elle, présente les modes de communications de Contiki parmi lesquels la couche basse consommation Rime qui implémente la communication multisauts.

2.1 Fonctionnement interne de Contiki

Cette section présente le fonctionnent hybride du système Contiki.

2.1.1 Gestion des événements

Dans les environnements à mémoire réduite, les architectures basées sur un modèle multitâche (multithreadé) consomment la grande partie de la ressource mémoire car chaque thread dispose de sa propre pile [18]. Afin de résoudre le problème de la mémoire dû à l'utilisation de threads, les systèmes basés sur le fonctionnement événementiel voient le jour [15].

Dans Contiki, un processus est exécuté quand il reçoit un événement. Il existe deux types d'événements : les événements asynchrones et les événements synchrones [34].

Evénements asynchrones

Les événements asynchrones sont envoyés au processus de réception quelque temps après qu'ils ont été postés. Entre leur affectation et de leur envoie, les événements asynchrones sont organisés sur une file d'attente d'événements à l'intérieur du noyau Contiki.

Les événements de cette file d'attente sont envoyés aux processus destinataires par le noyau de Contiki. Le noyau parcourt la file d'attente et distribue les événements de la file aux processus correspondants par invocation de ces derniers.

Le destinataire d'un événement asynchrone peut être soit un processus spécifique, ou tous les processus en cours d'exécution. Lorsque le récepteur est un processus spécifique, le noyau appelle ce processus afin de délivrer l'événement. Lorsque le récepteur d'un événement est réglé pour être tous les processus dans le système, le noyau délivre séquentiellement le même événement à tous les processus, les uns après les autres et s'il y a plusieurs événements en attente dans la file d'événements, l'attribution de ceux-ci est déclenchée de manière FIFO (Figure 2.1).

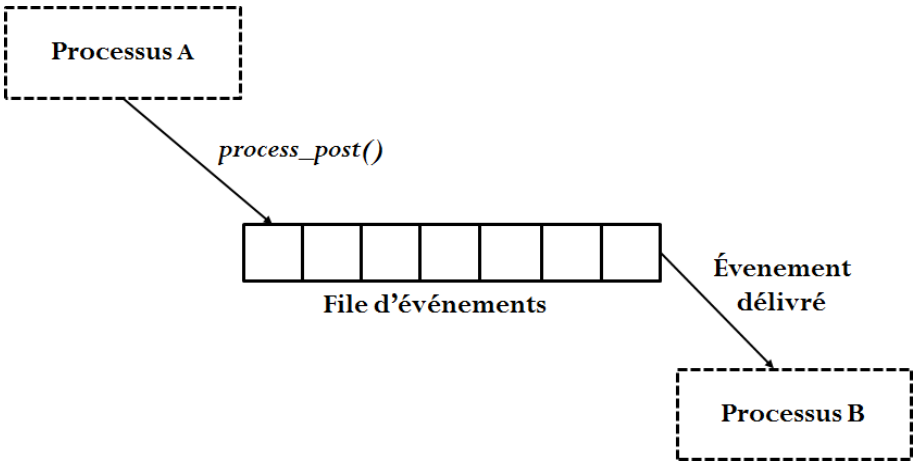


FIGURE 2.1 – Fonctionnement événements asynchrones [34]

Événements asynchrones sont postés à l'aide de la fonction *process_post()*. Elle vérifie d'abord la taille de la file d'événements pour déterminer s'il y a une place libre pour l'événement. Si oui, la fonction insère l'événement à la fin de la file d'attente et s'arrête ; si non, la fonction renvoie une erreur.

Evénements synchrones

Contrairement aux événements asynchrones, les événements synchrones sont livrés directement quand ils sont postés. Ces événements sont affectés à un processus spécifiques à l'aide de la fonction *process_post_synch()* (Figure 2.2).

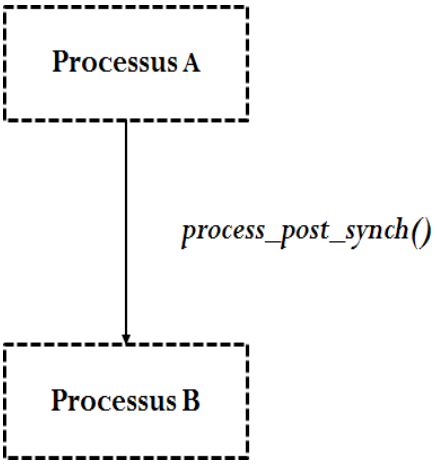


FIGURE 2.2 – Fonctionnement événement synchrones [34]

Comme les événements synchrones sont livrés immédiatement, posté un tel événement est équivalent à un appel de fonction : le processus auquel l'événement est destiné sera directement invoqué, et le processus qui a envoyé l'événement est bloqué jusqu'à ce que le processus de réception ait terminé le traitement de l'événement. Cependant, le processus de réception n'est pas informé si l'événement publié est synchrone ou asynchrone. La Figure 2.3 résume la gestion des événements par le système Contiki.

De par son architecture hybride, Contiki en plus d'avoir un fonctionnement événementiel, permet aussi une préemption de threads.

Multithrerading préemptif

Dans Contiki, le multithreading est implémenté par une bibliothèque au-dessus de son noyau événementiel. Toutefois, cette bibliothèque peut être associée explicitement aux applications qui disposent d'un modèle de fonctionnement multithreadé. La bibliothèque *mt.h* [1] de Contiki est divisée en deux parties : une partie indépendante qui joue le rôle d'interface avec son noyau événe-

1. contiki-2.7/core/sys/mt.h

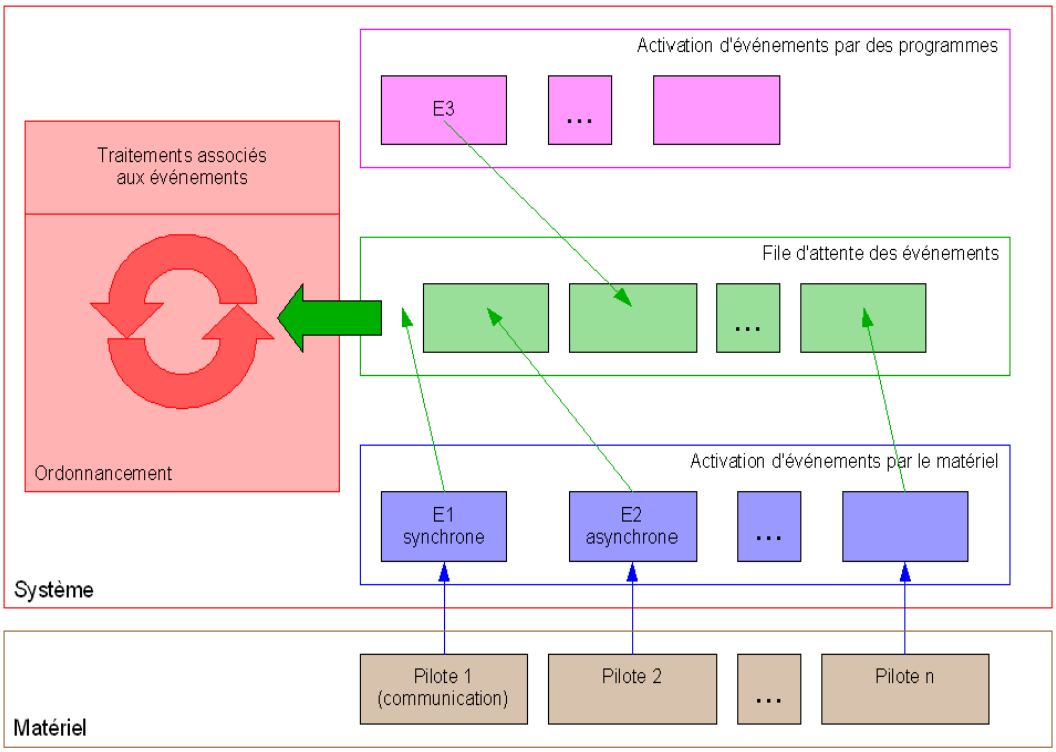


FIGURE 2.3 – Gestion des événements par Contiki [35]

mentiel ; et une plateforme spécifique qui implémente les primitives de préemption et la gestion de la pile. Habituellement, la préemption est implémentée en utilisant une interruption temporelle (à l'aide d'un minuteur) qui enregistre les registres du processeur sur la pile.

Contrairement aux processus de Contiki où chaque thread nécessite une pile séparée, La bibliothèque *mt.h* fournit une pile nécessaire pour la gestion des fonctions. Les threads s'exécutent sur leur propre pile jusqu'à ce qu'ils se terminent ou qu'ils soient préempter.

Les différentes fonctions de la bibliothèque *mt.h* sont présentées à la Figure 2.4. En plus de la fonction d'initialisation *mt_init()*, elle se compose de trois fonctions qui peuvent être appelées à partir d'un thread en exécution (*mt_yield()*, *mt_exit()* et *mt_stop()*) ; et de deux autres qui sont appelées pour exécuter un thread (*mt_start()* et *mt_exec()*). la fonction *mt_exec()* effectue la programmation réelle d'un thread et est appelée à partir d'un gestionnaire d'événement. La gestion de la préemption entre les threads est rendue possible grâce à la fonction *mt_yield()*.

La gestion des événements et le multithreading préemptif sont implémentés dans Contiki à l'aide des processus.

Void **mt_init** ()	Initializes the multithreading library.
Void **mt_remove** ()	Uninstalls library and cleans up.
Void **mt_start** (struct mt_thread *thread, void(*function)(void*), void *data)	Starts a multithreading thread
Void **mt_exec** (struct mt_thread *thread)	Execute parts of a thread.
Void **mt_yield** ()	Voluntarily give up the processor
Void **mt_exit** ()	Exit a thread.
Void **mt_stop** (struct mt_thread *thread)	Stop a thread

FIGURE 2.4 – Fonctions de la bibliothèque *mt.h* [36]

2.1.2 Processus dans Contiki

Contextes d'exécution

Le code d'un programme dans Contiki peut fonctionner dans l'un des contextes d'exécution suivants : coopératif ou préemptif. L'exécution d'un code en mode coopératif est gérée de manière séquentielle sans interruption (*run-to-completion*). Par contre, un code s'exécutant en mode préemptif peut arrêter le code coopératif à tout moment. Lorsque le code préemptif arrête celui coopératif, ce dernier ne sera pas repris tant que l'exécution du code préemptif ne soit terminée. Le concept de ces deux contextes d'exécution est illustré à la Figure 2.5.

Les processus s'exécutent toujours en mode coopératif, toutefois le mode préemptif est utilisé par les gestionnaires d'interruption et par des tâches en temps réel qui ont été prévues pour un délai précis défini dans Contiki par le module Timer [2].

Son but est d'invoquer un processus lorsqu'il doit être exécuté (après réception d'un événement). L'ordonnanceur de processus appelle le processus en exécutant la fonction qui implémente le code du processus. Toute invocation de processus dans Contiki est faite en réponse d'un événement posté qui a été affecté à un processus, ou une demande requête d'invocation du processus. L'ordonnanceur de processus envoie l'identifiant de l'événement au processus qui est invoqué.

2. contiki-2.7/core/sys/timer.h

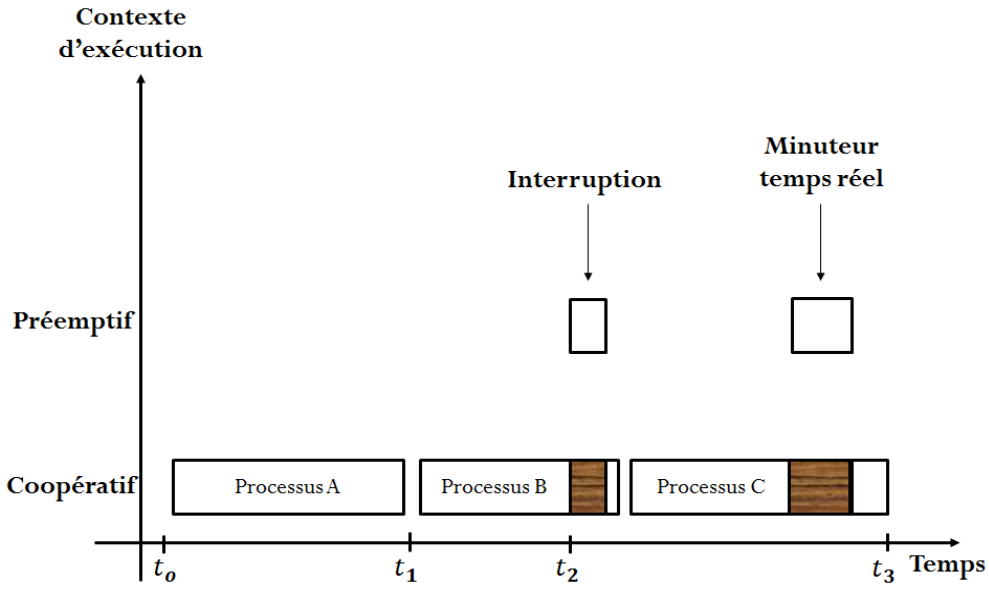


FIGURE 2.5 – Contextes d'exécution dans Contiki [37]

La Figure 2.6 récapitule le fonctionnement de l'ordonnanceur de Contiki.

Structure d'un processus

Les processus dans Contiki sont l'équivalent de tâches dans TinyOS. Un processus se compose de deux parties : le Process Control Block (PCB) et le Process Thread (PT) [37]. Le PCB est stockée dans la RAM et contient des informations sur le processus en exécution tandis que le PT est le code du processus et est implémenté dans la mémoire flash.

Process Control Block Le PCB est utilisé uniquement par le noyau et n'est pas accessible directement par les processus.

Le PCB est léger en ce sens qu'il ne nécessite que quelques octets de mémoire. Aucun des champs de cette structure ne doit être accessible directement. Seules les fonctions de gestion des processus devraient y avoir accès.

Un PCB n'est ni déclaré, ni défini directement mais plutôt à travers la macro *PROCESS()*. Cette macro prend deux paramètres : la variable *nom* du PCB, qui est utilisé lors de l'accès au processus, et un *nom textuel*, qui est utilisé dans le débogage et lors de l'impression des listes de processus actifs aux utilisateurs. La définition du bloc de contrôle de processus correspondant à l'exemple

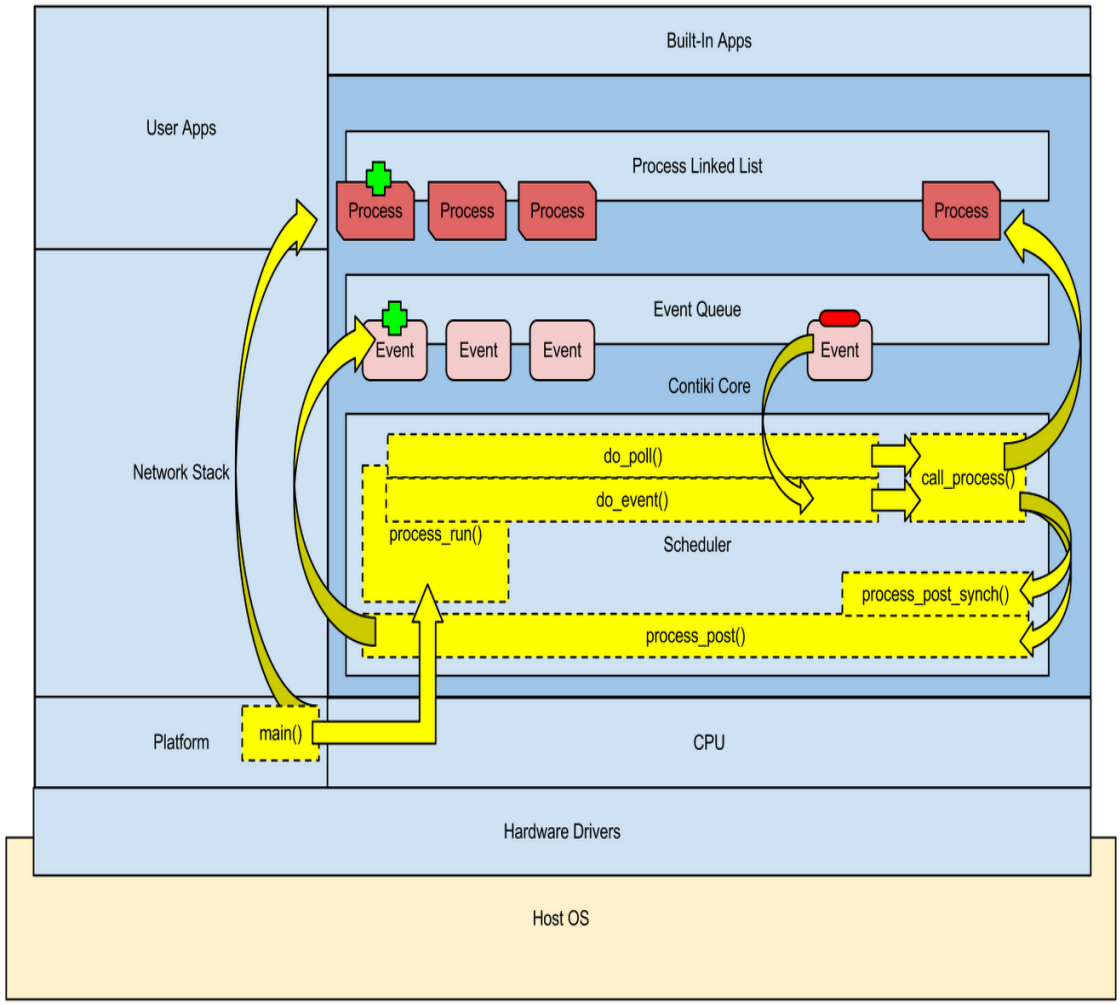
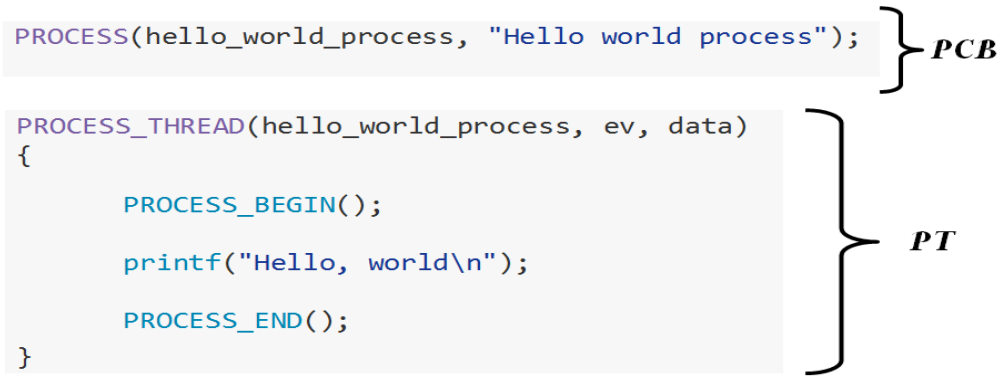


FIGURE 2.6 – Ordonnanceur de Contiki [37]

représenté ci-dessous.

