

République Algérienne Démocratique et Populaire
Ministère de l'Enseignement Supérieur et de la Recherche Scientifique

Université Saâd DAHLAB de Blida



Faculté des sciences

Département d'informatique

Mémoire présenté par :

M^{lles} MEKAHLIA Fatma Zohra et MERABTINE Nassima

En vue de l'obtention d'un diplôme de Master

Domaine : Mathématique et Informatique

Filière : Informatique

Spécialité : Ingénierie des Logiciels

Sujet :

*Synchronisation du Temps dans
les Réseaux de Capteurs Sans Fil*

Soutenu le : 19 Septembre 2012, devant le jury composé de :

M ^f BENNOUAR Djamal, Maître de Conférence A, USDB	Président
M ^f CHERIF ZAHAR Amine, Maître Assistant A, USDB	Examineur
M ^{lle} MILOUD Amal, Maître Assistant B, USDB	Examinatrice
M ^f DJENOURI Djamel, Maître de Recherche A, CERIST	Promoteur

Organisme d'accueil :

CERIST – Centre de Recherche sur l'Information Scientifique et
Technique – Ben Aknoun

Remerciements

Tout d'abord, nous tenons à rendre grâce à DIEU tout puissant pour nous avoir donné le courage et la détermination nécessaire pour finaliser ce travail.

Nous tenons à remercier avec gratitude notre encadreur Docteur Djamel DJENOURI qui a endossé son rôle de la meilleure façon qui soit. Nous retiendrons sa disponibilité, son aide indéfectible, ses conseils avisés et ses idées riches ainsi que sa sympathie et ses encouragements. Nous voulons le remercier aussi de nous avoir fait bénéficier de son savoir et de son expérience tout au long de la période de notre stage, et également pour la mise à disposition de la documentation pertinente et le matériel nécessaire.

Nous exprimons nos sincères remerciements et notre profonde reconnaissance à Monsieur Samir DOUDOU pour son aide très précieuse et pour le temps qu'il a bien voulu nous consacrer.

Nos remerciements vont également à tous les chercheurs du laboratoire 'réseaux de capteurs' au sein du CERIST notamment Mr Nour Eddine LASLA, et Abdelraouf OUDJAOUT, pour leur accueil et leurs renforts.

Nous tenons à remercier le Dr Djamel BENNOUAR pour l'honneur qu'il nous a fait de présider notre jury de soutenance, ainsi que Mr Amine CHERIF ZAHAR et Mlle Amal MILOUD pour avoir bien voulu accepter d'examiner et de juger ce travail.

Par ailleurs ; nous rendons un vibrant hommage à l'ensemble du corps professoral du département d'informatique de l'université Saâd DAHLAB de Blida qui ont contribué activement à notre formation pendant notre cursus universitaire.

Nous ne pouvons pas terminer sans remercier encore une fois nos parents respectifs qui, par leur amour et leur soutien nous ont permis de mener à terme ce travail.

Dédicaces

Je dédie ce modeste travail :

A mes très chers parents en témoignage de ma profonde gratitude et mon incontestable reconnaissance, pour leurs sacrifices, la confiance qu'ils m'accordent, leurs soutien permanent et tout l'amour dont ils m'entourent.

A mes très chers grands parents que dieu les garde en santé.

A mes très chers frères et sœur, Tahar, Abdelhakim et Assia ainsi qu'à son époux Mohamed.

A mon adorable neveu Anes que dieu le protège.

A tous mes oncles, tantes, cousins et cousines.

A Meriem, Latifa, Khadidja, Amina Malki et Amina Toubelsghir, ainsi que tous mes amis que je ne peux pas, malheureusement, les cités tous.

A mon binôme Fatma Zohra ainsi que toute sa famille.

A tous mes collègues de la promotion 2011/2012, ainsi que mes collègues dans le projet au sein du CERIST.

Nassima

Dédicaces

Je tiens à dédier ce modeste travail

A la femme qui a donné de son mieux pour moi et qui a longtemps attendu ce jour: ma chère maman que DIEU la protège et lui procure la santé et le bonheur.

A l'homme qui a durement travaillé pour que je puisse réaliser l'un de mes rêves et de ses rêves: mon cher papa que DIEU le protège et lui procure la santé et le bonheur.

A mes très chers frères et sœurs, ma très chère belle-sœur et à mes adorables neveux Alàa et Adam auxquels je souhaite une vie pleine de bonheur, réussite et prospérité.

A tous mes oncles, tantes, cousins et cousines.

Aussi à mes amis Khadidja, Melyara, Imene, Nawel, Hafsa, Amira, Latifa, Amina, Messaouda, Karima, Cherifa, Fatma Zohra et Zineb, sans oublier mes amis de section Yassine, Anouar, Lotfi, Mohamed, Ismail et Allaa Eddine.

A mes collègues de projet au sein du CERIST Aghiles et son binome Adlane, Billel et Hakim sans oublier Nassima, mon acolyte dans ce travail ainsi que toute sa famille.

A toute personne qui m'a aidé un jour à parvenir jusqu'ici, en espérant être toujours à la hauteur de leurs attentes et de leurs espérances.

Que la paix d'ALLAH soit avec tous et que DIEU nous réunisse dans son vaste paradis inchaALLAH.

Fatma Zohra

ملخص

الطبيعة التعاونية لشبكات الحساسات اللاسلكية تجعل الحاجة إلى وجود مفهوم مشترك للوقت في الشبكة شرطاً أساسياً لمعظم التطبيقات. هذا المفهوم يمكن تجسيده بتوظيف بروتوكول التزامن الزمني الذي يأخذ بعين الاعتبار مختلف القيود مثل: الطاقة المحدودة، مساحة الذاكرة المحدودة و عطل أجهزة الحساسات اللاسلكية.

في إطار مشروع البحث الوطني المعنون ب "تسيير حركة المرور ديناميكياً باستعمال الحساسات". يكمن مشروعنا في تزامن الوقت. من أجل هذا، قمنا بدراسة مجموعة من بروتوكولات مزامنة الوقت الموجهة لشبكة الحساسات اللاسلكية، ثم قمنا بتطوير، تنفيذ و تجريب نموذج يأخذ بعين الاعتبار عطل هذه الأجهزة على مجموعة من العقد. فقد بينت النتائج المتحصّل عليها أنّ تنفيذنا يعطي دقة عالية، في حدود الميكرو ثانية، كما بينت هذه النتائج درجة عالية من الاستقرار فيما يخص النموذج الثاني.

الكلمات الرئيسية

الحساسات الصغيرة، شبكات الحساسات اللاسلكية، بروتوكول تزامن الوقت، إنحراف الساعة، طرق التقدير.

Résumé

La nature coopérative des réseaux de capteurs sans fil (WSN) rend le besoin d'une notion du temps commune dans le réseau une exigence primordiale pour la majorité des applications. Ceci peut être assuré par un protocole de synchronisation du temps, qui doit prendre en considération diverses contraintes telles que l'énergie limitée, l'espace mémoire réduit et les pannes des micro-capteurs.

Dans le cadre d'un projet de recherche national « la gestion dynamique du trafic routier en utilisant les réseaux de capteurs sans fil », notre projet consiste à assurer la synchronisation du temps des micro-capteurs. Pour ce faire, nous avons étudié un ensemble de protocoles de synchronisation du temps destinés aux WSNs, puis nous avons développé, implémenté et testé une variante tolérante aux fautes du protocole R^4_{syn} dans des micro-capteurs réels de type MICAz. Les résultats montrent que notre implémentation offre une très bonne précision, à l'ordre de quelques microsecondes, et une grande stabilité pour le modèle *skew/offset*.

Mots clés

Micro-capteur, Réseaux de capteurs sans fil, Protocole de synchronisation du temps, Déviation des horloges, Méthodes d'estimation, TinyOS, NesC.

Abstract

The cooperative nature of wireless sensor networks (WSN) makes the need for a common time reference in the network a key requirement for most applications. This can be assured by a time synchronization protocol, which must take into account various constraints such as the limitation in energy supply, memory, and the node failure.

As part of a national research project, "road traffic management using wireless sensor networks", our task has been focusing on providing time synchronize to sensor motes in WSN. First, a set of WSN time synchronization protocols have been studied, then a fault-tolerant variant of the R4syn has been developed, implemented and tested in MICAz sensor motes. The results are encouraging and demonstrates high accuracy of the implemented protocol (in the order of few microseconds), and show more stability for the join *skew/offset* model, i.e. the model that takes the *drift* phenomenon into account.

Keywords

Sensor mote, Wireless Sensor Network, Time Synchronization Protocol, Clock Drift, estimation methods, TinyOS, NesC.

Liste des Acronymes

ADC	<i>Analog to Digital Converter</i>
CENS	<i>Center for Embedded Networking Sensing</i>
EEPROM	<i>Electrically Erasable Programmable Read Only Memory</i>
Eq	<i>Equation</i>
ETSM	<i>Efficient Time Synchronization Mechanism for Wireless Multi Hop Networks</i>
FIFO	<i>First In First Out</i>
FTSP	<i>Flooding Time Synchronization Protocol</i>
GPS	<i>Global Position System</i>
ID	<i>Identifiant</i>
IEEE	<i>Institute of Electronic and Electronics Engineers</i>
ISM	<i>Industrial Scientific and Medical</i>
MAC	<i>Media Access Control</i>
MIB	<i>Mote Interface Board</i>
MLE	<i>Maximum Likelihood Estimators</i>
NTP	<i>Network Time Protocol</i>
PFM	<i>Program Flash Memory</i>
PNR	<i>Projet National de Recherche</i>
RAM	<i>Random Access Memory</i>
RBS	<i>Reference Broadcast Synchronization</i>
ROM	<i>Read-Only Memory</i>

Liste des Tableaux

- p. 06 - Tableau 1.1 : Différentes plateformes des micro-capteurs.*
- p. 60 – Tableau 4.1 : Paramètres du protocole.*
- p. 69 - Tableau 4.2 : Paramètres de mécanismes de tolérance aux fautes.*

Table des Matières

Introduction Générale

1.	<i>Contexte du Travail</i>	1
2.	<i>Cadre du Projet</i>	2
3.	<i>Problématique et Motivations</i>	2
4.	<i>Objectif du Travail</i>	3
5.	<i>Organisation du Mémoire</i>	3

Partie I: Etat de l'Art

Chapitre I : Généralités sur les Réseaux de Capteurs Sans Fil

1.	<i>Introduction</i>	6
2.	<i>Micro-capteur « Mote »</i>	6
2.1.	<i>Présentation d'un Micro-capteur</i>	6
2.2.	<i>Architecture Physique d'un Micro-capteur « Hardware »</i>	7
2.3.	<i>Caractéristiques Physiques d'un Micro-capteur</i>	8
2.3.1.	<i>Autonomie et Adaptabilité</i>	8
2.3.2.	<i>Ressources Energétiques et Matérielles Limitées</i>	8
2.3.3.	<i>Taux de Transfert Limité</i>	9
3.	<i>Réseaux de Capteurs Sans Fil</i>	9
3.1.	<i>Architecture d'un WSN</i>	9
3.2.	<i>Pile Protocolaire d'un Micro-capteur</i>	9

3.3.	<i>Facteurs et Contraintes de Conception d'un WSN</i>	10
3.3.1.	<i>Auto-Configuration</i>	11
3.3.2.	<i>Facteur d'Echelle « Scalability »</i>	11
3.3.3.	<i>Absence d'Infrastructure</i>	11
3.3.4.	<i>Contrainte d'Energie</i>	11
3.3.5.	<i>Ressources Matérielles Limitées</i>	12
3.3.6.	<i>Topologie Dynamique</i>	12
3.3.7.	<i>Agrégation de Données</i>	12
3.3.8.	<i>Tolérance aux Pannes</i>	12
3.4.	<i>Types des Flux de Données dans un WSN</i>	13
3.4.1.	<i>Périodique</i>	13
3.4.2.	<i>Événementiel</i>	13
3.4.3.	<i>Requête/Réponses</i>	13
3.4.4.	<i>Hybride</i>	13
4.	<i>Domaines d'Application des Réseaux de Capteurs Sans Fil</i>	13
4.1.	<i>Applications Environnementales</i>	14
4.2.	<i>Applications Militaires</i>	14
4.3.	<i>Applications Médicales</i>	14
4.4.	<i>Applications de Gestion du Trafic Routier</i>	14
4.5.	<i>Applications Commerciales</i>	15
4.6.	<i>Applications Domestiques</i>	15
4.7.	<i>Application à la Sécurité</i>	15

4.1.	<i>Timing-sync Protocol for Sensor Networks « TPSN »</i>	24
4.1.1.	<i>Description</i>	24
4.1.2.	<i>Principe</i>	25
4.1.3.	<i>Phase de Découverte de Niveau « Level Discovery Phase »</i>	25
4.1.4.	<i>Phase de Synchronisation « Synchronization Phase »</i>	26
4.1.5.	<i>Estimation des Paramètres de Synchronisation</i>	27
4.2.	<i>Flooding Time Synchronization Protocol « FTSP »</i>	28
4.3.	<i>Reference Broadcast Synchronization « RBS »</i>	28
4.3.1.	<i>Description</i>	28
4.3.2.	<i>Principe</i>	29
4.3.3.	<i>Estimation des Paramètres de Synchronisation</i>	29
4.3.4.	<i>Synchronisation Globale</i>	30
4.4.	<i>Relative Referencless Receiver/Receiver Time Synchronisation in Wireless Sensor Networks « R⁴Syn »</i>	31
4.4.1.	<i>Description</i>	31
4.4.2.	<i>Principe</i>	31
4.4.3.	<i>Estimation des Paramètres de Synchronisation</i>	32
4.4.4.	<i>Synchronisation Globale</i>	33
4.5.	<i>Efficient Time Synchronization Mechanism for Wireless Multi Hop Networks « ETSM »</i>	34
4.5.1.	<i>Sélection des Nœuds Emetteurs</i>	34
4.5.2.	<i>Synchronisation des Nœuds</i>	34
4.5.3.	<i>Estimation des Paramètres de Synchronisation</i>	35

4.3.1.1.	<i>Description</i>	44
4.3.1.2.	<i>Algorithme</i>	44
4.3.2.	<i>Continuité de la Synchronisation</i>	45
4.3.2.1.	<i>Description</i>	45
4.3.2.2.	<i>Algorithme</i>	46
4.3.3.	<i>Granularité de la Synchronisation</i>	47
4.3.3.1.	<i>Description</i>	47
4.3.3.2.	<i>Algorithme</i>	48
4.4.	<i>Conversion</i>	448
5.	<i>Conclusion</i>	49

Chapitre IV : Implémentation et Tests Réels

1.	<i>Introduction</i>	51
2.	<i>Plateforme d'Implémentation</i>	51
2.1.	<i>Plateforme Matérielle</i>	51
2.1.1.	<i>Micro-capteur</i>	51
2.1.2.	<i>Programmateur « Programming Board »</i>	53
2.1.3.	<i>Carte d'Extension «Data Acquisition Board »</i>	53
2.1.4.	<i>Arduino UNO</i>	53
2.2.	<i>Plateforme Logicielle</i>	53
2.2.1.	<i>Système d'Exploitation TinyOS</i>	53
2.2.2.	<i>Langage de Programmation NesC</i>	54

2.2.3.	<i>Simulateur Avrora</i>	54
3.	<i>Réalisation de R⁴ Syn</i>	54
3.1.	<i>Horloge Utilisée</i>	54
3.1.1.	<i>Horloge de Microcontrôleur</i>	55
3.1.2.	<i>Horloge Externe</i>	55
3.2.	<i>Implémentation</i>	55
3.2.1.	<i>Offset-only</i>	56
3.2.1.1.	<i>TimeSyncP</i>	56
3.2.1.2.	<i>TimeSyncTreatmentOffsetOnlyP</i>	57
3.2.1.3.	<i>TimeSyncEstimOffsetOnlyP</i>	57
3.2.1.4.	<i>CC2420ReceiveP</i>	57
3.2.2.	<i>Skew/Offset</i>	58
4.	<i>Tests Réels de R⁴ Syn</i>	59
4.1.	<i>Paramètres du Protocole</i>	59
4.2.	<i>Offset-Only</i>	60
4.2.1.	<i>Précision</i>	60
4.2.1.1.	<i>Première Méthode</i>	61
4.2.1.2.	<i>Deuxième Méthode</i>	62
4.2.1.3.	<i>Troisième Méthode</i>	64
4.2.2.	<i>Stabilité</i>	65
4.3.	<i>Skew/Offset</i>	66
4.3.1.	<i>Précision</i>	66

4.3.2.	<i>Stabilité</i>	68
4.4.	<i>Comparaison Entre les deux Modèles</i>	68
4.5.	<i>Tolérance aux Fautes</i>	69
4.5.1.	<i>Continuité</i>	69
4.5.2.	<i>Granularité</i>	70
5.	<i>Application Oscilloscope</i>	71
6.	<i>Conclusion</i>	72
	 <i>Conclusion Générale et Perspectives</i>	 75

Annexes

Annexe A : Système d'Exploitation TinyOS

Annexe B : Langage de Programmation NesC

Annexe C : Simulateur Avrora

Annexe D : Procédures d'Installation

Références Bibliographiques et Webographiques

Introduction

Générale

L'instabilité de l'oscillateur entraîne un changement dans l'unité de mesure du temps. Ceci dit, les valeurs d'horloges observées dans les différents nœuds du réseau, à un instant T , peut être différentes.

Par conséquent, le temps $T(t)$ d'un nœud " n " peut être modélisé comme suit:

$$T_n(t) = a_n * t + b_n$$

Où a_n est la déviation « *skew* » de l'horloge de nœud " n ", représentant la vitesse à laquelle l'horloge progresse à l'instant " t ". Celle-ci peut changer par rapport à la fréquence initiale à cause de l'instabilité de l'oscillateur. b_n est le décalage entre l'horloge de nœud " n " et le temps réel donné par une référence du temps global, telle que UTC (Universal Time Coordinated), à l'instant " t ".

4. Objectif du Travail

L'objectif prévu de ce projet est la mise en œuvre d'un outil de conversion du temps, qui permet à un nœud X du réseau de convertir sa valeur d'horloge, à n'importe quel instant T , à la valeur d'horloge qui aura été générée dans un nœud Y, avec une précision de l'ordre de microsecondes. Ceci en implémentant un protocole de synchronisation du temps distribué, puis le tester sur des micro-capteurs réels pour fixer empiriquement ses paramètres. L'implémentation consiste aussi à développer des mécanismes pour assurer la tolérance aux pannes des micro-capteurs. Ces mécanismes doivent garantir la continuité de la synchronisation malgré la défaillance d'un ou de plusieurs nœuds dans le réseau. L'implémentation doit, également, tolérer la perte de messages. Cette dernière ne doit pas affecter la granularité de la synchronisation.

5. Organisation du Mémoire

Ce document est organisé comme suit :

Le premier chapitre s'inscrit dans le contexte d'une étude globale sur les réseaux de capteurs sans fil. Pour cela, nous présenterons leurs éléments de base, l'architecture de communication, les principales caractéristiques et contraintes qui influencent la conception

de ce genre de réseaux. Nous discuterons également leurs différents domaines d'application.

Le deuxième chapitre décrit la synchronisation temporelle dans les réseaux de capteurs sans fil, dans lequel nous présenterons les différentes notions de base sur la synchronisation temporelle, ainsi qu'une étude sur un ensemble de protocoles de synchronisation du temps conçus spécialement pour les réseaux de capteurs sans fil.

Le troisième chapitre intitulé "Protocole Développé" sera consacré à la description des caractéristiques du protocole R^4 *syn*, ainsi que sa conception.

Dans le dernier chapitre "Implémentation et Tests Réels", nous présenterons l'environnement matériel et logiciel utilisé dans ce travail, ainsi que la description de notre implémentation sur des micro-capteurs réels de type MICAz. Par la suite, nous allons révéler les différents résultats obtenus par des expérimentations réelles.

En fin, la conclusion de ce mémoire synthétisera nos principales contributions et donnera quelques perspectives à notre travail.

Partie I :

Etat de l'Art

Chapitre 01 :

*Généralités sur les
Réseaux de Capteurs
Sans Fil*

1. Introduction







Un réseau de capteurs sans fil (WSN) est un type spécifique des réseaux ad-hoc où les nœuds sont des micro-capteurs intelligents. Dans ce premier chapitre, nous allons présenter les éléments constitutifs des WSNs, ainsi que leurs architectures de communication. Nous discuterons également les principaux facteurs qui influencent la conception des WSNs. A la fin, nous présenterons les différents domaines d'application où nous pourrions tirer profit de déploiement d'un WSN.