Process Thread le Process Thread contient le code du processus. C'est un processus léger qui est appelée à partir de l'ordonnanceur de processus. Un exemple de processus est illustré ci-dessous.



Ces processus légers sont appelés protothreads.

Les protothreads dans les processus

Un protothread est un processus léger qui donne un moyen de structurer le code d'une manière afin permet au système de faire fonctionner d'autres activités lorsque le code est en attente [15,38]. Le concept de protothreads a été développé dans le cadre de Contiki. Le protothread fournit un moyen aux fonctions C de travailler d'une manière qui est similaire aux threads, sans la surcharge de la mémoire car les threads créés partagent un même espace mémoire [15]. La réduction de la surcharge de mémoire est importante sur les systèmes à mémoire limitées sur lesquels Contiki s'exécute. Un protothread est une fonction régulière C, elle commence et se termine avec deux macros spéciales, *PT_BEGIN()* et *PT_END()*. Entre ces macros, un ensemble de fonctions peut être utilisé (Figure 2.7).

La Figure 2.8 montre l'implémentation des protothreads dans les processus de Contiki [3].

En plus de processus légers, Contiki implémente une technique de gestion de la radio pour consommer moins d'énergie.

3. Dans contiki-2.7/core/sys/process.h

#define **PT_INIT**(pt)	Initialize a protothread.
#define **PT_THREAD**(name_args)	Declaration of a protothread.
#define **PT_BEGIN**(pt)	Declare the start of a protothread inside the C function implementing the protothread.
#define **PT_END**(pt)	Declare the end of a protothread.
#define **PT_WAIT_UNTIL**(pt, condition)	Block and wait until condition is true.
#define **PT_WAIT_WHILE**(pt, condition)	Block and wait while condition is true.
#define **PT_WAIT_THREAD**(pt, thread)	Block and wait until a child protothread completes.
#define **PT_SPAWN**(pt, child, thread)	Spawn a child protothread and wait until it exits.
#define **PT_RESTART**(pt)	Restart the protothread.
#define **PT_EXIT**(pt)	Exit the protothread.
#define **PT_SCHEDULE**(f)	Schedule a protothread.
#define **PT_YIELD**(pt)	Yield from the current protothread.
#define **PT_YIELD_UNTIL**(pt, cond)	Yield from the protothread until a condition occurs.

FIGURE 2.7 – Opérations sur les protothreads [39]

```
PROCESS_BEGIN(); // Declares the beginning of a process' protothread.
PROCESS_END(); // Declares the end of a process' protothread.
PROCESS_EXIT(); // Exit the process.
PROCESS_WAIT_EVENT(); // Wait for any event.
PROCESS_WAIT_EVENT_UNTIL(); // Wait for an event, but with a condition.
PROCESS_YIELD(); // Wait for any event, equivalent to PROCESS_WAIT_EVENT().
PROCESS_WAIT_UNTIL(); // Wait for a given condition; may not yield the process.
```

```
#define PROCESS_BEGIN()              PT_BEGIN(process_pt)
#define PROCESS_END()                PT_END(process_pt)
#define PROCESS_EXIT()               PT_EXIT(process_pt)
#define PROCESS_WAIT_EVENT()         PT_WAIT_UNTIL(process_pt, c)
#define PROCESS_WAIT_EVENT_UNTIL(c)  PROCESS_YIELD_UNTIL(c)
#define PROCESS_YIELD()              PT_YIELD(process_pt)
#define PROCESS_YIELD_UNTIL()        PT_YIELD_UNTIL(process_pt, c)
#define PROCESS_WAIT_UNTIL(c)        PT_WAIT_UNTIL(process_pt, c)
```

FIGURE 2.8 – Implémentation des protothreads dans les processus

2.1.3 RDC et consommation d'énergie

Dans les RCSFs, une politique de gestion de l'éveil de la radio permet une réduction de la consommation : C'est le Duty-cycling [7].

Technique du Duty-cycling [1]

Cette technique est principalement utilisée dans l'activité réseau. Le moyen le plus efficace pour conserver l'énergie est de mettre la radio de l'émetteur en mode veille (*low-power*) à chaque fois que la communication n'est pas nécessaire. Idéalement, la radio doit être éteinte dès qu'il n'y a plus de données à envoyer et ou à recevoir, et devrait être prête dès qu'un nouveau paquet de données doit être envoyé ou reçu. Ainsi, les noeuds alternent entre les périodes d'activité et de sommeil de leur radio. Ce comportement est généralement dénommé Duty-cycling. Le *Duty-cycling* se défini comme étant la fraction de temps où les noeuds sont actifs.

Comme les noeuds-capteurs effectuent des tâches en coopération, ils doivent coordonner leur date de sommeil et de réveil. Un algorithme d'ordonnancement Sommeil/Réveil accompagne le

Duty-cycling. Il s'agit généralement d'un algorithme distribué reposant sur les dates auxquelles des noeuds décident de passer entre l'état actif et l'état sommeil. Il permet aux noeuds voisins d'être actifs en même temps, ce qui rend possible l'échange de paquets, même si les noeuds ont un faible *duty-cycle* (c'est-à-dire qu'ils dorment la plupart du temps), on parle de réseaux mostly-off [4] [1]. Dans Contiki, des stratégies pour réduire au minimum le cycle de fonctionnement d'une radio sont typiquement employés dans la couche MAC, cependant, Contiki divise la couche MAC en deux composants : les pilotes MAC et RDC.

Le pilote MAC (CSMA) a pour objectif spécifique la détection des collisions radio. Contiki implémente plusieurs protocoles sur la couche RDC : ContikiMAC et X-MAC. ContikiMAC ayant la plus faible latence du fait de la gestion dynamique du *duty-cycling* [40]. Le protocole ContikiMAC est le protocole par défaut de la couche RDC.

Consommation

Dans les RCSF, la communication et la consommation d'énergie s'entrelacent. Le module de communication est le composant qui consomme le plus dans un noeud de capteur [34]. La bonne définition du *duty-cycling* permettra la minimisation de la consommation énergétique. Ainsi l'évaluation de la consommation d'un noeud de capteur s'effectue lorsqu'un capteur allume sa radio pour recevoir ou pour envoyer un paquet.

2.2 Communications dans Contiki

Cette section présente les différents modes de communication utilisés dans Contiki.

2.2.1 Structure *packetbuffer*

Le système Contiki implémente une structure qui est responsable de l'acheminement des paquets en fonction de leur destination. Cette structure est *packetbuffer* qui permet d'encapsuler les paquets entrants et sortants. A chaque *packetbuffer*, correspond un unique paquet. Il se compose de deux parties (Figure 2.9) : 1 partie entête et 1 partie donnée.

4. Contrairement aux réseaux *mostly-on* où les noeuds sont en mode éveil la plupart du temps

FIGURE 2.9 – Structure du packetbuffer [41]

Pour un paquet sortant, l'entête est stockée dans la partie entête du *packetbuffer* et sa partie donnée se trouve dans le bloc de données du *packetbuffer* comme indiqué à la Figure 2.10.

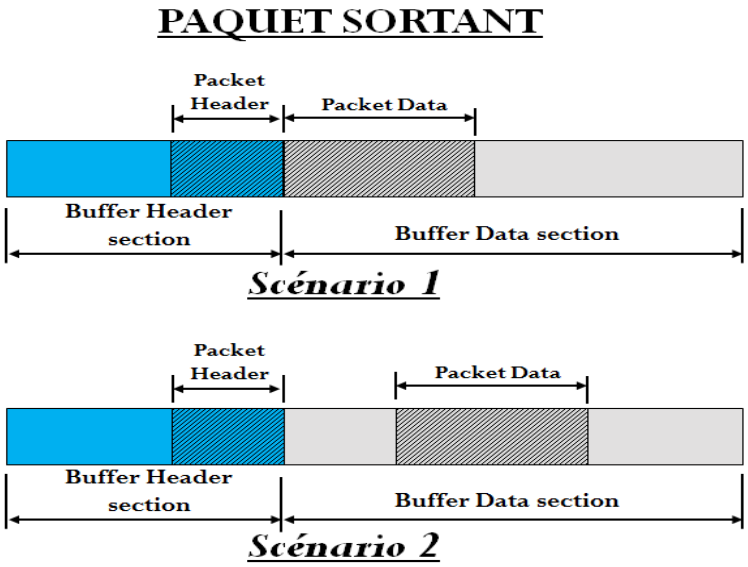


FIGURE 2.10 – Packetbuffer pour les paquets sortants [41]

Par contre, pour un paquet entrant (Figure 2.11), l'entête et les données du paquet sont stockés dans la partie donnée du *packetbuffer*. La taille du buffer est par défaut 178 octets (définis dans *~/contiki-2.7/core/net/packetbuf.h*).

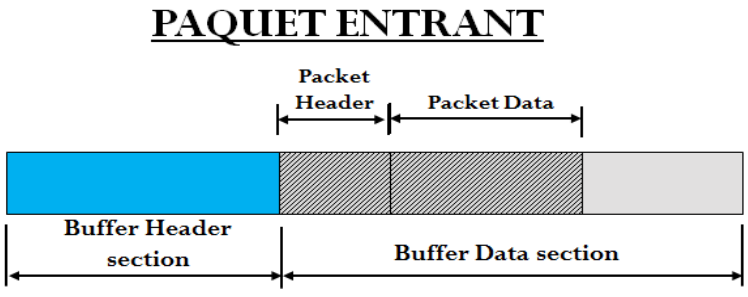


FIGURE 2.11 – Packetbuffer pour les paquets entrants [41]

La pile globale de communication de Contiki est représentée à la Figure 2.12.

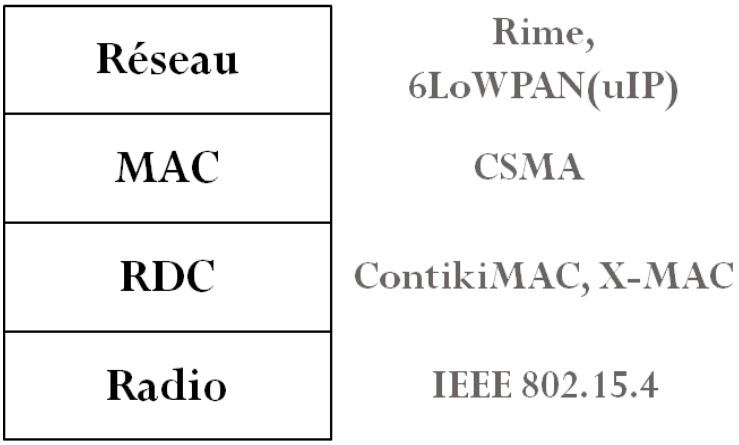


FIGURE 2.12 – Architecture des communications dans Contiki [42]

2.2.2 Pile de communication uIP

La couche uIP (micro IP), orientée Internet offre les services essentiels du protocole IP tels que HTTP, FTP, DHCP. La pile uIP permet la communication en utilisant la suite de protocoles TCP/IP, même sur de petits microcontrôleurs 8 bits. Utilisant la technologie IEEE 802.15.4 qui dispose de 127 octets comme taille maximale pour une trame, Contiki qui intègre la technologie IPv6 (dont l'entête est de 40 octets) doit fragmenter un paquet IPv6 en plusieurs trames, la couche d'adaptation 6LoWPAN a été intégrée (Figure 2.13). La couche réseau implémente le routage proprement dit où le protocole de routage RPL [43] y est implémenté.

La pile uIP qui est orientée Internet nécessite plusieurs ressources pour son fonctionnement (couche adaptation, réseau et transport) contrairement à la couche Rime qui fonction directement au-dessus de la couche physique.

2.2.3 Couche de communication Rime [17]

Rime est une couche de communication pour RCSF qui est conçu pour réduire de façon significative la complexité par rapport à la couche uIP pour le développement de protocoles de communication (Figure 2.14(a)) facilitant la réutilisation du code. Rime est implémentée dans Contiki, avec une code de moins de deux kilo-octets et pour les besoins de données de quelques dizaines d'octets, ainsi elle consommera moins de mémoire que le couche uIP (Figure 2.14(b)). Les couches sont conçus pour être extrêmement simple, à la fois en termes d'interface et de mise en œuvre.

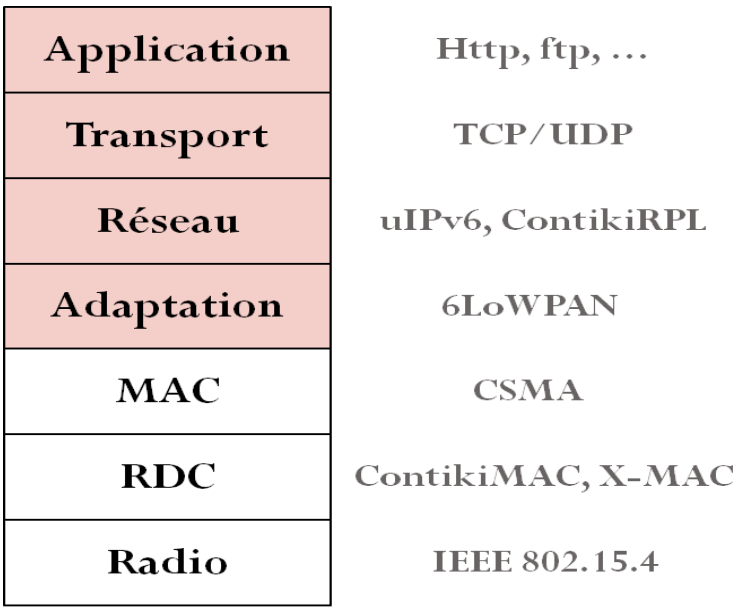


FIGURE 2.13 – Pile uIP [38]

Characteristic \ Stack	Rime	uIP
Complexity	low	high
Code Size	low	high
Energy Consumption	low	high
Funcionality	low	high
Documentation	less	more

(a)

Memory (bytes) \ Stack	Rime	uIP & IPv6
Program	20728 (15.8%)	37416 (28.5%)
Data	1776 (10.8%)	4076 (24.8%)
EEPROM	8 (0.2%)	8 (0.2%)

(b)

FIGURE 2.14 – Comparaison Rime et uIP [24]

Rime s'organise en couches comme présenté à la Figure 2.15. Chaque couche étant conçue pour être extrêmement simple [17].

L'anonymous brodcast (*abc*) [5] : Il envoie simplement un paquet par l'intermédiaire du pilote radio, reçoit tous les paquets à partir du pilote radio et les transmet à la couche supérieure. C'est la primitive la plus basique et est utilisée par toutes les autres primitives.

5. Contiki-2.7/core/net/rime/abc

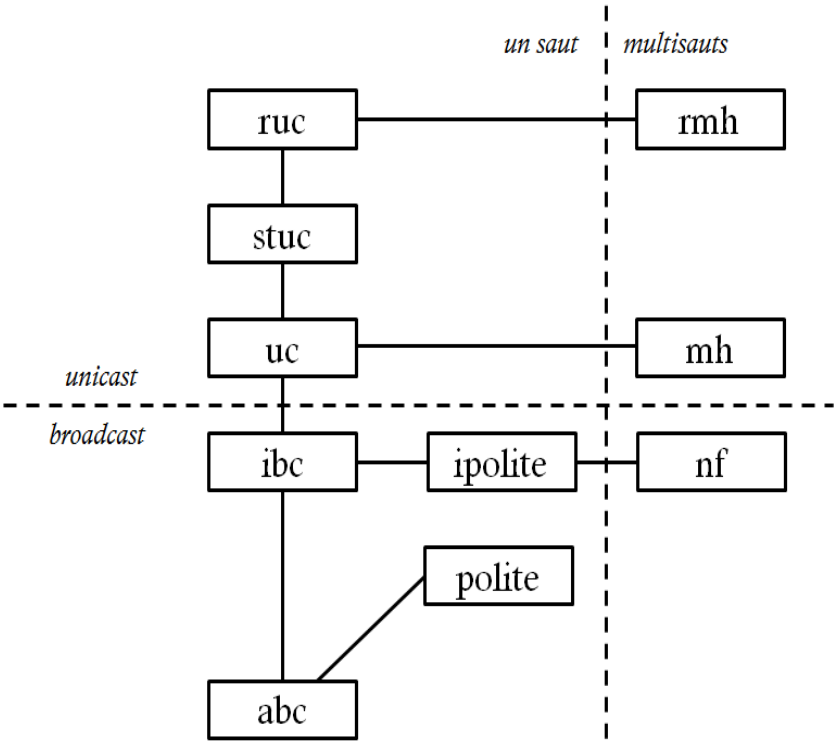


FIGURE 2.15 – Organisation de la couche Rime [38]

L'identified broadcast (*ibc*) [6] : Elle effectue la même tâche que la primitive *abc* tout en incluant dans le packetbuffer l'adresse de l'émetteur.

L'unicast (*uc*) [7] : Cette primitive s'appuie sur le *ibc* et y ajoute l'adresse du destinataire. Si l'adresse de destination du paquet ne correspond pas à l'adresse du noeud, le paquet est rejeté.

Le stubborn unicast (*stuc*) [8] : Elle permet d'envoyer à plusieurs reprises pendant une période de temps donnée un paquet à un nœud, jusqu'à ce qu'une couche supérieure annule la transmission. Ce module n'est généralement pas utilisé telle qu'est, mais plutôt par le module suivant.

Le reliable unicast (*ruc*) [9] : Il envoie un paquet à l'aide du module *stuc* et attend l'accusé de réception d'un paquet. Lorsque l'accusé est reçu, il arrête la transmission en continu du paquet. Un

6. Contiki-2.7/core/net/rime/broadcast
7. Contiki-2.7/core/net/rime/unicast
8. Contiki-2.7/core/net/rime/stunicast
9. Contiki-2.7/core/net/rime/runicast

certain nombre de retransmission maximale est spécifié par le programmeur afin d'éviter l'envoi infini.

Le *polite* [10] : Cette primitive envoie en broadcast à un saut un paquet dans un intervalle de temps donné si aucun paquet avec la même en-tête n'a été reçu dans cet intervalle.

Le *ipolite* [11] : C'est une variante de *polite* dans laquelle l'information sur l'émetteur du paquet est ajoutée.

Le multihop (*mh* et *rmh*) [12] : Ces deux variantes fournissent les outils pour effectuer du multisauts et ne sont pas des algorithmes de routage. Il demande à la table de routage l'identifiant du prochaine saut et envoie le paquet à l'aide de *uc*. Quand il reçoit un paquet, si le nœud est la destination alors le paquet est passé à la couche supérieure, sinon il demande à nouveau à la table de routage le prochain saut et relaie le paquet vers ce dernier.

Le network flooding (*nf*) [13] : Envois un paquet à tous les nœuds du réseau, en utilisant *polite* à chaque saut afin de réduire le nombre de transmission redondante. Il définit de bout en bout sur les paquets sortants l'expéditeur et son identifiant afin d'éviter les retransmissions.

La couche Rime supporte à la fois les primitives de communication simple-saut et multisauts, où *abc* est la primitive de bas niveau.

Les Communications dans la couche Rime utilisent différents canaux logiques. Chaque canal possède son propre ensemble de protocoles et d'attributs. Ces canaux logiques sont ouvertes au moment de l'exécution et les parties qui communiquent doivent convenir à l'avance sur l'ensemble particulier de protocoles à utiliser pour un canal particulier. Ce sont des nombres sur 16 bits (toutes les valeurs inférieures à 128 sont réservées au noyau) [44]. Par exemple, deux applications fonctionnant sur deux noeuds différents communiquent entre eux en s'accordant sur un même canal logique.

10. Contiki-2.7/core/net/rime/polite
11. Contiki-2.7/core/net/rime/ipolite
12. Contiki-2.7/core/net/rime/multihop et Contiki-2.7/core/net/rime/rmh
13. Contiki-2.7/core/net/rime/netflood

2.3 Conclusion

Dans ce chapitre, nous avons présenté le système d'exploitation hybride pour RCSF Contiki qui dispose d'un noyau événementiel et permet une exécution préemptive de threads. Il permet également deux modes de communication aux utilisateurs : un mode orienté Internet et un autre utilisant la radio embarquée dans les capteurs. Ce deuxième mode implémenté par la couche Rime permet cependant d'avoir une meilleure consommation que la première couche. Dans la suite de notre document nous utiliserons la couche faible consommation Rime.

Le chapitre suivant présente les expérimentations sur la gestion du multitâche à travers une communication multisauts implémentée dans la couche Rime qui présente une meilleure consommation que le couche uIP.

CHAPITRE 3

EXPÉRIMENTATIONS SUR LA GESTION DU MULTITÂCHE DANS RIME

Ce chapitre présente les tests réalisés sur les architectures Normal et Thread lors d'un envoi multisauts implémenté dans la couche faible consommation Rime. Il s'articule autour de trois parties : la première présente l'environnement expérimental ; la seconde fait un éclaircissement sur la méthodologie ; la dernière partie expose les résultats obtenus après les expérimentations.

3.1 Environnement expérimental

Dans cette section nous présentons les outils matériels et logiciels utilisés pour réaliser nos expérimentations.

3.1.1 Architecture de la machine

Les expérimentations ont été menées sur un ordinateur HP envy 15 Notebook. Il dispose d'un processeur Intel core i7-4710HQ quad-core de la famille Haswell [1] cadencé à 2.5GHz et se dote d'une mémoire de 8GB et 1 Téra de stockage.

3.1.2 Système d'exploitation

Les expérimentations sont réalisées dans le système Linux Ubuntu 14.04 i386 LTS (Trusty Tahr) qui est une version stable d'Ubuntu obtenue à travers le miroir de l'Université de Ngaoun-

1. 4ième génération de processeur Intel core i7

déré (http ://iso.univ-ndere.cm/ubuntu/). Sur ce système, nous avons utilisé contiki-2.7 [2] qui est la dernière version du système à ce jour.

3.1.3 Environnement de simulation

Nous utilisons l'émulateur **COOJA** [3] [16]. Ce dernier est intégré au système d'exploitation Contiki (contiki-2.7/tools/cooja), ne disposant pas de capteur réel, il émule le comportement de certains capteurs parmi lesquels TelosB qui fournit de meilleures rapport qualité/prix (Tableau 3.1)

TABLEAU 3.1 – Présentation des capteurs intégrés dans COOJA [45]

Capteurs	Mémoire Flash	RAM	Débit	Tension Max	Prix (FCFA)
MicaZ	512 Kb	4 Kb	250 Kpbs	2.7 V	60 000
Mica2	512 Kb	4 Kb	38.4 Kpbs	2.7 V	70 900
TelosB	1024 Kb	10 Kb	250 Kpbs	1.8 V	50 500

COOJA permet d'analyser le fonctionnement d'une application dans un capteur, son utilisation est un atout dans l'estimation du coût de déploiement d'un RCSF car il permet également en fonction du capteur de connaître la zone de communication et d'interférence (Figure 3.1). COOJA facilite en outre le débogage d'applications.

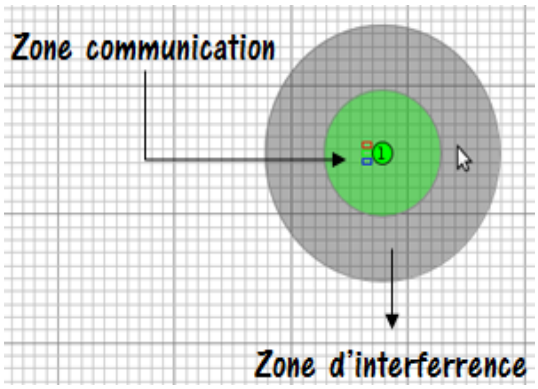


FIGURE 3.1 – Zone de communication d'un capteur TelosB

L'émulateur COOJA fournit à l'utilisateur la capacité de simuler un réseau de capteurs en utilisant les couches Rime [4] et uIP [5]. L'évaluation de la consommation énergétique se fait à l'aide

2. http ://www.sics.se/contiki-2.7
3. COOJA = COntiki Os JAva developpé en Java
4. Fonctionnalités dans contiki-2.7/core/net/rime/
5. Implémenter dans contiki-2.7/core/net/

du module powertrace [6]. En fonction du temps mis par le capteur en mode radio lors d'un envoi (Tx) ou de la réception (Rx) d'un paquet, permet une estimation de la consommation d'énergie d'un capteur.

Pour lancer COOJA, il suffit de se diriger en mode terminal vers le répertoire */cooja* et lancer le simulateur.

cd /contiki-2.7/tools/cooja

ant run

La Figure 3.2 représente la fenêtre de COOJA après la création d'une nouvelle simulation constitué de 5 principaux champs :

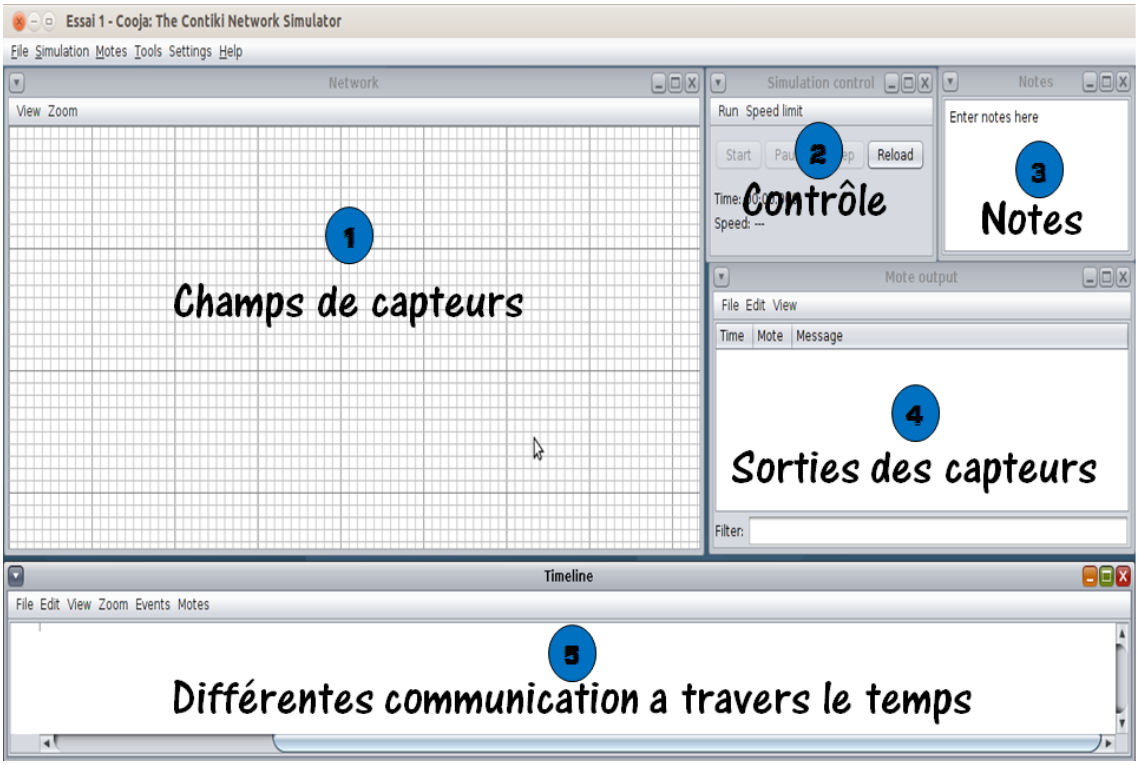


FIGURE 3.2 – Différentes zones dans une simulation de COOJA

1 Network : Il présente les différents capteurs dans le RCSF. Ces derniers sont représentés à l'aide de coordonnées.

2 Simulation control : Il permet le lancement, la suspension et la reprise d'une simulation.

3 Notes : Cette zone permet au programmeur de poster des notes concernant la simulation.

4 Mote output : Il imprime toutes les sorties des capteurs.

6. contiki-2.7/apps/powertrace/

5 Timeline : Il affiche les différentes communications (réception, transmission, collision et mise en veille du capteur) à travers le temps.

3.2 Objectifs expérimentaux et méthodologie

3.2.1 Objectifs expérimentaux

Le but des expériences réalisées est l'étude de l'efficacité d'un envoi multisauts en utilisant les architectures Normal et Thread sous Contiki . Il sera question pour nous d'étudier pendant une période le nombre de message que peut recevoir un noeud lorsqu'il est mitraillé d'envois. Une observation sur la consommation énergétique est également faite.

3.2.2 Méthodologie

Afin d'effectuer les expérimentations sur l'envoi multisauts, nous avons utilisé la bibliothèque par défaut *multihop.h*[7]dans un premier temps, dans notre réseau expérimental. Pour cela nous considérons un noeud central (*inter*) qui devra recevoir des paquets envoyés par certains noeuds (*senders*) et les transférer vers un autre noeud récepteur (*receiver*). L'idée étant qu'un noeud *sender* ne peut pas communiquer directement avec un noeud *receiver*. Ainsi, il envoie un paquet en multisauts à travers le noeud *inter*. Les positions des noeuds du réseau expérimental étudié (Figure 3.3) sont données par le Tableau 3.2.

- Le noeud 1 est *l'inter*
- Les noeuds 2 à 16 sont les *senders*
- Le noeud 17 est un *receiver*

Dans un second temps, nous modifions la bibliothèque *multihop.h* de telle sorte que après la réception d'un paquet à transférer par le noeud inter, ce dernier créé un thread responsable de transférer le paquet du *noeud sender* au *noeud receiver*. Les noeuds utilisés sont des capteurs *TelosB* qui présentent de meilleure performance (Tableau 3.1) et implémentent le protocole ContikiMAC dans son RDC.