2. Micro-capteur « Mote »

2.1. Présentation d'un Micro-capteur

Le micro-capteur est un petit dispositif électronique capable de récolter, traiter et de transmettre des données environnementales à une autre entité (capteurs, station de base, etc.) via les ondes radio, et d'une manière autonome [W05]. Il existe plusieurs modèles commercialisés dans le marché. Parmi les plus célèbres, nous pouvons citer, MICAz, IntelMote2, TinyNode, etc.

Les caractéristiques des différents composants de certaines plateformes des micro-capteurs sont représentées dans le tableau suivant [W01, W02, W03, W04].

Plateforme / Propriété	 MicaZ	 Mica2	 TelosB	 Mica2dot	 IRIS	 Imote2
Micro-contrôleur	ATMega 128L	ATMega 128L	TI MSP 430	ATMega 128L	ATMega128 1	Intel PXA271
fréquence de processeurs(MHz)	8	8	16	8	16	13 – 416
débit de transfert(Kbps)	250	38.4	250	38.4	250	250
RAM	4 KB	4 KB	10 KB	4 KB	8 KB	32 MB
PFM(Kbytes)	128	128	48	128	128	256
EEPROM	4 KB	4 KB	16 KB	4 KB	4 KB	32 MB

Interface radio	CC2420	CC1000/ AT45DB	STM25P	CC1000/ AT45DB	RF230/ AT45DB	CC2420/ DA9030/
bande de fréquence(MHz)	2400 - 2483.5	315 – 916	2400 - 2483.5	315 – 916	2405 - 2480	2400 - 2483.5
Batterie	2x AA	2x AA	2x AA	Coin (CR2354)	2x AA	3x AAA
Interface utilisateur	3 LEDs	3 LEDs	USB	1 LED	3 LEDs	USB, Camera

Tableau 1.1. Différentes plateformes des micro-capteurs

2.2. Architecture Physique d'un Micro-capteur « Hardware »

Le micro-capteur est composé de quatre unités principales (voir *figure 1.1*) [ASC02].

Unité de Captage "Sensing Unit" : Elle se compose de deux sous-unités, les capteurs et les convertisseurs analogiques/numériques "ADC". Les capteurs permettent de capter les mesures analogiques sur le phénomène observé, telles que la température, la luminosité, la vibration, le son, etc. et les transmettent aux convertisseurs. Les ADCs convertissent les signaux analogiques en signaux numériques, qui seront par la suite fournis à l'unité de traitement.

Unité de Traitement "Processing Unit" : Elle est constituée d'un microcontrôleur et d'une mémoire. Les microcontrôleurs utilisés par les motes sont à faible fréquence, et à faible consommation d'énergie. L'unité de traitement utilise un système d'exploitation spécial, qui respecte la miniaturisation des micro-capteurs. Cette unité est chargée de traiter les données collectées et d'exécuter les protocoles de communication qui permettent de faire collaborer un nœud avec les autres nœuds du réseau. Les informations traitées seront envoyées à l'unité de communication.

Unité de Communication "Transceiver Unit" : Elle est composée d'un émetteur/récepteur (module radio), permettant la communication entre les différents nœuds du réseau. Généralement, ces unités utilisent les ondes électromagnétiques, et opèrent sur les bandes de fréquence ISM.

Unité d'Energie " Power Unit " : Les trois unités citées précédemment sont alimentées via une batterie, généralement, non rechargeable ni remplaçable. Néanmoins, les unités

d'énergie peuvent être supportées par des photopiles (générateur d'énergie), qui permettent de convertir l'énergie lumineuse en courant électrique.

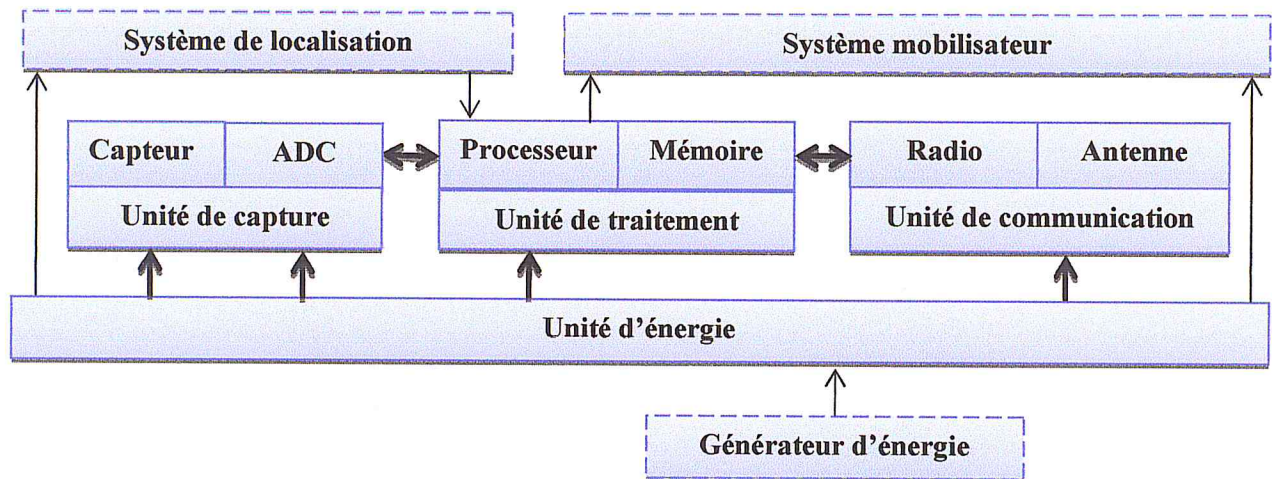
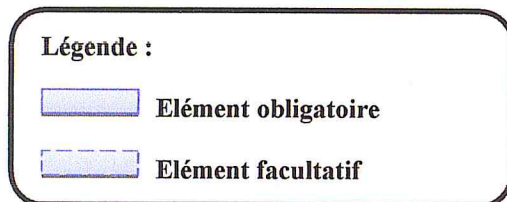


Figure 1.1. Architecture d'un micro-capteur (capteur intelligent)



2.3. Caractéristiques Physiques d'un Micro-capteur

2.3.1. Autonomie et Adaptabilité

Un micro-capteur intégré est complètement autonome ; il peut effectuer toutes les tâches qui lui sont affectées (captage, communication, etc.), sans avoir besoin d'un composant tiers. En revanche, il est moins adaptatif qu'un micro-capteur à carte qui peut recevoir plusieurs capteurs pour différentes grandeurs physiques [W05].

2.3.2. Ressources Energétiques et Matérielles Limitées

Un micro-capteur est un matériel de petite taille, caractérisé par des modestes ressources. L'énergie constitue le principal handicap des micro-capteurs. La limitation de la taille mémoire des motes est une autre caractéristique. Celle-ci est relative à la plateforme du micro-capteur utilisé. A titre d'exemple, la plateforme Telosb observe 10 ko de RAM, et 48 ko de PFM (Program Flash Memory) [W01]. Les micro-capteurs sont caractérisés ainsi, par des microcontrôleurs à faible fréquence.

2.3.3. Taux de Transfert Limité

Un faible débit de transfert de données n'est pas gênant pour un réseau de capteurs où les fréquences de transmission ne sont pas importantes. Le taux de transfert dans les micro-capteurs est limité à 250Kbps, taux théorique pour le protocole ZigBee sur une fréquence de 2.4 GHz [W01].

3. Réseaux de Capteurs Sans Fil

3.1. Architecture d'un WSN

Un WSN se compose généralement d'un grand nombre de micro-capteurs, capables de récolter et de transmettre des données environnementales d'une manière autonome. La position de ces nœuds n'est pas obligatoirement prédéterminée. Ils peuvent être dispersés aléatoirement à travers une zone géographique, appelée champ de captage, qui définit le terrain d'intérêt pour le phénomène capté. Les données captées sont acheminées grâce à un routage multi-saut à un nœud considéré comme un "point de collecte", appelé nœud puits (ou *sink*), ou encore station de base. Ce dernier peut être connecté à l'utilisateur du réseau via internet ou un satellite. Ainsi, l'utilisateur peut adresser des requêtes aux nœuds du réseau via le nœud *sink*, précisant le type de données [W05] (voir *figure 1.2*).

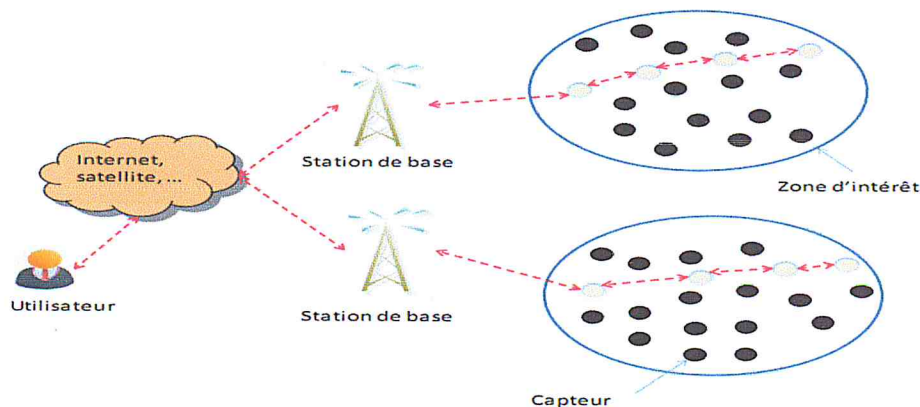


Figure 1.2. Schéma d'un réseau de capteurs sans fil

3.2. Pile Protocolaire d'un Micro-capteur

La pile protocolaire d'un micro-capteur [ASC02], illustrée par la *figure 1.3*, comprend la couche application « Application layer », la couche transport « Transport layer », la couche réseau « Network layer », la couche liaison de données « Data link layer », la couche

physique « Physical layer ». De plus, cette pile possède trois plans, le plan de gestion de la mobilité « Mobility management plane », le plan de gestion de l'énergie « power management plane », et le plan de gestion des tâches « Tasks management plane ».

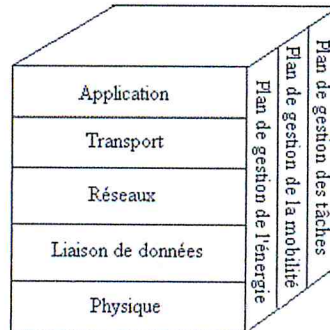


Figure 1.3. Pile protocolaire des micro-capteurs [ASC02]

Suivant la fonctionnalité des micro-capteurs, différentes applications peuvent être utilisées et bâties sur la couche application. La couche transport aide à gérer le flux de données si le réseau de capteurs l'exige. Elle permet de diviser les données issues de la couche application en segments pour les délivrer. Ainsi, la couche transport réordonne et rassemble les segments venus de la couche réseau avant de les envoyer à la couche application. La couche réseau prend soin de router les données fournies par la couche transport. Le protocole MAC « Media Access Control » de la couche liaison assure la gestion de l'accès au support physique. La couche physique assure la transmission et la réception des données au niveau bit.

En outre, les plans de gestion d'énergie, de mobilité et des tâches, surveillent et gèrent, respectivement, la consommation d'énergie, les mouvements et la répartition des tâches entre les nœuds-capteurs. Ces plans aident les nœuds-capteurs à coordonner les tâches de détection et à réduire l'ensemble de la consommation d'énergie [ASS02].

3.3. Facteurs et Contraintes de Conception d'un WSN

La conception et la réalisation des réseaux de capteurs sans fil sont influencées par plusieurs paramètres. Ces paramètres servent comme directives pour le développement des applications et des protocoles utilisés dans les WSNs [NDS04].

3.3.1. Auto-Configuration

Dans un WSN, les nœuds sont déployés soit d'une manière aléatoire (missile, avion, etc.), soit placés nœud par nœud par un humain ou un robot. Ceci à l'intérieur ou autour du phénomène à observer (surface volcanique, patient malade, route, terrain agricole, rivière, infrastructure, etc.). Ainsi, un nœud capteur doit avoir des capacités d'une part, pour s'auto-configurer dans le réseau, et d'autre part pour collaborer avec les autres nœuds dans le but de reconfigurer dynamiquement le réseau en cas de changement de topologie du réseau. L'auto-configuration est une caractéristique nécessaire dans les WSNs, surtout dans le cas du déploiement aléatoire des nœuds-capteurs, ou quand le réseau est déployé avec un grand nombre de nœuds.

3.3.2. Passage à l'Echelle « Scalability »

Le nombre de nœuds déployés dans une application peut atteindre des milliers. Dans ce cas, le réseau doit fonctionner avec une très grande densité de nœuds-capteurs. Un nombre aussi important de nœuds engendre beaucoup de transmissions inter nodales, et nécessite que la station de base soit équipée de mémoire suffisante pour stocker les informations reçues. Un réseau de capteurs doit être scalable; c'est-à-dire, possède la faculté d'accepter un très grand nombre de nœuds qui collaborent ensemble afin d'atteindre un objectif commun [ASC02].

3.3.3. Absence d'Infrastructure

Les réseaux Ad-hoc en général, et les réseaux de capteurs en particulier se distinguent des autres réseaux par la propriété d'absence d'infrastructure préexistante, et de tout genre d'administration centralisée.

3.3.4. Contrainte d'Energie

La caractéristique la plus critique dans les réseaux de capteurs est la limitation de ses ressources énergétiques. Comme nous l'avons vu dans la sous-section 2.2, chaque nœud-capteur du réseau est alimenté via une batterie. Le remplacement de ces batteries est une tâche difficile, voire impossible, vu la spécificité des applications des WSNs (militaires, sismiques, etc.). Par conséquent, la durée de vie d'un nœud est essentiellement dépendante de la durée de vie de la batterie. Afin de prolonger la durée de vie du réseau, une minimisation des dépenses énergétiques est exigée au niveau de chaque nœud, surtout en communication.

3.3.5. Ressources Matérielles Limitées

En plus de l'énergie, les nœuds-capteurs ont aussi une capacité de traitement et de mémoire limitée. En effet, les industriels veulent mettre en œuvre des capteurs simples, petits et peu coûteux, qui peuvent être achetés en masse. Ainsi, Afin de minimiser l'énergie consommée lors de transfert de données entre les nœuds, les micro-capteurs opèrent à bas débit (quelques dizaines de Kb/s).

3.3.6. Topologie Dynamique

La topologie des réseaux de capteurs peut changer au cours du temps pour les raisons suivantes :

- Les nœuds capteurs peuvent être déployés dans des environnements hostiles (champs de bataille par exemple); la défaillance d'un nœud capteur est donc très probable,
- Un nœud capteur peut devenir non opérationnel à cause de l'expiration de son énergie,
- Dans certaines applications, les nœuds capteurs et les stations de base sont mobiles.

3.3.7. Agrégation de Données

Dans les réseaux de capteurs, les données produites par les nœuds capteurs voisins sont très corrélées spatialement et temporellement. Ceci peut engendrer la réception d'informations redondantes par la station de base. Pour optimiser la consommation d'énergie, il faut réduire la quantité d'informations redondantes transmises par les micro-capteurs.

L'une des techniques utilisée pour réduire la transmission d'informations redondantes est l'agrégation de données [CON03]. Avec cette technique, les nœuds intermédiaires agrègent l'information reçue de plusieurs sources. Cette technique est connue aussi sous le nom de fusion de données.

3.3.8. Tolérance aux Pannes

Dans le cas de dysfonctionnement d'un nœud (manque d'énergie, interférences avec l'environnement d'observation, etc.), ou aussi en cas d'ajout de nouveaux nœuds capteurs dans le réseau, ce dernier doit continuer à fonctionner normalement sans interruption [HWi03]. Ceci explique le fait qu'un WSN n'adopte pas de topologie fixe, mais plutôt dynamique.

3.4. Types des Flux de Données dans un WSN

La transmission de données dans les WSNs peut se faire suivant plusieurs modèles dont on distingue quatre essentiels [Dap08] :

3.4.1. Périodique

Chaque micro-capteur dans le réseau envoie périodiquement des paquets aux destinations. La plupart des applications génèrent des trafics périodiques.

3.4.2. Événementiel

Les micro-capteurs envoient des paquets aux destinations lorsqu'ils détectent un événement. Par exemple, dans les applications de surveillance, un nœud-capteur n'envoie des paquets que s'il détecte un événement dans la zone qu'il surveille.

3.4.3. Requête/Réponses

Un contrôleur du réseau (*sink*) envoie une requête à un sous ensemble du réseau pour demander une information particulière. Cet ensemble envoie des paquets aux destinations en réponse à cette requête. Par exemple, dans une application de télésurveillance médicale, le médecin envoie une requête pour demander l'état actuel des signes vitaux d'un patient.

3.4.4. Hybride

Dans lequel une application utilise plusieurs types de trafic, décrits plus haut, à la fois. De nos jours, plusieurs applications utilisent ce modèle. Par exemple, l'application de télésurveillance médicale peut utiliser tous les types de trafic à la fois. Les micro-capteurs envoient d'une façon périodique l'état des fonctions vitales du patient. Si entre temps un problème est détecté, un événement est transmis au professionnel de santé. Ce dernier peut envoyer une requête pour vérifier l'état du patient à un instant donné.

4. Domaines d'Application des Réseaux de Capteurs Sans Fil

En ce début du 21^{ème} siècle, nous assistons à des bouleversements considérables dans le secteur des technologies de l'information, à la fois en termes applicatifs, usages et technologiques. En effet, Le besoin d'observer et de contrôler des phénomènes physiques tels que la température, la pression, etc. est essentiel pour de nombreuses applications industrielles et scientifiques. Il n'y a pas si longtemps, la seule solution pour acheminer les données du micro-capteur jusqu'au contrôleur central, était le câblage qui avait comme

principaux défauts d'être coûteux et encombrant. Aujourd'hui, grâce aux récents progrès des techniques sans-fils, de nouveaux produits exploitant les WSNs sont employés pour récupérer ces données environnementales. Des exemples d'applications potentielles dans des différents domaines sont exposés ci-dessous [MJS02], [SDZ07], [GIG07].

4.1. Applications Environnementales

Le contrôle des paramètres environnementaux par les réseaux de capteurs peut donner naissance à plusieurs applications. Par exemple, le déploiement des micro-capteurs thermiques dans une forêt peut aider à détecter un éventuel début de feu. Le déploiement des capteurs chimiques dans les milieux urbains, peut aider à détecter la pollution et analyser la qualité d'air. De même, le déploiement d'un WSN dans les sites industriels empêche les risques industriels tels que la fuite de produits toxiques (gaz, produits chimiques, éléments radioactifs, pétrole, etc.).

4.2. Applications Militaires

On peut penser à un réseau de capteurs déployé sur un endroit stratégique ou difficile à l'accès, afin de surveiller toutes les activités des forces ennemies, ou d'analyser le terrain avant d'y envoyer des troupes (détection d'agents chimiques, biologiques ou de radiations). Des tests concluants ont déjà été réalisés dans ce domaine par l'armée américaine dans le désert de Californie.

4.3. Applications Médicales

La médecine et le système de santé peuvent aussi profiter de l'application des WSNs. Les micro-capteurs peuvent capter des paramètres, tels que le rythme cardiaque et le taux de sucre dans le sang, puis transmettre ces informations à d'autres nœuds-capteurs (Actionneur) qui interviendront avant que la situation ne devienne critique.

4.4. Applications de Gestion du Trafic Routier

Un réseau de capteurs pourrait être utilisé dans un parking pour gérer automatiquement le stationnement des voitures. De même, des micro-capteurs pourraient être placés dans les entrées/sorties des autoroutes et des grandes villes pour réaliser des statistiques sur les flux de véhicules. Une autre application est celle de la gestion des feux de signalisation dans les intersections de façon dynamique, et le guide (routage) dynamique des abonnés pour éviter les zones encombrées.