7. contiki-2.7/core/net/rime/multihop.h

TABLEAU 3.2 – Coordonnées des noeuds dans le RCSF d'étude

Numéro Noeud	Axe (x-x')	Axe (y-y')
1	95.099	55.277
2	63.892	33.496
3	63.413	50.411
4	105.958	86.986
5	68.541	80.854
6	77.718	91.932
7	76.697	62.781
8	88.096	71.909
9	70.237	54.939
10	95.685	92.837
11	95.403	81.934
12	68.188	45.108
13	101.094	81.764
14	70.049	36.727
15	73.013	50.47
16	82.314	68.921
17	123.692	32.81

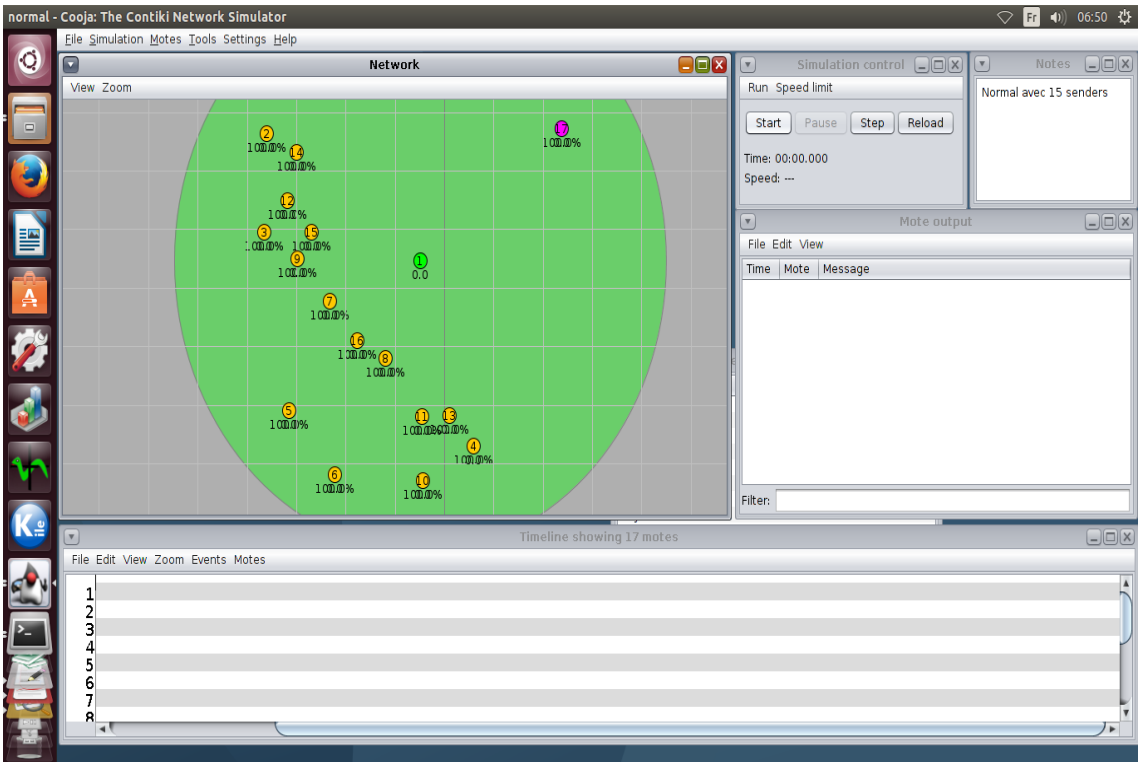


FIGURE 3.3 – Réseau expérimental dans COOJA

Ces deux modes de transfert sont analysés sur un échantillon de 30 cycles par *senders* où un cycle correspond à un envoi du *sender* et la période entre 2 cycles étant fixée à *2 secondes*. Pour chacun de ces tests, on varie le nombre de *senders* de 1 à 15, pendant 30 cycles afin d'observer le nombre de messages reçus par *l'inter* de même que sa consommation moyenne. Une dizaine de tests a été faite et présente des valeurs similaires. Toutefois, la moyenne des résultats est établie. Concernant l'évaluation de la consommation énergétique, nous utilisons la formule pour un capteur TelosB et donnée dans le Tableau 3.3 ; le taux d'utilisation de la radio est aussi évalué à l'aide de l'outil *PowerTracker* de Cooja (Figure 3.4)

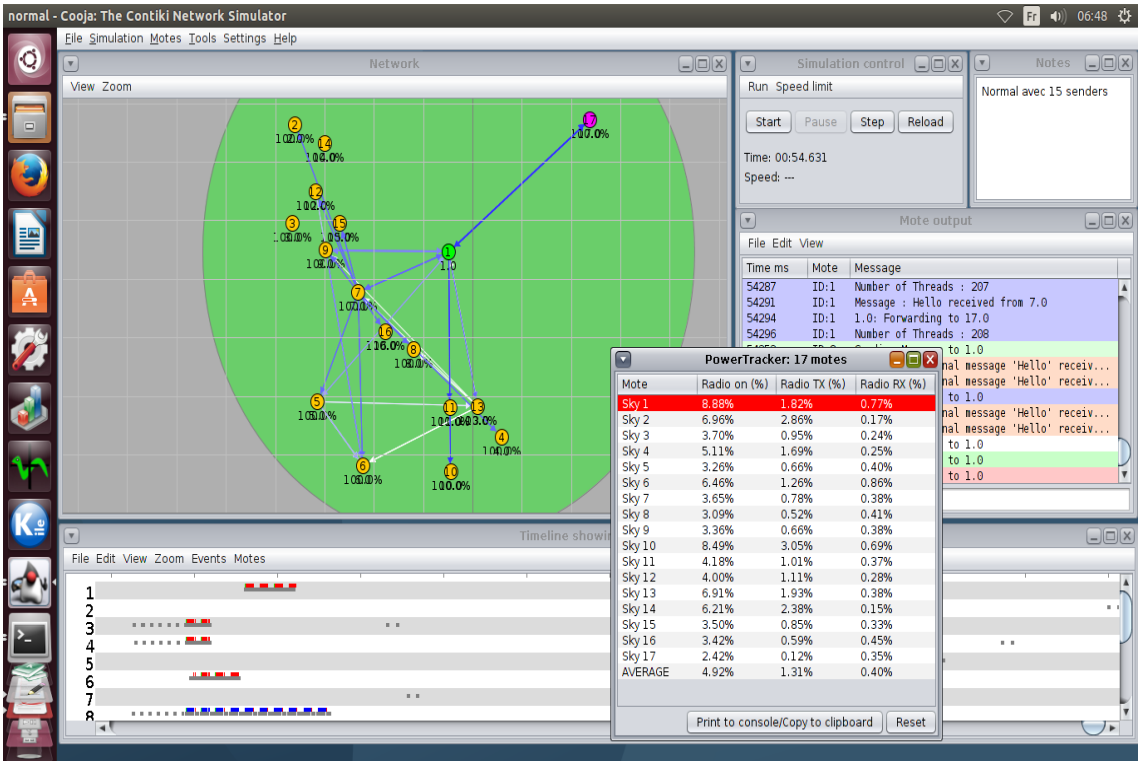


FIGURE 3.4 – Utilisation PowerTracker

TABLEAU 3.3 – Evaluation de la consommation énergique pour un capteur TelosB [46]

Power for RX, TX et cpu_ON
Power (mW) = ((rx + tx + ON)*20mA*3V) / (4096*runtime(seconds))

Où rx, tx et ON représente respectivement le temps passer par la radio en mode RX, TX et cpu_ON respectivement et s'obtiennent à l'aide de *powertrace* (Figure 3.5).

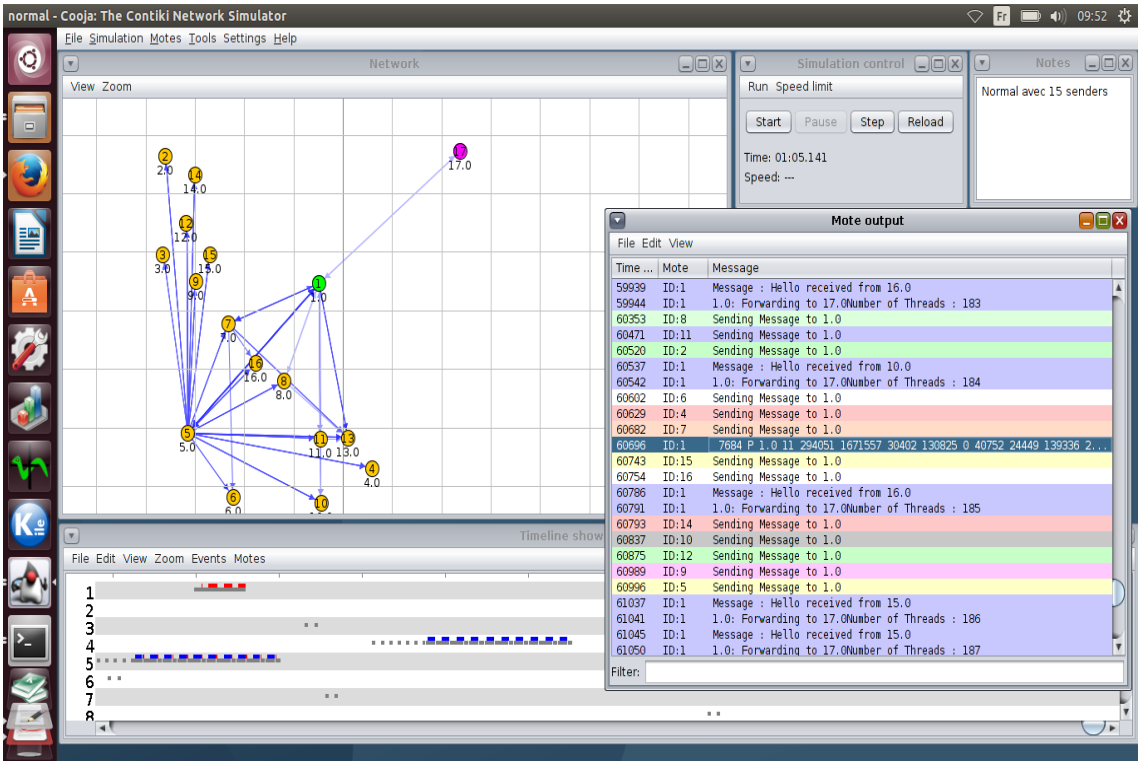


FIGURE 3.5 – Utilisation powertrace

3.2.3 Applications de test

Nous présentons les deux modèles d'envoi mutltisauts possibles dans Contiki que nous avons implémenter dans notre travail.

Envoi multisaut normal

Nous utilisons les fonctions proposées par le module ***multihop.h*** de façon native et implémentées comme suit :

- **void *multihop_open***(*struct multihop_conn *c, uint16_t channel, const struct multihop_callbacks *callbacks*) : Elle ouvre la connexion *c* sur le canal de communication *channel*. Le paramètre *callbacks* est programmable par l'utilisateur et il définit l'action d'un capteur après réception d'un paquet. C'est le cas de *Neighbor Discovery* [47] qui permet de construire la table de tous les voisins directs d'un noeud.
- **void *multihop_close***(*struct multihop_conn *c*) : Cette fonction ferme la connexion *c* préalablement ouverte.

– **int *multihop_send***(*struct multihop_conn *c, const rimeaddr_t *to*) : Cette fonction effectue un envoi *unicast* vers le noeud **to* qui parcours une table contenant les voisins du noeud sur lequel il s'exécute.

– **void *multihop_resend***(*struct multihop_conn *c, const rimeaddr_t *nexthop*) : Elle permet de renvoyer un paquet vers un noeud voisin

Les différents noeuds utilisés se connectent au canal *CHANNEL=150*, les différents *senders* envoient un paquet dans lequel est écrit "*hello*". Ce paquet est retransmis par *l'inter* et seul le noeud *receiver* peut ouvrir (car étant l'unique destinataire). Lorsque *inter* reçoit un paquet qu'il doit transférer, et si le *receiver* est son voisin, alors *inter* envoie le paquet grâce à la fonction *unicast_send(struct multihop_conn *c, const rimeaddr_t *receiver)* qui envoi directement le paquet vers *receiver* telle qu'expliquer dans la Figure 3.6.

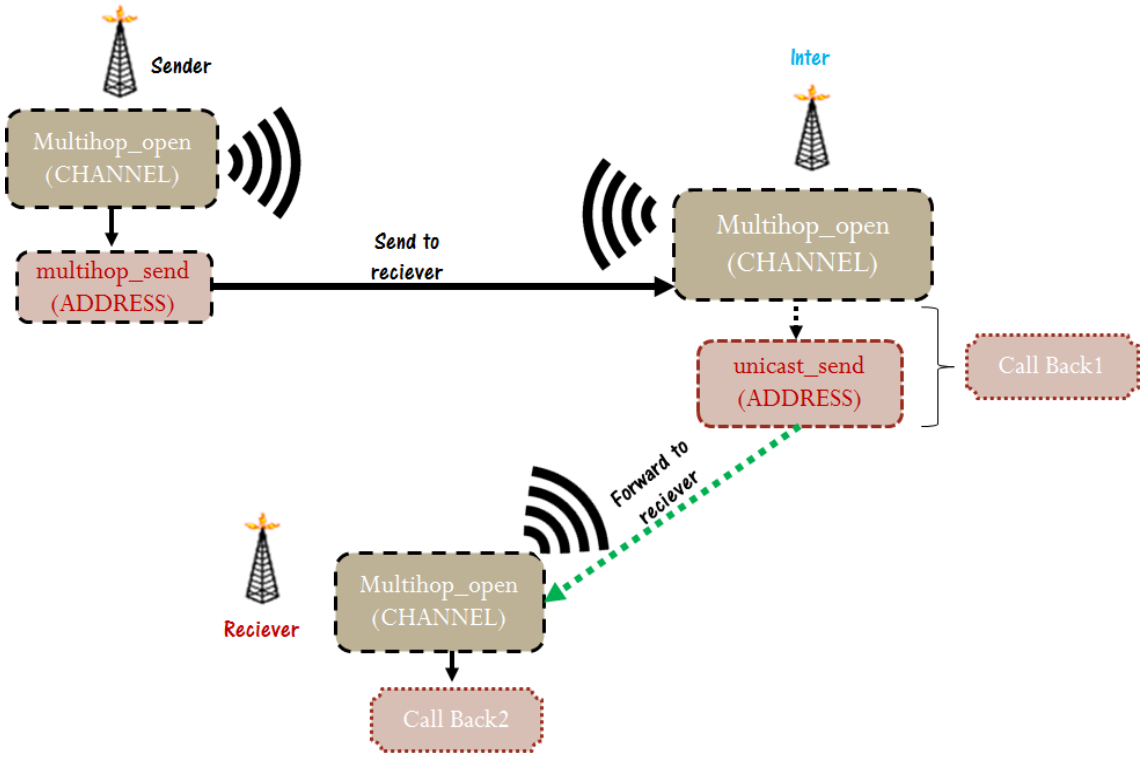


FIGURE 3.6 – Envoi multisauts normal (1 saut)

Envoi multisauts avec threads

Le noeud *inter* implémente la fonction dans laquelle un thread est créé lors de la réception d'un paquet et ce thread transmet le paquet au *receiver* grâce à la fonction *unicast_send* (Figure 3.7).

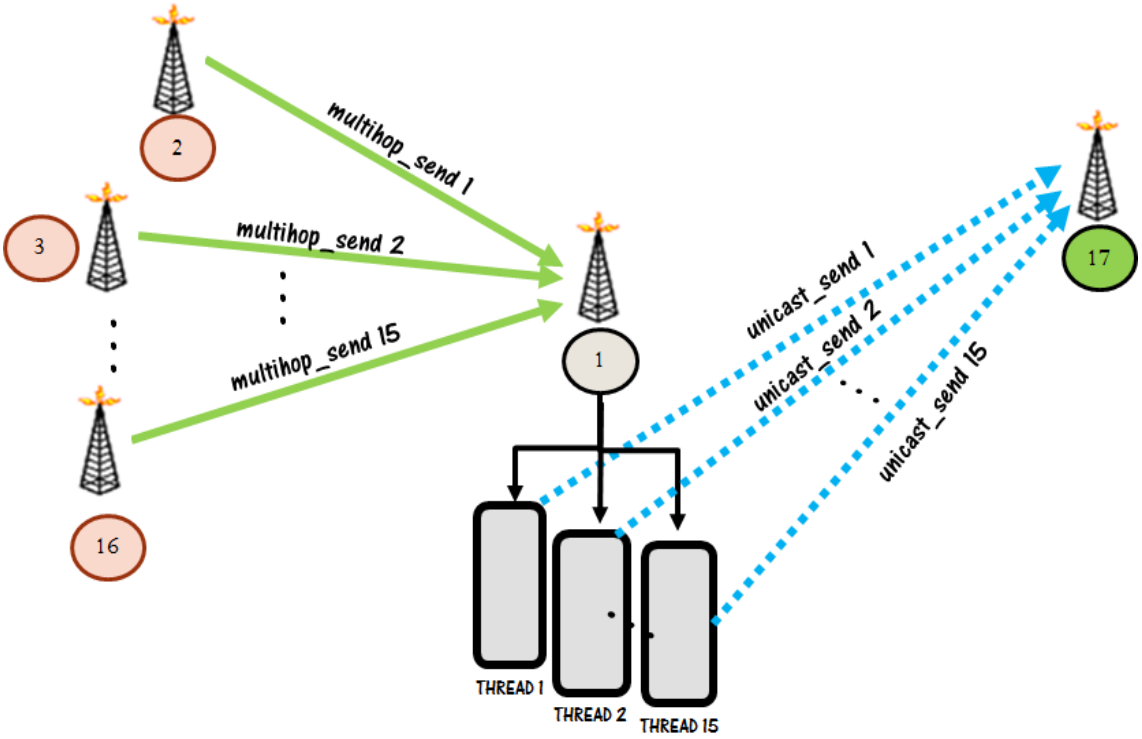


FIGURE 3.7 – Envoi multisauts avec threads

L'application des hypothèses précédentes nous permet d'obtenir les résultats suivants.

3.3 Résultats et discussions

Les résultats du Tableau 3.4 obtenus lors des expérimentations présentent le nombre de messages reçus par *inter* après 30 cycles en faisant varier le nombre de *senders* de 1 à 15. La Figure 3.8 présente l'évolution du nombre de messages reçus en fonction du nombre de capteur lors de l'envoi multisauts implémenté dans les architectures Normal et Thread de façon graphique.

Les résultats obtenus à la Figure 3.8 montrent que l'envoi multisauts avec threads permet à un noeud de recevoir plus de paquet par rapport à l'envoi normal durant un même intervalle de temps. En effet, l'envoi avec threads reçoit 5,37% messages en plus que celui normal.