4.5. Applications Commerciales

Il est possible d'intégrer des nœuds-capteurs au processus de stockage et de livraison. Le réseau ainsi formé pourra être utilisé pour connaître la position, l'état et la direction d'un paquet. Il devient alors possible pour un client qui attend la réception d'un paquet, d'avoir un avis de livraison en temps réel et de connaître la position actuelle du paquet. Pour les entreprises manufacturières, les réseaux de capteurs permettront de suivre le procédé de production à partir des matières premières jusqu'au produit final livré. Grâce aux WSNs, les entreprises pourraient offrir une meilleure qualité de service tout en réduisant leurs coûts. Dans les immeubles, le système de climatisation peut être conçu en intégrant plusieurs micro-capteurs dans les tuiles du plancher et les meubles. Ainsi, la climatisation pourra être déclenchée seulement aux endroits où il y a des personnes présentes et seulement si c'est nécessaire.

4.6. Applications Domestiques

Avec le développement technologique, les micro-capteurs peuvent être embarqués dans des appareils, tels que les aspirateurs, les fours à micro-ondes, les réfrigérateurs, etc. Ces micro-capteurs embarqués peuvent interagir entre eux et avec un réseau externe via internet pour permettre à un utilisateur de contrôler les appareils domestiques, localement ou à distance. Le déploiement des capteurs de mouvement et de température dans les futures maisons dites intelligentes permet d'automatiser plusieurs opérations domestiques telles que: la lumière s'éteint et la musique se met en état d'arrêt quand la chambre est vide, la climatisation et le chauffage s'ajustent selon les points multiples de mesure, le déclenchement d'une alarme par le nœud-capteur anti-intrusion quand un intrus veut accéder à la maison.

4.7. Application à la Sécurité

Les WSNs peuvent diminuer considérablement les dépenses financières consacrées à la sécurisation des lieux et des êtres humains. Ainsi, l'intégration des micro-capteurs dans de grandes structures telles que les ponts ou les bâtiments aidera à détecter les altérations dans la structure suite à un séisme ou au vieillissement de la structure. Le déploiement d'un réseau de capteurs de mouvement peut constituer un système d'alarme qui servira à détecter les intrusions dans une zone de surveillance.

5. Conclusion

Les réseaux de capteurs sans fil présentent un intérêt considérable, et une nouvelle étape dans l'évolution des technologies de l'information et de la communication. Cette nouvelle technologie suscite un intérêt croissant vu la diversité de leurs domaines d'application tels que la santé, l'environnement, l'industrie, etc. Dans ce chapitre, nous avons présenté les WSNs, leurs architectures de communication, la pile protocolaire des micro-capteurs et leurs diverses applications. Cependant, nous avons remarqué que plusieurs facteurs et contraintes compliquent la gestion de ce type de réseaux. En effet, les réseaux de capteurs se caractérisent par des ressources matérielles limitées, surtout ce qui concerne l'énergie. Ainsi, les WSNs sont confrontés à une multitude de challenges, parmi lesquels la synchronisation du temps que nous allons traiter d'une manière approfondie dans le chapitre suivant.

Chapitre 02 :
*La Synchronisation
Temporelle dans les
Réseaux de Capteurs
Sans Fil*

1. Introduction

Dans ce chapitre, nous présenterons en premier lieu le besoin de la synchronisation du temps dans les WSNs. Ensuite, nous allons introduire les concepts de base de la synchronisation temporelle. Nous étudierons également quelques protocoles de synchronisation du temps dédiés spécialement aux WSNs, c'est-à-dire, qu'ils prennent en considération la miniaturisation des micro-capteurs, et leurs contraintes de conception. Et enfin, nous terminerons par une conclusion.

2. Besoin de Synchronisation du Temps dans les WSNs

La synchronisation du temps est un service qui vise à construire une notion du temps commune dans le réseau, ou encore à la construction d'un convertisseur du temps qui permet à chaque nœud d'estimer les valeurs d'horloges générées dans les différents nœuds du réseau. Ceci représente un challenge important et primordial pour les applications collaboratives, ainsi que pour certains protocoles de base. Par exemple, les micro-capteurs collaborent entre eux pour effectuer une tâche de détection complexe telle que la fusion de données qui sert à regrouper les données collectées par les différents nœuds en un seul résultat significatif. Ceci peut être utilisé pour tracer le chemin d'un véhicule, ou les nœuds captent la position et le temps de détection du véhicule et les envoient à une station de base, qui à son tour combine toutes les informations pour tracer le chemin complet. Il est évident que si les nœuds ne sont pas synchronisés, le chemin sera inexact.

L'implémentation du mécanisme dit *duty-cycling* pour économiser la consommation d'énergie est un autre exemple ; les micro-capteurs peuvent être mis en veille (éteindre le module de communication sans fil) à des moments appropriés, et de se réveiller lorsque c'est nécessaire. Les nœuds doivent alors passer en veille, et se réveillent à des heures coordonnées. Ce mécanisme de coordination doit assurer que l'interface radio d'un nœud ne soit pas éteinte lorsqu'il est censé recevoir certaines données. Cela exige une synchronisation précise entre les micro-capteurs.

3. Concepts de Base

3.1. Modèle d'Horloge

L'horloge est définie comme étant un ensemble de composants matériels et logiciels utilisés pour fournir un temps exact, stable et fiable. L'oscillateur est le composant principal dans l'horloge. Chaque oscillateur est défini par une fréquence exprimée en Hertz. Les horloges utilisées dans les composants électroniques modernes sont caractérisées par deux paramètres très importants:

- **La précision** : c'est la différence entre les fréquences d'oscillations théoriques et réelles. Cet écart est appelé erreur de fréquence, il est donnée par le fabricant de l'oscillateur, et il est généralement de l'ordre de 10^{-5} pour les oscillateurs utilisés dans les composants électroniques des miro-capteurs modernes [WLC08].
- **La stabilité** : c'est la tendance de l'oscillateur à garder la même fréquence avec le temps. L'instabilité de l'oscillateur peut être à court terme et due aux facteurs d'environnement tels que les variations de la température, la pression, la tension d'alimentation, etc., ou à long terme et due à des effets plus subtils tels que le vieillissement de l'oscillateur.

La conséquence directe de l'instabilité de l'oscillateur, est le *skew*. Ce dernier, représente la fréquence d'une horloge à un instant « t ». Celle-ci peut changer par rapport à la fréquence initiale, d'où son appellation *skew* ou déviation.

La déviation continue des fréquences entraîne un décalage « *offset* » entre la valeur donnée par une horloge et le temps réel donné par une référence du temps telle que l'UTC.

La différence entre les fréquences de deux nœuds du réseau est appelée *skew relatif* (*relative skew*). De même, la différence entre les valeurs d'horloges données par ces deux nœuds est appelée *l'offset relatif* (*relative offset*).

Toutefois ; chaque nœud, soit n_1 , peut estimer à n'importe quel instant « t » la valeur d'horloge C_2 générée dans un autre nœud (n_2) du réseau en utilisant sa propre horloge C_1 via l'équation suivante :

$$\left[C_2(t) = a_{1,2} * C_1(t) + b_{1,2} \quad (2.1) \right]$$

Où $a_{1,2}$, $b_{1,2}$ sont le *skew* relatif et l'*offset* relatif, respectivement.

3.2. Exigences de la Synchronisation du Temps dans les WSNs

Beaucoup de travaux de recherche ont été consacrés pour la conception des nouveaux algorithmes de synchronisation spécialement adaptés aux réseaux de capteurs sans fil. Ces nouveaux algorithmes doivent prendre en considération plusieurs facteurs ou exigences. Ces derniers peuvent être considérés comme des indicateurs pour évaluer les schémas de synchronisation dédiés aux WSNs. Ci-dessous, nous citons ces facteurs [SiY04]:

a) Efficacité Energétique « Energy Efficiency » :

L'économie d'énergie est l'une des problématiques majeures dans les WSNs. En effet, la recharge des sources d'énergie est souvent trop coûteuse, et parfois impossible. Par conséquent, tous les protocoles conçus pour les WSNs, y compris les protocoles de synchronisation du temps, doivent minimiser au maximum la consommation énergétique.

b) Passage à l'Echelle « Scalability » :

La plupart des applications des réseaux de capteurs sans fil nécessitent le déploiement d'un grand nombre de nœuds-capteurs. De ce fait, les algorithmes de synchronisation doivent s'adapter aux réseaux de forte densité ou à l'augmentation du nombre de nœuds.

c) Précision « Precision » :

Le besoin de la précision peut varier considérablement selon le but de l'application. Pour certaines applications, un simple ordre entre les événements et les messages peut suffire (un temps logique [Lam78]), alors que pour d'autres, l'exigence de la précision de synchronisation peut être de l'ordre de quelques microsecondes, en raison de leur couplages étroit avec le monde physique.

d) Robustesse « Robustness »:

Un réseau de capteurs est généralement laissé sans surveillance pendant des longues périodes de fonctionnement, éventuellement, dans des environnements hostiles. En cas de panne ou de défaillance d'un petit nombre de nœuds, l'algorithme de synchronisation doit rester valide et fonctionnel pour le reste du réseau.

e) Durée de Vie de la Synchronisation « Life Time » :

La synchronisation du temps entre les nœuds fournie par un algorithme de synchronisation peut être instantanée, ou peut durer longtemps. Certaines applications des WSNs, telles que

la détection d'urgence ou la détection de fuite de gaz nécessitent la communication immédiate avec le *sink*. Dans ce type d'applications et lors de la détection de telles situations d'urgence, la pré-synchronisation des nœuds est exigée à tout moment.

f) Portée « Scope » :

Un algorithme de synchronisation peut fournir une synchronisation du temps global pour tous les nœuds du réseau, ou seulement une synchronisation locale, entre les nœuds voisins géographiquement. La synchronisation globale est difficile, et très coûteuse dans les réseaux de capteurs denses (en termes de consommation d'énergie et d'utilisation de la bande passante). Cependant, une synchronisation globale peut être nécessaire pour fusionner les données provenant des nœuds distants, dans un ordre chronologique.

3.3. Sources d'Erreurs

Toutes, les méthodes de synchronisation du temps dans les WSNs comptent sur l'échange de messages entre les nœuds. La cause majeure d'erreur dans les protocoles de synchronisation provient de l'estimation non-déterministe de la latence des messages échangés. Quand un nœud génère un estampille « *timestamp* » pour la synchronisation, le paquet portant cet estampillage fera face à une quantité variable de retard jusqu'à son arrivée à la destination prévue. Ce retard empêche le récepteur de comparer, avec exactitude, sa valeur d'horloge locale avec celle de l'expéditeur, comme montre la *figure 2.1*. Ces sources d'erreurs peuvent être divisées en quatre catégories différentes [SBK05]:

3.3.1. Temps d'Envoi « *Send Time* »

C'est le temps nécessaire pour la construction d'un paquet de synchronisation. Il comprend la latence du système d'exploitation (par exemple, les changements du contexte), et le temps écoulé pour transférer le message à l'interface réseau.

3.3.2. Temps d'Accès « *Access Time* »

Il s'agit du temps d'attente pour accéder au canal de transmission (contention). Chaque paquet transmis est confronté à des retards dans la couche de contrôle d'accès au médium (MAC). Ce temps dépend du protocole MAC utilisé. De plus, les retransmissions dues aux problèmes d'interférence ou de collision font augmenter ce temps d'accès.

3.3.3. Temps de Propagation « *Propagation Time* »

C'est le temps nécessaire pour la transmission du message entre les interfaces réseau de l'émetteur et de récepteur. Dans les réseaux sans fil où la portée est limitée à quelques dizaines de mètres, le temps de propagation est négligeable quand il s'agit du même domaine de diffusion (un seul saut).

3.3.4. Temps de Réception « *Receive Time* »

Il s'agit du temps nécessaire à l'interface réseau du récepteur pour la réception du message et la notification du système d'exploitation de son arrivée. C'est typiquement le temps nécessaire pour générer un signal de réception du message. Il contient aussi, le temps écoulé dans le transfert de message à la couche application.

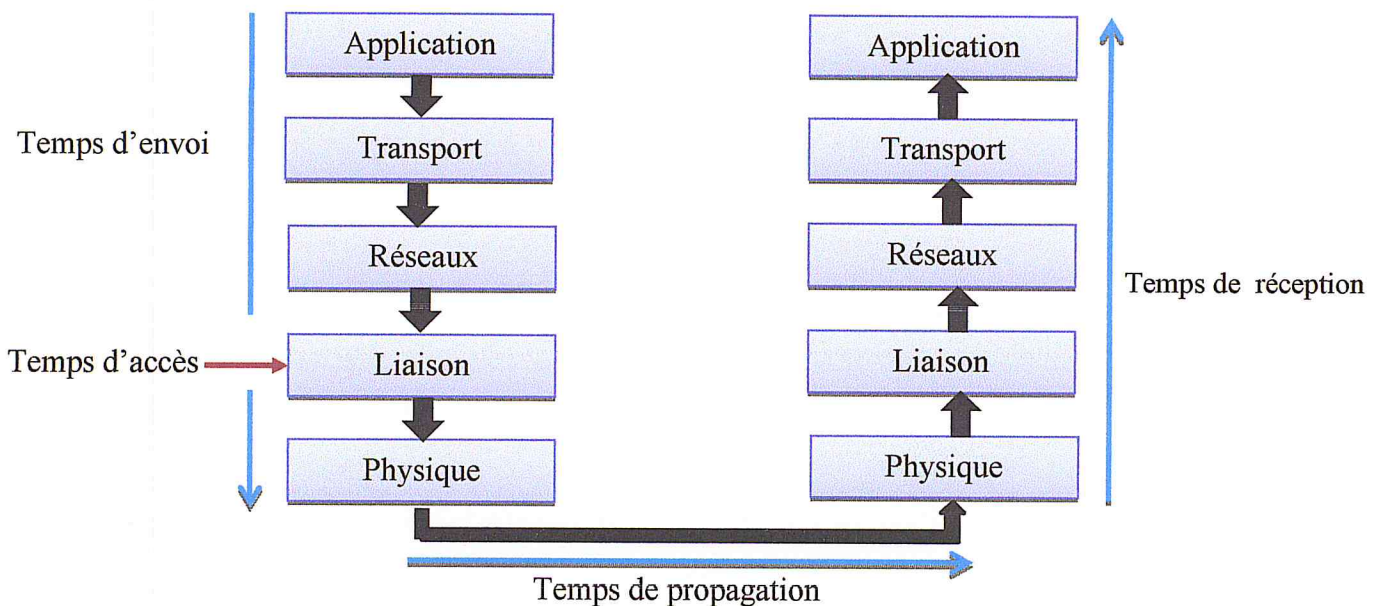


Figure 2.1. Sources d'erreurs communes dans les méthodes de synchronisation.

3.4. Classification des Protocoles de Synchronisation du Temps

Les protocoles de synchronisation du temps peuvent être classés selon plusieurs critères [SBK05], [SiY04] :

3.4.1. Maître/Esclave versus Peer-to-Peer

Il existe des applications qui nécessitent la synchronisation de tous les nœuds, esclaves, avec un seul nœud maître. Cette approche a besoin d'un algorithme d'élection du chef en cas de défaillance du nœud maître, mais elle est facile à mettre en œuvre. Par contre,

l'approche peer-to-peer permet une synchronisation dynamique entre un couple de nœuds lorsque c'est nécessaire. Cette approche est plus robuste et plus flexible mais elle est très difficile à mettre en œuvre.



Figure 2.2. Synchronisation Maître/esclave versus peer-to-peer

3.4.2. Sender / Receiver versus Receiver / Receiver

La plupart des protocoles de synchronisation du temps sont basés sur l'approche sender/receiver (émetteur/récepteur). Dans cette approche, le nœud émetteur envoie périodiquement un horodatage (*timestamp*) au récepteur. Après l'échange d'un certain nombre de messages, le récepteur peut estimer son décalage par rapport à l'émetteur. Dans la deuxième approche, un nœud tiers est utilisé pour synchroniser un ensemble de récepteur entre eux.

3.4.3. Synchronisation Relative versus Synchronisation Absolue

Il existe des applications qui nécessitent seulement un temps relatif. Dans ce cas, les nœuds du réseau doivent être synchronisés entre eux; c'est-à-dire, chaque nœud doit être capable d'estimer les valeurs d'horloges générées dans les autres nœuds du réseau. Par exemple, pour une application qui détecte la durée de propagation du son, il suffit que les nœuds qui captent le début du son soient synchronisés avec ceux qui captent sa fin pour pouvoir calculer la différence des temps de détection et évaluer la durée de propagation. Par contre, il existe d'autres applications qui nécessitent un temps absolu. Dans ce deuxième cas, il faut synchroniser les capteurs par rapport à une référence du temps externe telle que l'UTC.

3.4.4. Synchronisation Locale versus Synchronisation Globale

Dans le mécanisme de la synchronisation locale (un seul-saut), on suppose que tous les nœuds sont voisins les uns des autres. C'est-à-dire, les nœuds ne couvrent pas une zone géographique plus grande que celle couverte par la portée de leurs interfaces radios (quelques dizaines de mètres). Par contre, le mécanisme de la synchronisation globale (multi-sauts) permet de synchroniser tous les nœuds du réseau, quel que soit leurs dispersions géographique. Cela est faisable, par exemple, si nous avons un nœud passerelle, qui appartient à deux domaines de diffusion successifs.

3.4.5. Correction d'Horloge versus Horloge Relative

La première catégorie vise à construire une notion du temps commune à travers le réseau. Ceci, en modifiant les valeurs d'horloges des nœuds après avoir estimé les paramètres de synchronisation (*drift* relatif et/ou *offset* relatif). Les protocoles de la deuxième catégorie représentent des convertisseurs du temps. Pour ce faire, les horloges des micro-capteurs s'exécutent de façon indépendante tout en gardant en mémoire les paramètres de la synchronisation. Ces paramètres doivent être mis-à-jour régulièrement. De cette façon, chaque nœud X du réseau peut convertir, à n'importe quel instant, sa valeur d'horloge à la valeur d'horloge qui aura été générée dans un nœud Y du réseau.

3.4.6. Synchronisation Centralisée versus Synchronisation Décentralisée

Il existe des protocoles de synchronisation du temps qui permettent à un seul nœud, fixé au préalable, de lancer l'échange des messages de synchronisation entre les nœuds du réseau. De ce fait, le protocole ne continuera jamais à s'exécuter dans le cas de défaillance de ce nœud. Par contre, les protocoles de synchronisation du temps décentralisés ne tentent pas de fixer une référence. Cette deuxième catégorie garantit la continuité de l'exécution de l'algorithme tant qu'il existe deux nœuds qui fonctionnent dans le réseau.

4. Protocoles de Synchronisation du Temps dans les WSNs

Comme mentionné auparavant, les protocoles de synchronisation du temps dédiés aux réseaux filaires ne sont pas appropriés aux WSNs à cause des contraintes matérielles de ces derniers. Nous présenterons dans ce qui suit, quelques protocoles de synchronisation du temps proposés spécialement pour les réseaux de capteurs sans fil.

4.1. Timing-sync Protocol for Sensor Networks « TPSN »

4.1.1. Description

TPSN [GKS03] est un protocole de synchronisation du temps destiné au WSNs. Il est basé sur l'approche sender/receiver. Ce protocole vise à fournir une synchronisation du temps suivant le modèle « Always-on », c'est-à-dire, chaque nœud maintient une horloge synchronisée à un nœud référence dans le réseau. De ce fait, TPSN permet d'établir une notion du temps commune pour tous les nœuds du réseau. Ceci en se basant sur la correction des horloges. Le protocole TPSN suit l'approche maître/esclave, et il est adapté

aux réseaux multi-hop de forte densité. Il appartient à l'ensemble des protocoles de synchronisation décentralisés, car, le nœud racine peut être choisi par l'application d'un algorithme d'élection du chef « leader election algorithm ». TPSN utilise uniquement les liens bidirectionnels, afin de synchroniser les nœuds.

4.1.2. Principe

TPSN fonctionne en deux phases, la phase de découverte de niveau et la phase de synchronisation. Dans la deuxième phase, chaque nœud de niveau « i » doit être capable de communiquer au moins avec un seul nœud de niveau « $i-1$ ». A la fin de la phase de synchronisation, tous les nœuds du réseau vont être synchronisés avec le nœud racine.

4.1.3. Phase de Découverte de Niveau « *Level Discovery Phase* »

Cette phase survient lorsque le réseau est déployé. Elle a pour but de créer une topologie hiérarchique dans le réseau, où chaque nœud est attribué à un niveau. Un seul nœud est affecté au niveau « 0 », c'est le nœud racine. Ce nœud peut être le *sink*, comme il peut être choisi via un algorithme d'élection du chef. Ainsi, la racine peut être équipée par un GPS. Dans ce cas, les nœuds du réseau vont être synchronisés par rapport au monde physique.