TABLEAU 3.4 – Résultats nombres de messages reçus Normal-Thread

Nbre de Senders	Msg-Envoyés	Msg-Reçus-Normal	Msg-Reçus-Threads
1	30	30	30
2	60	42	56
3	90	57	70
4	120	90	94
5	150	90	120
6	180	134	151
7	210	146	165
8	240	177	179
9	270	179	202
10	300	172	205
11	330	201	213
12	360	198	218
13	390	189	177
14	420	169	237
15	450	188	218

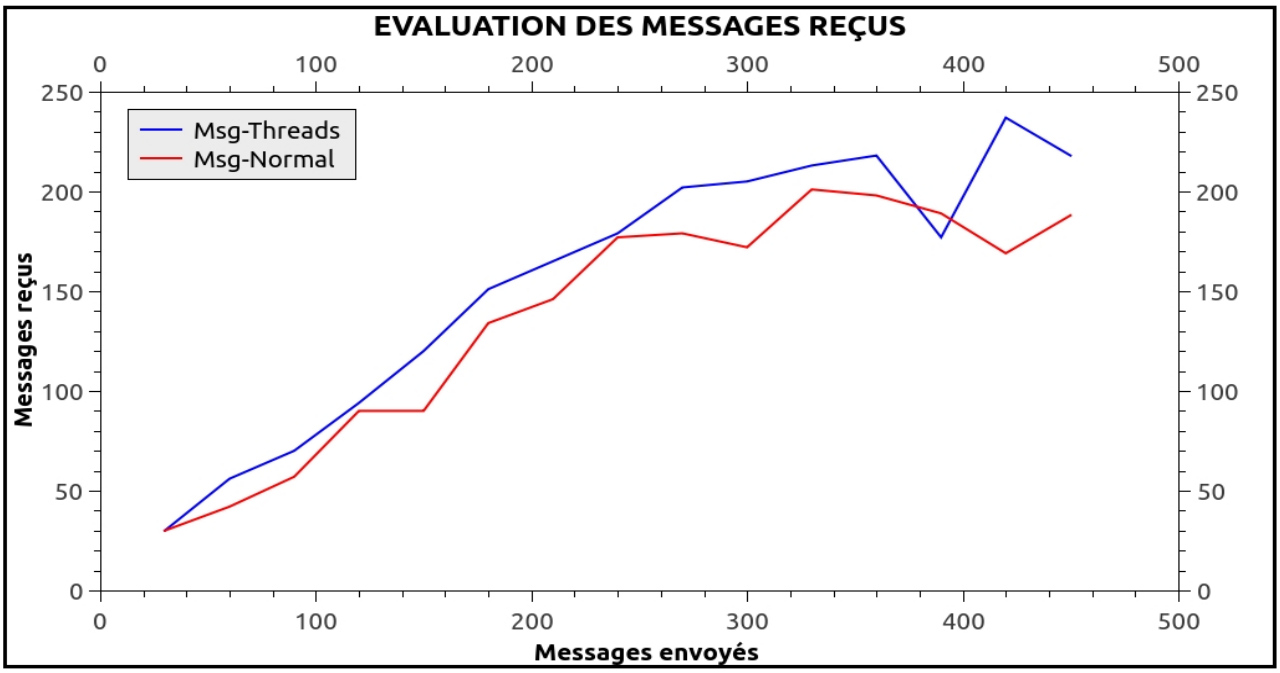


FIGURE 3.8 – Comparaison nombre de messages reçus Normal-Thread

Dans le Tableau 3.5, une comparaison entre le module Normal et celui utilisant les threads est présentée par rapport au taux d'utilisation de la radio (en pourcentages %) en mode écoute (ON), réception (RX) et transmission (TX). La Figure 3.9 en présente l'évolution graphique en fonction du nombre de *senders*. On y remarque que l'architecture implémentant les threads utilise moins sa radio, par conséquent elle l'allume la moins par rapport à l'architecture normale. Ce résultat pourrait s'expliquer par le fait que le module implémentant les threads permet un traitement plus rapide des données, soit de retransmettre plus rapidement les paquets à destination du *receiver*.

TABLEAU 3.5 – Résultats taux utilisation radio Normal-Thread

Nbre de Senders	**Taux Normal(%)**	**Taux Threads(%)**
1	3.93	2.26
2	5.19	4.03
3	6.33	5.33
4	8.85	7.56
5	8.47	7.32
6	9.51	7.01
7	10.42	8.99
8	11.35	9.5
9	11.12	11.4
10	11.57	11.49
11	12.82	11.47
12	11.59	11.23
13	12.25	10.35
14	11.61	12.15
15	11.63	11.4

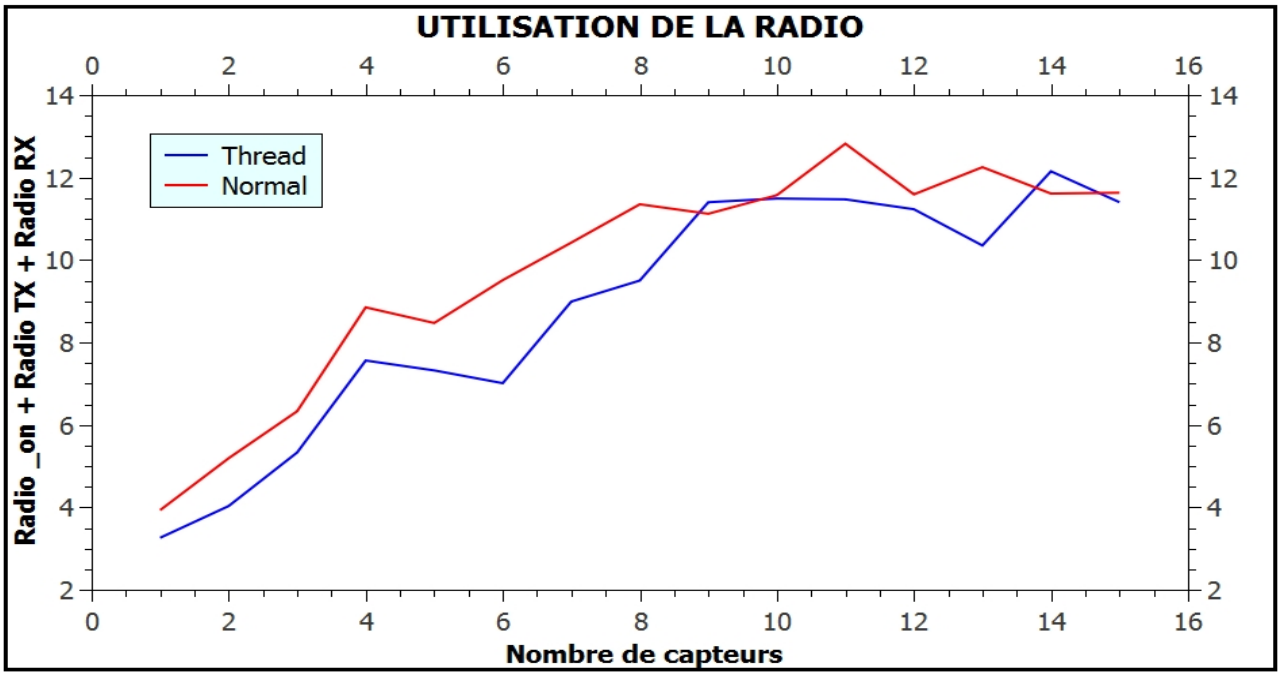


FIGURE 3.9 – Comparaison taux utilisation radio Normal-Thread

Le Tableau 3.6 présente la comparaison de la consommation en milliWatts (mW) de la consommation énergétique du noeud *l'inter* entre les envois multisauts normal et celui avec threads. On remarque que les consommations entre ces deux architectures sont dans l'ensemble identiques. Toutefois, il est important de noter que la consommation énergétique de l'architecture avec threads est plus conséquente lorsque le nombre de *senders* est réduit (moins de 8) et s'explique par le fait que chaque thread créé dispose d'une pile mémoire. La Figure 3.10 permet d'illustrer ces résultats graphiquement. Le chevauchement observé à la Figure 3.10 montre qu'entre l'envoi multisauts normal et celui avec threads il n'y a pas de meilleur, le modèle normal consommant en moyenne 0.075 mW de plus que celui avec threads. Donc l'utilisation de l'envoi multisauts avec threads ne consomme pas plus d'énergie que l'envoi multisauts normal.

TABLEAU 3.6 – Résultats consommations énergétiques Normal-Thread

Nbre de Senders	Consommation Normal(mW)	Consommation Threads(mW)
1	2.5956	4.5877
2	3.1021	3.4757
3	3.0470	5.7267
4	3.9621	4.70905
5	4.5368	5.1678
6	7.2492	6.4264
7	9.3301	9.6382
8	7.9583	10.2538
9	6.1979	8.1779
10	8.4618	9.1186
11	10.3861	10.1320
12	12.9091	9.3255
13	12.8760	8.0336
14	8.6349	5.5132
15	9.4236	9.3250

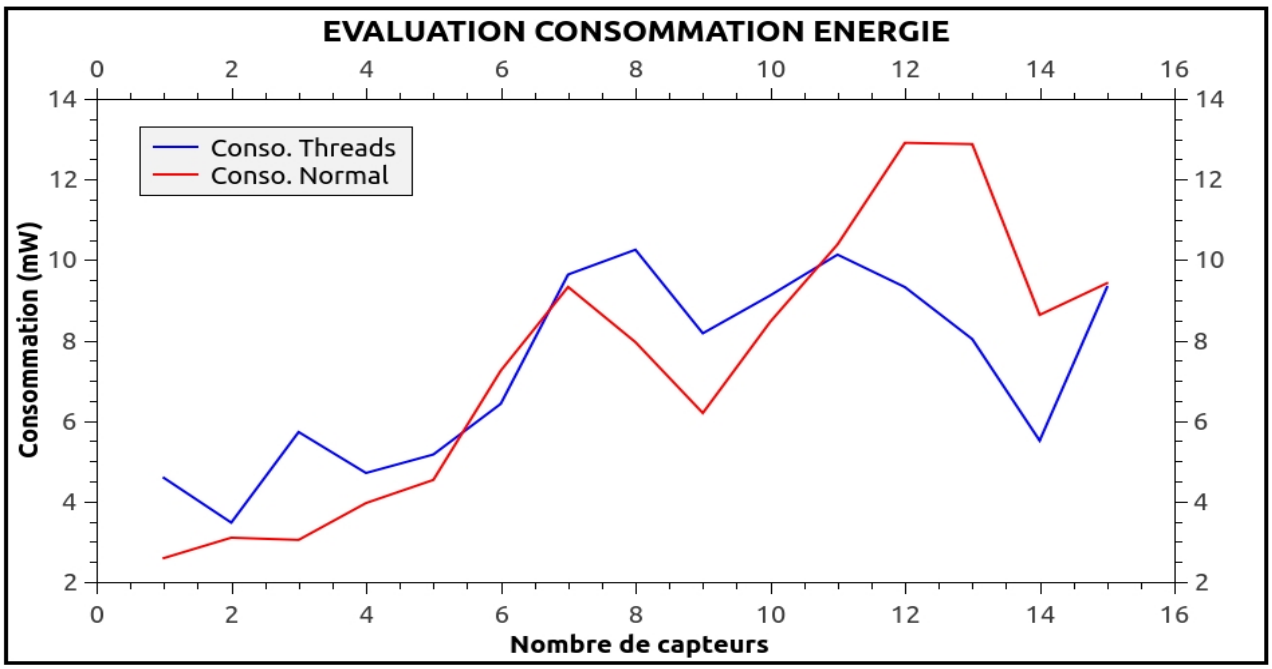


FIGURE 3.10 – Comparaison consommations énergétiques Normal-Thread

3.4 Conclusion

Dans ce chapitre, nous avons menés des expérimentations sur deux méthodes d'envoi multi-sauts (Normal et Thread) en faisant varier le nombre de *senders* qui envoient chacun des paquets à travers *inter* à destination de *receiver*. Nous avons remarqué que malgré le fait que l'utilisation de threads dans les RCSF est coûteuse en termes de ressources, cette dernière offre une meilleure performance en terme de rapidité de traitement (reçoit 5,37% de message en plus qu'une architecture normale).

Dans le prochain chapitre, nous proposons une nouvelle architecture multithreadée facile d'utilisation à l'aide des protothreads qui permet une meilleure consommation énergétique.

CHAPITRE 4

PROPOSITION D'UNE ARCHITECTURE EFFICACE MULTITHREADÉE POUR LES RCSFS

A l'issue des expérimentations menées dans le chapitre précédent, il en ressort que l'utilisation du multhreading lors de l'envoi multisauts est efficace pour plusieurs requêtes, toutefois, elle présente une consommation plus importante.

Dans ce chapitre, nous présentons dans un premier temps notre architecture alternative *Prototype* pour la gestion multitâche. Dans un second temps, une présentation des fonctions de ces architectures est faite avant de terminer par la validation des résultats obtenus.

4.1 Architecture proposée

Dans cette section, l'architecture multithreadée que nous proposons lors d'envoi multisauts est présentée. La Figure 4.2 illustre le fonctionnement interne de cette architecture.

Dans la Figure 4.1, lorsqu'un noeud reçoit un paquet à retransmettre ce dernier créé un autre processus qui sera responsable de la retransmission. Sachant que dans le système Contiki, un processus est l'équivalent d'un protothread, le premier protothread *PT_recv* est responsable de l'écoute sur un canal de communication. C'est ce dernier qui reçoit la requête d'un *sender*, s'il s'agit d'une retransmission alors un protothread *PT_send* est créé qui finalise l'envoi vers le noeud destinataire. Tout d'abord le noeud se met en écoute sur un canal de communication avec *multihop_open()*, puis à l'arrivée d'un nouveau paquet, la structure *packetbuffer* permet de savoir s'il

s'agit ou pas d'une retransmission. Si oui, la fonction *callback* est alors exécutée et déclenche à l'aide de la fonction *process_start()* le protothread *PT_send* qui effectue la livraison du paquet grâce à *unicast_send()* avant de s'arrêter comme montrer à la Figure 4.2.

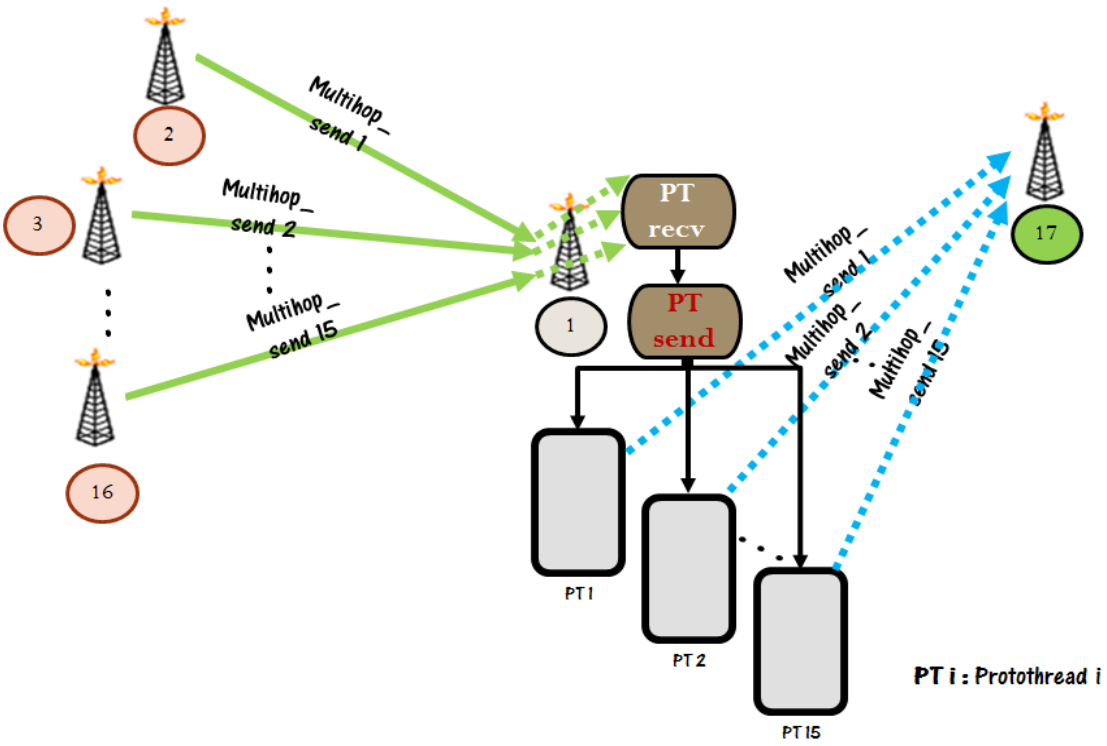


FIGURE 4.1 – Réseau utilisant le module *prototype*

4.2 Fonctions de la bibliothèque

Le module *prototype* que nous présentons utilise les mêmes fonctions de base que la bibliothèque *multihop.h* à la seule différence que le mécanisme de fonctionnement utilise les fonctions de la Figure 4.3.

- ***recv_uc2()*** : Elle représente la fonction callback2 qui est exécutée uniquement par le destinataire final (dans notre cas *receiver*). Elle lit et affiche la donnée du packetbuffer qui a été modifiée par l'expéditeur.
- ***forward()*** : Elle permet d'effectuer la retransmission et s'exécute dans *PT_send* et est responsable de la livraison du paquet au destinataire. Elle est déclenchée par le noeud qui effectue la retransmission (*inter*).

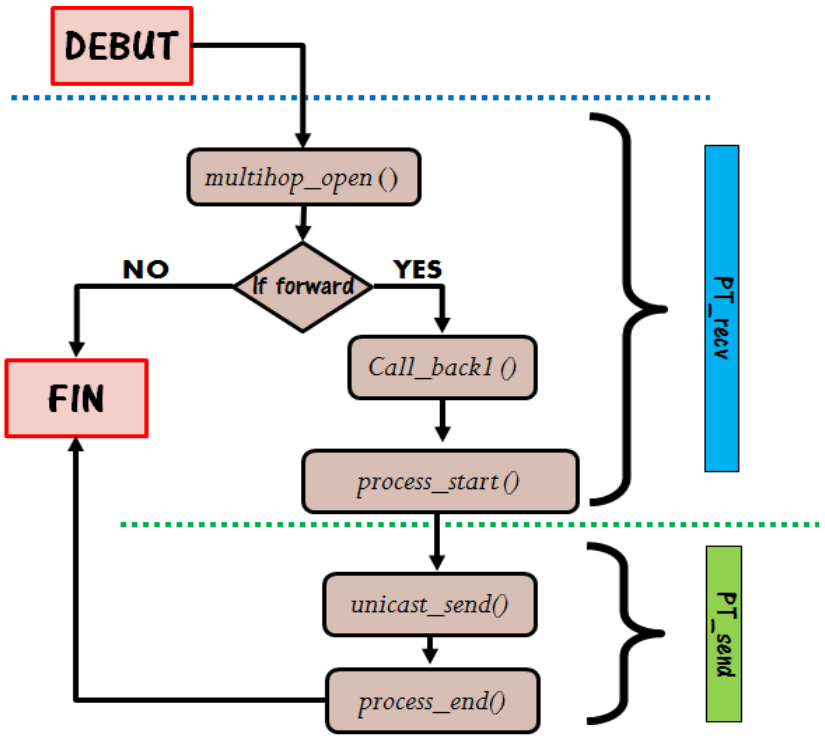


FIGURE 4.2 – Architecture de l'envoi multisauts dans *prototype*

- ***process_start()*** : Elle permet de déclencher instantanément *PT_send* dans *callback1*, en postant un événement synchrone.
- ***recv_mc()*** : Elle équivaut à *callback1* et se déclenche à la suite d'un paquet à transmettre. Elle contient la fonction *process_start()* qui démarre *PT_send*.

L'implémentation des protothreads *PT_recv* et *PT_send* est donnée par la Figure 4.4. *PT_recv* est démarré dès le début de l'application (AUTOSTART_PROCESSES(&PT_recv)) et met le noeud en écoute sur le canal *CHANNEL* (*multihop_open(&mc, CHANNEL, &multihop_callbacks1)*). *PT_send* quant à lui effectue juste la transmission finale du paquet vers le noeud destinataire. La fonction *powertrace_start(CLOCK_SECOND * 5)* permet d'évaluer la consommation énergétique du noeud toutes les 5 secondes.

```
/*---------------------------------------------------------------------------*/
static void
recv_uc2(struct unicast_conn *c, const rimeaddr_t *from)
{
        printf(" Final message %s received from %d.%d\n",
         (char *)packetbuf_dataptr(), from->u8[0], from->u8[1]);
}
/*---------------------------------------------------------------------------*/
static const struct multihop_callbacks multihop_callbacks2 = {recv_uc2};
static struct multihop_conn mc;
static struct unicast_conn uc;
/*---------------------------------------------------------------------------*/
int
forward(struct multihop_conn *c, const rimeaddr_t *receiver)
{
  printf("%d.%d: Forwarding to %d.%d ",
         rimeaddr_node_addr.u8[0],rimeaddr_node_addr.u8[1],
         receiver->u8[0], receiver->u8[1]);
  packetbuf_set_addr(PACKETBUF_ADDR_RECEIVER, receiver);
  return broadcast_send(&c->c);
}
/*---------------------------------------------------------------------------*/
int
process_start(struct process *p, const char *arg)
{
  struct process *q;
  p->next = process_list;
  process_list = p;
  p->state = PROCESS_STATE_RUNNING;

 /* Post a synchronous initialization event to the process. */
  process_post_synch(p, PROCESS_EVENT_INIT, (process_data_t)arg);

        return 0;
}
/*---------------------------------------------------------------------------*/
static void
recv_mc(struct multihop_conn *c, const rimeaddr_t *from)
{
        process_start (&PT_send,"");
}
/*---------------------------------------------------------------------------*/
```

FIGURE 4.3 – Fonctions utilisées dans *Prototype*

```
/*---------------------------------------------------------------------------*/
PROCESS(PT_recv, "Protothread d'écoute");
PROCESS(PT_send, "Protothread d'envoi");
AUTOSTART_PROCESSES(&PT_recv);
/*---------------------------------------------------------------------------*/
PROCESS_THREAD(PT_recv, ev, data)
{
  PROCESS_EXITHANDLER(multihop_close(&mc);)

  PROCESS_BEGIN();

        multihop_open(&mc, CHANNEL, &multihop_callbacks1);
        powertrace_start(CLOCK_SECOND * 5); // Evaluation consommation

  PROCESS_END();
}
/*---------------------------------------------------------------------------*/
PROCESS_THREAD(PT_send, ev, data)
{
        static rimeaddr_t addr1;

                addr1.u8[0]=17; // Adresse du noeud receiver
                addr1.u8[1]=0;

  PROCESS_BEGIN();

                unicast_open(&uc, CHANNEL, &multihop_callbacks2);
                forward (&uc, &addr1);

  PROCESS_END();

}
/*---------------------------------------------------------------------------
```

FIGURE 4.4 – Fonctions implémentant *PT_recv* et *PT_send*

4.3 Validation expérimentale

Dans cette section, nous présentons les résultats de notre architecture *prototype* par rapport aux architectures Normal et Thread lors de l'envoi multisauts suivant 3 facteurs : La réception de messages, le taux d'utilisation de la radio et la consommation énergétique.

4.3.1 Réception des messages

Normal - Prototype

Le Tableau 4.1 ci-dessous présente la comparaison du nombre de messages reçus entre l'architecture *Normal* et celle *Prototype* que nous proposons. Nous remarquons que notre module Prototype reçoit plus de messages que celui Normal sur le même nombre de cycle. En effet, notre architecture observe un taux de réception égal à 70.95% contre 66.96% pour le module Normal, soit une différence de 3.99%. L'explication de cette différence pourrait être que l'architecture proposée traite plus rapidement les différentes requêtes reçues de par sa structure multithreadée, par conséquent, transfère plus vite un paquet vers son destinataire. Ce résultat s'illustre à la Figure 4.5.

TABLEAU 4.1 – Résultats nombres de messages reçus Normal-Prototype

Nbre de Senders	Msg-Envoyés	Msg-Reçus-Normal	Msg-Reçus-Prototype
1	30	30	30
2	60	42	50
3	90	57	66
4	120	90	94
5	150	90	110
6	180	134	157
7	210	146	172
8	240	177	192
9	270	179	198
10	300	172	220
11	330	201	208
12	360	198	203
13	390	189	186
14	420	169	187
15	450	188	218

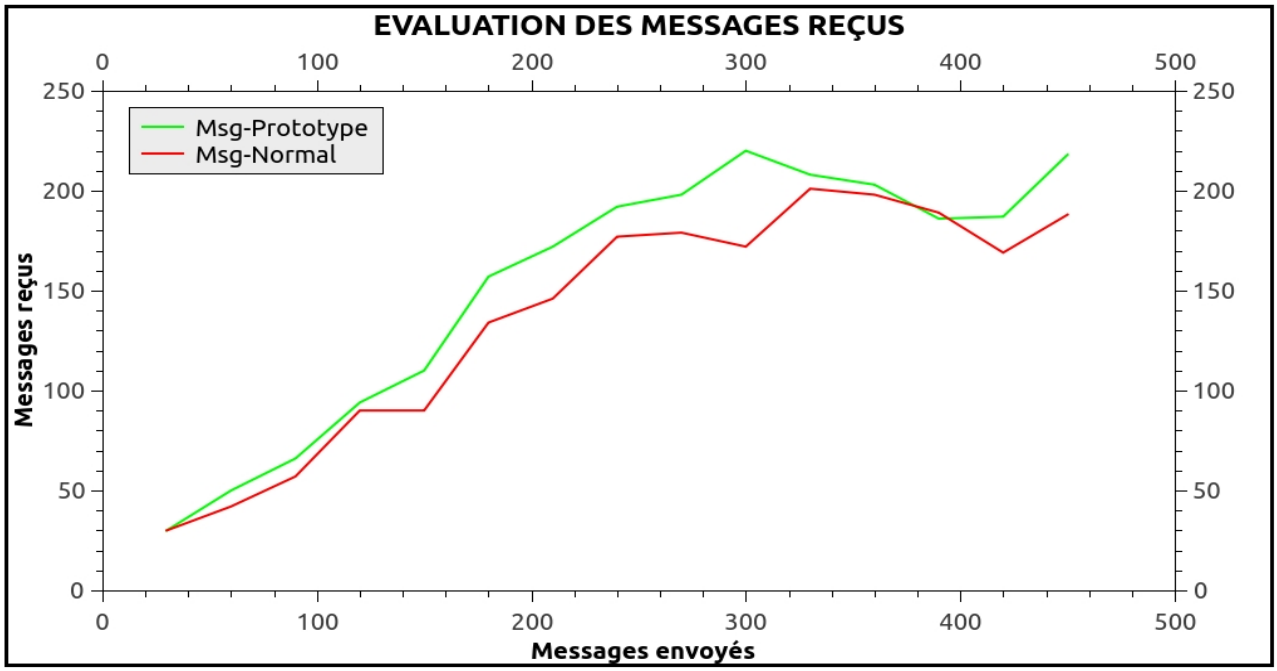


FIGURE 4.5 – Comparaison nombre de messages reçus Normal-Prototype

Thread - Prototype

Dans Le Tableau 4.2 est présente une comparaison du nombre de messages reçus entre les modules *Thread* et *Prototype* implémentés lors d'envois multisauts. Nous remarquons que ces deux architectures présentent des performances sensiblement identiques. L'entrelacement entre les deux courbes observé à la Figure 4.6 ne permet pas de mettre en avant une de ces architectures et montre également qu'entre celles-ci, il n'y a réellement pas une qui surpasse l'autre. Ce résultat s'explique par le fait que ces deux architectures se basent sur des structures multithreadées pratiquement identiques.

TABLEAU 4.2 – Résultats nombres de messages reçus Thread-Prototype

Nbre de Senders	Msg-Envoyés	Msg-Reçus-Thread	Msg-Reçus-Prototype
1	30	30	30
2	60	56	50
3	90	70	66
4	120	94	94
5	150	120	110
6	180	151	157
7	210	165	172
8	240	179	192
9	270	202	198
10	300	205	220
11	330	213	208
12	360	218	203
13	390	177	186
14	420	237	187
15	450	218	218

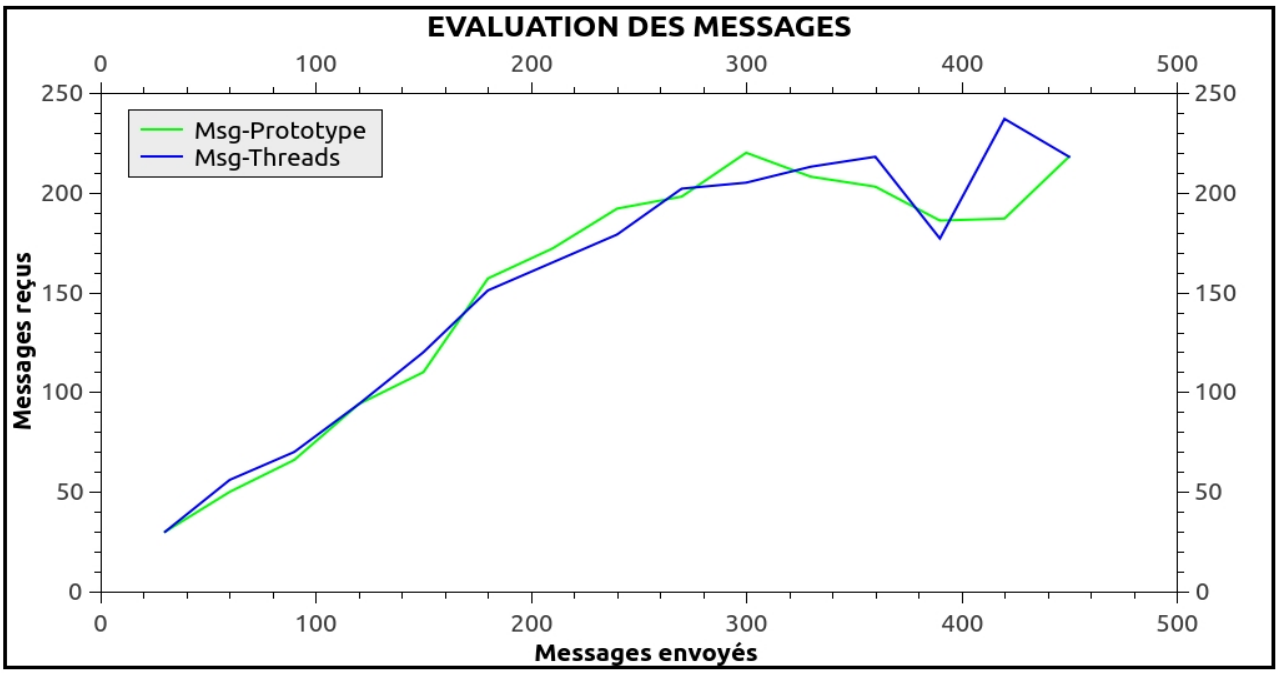


FIGURE 4.6 – Comparaison nombre de messages reçus Thread-Prototype

La Figure 4.7 ci-dessous représente un récapitulatif entre les différents modules Normal, Thread et Prototype par rapport au nombre de messages reçus.

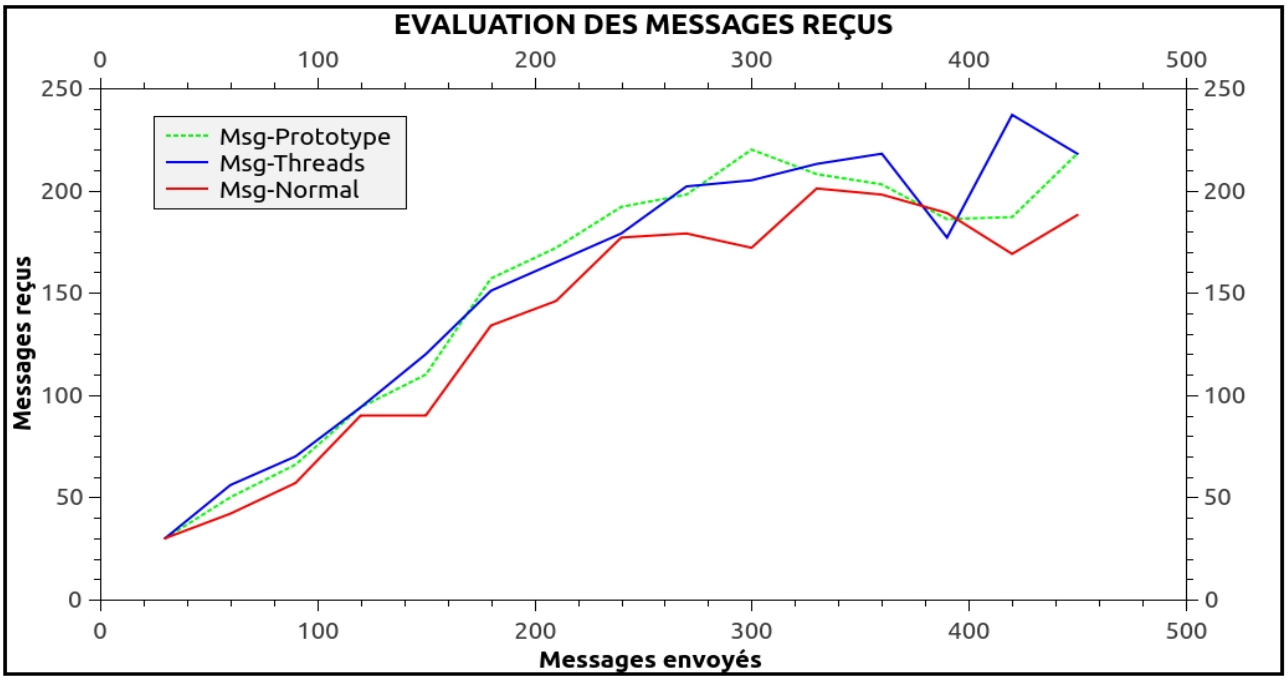


FIGURE 4.7 – Comparaison nombre de messages reçus Normal-Thread-Prototype

4.3.2 Utilisation de la radio

Normal - Prototype

Le Tableau 4.3 présente une comparaison du taux d'utilisation de la radio entre les architectures Normal et Prototype lors d'envoi multisauts. Nous remarquons que l'architecture proposée observe un taux d'utilisation de sa radio quasi inférieur par rapport à celui de l'architecture Normal. L'infériorité du taux d'utilisation de la radio dans architecture Prototype par rapport à celui Normal s'explique par le fait que l'architecture Prototype traite plus rapidement les données et par conséquent, elle utilise moins sa radio pour des communications (envoi et réception). L'adaptation de la mise en veille de la radio est rendue possible grâce au protocole ContikiMAC qui gère dynamiquement le *duty-cycling*. La Figure 4.8 présente graphiquement ces résultats.