Au déploiement du réseau, le nœud racine est auto-attribué au niveau 0, et il initie la phase de découverte du niveau par la diffusion d'un paquet « *level_discovery* » à ces voisins qui appartiennent à son domaine de diffusion. Le paquet envoyé contient l'identifiant, et le niveau de l'expéditeur. A la réception du paquet *level_discovery* par les voisins du nœud racine, chacun doit s'auto-attribuer à un niveau. Ce dernier doit être égal au niveau reçu plus 1. Après avoir établi leur propre niveau, chaque récepteur diffuse un paquet « *level_discovery* » qui contient son propre niveau et son identifiant.

Pour assurer l'appartenance d'un nœud à un seul niveau, chaque nœud doit négliger le paquet « *level_discovery* » après la première réception. A la fin de l'exécution de cette phase, une structure hiérarchique sera créée. La *figure 2.3*, peut être le résultat après l'exécution de la phase de découverte du niveau.

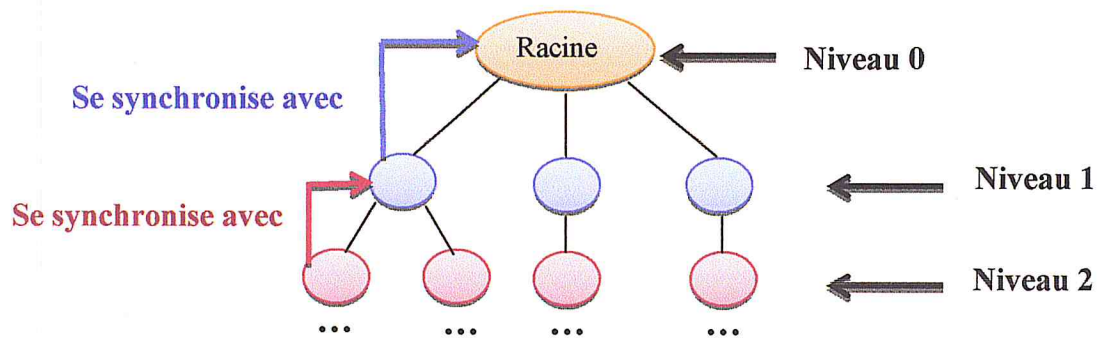


Figure 2.3. Topologie hiérarchique des nœuds

Chaque nœud doit être une partie de la topologie hiérarchique, afin qu'il puisse être synchronisé avec le nœud racine. Cependant, il se peut qu'un nœud ne soit pas attribué à un niveau à cause de l'un des scénarios suivants. Premièrement, un nœud peut rejoindre le réseau après l'exécution de la première phase. Ainsi, même si ce nœud est présent dès le déploiement du réseau, il peut rater la réception du paquet « level_discovery » à cause des collisions. Pour remédier à ces deux situations, chaque nœud déployé doit attendre un certain temps. S'il ne reçoit pas le paquet « level_discovery » dans ce délai, il diffuse un message « level_request ». A la réception de ce message par ses voisins, ces derniers répondent en envoyant leurs propre niveau. Enfin, le nœud s'attribue au niveau reçu.

4.1.4. Phase de Synchronisation « Synchronization Phase »

Après l'exécution de la première phase, le nœud racine déclenche la phase de synchronisation en diffusant le message « time_sync ». A la réception de ce message par les nœuds du niveau « 1 », chaque nœud doit enregistrer sa valeur d'horloge dans une variable **T1**, attendre un certain temps, puis envoyer un paquet « synchronization_pulse » vers le nœud racine. Ce paquet contient le niveau de l'émetteur, et sa valeur d'horloge lors de l'envoi de ce paquet. Le temps d'attente entre la réception de « time_sync » et l'envoi de « synchronization_pulse », permet d'éviter les collisions.

A la réception du paquet « synchronization_pulse », le nœud racine doit enregistrer sa valeur d'horloge dans une variable **T2**, attendre un certain temps, réenregistrer la valeur d'horloge dans **T3**, puis envoyé un accusé de réception au nœud de niveau 1 (l'émetteur du paquet «synchronization_pulse»). Cet accusé contient en plus du niveau du nœud racine, les trois valeurs d'horloges : **T1**, **T2** et **T3**. A la réception de l'accusé par les nœuds du

niveau 1, chacun d'eux enregistre sa valeur d'horloge locale dans la variable T4. Nous allons présenter dans la section ultérieure, comment les nœuds de niveau 1 utilisent ces variables pour ajuster leurs valeurs d'horloges.

Après la synchronisation des nœuds du niveau 1 avec la racine, chaque nœud du niveau « i » doit se synchroniser aux nœuds du niveau « i-1 ». Les nœuds du niveau « i » doivent attendre un certain temps, avant d'envoyer le paquet « synchronization_pulse » au nœud du niveau « i-1 ». Ce temps permet de s'assurer que les nœuds de niveau « i-1 » ont terminé l'exécution de la phase de synchronisation. A noter, qu'un nœud ne renvoie un accusé du paquet « synchronization_pulse », sauf s'il est déjà synchronisé avec les nœuds du niveau « i-1 ». Ceci garantit que les nœuds des différents niveaux ajustent leurs horloges par rapport à celle du nœud racine. Le processus de synchronisation des nœuds des niveaux « i », avec ceux du niveau « i-1 », est pareil à celui de synchronisation des nœuds du niveau « 1 » avec la racine.

4.1.5. Estimation des Paramètres de Synchronisation

Le protocole TPSN permet d'estimer l'*offset* relatif seulement. La *figure 2.4* représente l'échange de messages « synchronization_pulse » (M1), envoyés par le nœud A du niveau « i » et l'accusé de réception (M2), envoyé par le nœud B du niveau « i-1 ».

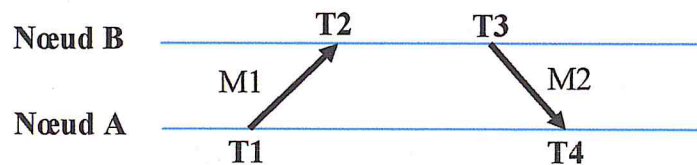


Figure 2.4. Echange de deux messages entre les deux nœuds A et B

Le nœud « A » veut ajuster sa valeur d'horloge avec celle du nœud « B ». Pour cela, le nœud A utilise les valeurs T1, T2, T3, T4 pour calculer l'*offset* relatif et le délai de propagation comme suit :

$$\Delta = \frac{(T2-T1)-(T4-T3)}{2} \quad (2.2) \quad ; \quad d = \frac{(T2-T1)+(T4-T3)}{2} \quad (2.3)$$

« Δ » représente l'*offset* relatif entre les deux nœuds « A » et « B », et « d » est le délai de propagation entre le nœud A et le nœud B.

temps, qui permet à chaque nœud de garder en mémoire les paramètres de synchronisation (*skew* relatif, *offset* relatif) sur l'ensemble des nœuds du réseau.

4.3.2. Principe

L'idée principale de RBS est d'utiliser un tiers pour la synchronisation. Au lieu de synchroniser un émetteur à un récepteur, ce protocole utilise un tiers (référence) pour synchroniser un ensemble de récepteurs entre eux. Le choix de la référence, est idem au choix de la racine dans le protocole TPSN (le nœud *sink*, ou un nœud sélectionné via un algorithme d'élection du chef). Dans le cas de la synchronisation locale, la référence diffuse périodiquement une balise de synchronisation, qui ne contient aucun *timestamp*. Cette balise va être reçue par les nœuds qui se situent dans le même domaine de diffusion que la référence.

Chaque nœud reçoit la balise de synchronisation, doit enregistrer ce temps de réception, selon son horloge locale. Ensuite, les nœuds échangent ces temps de réception entre eux. Après la réception d'un ensemble de balises de synchronisation, chaque nœud peut estimer les paramètres de la synchronisation par rapport aux autres nœuds du réseau.

4.3.3. Estimation des Paramètres de Synchronisation

Le premier modèle de RBS permet d'estimer l'*offset* seulement. Après la réception d'un certain nombre de balises de synchronisation, tous les nœuds qui appartiennent au même domaine de diffusion peuvent estimer leurs décalages les uns aux autres comme suit :

$$\forall i, j \in n : \text{Offset} [i, j] = \frac{1}{m} \sum_{k=1}^m (T_{j,k} - T_{i,k}) \quad (2.10)$$

Où « *i* » et « *j* » sont deux nœuds du même domaine de diffusion. « *n* » étant le nombre de nœuds dans le réseau. « *m* » est le nombre de balises de synchronisation envoyées par la référence, et $T_{r,b}$ est le temps de réception de la balise *b* par le nœud *r*.

Le deuxième modèle de RBS permet d'estimer les deux paramètres de la synchronisation (*skew* et *offset*). Pour cela, RBS utilise la méthode de régression des moindres-carrés linéaires « least-squares linear regression » [PPi02]. Le but est de trouver la meilleure ligne d'ajustement, en utilisant les temps de réception des balises de synchronisation comme échantillon. Le *skew* relatif et l'*offset* relatif sont récupérés, respectivement, à partir de la pente et l'intercepte de la ligne. Cette ligne suppose implicitement que la fréquence

Pour prendre en considération le problème du *drift*, le protocole TPSN peut utiliser l'un des estimateurs décrit en détail dans [NCS07].

4.2. Flooding Time Synchronization Protocol « FTSP »

FTSP [MKS04] est un protocole de synchronisation du temps fondé sur l'approche Sender/Receiver. Il organise une arborescence ad-hoc de synchronisation selon laquelle une racine est sélectionnée comme une source du temps. Ce protocole utilise la technique unidirectionnelle pour synchroniser les nœuds du réseau.

La phase initiale de FTSP consiste à choisir le nœud racine, qui représente une source du temps pour tout le réseau. Le processus d'élection du nœud racine doit sélectionner le nœud qui possède l'identifiant le plus petit. L'élection du nœud racine est suivie par une fenêtre de synchronisation, qui s'écoule pendant un intervalle du temps « I ». Le choix de cet intervalle est dicté par les exigences de précision de l'application développée.

La racine diffuse un paquet « synchronization », qui sera reçu par tous les nœuds voisins. Le paquet doit contenir trois champs, timestamp, rootID et le champ seqNum. Le champ d'horodatage représente le temps de l'expéditeur lors de l'envoi du paquet, le rootID représente l'identificateur de la racine, et le seqNum est un numéro de séquence incrémenté uniquement par la racine au début de chaque tour de synchronisation. A la réception du paquet de synchronisation, chaque nœud doit calculer les paramètres de synchronisation, pour déterminer son décalage par rapport à l'expéditeur, corriger son horloge, et diffuser un message « synchronization ». Ainsi, le temps est effectivement inondé à travers le réseau.

4.3. Reference Broadcast Synchronization « RBS »

4.3.1. Description

Contrairement à la majorité des protocoles de synchronisation du temps existants, RBS [EGE02] est basé sur l'approche *receiver/receiver*. Ce protocole permet, principalement, une synchronisation relative entre les nœuds du réseau. RBS appartient à la catégorie des protocoles peer-to-peer. Il fournit une synchronisation locale, ainsi que sur des chemins multi sauts. C'est un protocole centralisé, vu que le même nœud référence est responsable de l'émission des balises « beacons » de synchronisation à chaque fois. Le protocole RBS ne nécessite aucune modification dans les valeurs d'horloges. C'est un convertisseur du

d'horloge est stable ; c'est-à-dire, l'*offset* évolue à un rythme constant. Comme nous l'avons mentionné précédemment, la fréquence des oscillateurs réels change au fil du temps. Pour remédier à ce problème, le protocole RBS n'utilise que les derniers échantillons (en supposant que les fréquences sont stables pendant une petite durée).

4.3.4. Synchronisation Globale

RBS propose une extension multi-sauts. L'idée est d'utiliser des nœuds passerelles qui appartiennent à deux domaines de diffusion successifs. Ces passerelles ont le rôle de garantir la synchronisation via des chemins multi-sauts.

Considérant l'exemple illustré par la *figure 2.5* Les nœuds « A » et « B » sont deux références de synchronisation. Ces nœuds forment deux domaines de diffusion, le domaine de diffusion du nœud « A » contient les nœuds {1, 2, 3, 4}, et celui du nœud « B » contient les nœuds {4, 5, 6, 7}. Le nœud « 4 » appartient aux deux domaines de diffusion, c'est le nœud passerelle. En profitant de la position du nœud « 4 », on peut comparer des évènements qui surviennent dans les nœuds « 2 » et « 6 », par exemple.

Au début, le protocole de synchronisation locale doit être exécuté. Les nœuds « 2 » et « 6 » reçoivent les balises de synchronisation dans les temps PA et PB, respectivement. Si le nœud « 2 » a observé l'évènement E2, deux unités de temps après PA, et le nœud « 6 » a observé l'évènement E6 au moment PB - 4. Ces nœuds peuvent consulter le nœud passerelle « 4 » (lors du besoin d'une synchronisation relative) pour savoir que l'impulsion diffusée par « A » a eu lieu, 10 unités de temps après l'impulsion diffusé par « B ». En résumer :

$$E2 = PA + 2 \quad (2.11)$$

$$E6 = PB - 4 \quad (2.12)$$

$$PA = PB + 10 \quad (2.13)$$

En remplaçant (2.12) dans (2.13) et (2.13) dans (2.11), on trouve : $E2 = E6 + 16$. De cette façon, les nœuds « 2 » et « 6 » ont suffisamment d'information pour pouvoir comparer leurs évènements. Cependant, ce mécanisme permet la comparaison des évènements survenant dans les différents nœuds du réseau, plutôt que la conversion des horloges.

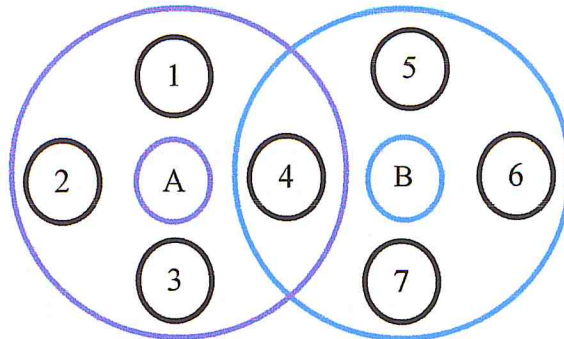


Figure 2.5. Schéma d'un réseau multi-sauts

4.4. Relative Referencless Receiver/Receiver Time Synchronisation in Wireless Sensor Networks « R⁴Syn »

4.4.1. Description

Le protocole R⁴ Syn proposé d'abord dans [Dje11], puis complété dans [Dje12], est basé sur RBS (l'approche *Receiver/Receiver*). De même que RBS, ce protocole est un convertisseur du temps, il permet une synchronisation locale et globale. Il appartient à l'ensemble des protocoles *peer-to-peer*. Ainsi, R⁴ Syn permet de fournir une synchronisation relative entre les nœuds du réseau. Cependant, ce protocole est totalement décentralisé, il ne dépend d'aucune référence fixe. Le rôle de la référence est balancé entre tous les nœuds du réseau.

4.4.2. Principe

Ce protocole fonctionne en cycle. Dans chacun, les nœuds diffusent séquentiellement des messages de synchronisation (*beacons*, ou balises). Le *beacon* va être reçu par les nœuds qui se trouvent dans le même domaine de diffusion que l'émetteur, (l'extension multi-sauts de ce protocole va être présentée dans la sous-section 4.4.4). L'ordre des diffusions, peut être déterminé en utilisant les identificateurs des nœuds. Quand tous les nœuds du réseau envoient une balise de synchronisation, l'initiateur du protocole déclenche un nouveau cycle.

Lors de la réception des messages de synchronisation, chaque nœud doit enregistrer le temps de réception selon son horloge locale. Dans le premier cycle, les nœuds diffusent des messages vides. A partir du deuxième cycle, chaque balise diffusée, doit transporter $(n - 1)$ *timestamps*, où n est le nombre de nœuds dans le réseau à synchroniser. Ce sont les temps de réception des balises reçues dans le cycle précédent. Pour illustrer le fonctionnement de

Dans ces formules, n_1 représente le nœud source, qui veut estimer les paramètres de synchronisation par rapport au nœud distant n_h . $Skew_{n_{i+1} \rightarrow n_i}$ représente le *skew* relatif entre les nœuds, n_{i+1} et n_i , idem pour l'*offset*.

4.5. Efficient Time Synchronization Mechanism for Wireless Multi Hop Networks « ETSM »

ETSM [WLC08], appartient à la classe des protocoles Receiver / Receiver. Contrairement à RBS sur lequel est basé tous les protocoles de cette classe, ETSM suit l'approche maître / esclave. Par conséquent, ce protocole synchronise tous les nœuds du réseau par rapport à un nœud référence. Cette propriété permet d'obtenir, facilement, une synchronisation absolue. Mise à part l'approche maître / esclave, ETSM partage les mêmes caractéristiques avec le protocole RBS. Pour la synchronisation locale, où tous les nœuds sont voisins les uns des autres, le protocole ETSM comporte deux étapes principales ; la sélection des nœuds émetteurs et la synchronisation.

4.5.1. Sélection des Nœuds Emetteurs

Dans un réseau à un saut, la synchronisation des différents nœuds par rapport à une référence nécessite deux émetteurs des messages de synchronisation. Le premier est le nœud référence lui-même " N_R " (référence de temps), par contre le deuxième doit être choisi de la manière suivante: le nœud référence transmet un message d'avertissement M_{avr} pour indiquer sa présence et pour déclencher le processus de synchronisation. Le message M_{avr} contient l'identifiant du " N_R " et un numéro de séquence. A la réception de ce message, chaque nœud choisit un temps aléatoire borné avant d'essayer de retransmettre le message M_{avr} . Le premier nœud qui réussit à transmettre ce message (celui qui a choisi le temps aléatoire le plus court) est considéré comme le deuxième nœud émetteur " N_E " (référence de message de synchronisation). A la réception du deuxième message d'avertissement, les autres nœuds annulent leur émission du message M_{avr} et se considèrent clients (esclaves) du mécanisme de synchronisation.

4.5.2. Synchronisation des Nœuds

Le nœud émetteur, N_E , déclenche des cycles de synchronisation périodiquement. Dans chacun, il transmet des messages de référence " M_{ref} ". Ces messages contiennent le numéro

ce protocole, on considère un réseau avec quatre nœuds. La *figure 2.6* illustre le déroulement du cycle « j » où $j > 1$. $B_{i,j}$ désigne la balise diffusée par le nœud i dans le cycle, j, et $t_{i,j}^k$ est le temps de réception de la balise diffusée par le nœud " i " dans le cycle " j " selon l'horloge locale de nœud " k ".

Comme nous l'avons mentionné plus haut, chaque balise diffusée contient les temps des réceptions des balises du cycle précédent. Par exemple, la balise $B_{1,j}$ contient $(t_{2,j-1}^1, t_{3,j-1}^1, t_{4,j-1}^1)$. Ces *timestamps* vont être utilisés par la suite comme des échantillons pour calculer les paramètres de synchronisation.

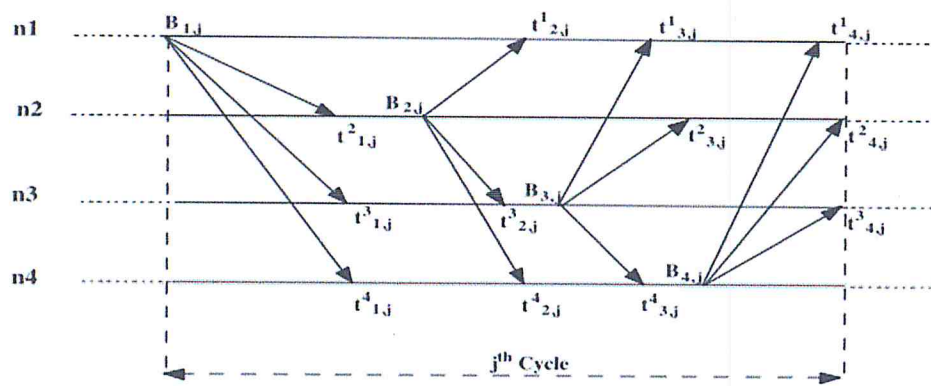


Figure 2.6. Diffusion des messages de synchronisation dans un seul cycle [Dje11]

4.4.3. Estimation des Paramètres Synchronisation

Le premier modèle de R⁴ Syn permet d'estimer l'offset seulement, en utilisant l'équation Eq.2.10.