TABLEAU 4.3 – Résultats taux utilisation radio Normal-Prototype

Nbre de Senders	Taux Normal(%)	Taux Prototype(%)
1	3.93	2.2
2	5.19	5.75
3	6.33	5.41
4	8.85	7.21
5	8.47	7.99
6	9.51	7.84
7	10.42	7.98
8	11.35	9.63
9	11.12	11.78
10	11.57	11.85
11	12.82	11.19
12	11.59	10.7
13	12.25	11.74
14	11.61	9.75
15	11.63	11.75

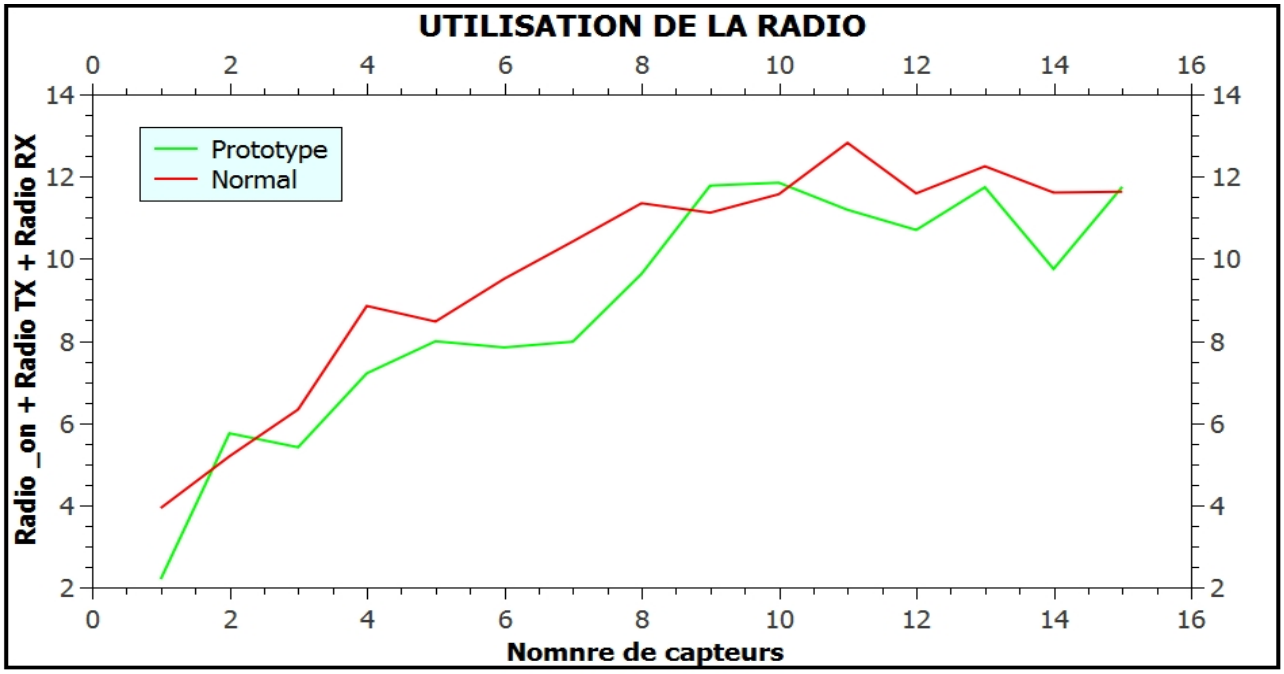


FIGURE 4.8 – Comparaison taux utilisation radio Normal-Prototype

Thread - Prototype

Les architectures Thread et Prototype ayant pratiquement un même nombre de réception de messages (Figure 4.6), elles auront sensiblement la même rapidité de traitement et par conséquent les taux d'utilisation de la radio similaires (Tableau 4.4). Le chevauchement des courbes observé dans la Figure 4.9 permet d'illustrer graphiquement ces résultats.

TABLEAU 4.4 – Résultats taux utilisation radio Thread-Prototype

Nbre de Senders	Taux Thread(%)	Taux Prototype(%)
1	3.26	2.2
2	4.03	5.75
3	5.33	5.41
4	7.56	7.21
5	7.32	7.99
6	7.01	7.84
7	8.99	7.98
8	9.5	9.63
9	11.4	11.78
10	11.49	11.85
11	11.47	11.19
12	11.23	10.7
13	10.35	11.74
14	12.15	9.75
15	11.4	11.75

La Figure 4.10 présente le taux d'utilisation de la radio dans les trois architectures Normal, Thread et Prototype.

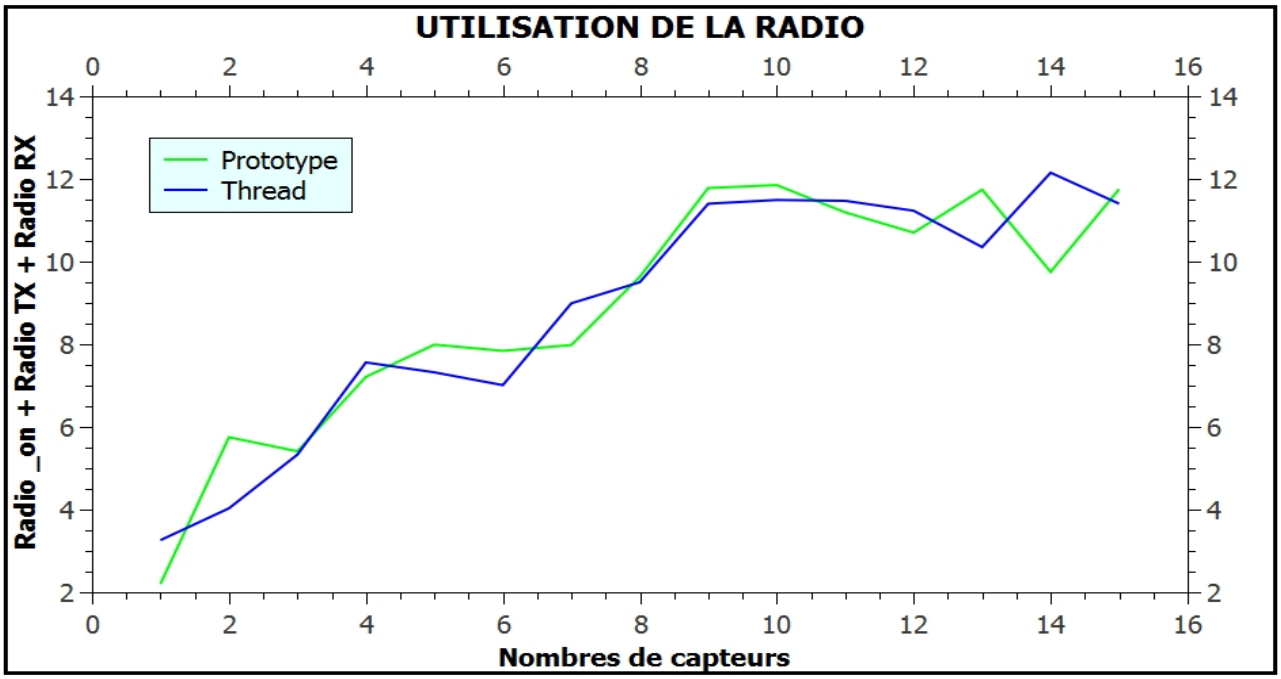


FIGURE 4.9 – Comparaison taux utilisation radio Thread-Prototype

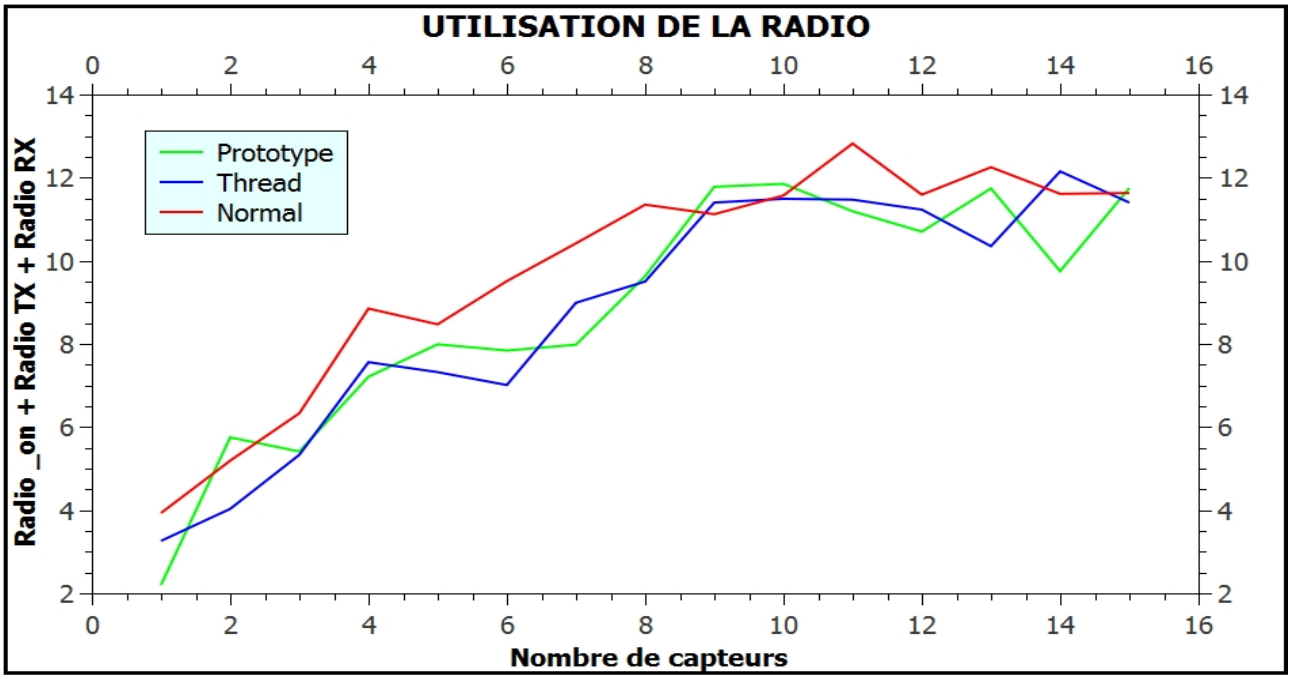


FIGURE 4.10 – Comparaison taux utilisation radio Normal-Thread-Prototype

4.3.3 Consommation énergétique

Normal - Prototype

Le Tableau 4.5 présente une comparaison de la consommation énergétique en mW entre les architectures Normal et Prototype. Ce tableau montre que dans la majorité des cas l'architecture Prototype que nous proposons observe une plus basse consommation énergétique que l'architecture Normal, soit au total 11.7348 mW en moins. L'architecture Normal consomme en moyenne 0.78234 mW de plus que celle proposée. Cette faible différente se traduit par le fait que ces deux modules utilisent des protothreads qui utilisent moins de mémoire puisque partageant un même espace mémoire. La Figure 4.11 permet une illustration graphique de ces résultats.

TABLEAU 4.5 – Résultats consommations énergétiques Normal-Prototype

Nbre de Senders	Consommation Normal(mW)	Consommation Prototype(mW)
1	2.5956	1.6882
2	3.1021	2.8172
3	3.0470	3.6935
4	3.9621	3.8473
5	4.5368	3.8130
6	7.2492	6.2966
7	9.3301	8.6741
8	7.9583	6.6649
9	6.1979	5.2931
10	8.4618	11.5380
11	10.3861	5.2282
12	12.9091	9.3529
13	12.8760	10.4199
14	8.6349	7.4984
15	9.4236	9.8375

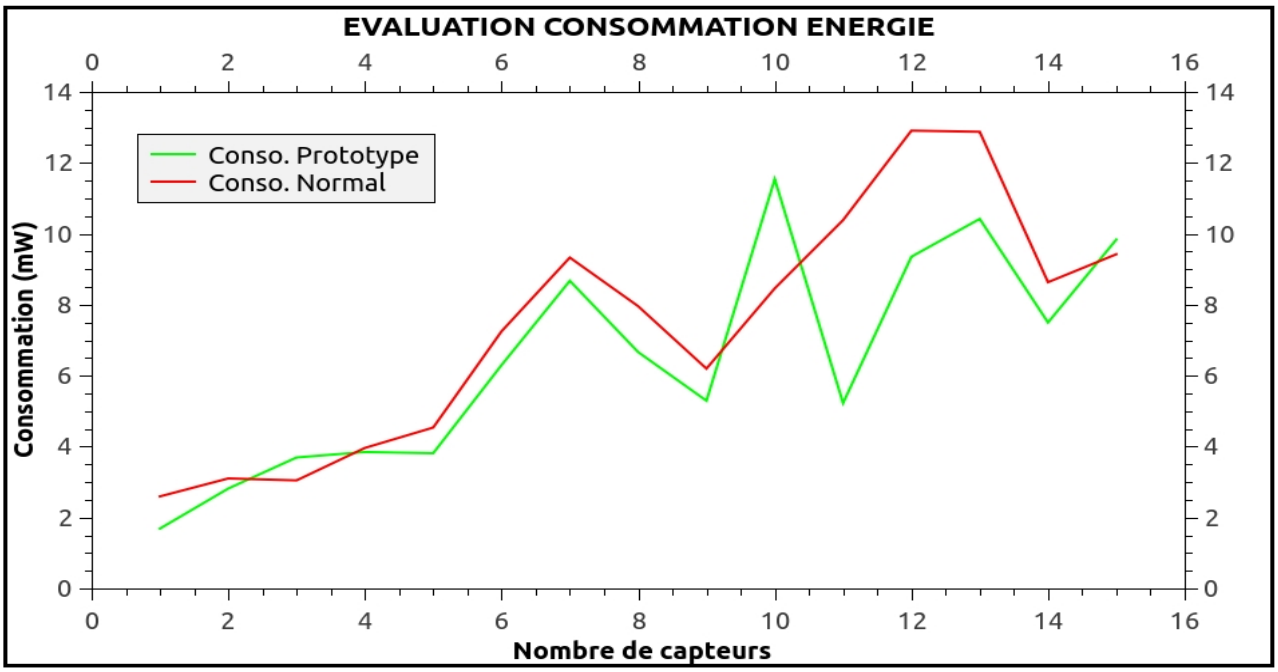


FIGURE 4.11 – Comparaison consommations énergétiques Normal-Prototype

Thread - Prototype

Il est présenté dans la Tableau 4.6 une comparaison des consommations énergétiques entre les architectures Thread et Prototype. Nous remarquons que le module Prototype consomme moins d'énergie que le celui Thread, soit une consommation totale de 12.95 mW en moins. Ainsi donc le module Thread observe une consommation moyenne de 0.86 mW en plus par rapport au module Prototype. Malgré l'entrelacement des courbes observé à la Figure 4.12, on note que la consommation énergétique de notre architecture est généralement inférieure à la consommation de l'architecture avec threads.

TABLEAU 4.6 – Résultats consommations énergétiques Thread-Prototype

Nbre de Senders	Consommation Thread(mW)	Consommation Prototype(mW)
1	4.5877	1.6882
2	3.4757	2.8172
3	5.7267	3.6935
4	4.7090	3.8473
5	5.16784	3.8130
6	6.42643	6.2966
7	9.6382	8.6741
8	10.2538	6.6649
9	8.1779	5.2931
10	9.1186	11.5380
11	10.1320	5.2282
12	9.3255	9.3529
13	8.0336	10.4199
14	5.5132	7.4984
15	9.3250	9.8375

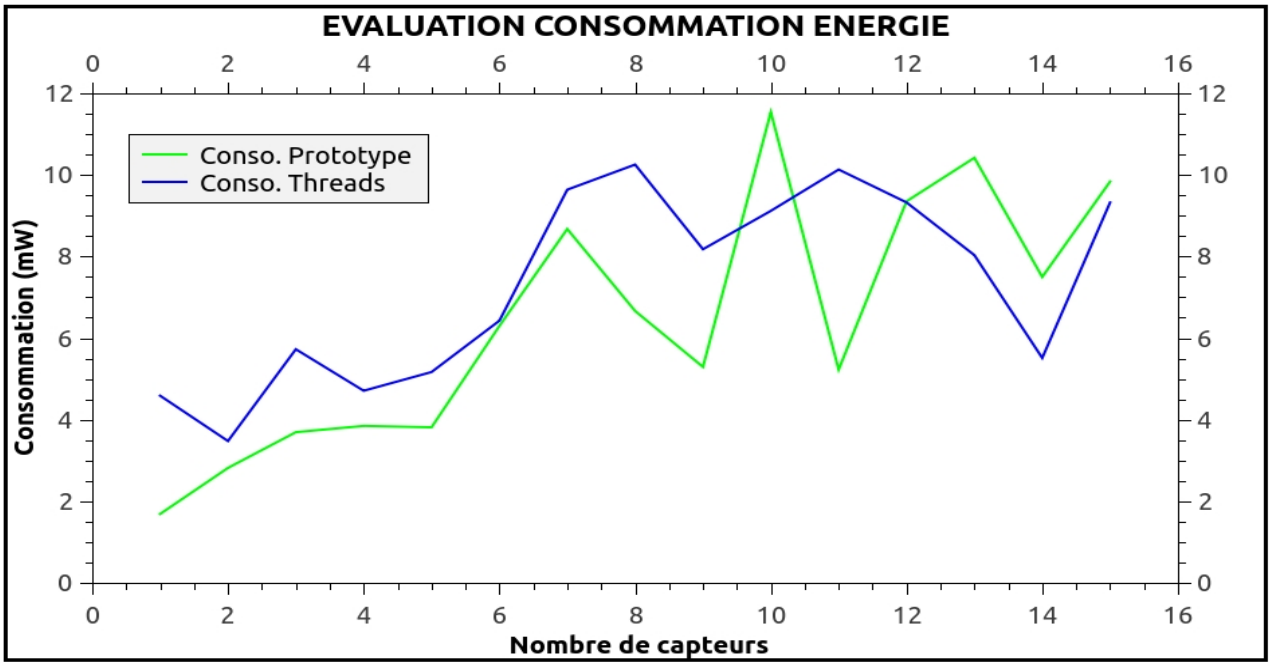


FIGURE 4.12 – Comparaison consommations énergétiques Thread-Prototype

La Figure 4.13 ci-dessous présente une comparaison des consommations énergétiques entre les 3 architectures étudiées.