R⁴ Syn possède une deuxième variante qui permet d'estimer les deux paramètres de la synchronisation. Les estimateurs utilisés par ce protocole sont basés sur le modèle mathématique, le maximum de vraisemblance MLE « Maximum Likelihood Estimators » [PPi02]. MLE est une méthode statistique, qui permet d'estimer un paramètre inconnu θ d'un modèle ou d'une loi de probabilité. Ainsi, c'est une fonction qui fait correspondre à une suite d'observations (un échantillon), une valeur estimée $\hat{\theta}$. Dans notre cas, le paramètre à estimer est le *skew* relatif / *offset* relatif. L'échantillon est les temps de réception des balises de synchronisation.

Soit deux nœuds n_1 et n_2 , qui appartiennent au même domaine de diffusion. Le nœud n_2 peut estimer les paramètres de synchronisation par rapport à n_1 via les formules suivantes:

$$Skew = \frac{\sum_{i=1}^k u_i \sum_{i=1}^k v_i - k \sum_{i=1}^k u_i v_i}{(\sum_{i=1}^k v_i)^2 - k \sum_{i=1}^k v_i^2} \quad (2.14)$$

$$Offset = \frac{1}{k} \left(\sum_{i=1}^k u_i - \frac{\sum_{i=1}^k u_i \sum_{i=1}^k v_i - k \sum_{i=1}^k u_i v_i}{(\sum_{i=1}^k v_i)^2 - k \sum_{i=1}^k v_i^2} \sum_{i=1}^k v_i \right) \quad (2.15)$$

u_i , v_i représentent les temps de réception de la balise " i " par le nœud, n_1 , n_2 , respectivement.

4.4.4. Synchronisation Globale

En plus de la synchronisation locale présentée ci-dessus, le protocole R⁴ Syn possède une extension multi-sauts [Dje12]. Pour ce faire, chaque nœud du réseau exécute le protocole de synchronisation avec ses voisins (une synchronisation locale). Dès que deux nœuds éloignés géographiquement ont besoin de se synchroniser, ils établissent une route de communication entre eux. Les nœuds routeurs (nœuds intermédiaires), avaient déjà exécuté la synchronisation locale, donc chaque routeur « i » possède les paramètres de synchronisation par rapport au routeur « i+1 ». Ces nœuds intermédiaires doivent transmettre ces paramètres aux nœuds communicants. A la fin, les nœuds communicants peuvent estimer les paramètres de synchronisation via les formules suivantes :

$$Skew_{n_1 \rightarrow n_h} = \prod_{i=1}^{h-1} Skew_{n_i \rightarrow n_{i+1}} \quad (2.16)$$

$$Offset_{n_1 \rightarrow n_h} = \sum_{i=2}^{h-1} \left[\left(\prod_{j=2}^i Skew_{n_{j-1} \rightarrow n_j} \right) Offset_{n_i \rightarrow n_{i+1}} \right] + Offset_{n_1 \rightarrow n_2} \quad (2.17)$$

du message dans le cycle, le nombre total de messages à émettre pendant ce cycle, et un numéro de séquence qui est incrémenté à chaque envoi du message de synchronisation pour le différencier des autres. Ces informations permettent à chaque nœud de détecter la fin de la transmission des messages de référence. Chaque récepteur, y compris le nœud référence, enregistre les temps de réception et le numéro de séquence des messages reçus. A la fin du cycle, chaque nœud envoie un message de réponse M_{rep} au nœud émetteur. Ce message contient les temps de réception des messages de synchronisation dans le dernier cycle. A la réception des messages " M_{rep} ", le nœud émetteur calcule les paramètres de la synchronisation de chaque nœud par rapport au nœud référence. Ensuite, il diffuse un message d'ajustement " M_{aju} " qui contient les adresses de chaque récepteur ainsi que ses paramètres de synchronisation. Ces derniers seront utilisés par la suite pour transformer les valeurs d'horloges locale à la valeur d'horloge qui aura été générée chez le nœud référence.

Une fois que le M_{aju} est reçu par la référence, elle commence son cycle de transmission de message " M_{ref} " afin de synchroniser le nœud émetteur N_E (le seul nœud non synchronisé dans le réseau). Ce cycle est identique à celui décrit ci-dessus avec l'unique différence que seuls les nœuds N_E et celui avec le plus petit ID répondront à ces messages. La raison de cette procédure est de minimiser la génération des messages de synchronisation. A la fin de ce cycle, tous les nœuds du réseau seront synchronisés.

4.5.3. Estimation des Paramètres de Synchronisation

Ce protocole permet d'estimer l'*offset* et le *skew* entre les horloges des nœuds et celle de référence de la même façon que le protocole RBS (voir la section 4.3.3).

4.5.4. Synchronisation Globale

Dans un réseau multi-sauts, le protocole ETSM comporte aussi deux étapes. La première consiste à sélectionner un ensemble de nœuds émetteurs qui doit couvrir tous les nœuds du réseau (chaque nœud doit être affecté à un nœud émetteur). La deuxième étape est la synchronisation. Cette-fois-ci, la synchronisation du temps est progressive, c'est-à-dire que les nœuds du i^{eme} saut ne se synchronisent qu'après ceux du $(i - 1)^{eme}$ saut. La synchronisation est basée sur le même mécanisme décrit dans la sous-section 4.5.2 à quelques différences près. En effet, un récepteur ne traite que les messages de référence envoyés par son émetteur maître. Ce qui permet de minimiser la charge de communication

dans le réseau tout en conservant la synchronisation de tous les nœuds. De plus, un récepteur ne tient compte que des fonctions d'ajustement proposées par un seul émetteur ce qui induit à un minimum d'incohérence.

5. Conclusion

La synchronisation du temps est un service vital pour plusieurs applications des WSNs. Elle permet aux nœuds du réseau de coopérer pour réaliser la tâche prévue de l'application.

Dans ce chapitre, nous avons d'abord présenté la problématique de la synchronisation temporelle ainsi que ses concepts de bases et les raisons qui justifient son étude. Ensuite, nous avons passé en revue cinq protocoles de synchronisation du temps conçus spécialement pour s'adapter aux contraintes de conception des WSNs, telles que les limitations matérielles des micro-capteurs. L'étude détaillée et la conception du protocole *R⁴syn* seront le sujet du chapitre suivant.

Partie II :
*Conception,
Implémentation et
Tests*

Chapitre 03 :
Protocole Développé

1. Introduction

Après avoir donné un aperçu général sur le fonctionnement et les différentes propriétés de certains protocoles de synchronisation du temps, nous allons consacrer ce chapitre à l'étude et la conception du protocole R^4syn . Nous présentons tout d'abord les caractéristiques qui justifient le choix de ce protocole. Ensuite, nous allons détailler les différentes phases de fonctionnement de R^4syn . Ainsi, nous allons développer un aspect très important dans les WSNs, qui est la tolérance aux fautes des micro-capteurs.

2. Caractéristiques du Protocole R^4syn

2.1. Complexité de Communication

2.1.1. Définition

Tous les protocoles de synchronisation du temps dans les systèmes distribués sont basés sur l'échange de messages entre les nœuds. La complexité de communication est le nombre de messages nécessaire pour effectuer la synchronisation. Elle représente une métrique importante dans l'évaluation des performances d'un protocole dédié aux WSNs ; vu qu'elle a une influence directe sur l'utilisation de la bande passante et en particulier sur la consommation énergétique. Plusieurs mesures expérimentales ont montré que le module de communication (interface radio) est la partie qui consomme le plus d'énergie. Selon [SBK05], l'énergie nécessaire pour transmettre 1 bit à travers 100 m est 3 joules. Celle-ci peut être utilisée pour exécuter environ 3 millions d'instructions.

2.1.2. Complexité du Protocole R^4syn

Considérons un réseau de n nœuds qui communiquent entre eux en *Broadcast*. Dans un seul cycle, R^4syn provoque la transmission de n messages de synchronisation (chaque nœud diffuse un seul message) et fournit $(n-2)$ échantillons pour chaque paire de nœuds. Le nombre de paires peut être calculé par la combinaison C_n^2 . Le nombre total d'échantillons fournis par ce protocole dans chaque cycle est donc $\frac{n(n-1)}{2} * (n - 2)$. Par contre, RBS offre un seul échantillon pour chaque paire de nœuds dans un cycle, c'est-à-dire, $\frac{n(n-1)}{2}$ échantillons au total.

Autrement dit, R^4syn a besoin de $O(N)$ transmissions pour produire $O(N^3)$ échantillons, i.e. un seul cycle, alors que RBS nécessite $O(N^2)$ transmissions pour produire le même nombre d'échantillons, i.e. $(n - 2)$ cycles [Dje12]. Ainsi, en greffant les *timestamps* dans les messages de synchronisation, R^4syn élimine les étapes supplémentaires qui consistent à l'échange des temps de réception après chaque message de synchronisation. Cela minimise considérablement la surcharge de communication « *overhead* » par rapport à RBS.

2.2. Approche Suivie

Une bonne précision est certainement une caractéristique souhaitable d'un protocole de synchronisation du temps. Celle-ci dépend, en plus de la qualité de l'estimateur utilisé, des quatre sources d'erreur suivantes ; i) temps d'envoi (*send time*), ii) temps d'accès (*access time*), iii) temps de propagation (*propagation time*) qui peut être d'ailleurs négligeable par rapport aux autres temps, et iv) temps de réception (*receive time*), voir Section 3.3 du chapitre précédent. Dans l'approche « sender/receiver », un nœud référence envoie périodiquement un message de synchronisation qui contient sa valeur d'horloge aux autres nœuds du réseau, dits récepteurs. Par conséquent, cette approche souffre des quatre sources d'erreur citées précédemment. En revanche, l'approche « receiver /receiver » permet de minimiser le chemin critique (voir *figure 3.1*). Cela, en profitant de la propriété de diffusion (*Broadcast*) de la couche physique ; si un nœud diffuse un message de synchronisation celui-ci va être reçu par tous les nœuds voisins dans, à peu près, le même temps. La seule source d'erreur dans cette approche est l'erreur du récepteur « *receive time* » (plus le temps de propagation qui est négligeable) [EGE02]. Le protocole R^4syn est basé sur l'approche *receiver / receiver* qui minimise le chemin critique, ce qui améliore théoriquement la précision.

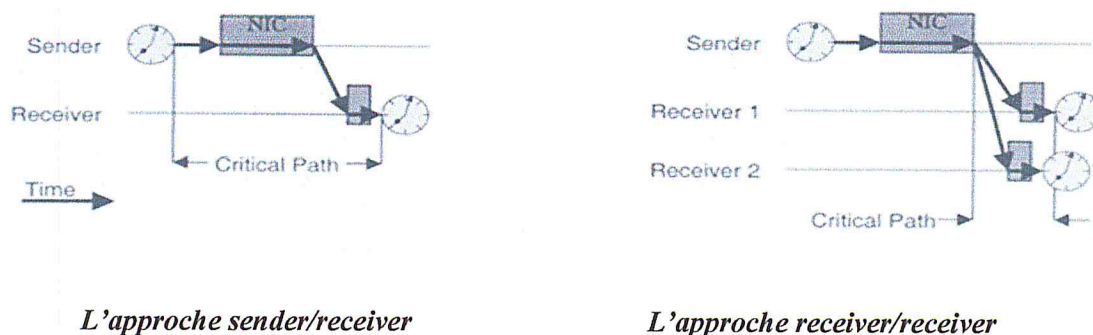


Figure 3.1. Sources d'erreurs dans les deux approches de synchronisation [EGE02].

2.3. Type de la Référence

L'approche receiver/receiver nécessite un nœud tiers (référence) responsable de l'émission des messages de synchronisation. Si cette référence est fixée au préalable, le protocole ne continuera pas à s'exécuter dans le cas de défaillance de ce nœud. En plus, certains protocoles de synchronisation du temps, RBS par exemple, ne permettent pas de synchroniser la référence avec les autres nœuds du réseau. *R⁴syn* est un protocole totalement décentralisé, il ne dépend d'aucune référence fixe. Le rôle de la référence est distribué sur tous les nœuds du réseau, et par conséquent tous ces nœuds seront synchronisés entre eux.

3. Tolérance aux Fautes

La tolérance aux fautes dans un réseau de capteurs est la capacité de ce dernier à maintenir ses fonctionnalités sans interruption et d'éviter la faille totale du système malgré la présence de quelques défaillances. Ces défaillances peuvent survenir par un manque d'énergie, dommages physiques des nœuds, ou encore à cause des interférences environnementales.

En effet, les réseaux de capteurs sont non fiables [ASC02], et les micro-capteurs sont caractérisés par leurs faibles couts de fabrication ; Ainsi, ils peuvent être déployés dans des environnements hostiles ce qui les rend vulnérables aux dégâts matériels comme les cassures.

De plus, les WSNs travaillent sur une bande de fréquences non propriétaire, cela rend leurs communications vulnérables aux problèmes d'interférences. Ces interférences conduisent à la perte des messages. Cette dernière peut être causée aussi par les collisions, vu le partage du médium de communication.

La non fiabilité doit être considérée comme une caractéristique ordinaire lors de la conception de n'importe quel protocole dédié aux réseaux de capteurs sans fil.

4. Conception

R⁴syn est un protocole totalement distribué où chaque nœud exécute le même code. Chaque nœud du réseau joue, à la fois, le rôle de la référence en diffusant les messages de synchronisation, et le rôle d'un client du mécanisme de synchronisation.

La version de R^4syn décrite ci-dessous permet de fournir une synchronisation locale entre les nœuds qui appartiennent au même domaine de diffusion. Pour cela, nous considérons un réseau de N nœuds, voisins les uns aux autres. Nous supposons que chaque nœud du réseau possède un identifiant unique et envoie les messages de synchronisation en diffusion (*broadcast*).

Le fonctionnement général du protocole R^4syn , décrit dans la section 4.4 du deuxième chapitre, peut être divisé en deux substances principales, la communication et l'estimation. Nous avons ainsi ajouté deux modules, un pour la conversion du temps et un deuxième pour tolérer les fautes des micro-capteurs.

4.1. Communication

4.1.1. Description

R^4syn fonctionne en cycles; dans chacun, tous les nœuds diffusent séquentiellement des messages de synchronisation. L'ordre d'envoi des messages va être déterminé en utilisant les identificateurs des nœuds.

Lors de la réception d'un message de synchronisation, chaque nœud doit vérifier si son identificateur est supérieur par un rang par rapport à l'émetteur, pour diffuser un nouveau message contenant son propre identificateur, après avoir attendu un certain temps "Temps_ER". Ce dernier représente l'intervalle du temps entre la réception et l'émission d'un message de synchronisation (ce temps va être fixé empiriquement dans la phase des tests). Ainsi, un nœud doit jouer le rôle de la référence une seule fois dans un cycle. Lorsque c'est le cas, il doit incrémenter sa variable locale "cycle". Cette variable contient le nombre de cycles auxquels le nœud a participé, elle est initialisée à 0.

Au début, un nœud doit être sélectionné pour lancer l'exécution du protocole. Dans la phase de communication initiale (le premier cycle), les nœuds diffusent des messages vides, qui ne contiennent que l'identificateur de l'émetteur.

A la réception d'un message de synchronisation, chaque nœud doit enregistrer le temps de réception selon son horloge locale. A partir du deuxième cycle, chaque *beacon* diffusé doit transporter $(n - 1)$ *timestamps*, ce sont les temps de réception des balises reçues dans le cycle précédent.

Ainsi, si le paquet reçu contient des *timestamps*, ces derniers doivent être récupérés et enregistrés. Ils représentent l'échantillon à base duquel les paramètres de synchronisation seront calculés par la suite.

La structure des messages échangés est la suivante :



Où ID représente l'identificateur de nœud émetteur. *Timestamps* est un tableau de taille N, qui contient les (n-1) *timestamps* reçus dans le cycle précédent.

4.1.2. Algorithme

Début :

A la réception du message de synchronisation (émetteur, Timestamps[])

 receiveTime ← H_i ; // H_i représente l'horloge locale du nœud " i "

 wait (Temps_ER)

Si ((cycle > 1) OU ((cycle = 1) ET (identificateur > émetteur)))

 Récupération des données contenues dans le vecteur *Timestamps*

Fin de Si.

Si (identificateur = ((émetteur + 1) %N))

Si(cycle = 0)

 Diffuser un message qui contient l'identificateur de l'émetteur.

Sinon

 Remplir le vecteur *Timestamps* par les n-1 temps de réception du cycle précédent.

 Diffuser un message qui contient l'identificateur de l'émetteur et le vecteur *Timestamps*.

Fin de Si.

 cycle ← cycle + 1

Fin de Si.

Fin.

FIN.

4.2. Estimation

Après l'échange d'un certain nombre de messages, chaque nœud aura suffisamment d'information pour estimer les paramètres de synchronisation (l'*offset* relatif et/ou le *skew* relatif) par rapport aux autres nœuds du réseau. Nous allons présenter les deux modèles d'estimateurs du protocole R^4syn proposés dans [Dje11], [Dje12].

4.2.1. Description

Les estimateurs sont basés sur le nombre d'échantillons utilisés dans la synchronisation " k ". Un échantillon utile pour la synchronisation est une paire des temps de réception d'un message reçu par les deux nœuds à synchroniser. Prenons, par exemple, l'échantillon $X = (temps_1, temps_2)$ et soit n_1 et n_2 les deux nœuds à synchroniser ; $temps_1$ représente le temps de réception du message m par le nœud n_1 , et $temps_2$ est le temps de réception du même message par le nœud n_2 . Dans le cas idéal (aucune perte de messages), k est comptabilisé comme suit :

$$k = cycle * (N-2) \quad (3.1)$$

Cette formule élimine les messages envoyés par les deux nœuds à synchroniser, car ils ne permettent pas de former un échantillon utile, d'où $(N-2)$.

Pour le modèle *offset-only*, l'estimateur consiste en une simple moyenne des différences des temps de réception. Par contre, le deuxième modèle permet d'estimer les deux paramètres de synchronisation en utilisant des estimateurs basés sur la méthode MLE. Les formules mathématiques de ces estimateurs sont présentées dans la section 4.4.3 du deuxième chapitre. Pour ce faire, nous allons utiliser un tableau de taille N pour calculer chaque terme de l'estimateur. Ces tableaux doivent être mis à jour à la réception de chaque message de synchronisation.

Les paramètres de synchronisation seront calculés après chaque fenêtre. Celle-ci représente un paramètre d'entrée qui comporte un certain nombre de cycles de synchronisation. La taille de cette fenêtre est une métrique qui sera fixée par des expérimentations réelles.

4.3.1.2. Algorithme

Début :

reçu \leftarrow FALSE

Au démarrage de nœud-capteur

Wait (Ω)

Si (reçu = FALSE)

Diffuser un message qui contient l'identificateur de l'émetteur.

Fin de Si.

Fin.

A la réception d'un message de synchronisation

reçu \leftarrow TRUE

Fin.

FIN.

4.3.2. Continuité de la Synchronisation

4.3.2.1. Description

La configuration du protocole R^4syn décrite antérieurement est basée sur l'échange de messages, où les nœuds diffusent à tour de rôle leurs balises de synchronisation. Si le nœud qui est censé jouer le rôle de la référence ne reçoit pas le paquet de synchronisation envoyé par son prédécesseur à cause d'une panne dans ce dernier ou si le message est perdu, l'exécution du protocole va être bloquée.

Pour remédier à ce problème, nous allons développer le mécanisme suivant ; au démarrage, chaque nœud doit déclencher un *Timer* (une minuterie) de durée Δ , qui représente le temps nécessaire pour le déroulement d'un cycle de synchronisation. A l'expiration de ce *Timer*, le nœud doit remplir le paquet à envoyer (s'il ne s'agit pas du premier cycle), puis diffuser ce paquet. Il doit ainsi incrémenter sa variable locale "cycle".

Pour ne pas violer la logique du protocole qui dicte qu'un nœud doit envoyer un seul message dans un cycle, la durée de ce *Timer* est augmentée par un certain temps ϵ . L'ajout de ce temps est pour assurer que le *Timer* ne va pas se déclencher avant la réception d'un

4.2.2. Algorithme

Début :

A la réception du message de synchronisation (émetteur, Timestamps[])

Mettre à jour les variables nécessaires pour le calcul des paramètres de synchronisation.

Si (cycle = fenêtre)

Calculer le nombre d'échantillons utilisés pour la synchronisation "k" .

Calculer *offset* et/ou *skew* relatifs en utilisant les formules (2.10) ou (2.14 et 2.15).

Fin de Si.

Fin.

FIN.

4.3. Mécanismes de Tolérance aux Fautes

Dans cette section, nous allons développer l'aspect tolérance aux fautes, à savoir les pannes des micro-capteurs et la perte de messages de synchronisation.

4.3.1. Déclenchement de la Synchronisation

4.3.1.1. Description

Comme nous l'avons dit dans la section 4.1 de ce chapitre, l'initiateur du protocole est fixé au préalable. Ce nœud peut tomber en panne avant le lancement du protocole. Dans ce cas, le protocole de synchronisation ne va pas démarrer. Pour remédier à cela, chaque nœud doit attendre un certain temps Ω , après son déploiement. S'il ne reçoit aucun message de synchronisation dans cette période, il doit lui-même initier l'exécution du protocole en diffusant un message de synchronisation. Le temps Ω est défini comme suit :

$$\Omega = (id - init) * (Temps_ER + Temps_Traitement) \quad (3.2)$$

"id" représente l'identificateur du nœud exécutant le code, et "init" est l'identificateur de l'initiateur. "Temps_Traitement" étant le temps nécessaire pour la réception, le traitement du message reçu et l'envoi d'un nouveau message de synchronisation. De même que "Temps_ER", la durée du "Temps_Traitement" va être fixée empiriquement.

message de synchronisation tardif, ou en cas d'une latence due à l'un des prédécesseurs de ce nœud.

La durée Δ est modélisée comme suit :

$$\Delta = (N - 1) * (Temps_ER + Temps_Traitement) + \epsilon$$

Cette conception garantit la continuité de la synchronisation, malgré la présence d'une ou de plusieurs défaillances dans le réseau.

4.3.2.2. Algorithme

Début :

Au démarrage de nœud-capteur

 Déclencher un *Timer* de durée Δ .

Fin.

A la réception du message de synchronisation (émetteur, Timestamps[])

 Arrêter le *Timer* courant.

Fin.

Lors de l'envoi d'un message de synchronisation

 Déclencher un *Timer* de durée Δ .

Fin.

A l'expiration du *Timer* Δ

Si (cycle = 0)

 Diffuser un message qui contient l'identificateur de l'émetteur.

Sinon

 Remplir le vecteur Timestamps par les n-1 temps de réception du cycle précédent.

 Diffuser un message qui contient l'identificateur de l'émetteur et le vecteur Timestamps.

Fin de Si

 cycle \leftarrow cycle + 1.

 Déclencher un *Timer* de durée Δ .

```

|
|   Si ( cycle = fenêtre )
|       Calculer le nombre d'échantillons utilisés pour la synchronisation "k " .
|       Calculer offset relatif et / ou skew relatif.
|   Fin de Si.
|
|   Fin
|
FIN.
```

4.3.3. Granularité de la Synchronisation

4.3.3.1. Description

La perte de messages de synchronisation ne cause pas seulement l'arrêt de l'exécution du protocole, mais elle affecte également la granularité de la synchronisation. En effet, la comptabilisation d'un message non reçu va altérer le calcul du *skew* et/ou l'*offset* relatifs, ce qui va conduire à une synchronisation erronée. Pour pallier à ce problème, nous devons compter le nombre d'échantillons effectivement utilisés.

Nous allons donc comptabiliser le nombre de messages de synchronisation reçus par chaque nœud dans le réseau. Pour ce faire, deux tableaux de taille "N", "messagesRecus" et "messagesPerdus" seront utilisés. A la réception d'un message de synchronisation, chaque case du tableau "messagesRecus" doit être incrémentée, sauf celle de l'émetteur (car le message envoyé par l'émetteur ne forme pas un échantillon pour ce dernier). Ainsi, le nombre de messages perdus chez chaque nœud peut être récupéré à partir du paquet reçu, si celui-ci contient des cases vides. Par conséquent, le tableau Timestamps envoyé dans le paquet doit être réinitialisé à zéro à chaque fois. Après l'estimation des paramètres de synchronisation, les tableaux utilisés doivent être, ainsi, réinitialisés à zéro.

Le nombre d'échantillons effectivement utilisés pour synchroniser le nœud " i " avec le nœud " j " est représenté par la formule suivante :

$$k = \text{cycle} * (N-2) - ((\text{cycle} * (N-2) - \text{messagesRecus}[j]) + \text{messagesPerdus}[j])$$

4.3.3.2. Algorithme

Début :

```
A la réception du message de synchronisation (émetteur, Timestamps[])
  Pour i ← 1 à N faire :
    Si (i ≠ émetteur)
      messagesRecus[ i ] ← messagesRecus[ i ] + 1 ;
    Fin de si.
  Fait.
  Si ((cycle > 1) OU (cycle = 1) ET (identificateur > émetteur))
    Pour i ← 1 à N faire :
      Si ( Timestamps[ i ] = 0)
        messagesPerdus[ référence ] ← messagesPerdus[ référence ] + 1 ;
      Fin de si.
    Fait.
  Fin de Si.
Fin.
FIN.
```

4.4. Conversion

Après le calcul des paramètres de synchronisation, et pour permettre à chaque micro-capteur d'estimer la valeur d'horloge $C_2(t)$ générée dans un autre nœud du réseau en utilisant sa propre horloge $C_1(t)$, nous allons utiliser l'équation d'estimation du temps proposée dans [SiY04], comme suit :

Début :

```
Au besoin de conversion du temps
  Si (modèle = offset-only)
     $C_2(t) = C_1(t) + b_{1,2}$ 
  Fin de Si.
```

```

|
| Si (modèle = skew/offset)
|   |
|   |  $C_2(t) = a_{1,2} * C_1(t) + b_{1,2}$ 
|   |
|   | Fin de Si.
|
| Fin.
|
FIN.
```

Où $a_{1,2}$, $b_{1,2}$ sont le *skew* relatif et l'*offset* relatif, respectivement, entre les deux nœuds à synchroniser.

5. Conclusion

Le protocole R^4syn représente une bonne solution pour la synchronisation du temps dans les WSNs, vu ses caractéristiques, à savoir sa légère complexité de communication, la décentralisation de la référence, et l'approche receiver/receiver dont il se base. Au cours de ce chapitre, nous avons développé une version tolérante aux fautes de ce protocole. Les mécanismes de tolérance aux fautes développés garantissent la continuité et la granularité de la synchronisation malgré la présence de quelques défaillances dans le réseau. Ceci sera montré dans le chapitre suivant, qui présentera les détails de notre implémentation de R^4syn ainsi que son évaluation sur une plateforme expérimentale réelle de réseaux de capteurs sans fil.

Chapitre 04 :
Implémentation
et
Tests Réels

1. Introduction

Dans ce chapitre, nous allons présenter l'environnement matériel et logiciel utilisé pour la mise en œuvre et les tests de $R^4 Syn$. Ensuite, nous allons décrire notre implémentation sur une plateforme réelle des micro-capteurs de types MICAz. Par la suite, nous présenterons les résultats obtenus suite à plusieurs scénarios de tests. Enfin, nous synthétiserons le tout par une conclusion.

2. Plateforme d'Implémentation

2.1. Plateforme Matérielle

2.1.1. Micro-capteur

Durant nos implémentations et expérimentations, nous avons utilisé la plateforme MICAz (Voir *figure 4.1* et *figure 4.2*) produite par Crossbow Technology. Elle comporte un microcontrôleur ATMEGA 128L sur 8-Bit avec 8MHz de fréquence, 4KB de RAM et 128 KB de mémoire FLASH. MICAz inclut une radio sans fil de type CHIPCON CC2420, qui fonctionne dans la bande de fréquence radio 2.4GHz avec un débit de 250 Kbits/s, et un intervalle d'amplification qui varie entre 1 et 30. Cette radio fonctionne sous la norme Zigbee IEEE 802.15.4, qui représente la norme la plus utilisée dans le domaine des WSNs.

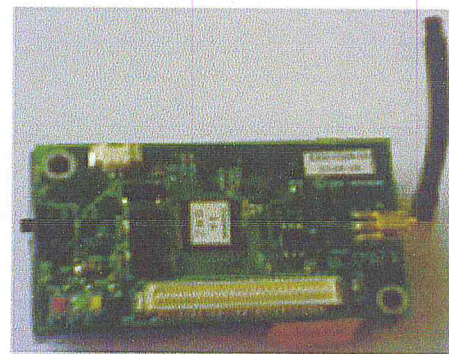
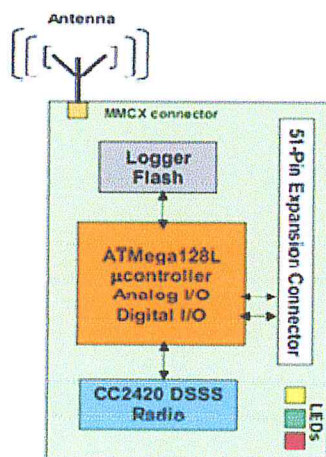


Figure 4.1. Architecture de MICAz [W01]

Figure 4.2. micro-capteur "MICAz"

2.1.2. Programmateur « Programming board »

Nous avons utilisé le programmeur MIB « Mote Interface Board » (voir *figure 4.3*), un dispositif matériel qui permet de connecter le mote MICAz à l'ordinateur à l'aide d'un câble

USB ou bien d'un port série. Le rôle du programmeur est la construction d'une station de base, qui permet d'acheminer les données collectées par les nœuds-capteurs vers un ordinateur central (un serveur ou un simple ordinateur). La station de base sert aussi à injecter les programmes dans les micro-capteurs afin de réaliser les tâches définies dans le code source. Ainsi, elle permet d'afficher sur écran les données envoyées par le micro-capteur, en temps réel.

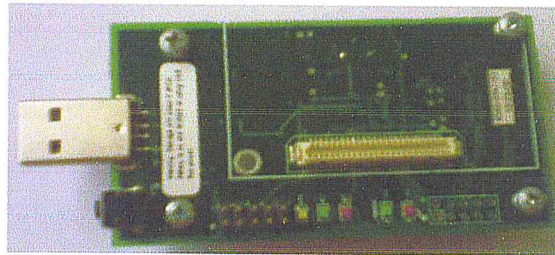


Figure 4.3. Programming Board MIB520

2.1.3. Carte d'Extension «Data Acquisition Board »

De même, nous avons utilisé la carte d'extension MDA300CA, qui permet de manipuler les pins du micro-capteur. Cette carte a été développée par le centre de la détection de réseaux embarqués (CENS) « Center for Embedded Networking Sensing », de l'université UCLA.

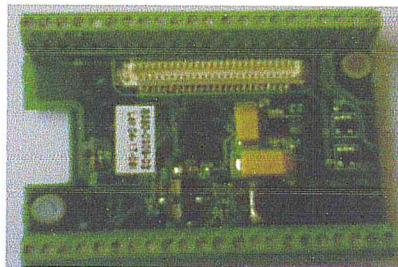


Figure 4.4. Data Acquisition Board MDA300CA

2.1.4. Arduino UNO

Dans la phase de tests, nous avons utilisé aussi la carte Arduino UNO (Voir *figure 4.5*), produite par la société informatique italienne Olivetti. Cette carte comporte un microcontrôleur ATMEGA 328P sur 8-Bit, avec 16MHz de fréquence, 2KB de RAM, 1KB de EEPROM et 32 KB de mémoire FLASH [W14].

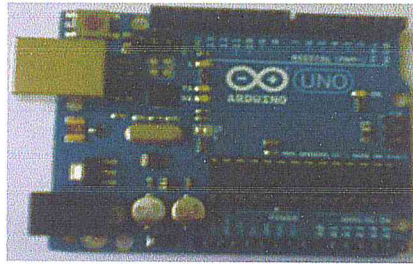


Figure 4.5. Arduino UNO

2.2. Plateforme Logicielle

2.2.1. Système d'Exploitation TinyOS

TinyOS « Tiny Operating System » [HSW00], [W05, W06, W07], est un système d'exploitation open-source développé à l'université de Berkeley en 2001. Sa conception a été entièrement dédiée pour s'adapter aux contraintes matérielles des micro-capteurs. TinyOS respecte une architecture basée sur une association de composants. Chaque composant correspond à un élément matériel : LED, *Timer*, etc., qui peut être réutilisé dans différentes applications. Quelques caractéristiques de ce système sont décrites par les points suivants (voir Annexe A pour plus de détails) :

- **Modèle d'exécution :** TinyOS exécute un seul programme constitué d'une sélection de composants systèmes et de composants développés spécifiquement pour l'application. Son modèle d'exécution est basé sur deux types de processus, les tâches et les pilotes d'interruption qui sont exécutés en réponse à une interruption matérielle. TinyOS ne dispose pas de mécanisme de préemption entre les tâches. Néanmoins, les pilotes d'interruption ont la capacité de préempter les tâches et les autres pilotes d'interruption [LeG09].
- **Consommation énergétique :** Pour répondre à la contrainte énergétique, TinyOS propose un fonctionnement événementiel ; c'est-à-dire qu'il ne devient actif qu'à l'apparition de certains événements, par exemple l'arrivée d'un message radio. Ainsi, lorsqu'aucune tâche n'est active, TinyOS se met automatiquement en veille.
- **Gestion de la mémoire :** TinyOS a une empreinte mémoire très faible puisqu' il ne prend que 300 à 400 octets de mémoire nécessaire à son installation, dans le cadre d'une distribution minimale [W08]. En plus de cela, il est nécessaire d'avoir 4Ko de mémoire libre qui se répartissent entre la pile et le stockage des variables globales.

2.2.2. Langage de Programmation NesC

Les applications qui s'appuient sur TinyOS et le système lui-même sont écrits en NesC [GLC03], [TLG07], [W09], un langage de programmation orienté composant syntaxiquement proche du langage C, conçu pour incarner les concepts et le modèle d'exécution de TinyOS. Ce langage est construit autour de deux éléments de base, qui sont les interfaces et les composants. Un composant peut fournir ou utiliser plusieurs interfaces différentes ou plusieurs instances d'une même interface. Les interfaces sont le seul point d'accès au composant. Elles sont bidirectionnelles vues qu'elles définissent un ensemble de fonctions qui peuvent être soit des commandes ou des événements. La commande doit être implémentée par le composant qui fournit l'interface tandis que l'évènement est implémenté par le composant utilisateur de l'interface. Deux types de composants existent, le module et la configuration. Les modules contiennent le code de l'application et les configurations servent à assembler les composants en connectant les interfaces utilisées par certains composants à celles fournies par d'autres [LeG09]. Plus de détails sur NesC, sont disponibles en Annexe B.

2.2.3. Simulateur Avrora

Avrora [TBL05], [W12] est un émulateur des WSNs, écrit en java. Il réalise une simulation complexe du microcontrôleur, des périphériques et de la communication radio. Ce simulateur offre plusieurs outils, par exemple l'analyse énergétique qui permet d'étudier la consommation d'énergie au niveau des nœuds-capteurs, ainsi que la durée de vie possible de la batterie utilisée. Cependant, Avrora n'implémente pas certains composants principaux comme un gestionnaire d'horloge. Ceci le rend incapable de modéliser une dérive d'horloge entre les nœuds du réseau. A noter que ce simulateur est spécifique aux plateformes MICA2 et MICAz. Plus de détails sur Avrora sont disponibles dans l'Annexe C de ce document.

3. Réalisation de R^4 Syn

3.1. Horloge Utilisée

MICAz est cadencé par deux types d'horloge, horloge interne (de microcontrôleur) et horloge externe (à cristal).

3.1.1. Horloge de Microcontrôleur

En mode d'horloge interne, l'horloge est basée sur un oscillateur de 8MHz de fréquence. Elle est caractérisée par une très bonne résolution, offrant une granularité de l'ordre de la microseconde, mais elle est moins stable à cause de sa déviation continue qui peut atteindre 100 microsecondes de décalage par seconde [FMT10].

Il existe trois *Timers* (minuterie) qui peuvent utiliser cette horloge, *Timer1* (16-bit), *Timer2* (8-bit) et *Timer3* (16-bit), où chaque *Timer* est manipulé par son propre composant qui est fourni par TinyOS.

3.1.2. Horloge Externe

En mode d'horloge externe, l'horloge est basée sur un oscillateur de 32KHz de fréquence [STG07]. Elle est caractérisée par une résolution moins bonne, limitée à l'ordre de quelques millisecondes. Cependant, elle est plus stable par rapport à celle de microcontrôleur, i.e. elle ne dérive pas beaucoup.

Il existe un seul *Timer* qui peut utiliser cette horloge, c'est le *Timer0* (8-bit). De même, le *Timer0* est utilisé par son propre composant délivré par TinyOS.

Dans notre implémentation, nous avons choisi la lecture de l'horloge de microcontrôleur lors de la réception d'un message de synchronisation et pour l'estimation de la valeur d'horloge générée dans un autre nœud. Notre choix est justifié par la bonne résolution de cette dernière par rapport à l'horloge externe. Concernant les *Timers* utilisés pour tolérer les fautes des micro-capteurs, nous avons choisi le composant qui est basé sur l'horloge à cristal qui est plus stable, et qui continue à progresser lorsque le micro-capteur passe en état de veille. Ceci assure que les *Timers* vont se déclencher dans les moments appropriés.

3.2. Implémentation

Notre implémentation de $R^4 Syn$ a été basée sur neuf composants qui permettent d'utiliser et de fournir un ensemble d'interfaces. Sachant que, nous avons implémenté deux versions tolérantes aux fautes de $R^4 Syn$, *Offset-only* et *Skew/Offset*, qui se différencient entre eux dans la partie estimation. L'architecture de notre application est présentée dans la *figure 4.6*.