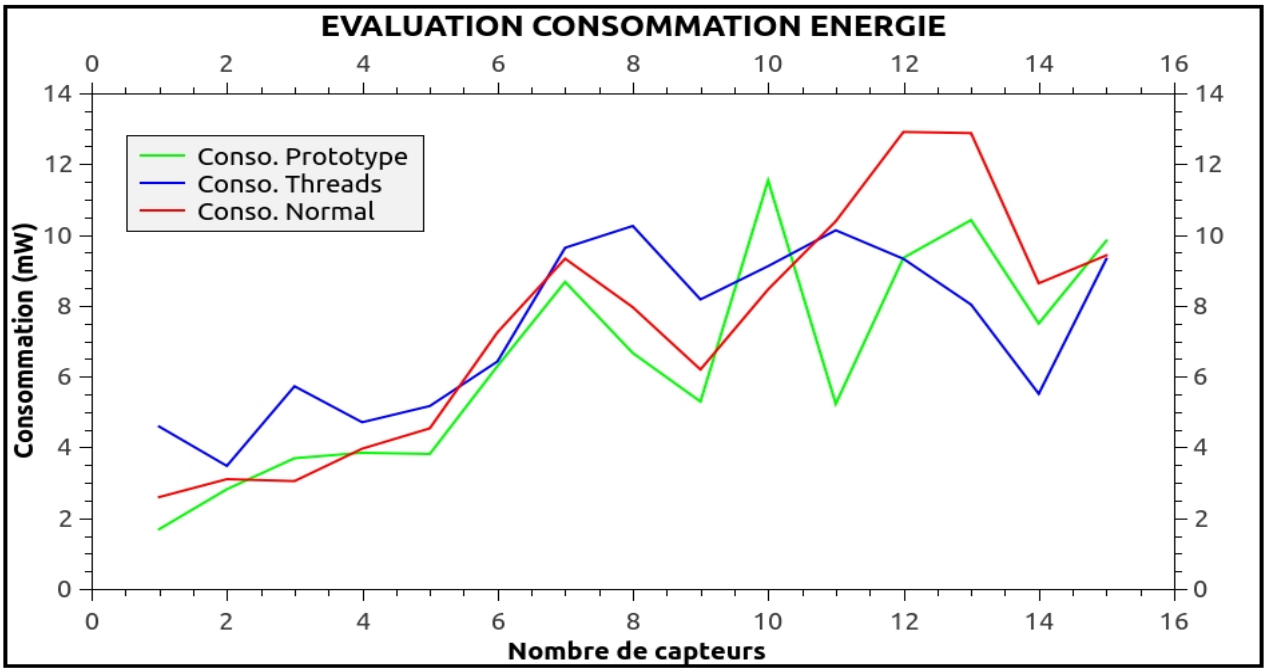


FIGURE 4.13 – Comparaison consommations énergétiques Normal-Thread-Prototype

4.4 Conclusion

Dans ce chapitre, nous avons proposé une nouvelle méthode d'envoi multisauts *prototype* simple d'utilisation qui se calque sur le multithreading cependant, au lieu de threads gourmands en mémoire elle utilise les protothreads qui partagent une même zone mémoire. Afin d'obtenir ces résultats une dizaine de tests similaires au chapitre 3 ont été réalisés.

Conclusion et Perspectives

Il a été question dans ce travail de redéfinir la notion de multithreading dans un monde de contraintes qu'est celui des RCSFs.

Afin d'y parvenir, nous avons utilisé un exemple d'envoi multisauts permettant de simuler un traitement multitâche dans le système d'exploitation Contiki. Tout d'abord, des expérimentations ont été réalisées sur les architectures Normal et Thread. Les résultats obtenus à la suite de ces expérimentations nous ont montré que l'architecture Thread présente une meilleure performance dans la rapidité de traitement de données par rapport à l'architecture Normal (5.37% en plus). Sur le plan énergétique, l'architecture Thread se montre plus gourmande que l'architecture Normal. A la suite de ces résultats, nous proposons une architecture hybride Prototype dont le fonctionnement est calqué sur l'architecture Thread mais utilise les protothreads à la place des threads qui utilisent plus la mémoire. Dans l'exemple utilisé, nous implémentons deux protothreads dont : l'un responsable des réceptions (*PT_recv*) et l'autre se chargeant de la transmission du paquet à son destinataire (*PT_send*). A la fin des tests, l'on remarque que notre architecture présente une capacité de traitement supérieure à celle normale (3.99% en plus) et sensiblement égale à l'architecture Thread. L'architecture proposée prend son sens réel au niveau de la consommation énergie. A l'aide de l'utilisation des processus légers que sont les protothreads, notre architecture observe une consommation inférieure aux architectures Normal (0.86 mW en moyenne) et Thread (0.78232 mW en moyenne). Notre architecture définit une nouvelle vision du multithreading dans les RCSFs.

D'autres critères doivent être pris en compte afin et d'améliorer notre architecture (Figures 4.6, 4.12). De plus, les tests étant réalisés uniquement sur le système Contiki et en environnement virtuel, il serait intéressant d'étendre ce nouveau paradigme du multithreading sur d'autres systèmes d'exploitation pour RCSF en conditions réelles pour valider ces résultats.

Références bibliographiques

[1] R. Kacimi. *Techniques de conservation d'énergie pour les réseaux de capteurs sans fil*. PhD thesis, Université de Toulouse (France), Juillet 2009.

[2] F. Khadar. *Contrôle de topologie dans les réseaux de capteurs : de la théorie à la pratique*. PhD thesis, Université de Lille 1 (France), Décembre 2009.

[3] Z. Tilak, N. B. Abu-Ghazaleh, and W. B. Heinzelman. *A taxonomy of wireless microsensor network models*. volume 6, pages 28–36. 2002.

[4] Y. Challal. *Réseaux de capteurs sans fil*. In *Systèmes intelligents pour le transport (SIT)*, volume 1. Novembre 2008.

[5] J. Eriksson. *Detailed simulation of heterogeneous Wireless Sensor Network*. PhD thesis, University of Uppsala (Sweden), 2009.

[6] J. L. Hill. *Wireless Sensor Networks*. PhD thesis, University of California-Berkeley(USA), 2003.

[7] W. Dargie and C. Poellabauer. *Fundamentals of Wireless Sensor Networks : theory and parctice*. Wiley series on Wireless communication and Mobile Computing, 2010.

[8] J. Beaudoux. *Auto-configuration et auto-adaptation de réseaux de capteurs sans fil dans le contexte de la télémédécine*. PhD thesis, Université de Strasbourg (France), 2013.

[9] Q. Wang and I. Balasingham. *Wireless Sensor Networks-An introduction*. *ISBN, In Tech*, 2010.

[10] C. Intanagonwiwat, R. Govindan, and D. Estrin. *A scalable and robust communication paradigm for sensor network*. In *ACM Mobicom'00*, pages 56–57. 2000.

[11] W. Dong, C. Chen, X. Liu, and J. Bu. *Providing OS support for Wireless Sensor Networks : challenges and approaches . IEEE Communications surveys and tutorials*, 12(4) :519–530, 2010.

[12] I. F. Akyildiz, W. Su, Y. Sankarasubramaniam, and E. Cayiri. *Wireless sensor network : a survey*. In *Broadband and wireless Networking Laboratory*, pages 393–422. December 2001.

[13] Y. Younes. *Minimisation d'énergie dans un réseau de capteurs*. Master's thesis, Université Mouloud Mammen de Tzi-Ouzou, Septembre 2012.

[14] G. De Sousa. *Etude en vue de la réalisation de logiciels bas niveau dédiés aux réseaux de capteurs sans fil : microsystèmes de fichiers*. PhD thesis, Université de Blaise Pascal-Clermont II (France), Octobre 2008.

[15] A. Dunkels, B. Grönvall, and T. Voigt. *Contiki-a lightweight and flexible Operating System for tiny networked*. Technical report, Swedish Institute of Computer Science (SICS), 2004.

[16] A. Sehgal. *Using the Contiki Cooja simulator*. Technical report, University of Bremen (Germany), October 2013.

[17] A. Dunkels. *Rime-a lightweight layered communication stack for sensor networks*. Technical report, Swedish Institute of Computer Science (SICS), 2007.

[18] A. Dunkels, O. Schmidt, T. Voigt, and M. Ali. *Protothreads : simplify event-driven programming of memory-constrained embedded systems*. Technical report, Swedish Institute of Computer Science (SICS), 2006.

[19] A. Dunkels, O. Schmidt, and T. Voigt. *Using protothreads for sensor node programming* . Technical report, Swedish Institute of Computer Science (SICS), 2005.

[20] H. Sundani, H. Li, V. K. Devebhaktuni, M. Alam, and P. Bhattacharya. *Wireless Sensor Network simulators : a survey and Comparisons*. *International Journal of Computer Networks (IJCN)*, 2(5) :248–265, 2011.

[21] H. Karl and A.Willig. *Protocols and architectures for Wireless Sensor Networks*. John Wiley and Sons, 2005.

[22] B. L. Titzef and J. Palsberg. *Nonintrusive precision instrumentation of microcontroller software*. *ACM*, 40(7) :59–68, July 2005.

[23] A. Montoya, D. C. Restrepo, and D. A. Ovalle. *Artificial Intelligence for Wireless Sensor Networks enhancement*. *In Tech*, 2010.

[24] *Le cardio pad*. http ://www.lemonde.fr/afrique/article/2015/04/02/cardiopad-la-tablette-imaginee-pour-sauver-des-vies-1-14_4608552_3212.html (dernière consultation le 11 octobre 2015).

[25] A. Mainwaning, D. Culler, J. Polastre, R. Szewczyk, and J.Anderson. *Wireless Sensor Networks for habitat monitoring*. *ACM international workshop on Wireless Sensor Networks and applications (WSNA)*, pages 88–97, 2002.

[26] L. Saraswat and P. S. Yadav. *A comparative analysis of Wireless Sensor Network Operating Systems*. In *5th National conference, INDIACom-2011*, March 2011.

[27] S. Bhatti, J. Carlson, H. Dai, J. Deng, J. Ross, A. Seth, B. Shucker, C. Gruenwald, A. Torgerson, and R. Han. *MANTIS OS : an embedded multi-threaded Operating System for Wireless Micro Sensor platforms*. *ACM Mobile Networks and Applications MONET*, August 2055.

[28] J. Hill, R. Szewczyk, A. Woo, S. Hollar, D. Culler, and K. Pister. *System architecture directions for networked sensors*. In *ASPLOS-IX 11/00 Cambridge (USA)*. November 2000.

[29] T. Reusing. Comparison of operating systems tinyos and contiki. In *Network Architectures and Services SN SS2012*, volume 02, August 2012.

[30] C. C. Han, R. Kumar, R. Shea, E. Kahler, and M. Srivastova. *A dynamic Operating System for sensor nodes*. *ACM Mobi Syst*, 2005.

[31] H. Cha, S. Choi, I. Jung, H. Kin, and H. Shin. *RETOS : Resilient Expandable an Threaded Operating System for Wireless Sensor Netwoks*. *ACM/IEEE IPSN*, 2007.

[32] V. Vanitha, V. Palanisamy, and G. Aravindohbabu N. Johnson and. *LiteOS based extended service oriented architecture for Wireless Sensor Networks*. *International Jornal of Computer and Electrical engineering*, 2(3), June 2010.

[33] A. Eswaran, A. Rowe, and R. Rajkumar. *Nano-RK : an energy-aware resource-centric RTOS for Sensor Networks*. In *4th International workshop on the web site evolution*, 2002.

[34] A. Dunkels, J. Eriksson, N. Finne, and N. Tsiftes. *Powertrace : network-level power profiling for low-power wireless networks*. Technical report, Swedish Institute of Computer Science (SICS), March 2011.

[35] *Contiki stack*. http ://fr.slideshare.net/adunkels/building-the-internet-of-things-with-thingsquare-and-contiki-day-2-part-2 (dernière consultation le 12 octobre 2015).

[36] *Multi-threading library* . http :// contiki.sourceforge.net/ docs/ 2.6/ a01669.html#_details (dernière consultation le 24 octobre 2015).

[37] *Contiki process*. https ://github.com/contiki-os/contiki/wiki/processes#polling (dernière consultation le 15 octobre 2015).

[38] A. Dunkels, F. Österling, and Z. He. *An adaptive communication architecture for Wireless Sensor Networks*. Technical report, Swedish Institute of Computer Science (SICS), November 2007.

[39] *Protothreads*. http ://contiki.sourceforge.net/docs/2.6/a01802.html#_details (dernière consultation le 24 octobre 2015).

[40] M. Michel and B. Quoitin. *ContikiMAC performance analysis*. Technical report, Computer Science Department, University of Mons (Belgium), july 2015.

[41] *Packet Buffer*. http ://anrg.usc.edu/contiki/index.php/packetbuffer_basics#void_packetbuf_clear_hdr.28vo (dernière consultation le 18 octobre 2015).

[42] H. Saad and J. Amel. *Developpement d'un système de surveillance de l'environnement à base d'un réseau de capteurs sans fil*. Master's thesis, Institut Supérieur d'Informatique de Mahdia (Tunisie), Juin 2013.

[43] T. Winter and P. Thubert. *RPL : IPv6 routing protocol for low power an lossy networks*. Technical report, IEFT, Draft, 2010.

[44] F. Moradi and A. Javaheri. *Clock synchronisation in sensor networks for civil security*. Master's thesis, University of Gothenburg(Swedish), March 2009.

[45] N. Lasla. *La gestion de clés dans les réseaux de capteurs sans-fil*. Master's thesis, Institut National de formation en Informatique (I.N.I) Oued-Smar (Algérie), Juin 2007.

[46] B. Bagula and Z. Erasmus. *iot emulation with COOJA* http ://wireless.ictp.it/school_2015/presentations/firstweek/ictp-cooja-presentation-version0.pdf. March 2015.

[47] M. A. M. Seliem, K. M. F. Elsayed, and A. Khattab. *Performance evaluation and optimization of Neighbor Discovery implementation over Contiki OS*. *IEEE Communications Magazine*, pages 96–101, April 2011.

[48] B. Krishnamacharie. *Networking wireless sensors*. Technical report, Cambridge University Press (USA), 2005.

[49] F. L. Lewis. *Wireless Sensor Networks*. In John Wiley and sons, editors, *Smart environments : technologies, protocols and applications*.

[50] H. Y. Zhou, K. M. Hou, J. Ponsonnaille, L. Gineste, J. Goudin, G. De Soussa, C. De Vaulx, J. J. Li, P. Chainais, R. Aufrère, A. Amanra, and J. P. Chenet. *Remote continuous cardiac arrhythmas detection and monitoring, transformation of healthcare with information technology and informatics* . *IOS*, 105 :112–120, 2004.

[51] M. Kuarilehto, T. Alho, M. Hännikaïnen, and T. I. Hämäläinen. *SensorOS : a new operating system for the critical Wireless Sensor Network applications*. *LNCS 4599*, pages 431–442, 2007.

[52] M. Samia and O. Souhila. *Implémentation et évaluation des schémas de routage sur une plate forme réelle de réseaux de capteurs sans fil* . Master's thesis, Université Abou Bakr Belkaud-Tlemcem (Algérie), Juin 2014.

[53] J. P. Leal Licudis, J. C. Abdala, G. G. Riva, and J. M. Finochietto. *Flexible Prototyping for Ad Hoc Wireless Sensor Network Protocols*. In *40th Jornadas Argentinas de Informática (JAIIO),Córdoba (Argentina)*, September 2011.

[54] *Contiki rime stack*. http ://www.log-a-tec.eu/software.html (dernière consultation le 12 octobre 2015).