3.2.1. Offset-only :

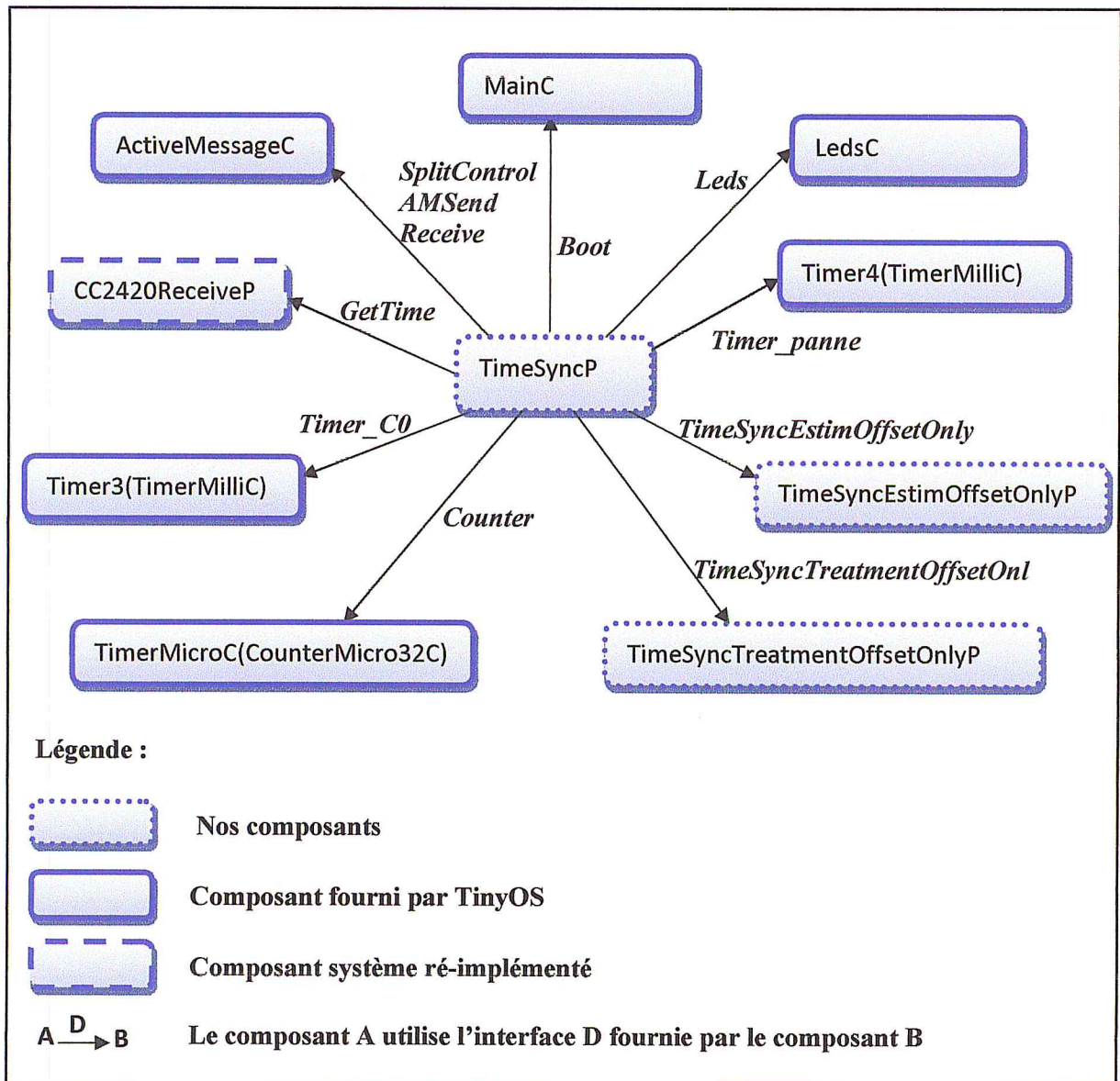


Figure 4.6. Architecture de notre implémentation

Dans ce qui suit, une description des différents modules sera présentée:

3.2.1.1. TimeSyncP

TimeSyncP représente le module principal qui permet d'initier le protocole lors du démarrage des différents nœuds du réseau. Il ne fournit aucune interface, tandis qu'il utilise les interfaces qui sont définies dans le schéma précédent. Pour assurer son fonctionnement, TimeSyncP fait appel à d'autres composants qui sont :

- **TimeSyncTreatmentOffsetOnlyP** et **TimeSyncEstimOffsetOnlyP** : qui offrent les deux interfaces **TimeSyncTreatmentOffsetOnly** et **TimeSyncEstimOffsetOnly**, respectivement.
- **CC2420ReceiveP** : version ré-implémentée du composant fourni par TinyOS.
- **MainC** : il permet de démarrer le micro-capteur à l'aide de son interface « Boot ».
- **LedsC** : il est utilisé pour gérer les Leds des micro-capteurs via l'interface led.
- **TimerMilliC** : il fournit l'interface *Timer* qui nous permet de déclencher un ensemble de traitements à l'expiration d'un intervalle du temps bien défini.
- **ActiveMessageC** : permet de gérer l'envoi et la réception des messages en utilisant ses deux interfaces **AMSend** et **Receive**, au niveau plus haut que celui de **CC2420ReceiveP**. En plus, elle sert à gérer la radio, l'allumer et l'éteindre, en utilisant l'interface **SplitControl**.
- **CounterMicro32C** : il fournit l'interface **Counter** qui nous permet de lire la valeur d'horloge du microcontrôleur à l'ordre de microseconde.

3.2.1.2. **TimeSyncTreatmentOffsetOnlyP**

Il représente le module responsable des différents traitements et de la mise à jour des données lors de la réception de nouveaux messages de synchronisation.

3.2.1.3. **TimeSyncEstimOffsetOnlyP**

C'est le module qui implémente les différentes méthodes d'estimation et de conversion du temps. Ce module utilise l'interface **Counter** fourni par TinyOS afin de lire sa valeur d'horloge lors de l'estimation.

3.2.1.4. **CC2420ReceiveP**

Ce module est fourni par TinyOS, il assure la gestion de l'interface radio CC2420 des micro-capteurs (y compris MICAz) dans un niveau bas. Nous avons ré-implémenté ce module en ajoutant la commande **GetTime**. Cette commande est chargée de la réalisation de l'estampillage « *Timestamping* » des messages de synchronisation reçus au niveau de la couche MAC, afin de minimiser les délais aléatoires du temps de réception des messages.

3.2.2. Skew/Offset

L'architecture de l'implémentation de la deuxième variante de $R^4 Syn$ est identique à celle de la première variante, la seule différence se trouve au niveau de l'implémentation des deux composants TimeSyncEstimSkewOffsetP et TimeSyncTreatmentSkewOffsetP. Ces deux derniers modules permettent d'implémenter les différentes méthodes qui calculent les deux paramètres de synchronisation, *Skew* et *Offset*, et la méthode d'estimation de la valeur d'horloge ainsi que les différents traitements nécessaires.

Durant l'implémentation de ce modèle, nous avons confronté un problème majeur dans la mise en œuvre des estimateurs, Eq.2.14 et Eq.2.15. Ces deux équations permettent de calculer le *skew* et l'*offset* respectivement. Elles comprennent des multiplications de très grands nombres qui représentent des valeurs d'horloges en microsecondes. Ces multiplications causent un débordement de type après quelques secondes de lancement du protocole. Le débordement de type nous empêche d'acquérir un nombre élevé d'échantillons afin de calculer les deux paramètres de synchronisation avec exactitude, et donc l'obtention d'une précision de l'ordre de microseconde ne serait pas possible. Pour remédier à ce problème, les formules d'estimation du *skew* et de l'*offset* ont été simplifiées.

Concernant l'Eq.2.14, il est remarquable que le numérateur et le dénominateur représente une différence de deux termes importants, mais les différences finales représentent deux petits nombres qui ne dépassent pas 32 bit, et qui ne posent pas le problème de débordement lors de leurs division. En effet, le problème est causé par le calcul des termes qui contiennent l'opération de produit. Pour le numérateur, le premier terme peut être considéré comme la somme du produit $(u_i v_j)$, composé de k^2 termes élémentaires, et le second représente la somme des produits $(u_j v_j)$ qui se répète k fois.

Par conséquent, au lieu de calculer les deux termes séparément puis les soustraire, il est plus pratique de retrancher chaque terme élémentaire à partir de son terme équivalent de la deuxième partie, par exemple soustraire $(u_i v_j)$ à partir de $(u_j v_j)$. Le même principe est appliqué au dénominateur, le résultat de la simplification est illustré par l'expression suivante :

$$\alpha_{mle} = \frac{\sum_{i=1}^k \sum_{j=1}^k u_i v_j - v_j u_j}{\sum_{i=1}^k \sum_{j=1}^k v_i v_j - v_j v_j} \quad (4.1)$$

La différence des grands produits est transformée maintenant en une somme de différence de petits produits, qui ne causent pas de problème lors de sa mise en œuvre. Pour le calcul de l'*offset*, il suffit de remplacer la nouvelle α_{mle} dans l'équation Eq.2.15, nous trouvons,

$$\beta_{mle} = \frac{1}{k} \left(\sum_{i=1}^k u_i - \alpha_{mle} * \sum_{i=1}^k v_i \right) \quad (4.2)$$

En plus du problème de débordement, le calcul du *skew* nécessite l'utilisation des nombres flottants (la valeur du *skew* est autour de 1). Cependant, les unités arithmétiques des microcontrôleurs utilisés par les nœuds-capteurs de type MICAz sont sur 8 bits, et donc ils ne permettent pas de calculer la virgule flottante. La distribution 2.x de TinyOS implémente une technique logicielle qui résout ce problème en offrant les deux types «*float*» et «*double*». Bien que le *double* est sur 64 bits, ce type est divisé en deux sous parties, la mantisse et l'exposant, chacune sur 32 bits. Cette taille ne suffit pas pour stocker les termes de calcul du *skew* et de l'*offset*, et elle re-provoque le problème du débordement même en utilisant les formules simplifiées. Pour cela, nous avons calculé la flottante en utilisant une arithmétique du calcul qui consiste à la multiplication du dominateur par une puissance de 10 (10,100, etc.), dans chaque opération de division. A la fin, la valeur d'horloge convertie doit être divisée sur la même puissance de 10 utilisée dans le calcul.

4. Tests Réels de $R^4 Syn$

Pour pouvoir évaluer les performances du protocole $R^4 Syn$, nous avons effectué une série de tests réels sur un réseau composé de trois nœuds-capteurs. Dans ce qui suit, nous présentons les résultats obtenus.

4.1. Paramètres du Protocole

Les résultats qui seront présentés par la suite sont obtenus en exécutant la configuration décrite dans le tableau suivant :

Modèle Paramètre	Offset-only	Skew/Offset
Fenêtre	1 cycle	2 cycles
Temps_ER	100 millisecondes	2 secondes

Tableau 4.1. Paramètres du protocole

Ces paramètres ont été fixés après l'exécution d'une série importante de tests. Pour le modèle *Offset-only*, nous avons remarqué que la précision s'abaisse avec la diminution de la durée nécessaire pour le déroulement d'un cycle de synchronisation. Cela est justifié par la vétusté des échantillons utilisés pour la synchronisation avec le temps, sachant que ce modèle ne modélise pas le *drift*. Par conséquent, les paramètres de ce modèle ont été fixés à un cycle par fenêtre et 100 millisecondes entre la réception et l'émission d'un message de synchronisation. D'autre part, le modèle *skew/offset* nécessite un peu plus de temps et un nombre d'échantillons plus élevé pour pouvoir estimer le *skew* et l'*offset* avec exactitude. De plus, nous avons fixé la valeur de la puissance de 10 utilisée pour garder la précision du calcul à cinq chiffres après la virgule. Nous avons remarqué que la précision de synchronisation s'améliore avec l'augmentation de la précision du calcul. Néanmoins, On ne peut pas aller au-delà de cinq chiffres après la virgule à cause du débordement.

4.2. *Offset-Only*

Dans cette section, nous allons évaluer notre implémentation (premier modèle) de $R^4 Syn$ selon deux métriques qui sont la précision et la stabilité. La précision ou encore l'erreur de la synchronisation représente la différence entre la valeur d'horloge réelle et celle estimée. La stabilité est la mesure de la variation de cette précision par rapport au temps, i.e. durée de vie de la synchronisation.

4.2.1. Précision

Pour une bonne évaluation, nous avons conçu trois méthodes de tests où chaque nouvelle méthode a été proposée dans le but de l'amélioration de la méthode précédente, jusqu'à l'obtention de résultats fiables.

4.2.1.1. Première Méthode

Au début, nous avons proposé cette méthode de tests pour avoir une idée globale sur le fonctionnement général du protocole.

Nous avons pensé à sélectionner un nœud qui permet d'envoyer un message d'estimation par *broadcast* après avoir terminé la phase de l'enregistrement des échantillons et du calcul des paramètres de synchronisation. L'idée était comme suit : à la fin de la fenêtre de synchronisation du nœud qui possède l'identifiant le plus grand ; ce dernier doit diffuser le message AM_ESTIM_MSG, que nous avons créé. Le choix de ce nœud était dans le but de garantir que tous les nœuds du réseau aient déjà calculé les paramètres de synchronisation.

A la réception de ce message nous avons choisi deux nœuds de manière aléatoire, par exemple le nœud qui possède l'ID 2 et celui qui possède l'ID 3. Le premier doit enregistrer et afficher sa valeur d'horloge et le deuxième doit estimer et afficher la valeur d'horloge du premier nœud. Une soustraction des deux valeurs est effectuée par la suite pour trouver la précision de synchronisation. Pour l'affichage des résultats obtenus par les deux nœuds, nous avons créé deux stations de base (MICAz + programming board).

Après la réalisation des tests de la première méthode, nous avons eu des résultats non satisfaisants vu que la précision du protocole a dépassée 100 microsecondes dans plusieurs séquences comme montre la *figure 4.7*, où l'axe des X représente les séquences et l'axe des Y représente la précision de synchronisation en microsecondes.

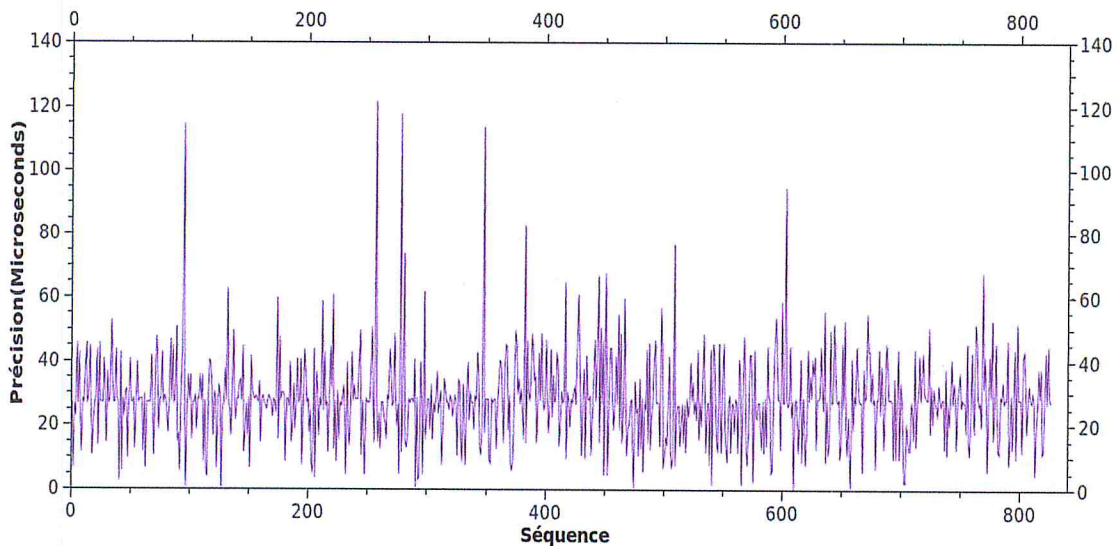


Figure 4.7. Précision du protocole dans la première expérience

Pour automatiser le calcul de la précision, nous avons réalisé un programme en langage *Python* qui permet de soustraire les deux valeurs affichées par les deux nœuds dans des fichiers LOG en utilisant la redirection d'affichage.

- **Inconvénient :** L'inconvénient majeur de cette méthode est l'indéterminisme des délais de réception de message de l'estimation. Cet indéterminisme peut s'expliquer par le scénario suivant : un nœud parmi les deux nœuds concernés par l'estimation reçoit le message `AM_ESTIM_MSG` avant le deuxième nœud, ce qui implique que la valeur d'horloge estimée par le nœud 3 ne reflète pas le temps exact de la lecture d'horloge du nœud 2. Ceci nous a poussés à chercher une autre méthode de test.

4.2.1.2. Deuxième Méthode

Dans cette méthode, nous avons utilisé la carte Arduino ainsi que la carte d'extension. Cette fois-ci nous avons étendu le scénario précédent par le câblage des deux nœuds avec la carte Arduino comme le montre la *figure 4.8*.

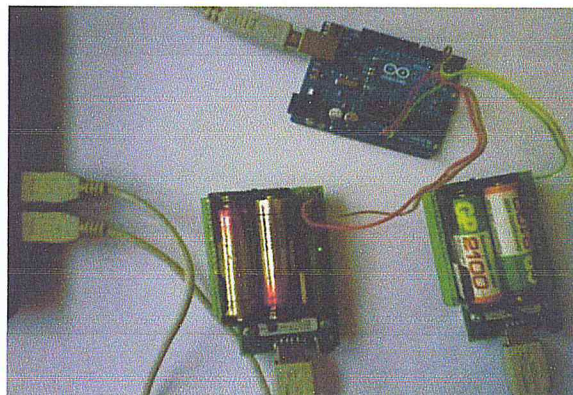


Figure 4.8. Deuxième méthode

La carte d'extension (*figure 4.9*) a été utilisée pour pouvoir manipuler les pins du micro-capteur. La figure suivante représente les différents pins de la carte d'extension. Dans cette deuxième expérience nous avons utilisé les pins indiqués par les deux flèches jaune et vert.

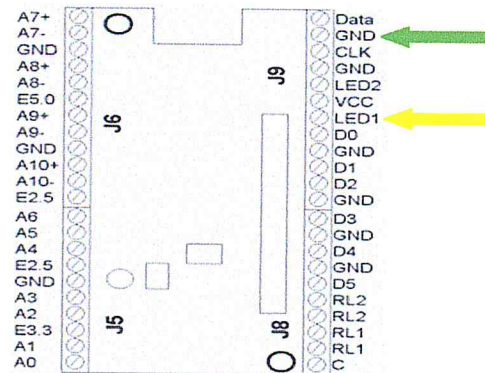


Figure 4.9. Pins de la carte d'extension

Considérons la *figure 4.8*, nous prenons le micro-capteur qui porte des batteries vert et blanc pour expliquer le câblage ; le câble jaune est associé au pin qui manipule la « led rouge » de MICAz afin de permettre au nœud 3 d'envoyer un signal juste après l'estimation. De l'autre côté d'Arduino, le câble est associé à l'interruption zéro (pin 2) pour permettre à Arduino de lire sa valeur d'horloge lors de la réception du signal. Le câble vert est associé au pin GND du micro-capteur afin de lui permettre d'afficher sur écran et idem pour le côté d'Arduino.

Après avoir répondu au message d'estimation par les deux nœuds comme expliqué dans la première méthode, chacun des deux micro-capteurs doit envoyer un signal à Arduino. Ceci est faisable en allumant la led rouge du micro-capteur. D'autre part, nous avons programmé Arduino de façon qu'il enregistre sa valeur d'horloge à chaque réception du signal. Comme la carte Arduino utilise une seule horloge lors de la lecture, la différence entre l'estampillage des deux signaux reçus représente la durée écoulée entre la lecture de l'horloge par le nœud 2 et l'estimation de cette valeur par le nœud 3. De ce fait, nous avons retranché cette différence donnée par Arduino à partir de la précision, comme le représente la formule suivante :

$$\text{Précision} = (t1 - t2) - (T1 - T2)$$

Où « Précision » représente la précision de la synchronisation, « t1 » et « t2 » représentent les valeurs d'horloges générées dans le nœud 2 et estimées par le nœud 3, respectivement. « T1 » étant la valeur d'horloge d'Arduino lors de la réception du signal envoyé par le nœud 2 qui a été envoyé juste après la lecture de l'horloge. De même, « T2 » représente la valeur d'horloge d'Arduino lors de la réception du signal envoyé par le nœud 3 (le signal a été envoyé juste après l'estimation). Donc, (T1 - T2) représente la variabilité des délais de

réception entre le nœud 2 et le nœud 3. Les résultats obtenus par cette méthode sont présentés dans la *figure 4.10*.

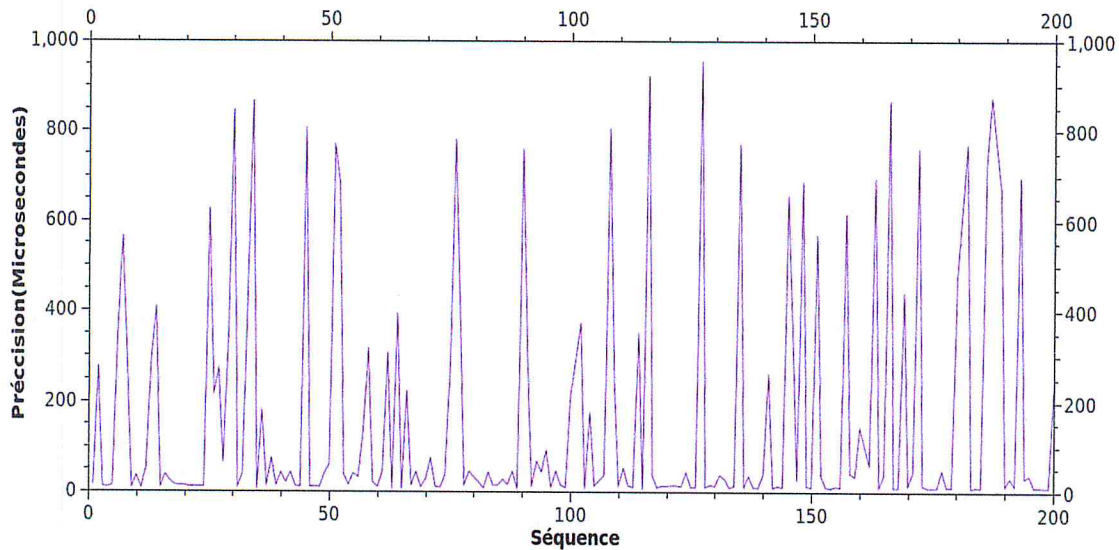


Figure 4.10. Précision du protocole dans la deuxième expérience

- **Inconvénient :** dans cette méthode, nous avons diminué la durée de latence des deux micro-capteurs dans un certain nombre de séquences. Mais parfois, il existe toujours une latence importante durant la réponse d'Arduino sur les deux signaux reçus (la valeur de $T1 - T2$). Cette latence est justifiée par la séquentialisation des deux signaux reçu dans un seul instant ; en d'autres termes si Arduino reçoit les deux signaux en même temps, il va les traiter l'un après l'autre ce qui implique un décalage important entre les deux temps de réception.

4.2.1.3. Troisième Méthode

Cette fois-ci, nous avons proposé le remplacement du message AM_ESTIM_MSG par un signal envoyé par Arduino, la réponse à ce signal se réalise au niveau le plus bas du matériel. De ce fait, nous étions obligées de manipuler les pins du micro-capteur pour permettre aux nœuds 2 et 3, respectivement, de lire et d'estimer les valeurs d'horloges lors d'une interruption matérielle. Nous avons câblé Arduino avec les deux nœuds, comme montre la *figure 4.11*. Avec cette méthode d'expérience, nous avons garanti que le signal de l'estimation va être reçu par les deux nœuds en même temps. En plus, la réponse à cet évènement va être exécutée sur le coup, en interrompant les instructions en cours d'exécution. Ainsi, les micro-capteurs vont répondre à ce signal au niveau matériel au lieu

que ça soit au niveau de la couche application, donc nous avons éliminé la variabilité des délais de réception entre les deux nœuds. Par conséquent, nous avons pu garantir que le nœud 2 et le nœud 3 vont réaliser en même temps, la lecture de la valeur d'horloge locale et l'estimation.

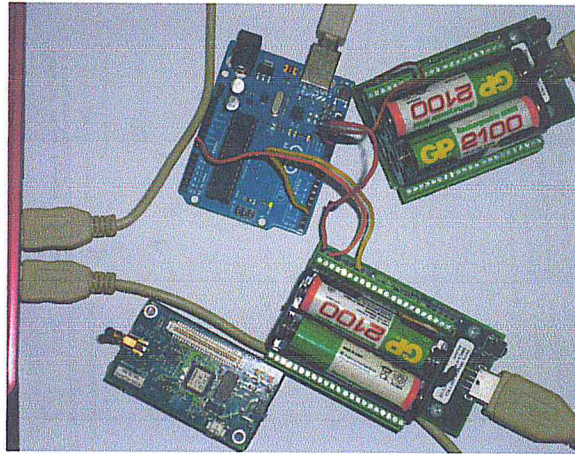


Figure 4.11. Troisième méthode

Les résultats obtenus par cette méthode sont représentés dans la *figure 4.12*. La précision de la synchronisation était de l'ordre de quelques microsecondes (entre 1 et 7) pendant toute la durée de test, avec une moyenne de 3.5 microsecondes.

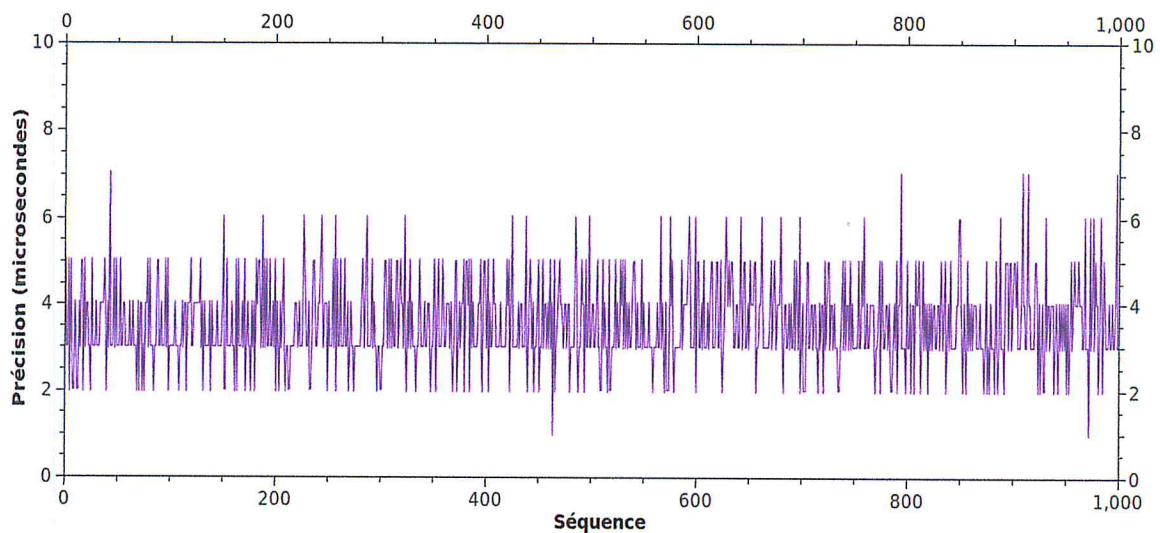


Figure 4.12. Précision du protocole dans la troisième expérience

4.2.2. Stabilité

Pour examiner la durée de vie de la synchronisation du modèle *offset-only*, nous avons arrêté l'exécution du protocole après la première fenêtre de synchronisation afin de garantir que le nœud 3 va toujours utiliser les anciens paramètres de synchronisation ; puis, nous avons mesuré la précision après chaque seconde. La *figure 4.13* montre une dégradation très rapide de la précision. Après une minute, la précision a dépassé 500 microsecondes, sachant que la précision initiale, qui est mesurée juste après le calcul des paramètres de synchronisation, était 1 microseconde. L'erreur de la synchronisation a atteint environ 10 millisecondes après 15 minutes de test. Cette rapidité de dégradation est liée à la nature de ce modèle qui permet d'estimer l'*offset* seulement, alors que la fréquence des oscillateurs réels change au fil du temps et peut atteindre 100 microsecondes de décalage par seconde.

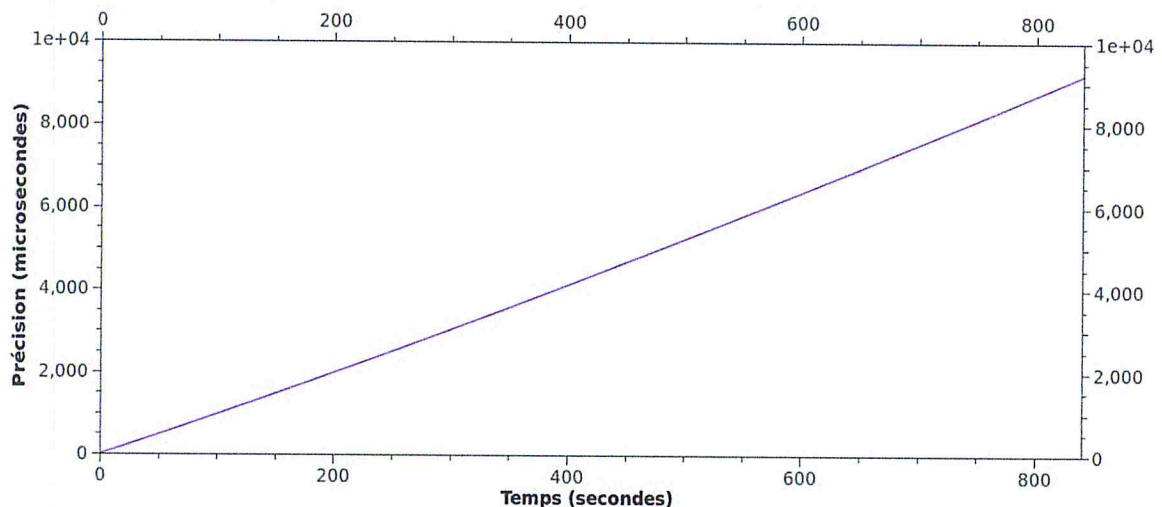


Figure 4.13. Stabilité du modèle *Offset-only*

4.3. Skew/Offset

Idem pour le premier modèle, nous allons évaluer les performances du modèle *skew/offset* par rapport aux deux critères, précision et stabilité.

4.3.1. Précision

Pour mesurer la précision de ce modèle, nous avons utilisé la troisième méthode de test conçue dans la sous-section 4.2.1.3 de ce chapitre. Comme montré dans la *figure 4.14*, l'erreur de la synchronisation varie entre 0 et 6 microsecondes, avec la plupart des valeurs comprises entre 2 et 4. La moyenne de la précision est 3.02 microsecondes.

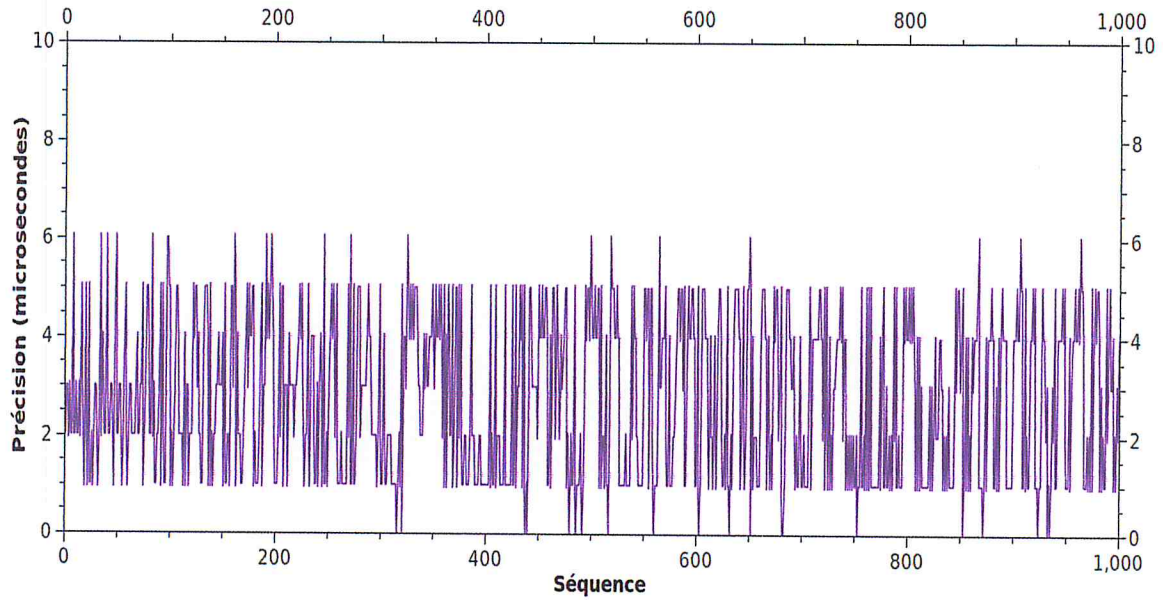


Figure 4.14. Précision du modèle skew/offset

Les valeurs de précision obtenues sont légèrement variables (changeant d'un test à un autre). C'est pour cette raison que nous avons ré-exécuté les tests de précision 13 fois. La *figure 4.15* représente la moyenne de ces tests. Cette moyenne est comprise entre 7.07 et 13.46 microsecondes. Les barres d'erreurs sont calculées avec un intervalle de confiance égal à 95%.

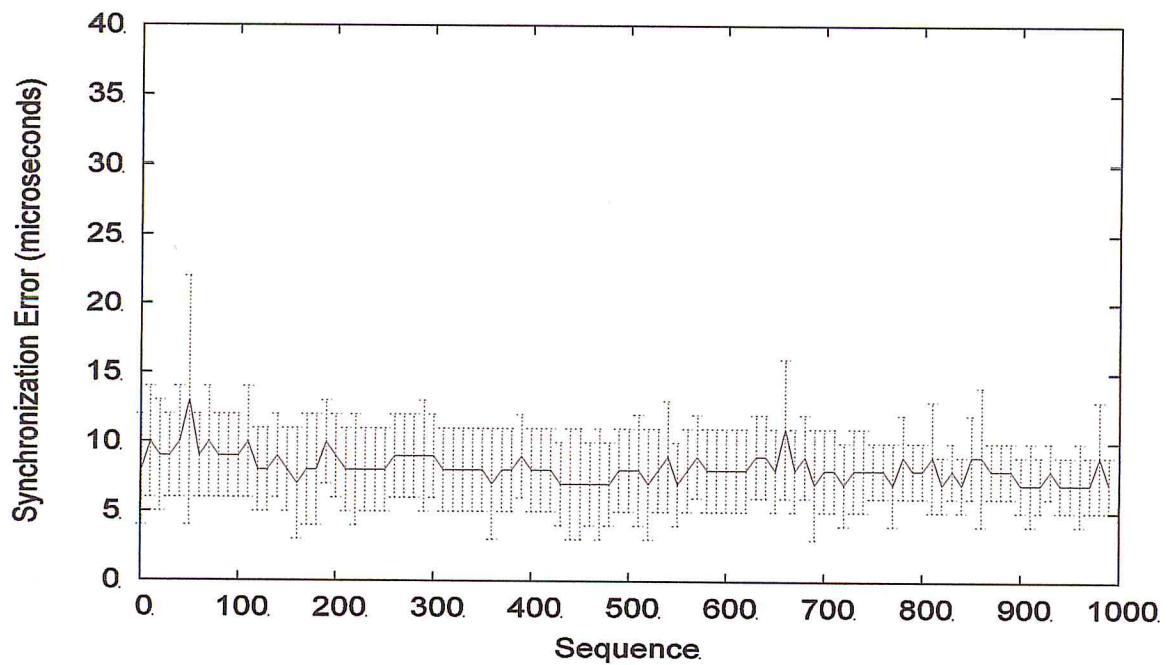


Figure 4.15. Moyenne de la précision du modèle skew/offset

4.3.2. Stabilité

La même expérience décrite dans la section 4.2.2 a été ré-exécutée, cette fois-ci en utilisant le modèle *Skew/Offset*. La *figure 4.16* représente l'erreur de synchronisation en fonction du temps. Nous remarquons une grande stabilité de ce modèle grâce à la modélisation du *drift*. La précision a été inférieure à 10 microsecondes pendant plus de 3 minutes, et elle n'a pas dépassé 330 microsecondes pendant les 15 minutes de test.

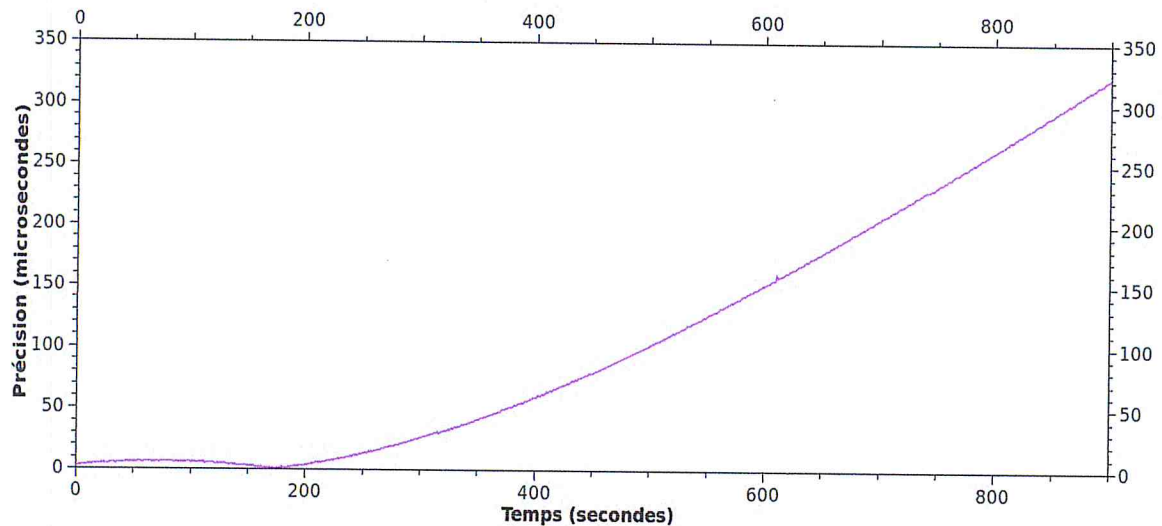


Figure 4.16. Stabilité du modèle *Skew /offset*

4.4. Comparaison entre les Deux Modèles

Les deux modèles, *offset-only* et *skew/offset*, ont donné des valeurs de précision similaires. Cependant, le modèle *skew/offset* a montré une grande stabilité par rapport au premier modèle (voir *figure 4.17*), qui nécessite l'émission continue des messages de synchronisation pour garder la même échelle de précision, chose qui n'est pratique pour les WSNs très limités en termes de ressources. Nous concluons donc que le second modèle est le plus efficace pour les applications des réseaux de capteurs, surtout celles qui nécessitent de préserver les horloges pré-synchronisées telles que la détection de fuites de gaz ou la détection d'intrusions. En revanche, le modèle *offset-only* a l'avantage de simplicité, et il peut être utile lors de sa mise en œuvre avec une horloge plus stable comme l'horloge externe, à condition que seule une faible précision est requise (par exemple, à l'ordre de la milliseconde).

Précision (microsecondes)

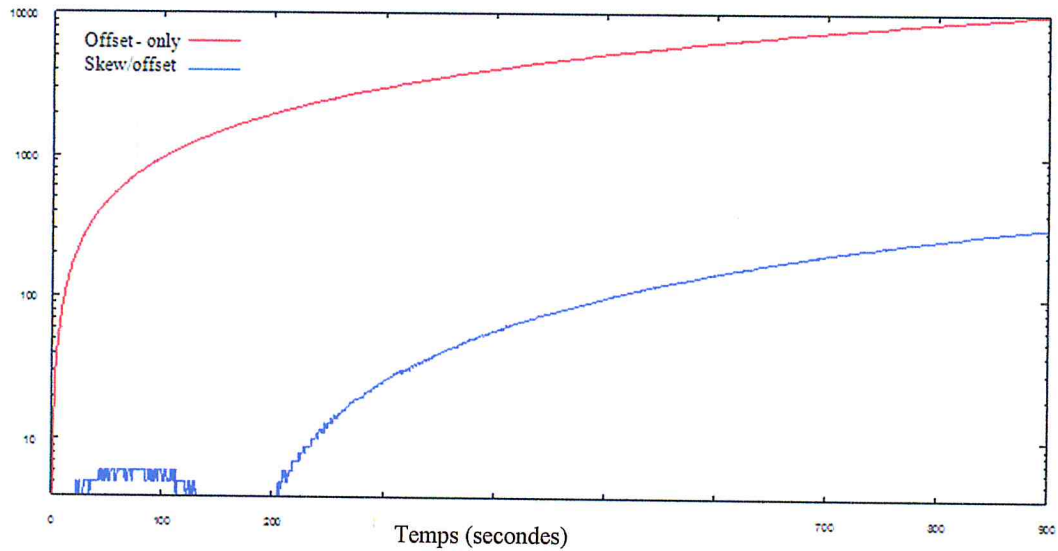


Figure 4.17. Comparaison de stabilité entre les deux modèles

4.5. Tolérance aux Fautes

Les mécanismes de tolérance aux fautes conçus dans la section 4.3 du chapitre précédent sont basés sur l'utilisation des *Timers* qui prennent en entrée les deux constantes Ω et Δ . La durée de ces constantes est dépendante des deux temps « Temps_Traitement » et « ϵ » qui sont fixés empiriquement comme suit :

Paramètre	Valeur
Temps_Traitement	500 microsecondes
ϵ	1 secondes

Tableau 4.2. Paramètres de mécanismes de tolérance aux fautes

Par la suite, nous allons tester le fonctionnement des mécanismes de tolérance aux fautes.

4.5.1. Continuité

Pour tester la continuité de la synchronisation, nous avons réalisé un code qui permet d'éteindre les micro-capteurs du réseau l'un après l'autre. Pour cela, nous avons utilisé un réseau de 6 nœuds ayant des identificateurs allant de 0 à 5. Au début, tous les nœuds du réseau fonctionnent normalement ; après un certain temps, nous avons éteint le nœud portant l'ID 1, puis le nœud 2 et ainsi de suite. Ce scénario permet de simuler le cas de la panne des micro-capteurs. Nous avons remarqué que le protocole a continué son exécution

et chaque nœud a envoyé le message de synchronisation dans son tour, même en cas d'éteignement de trois micro-capteurs successifs. De plus, les nœud-capteurs ont pu rejoindre le réseau aisément après la phase d'éteignement (voir *figure 4.18*).

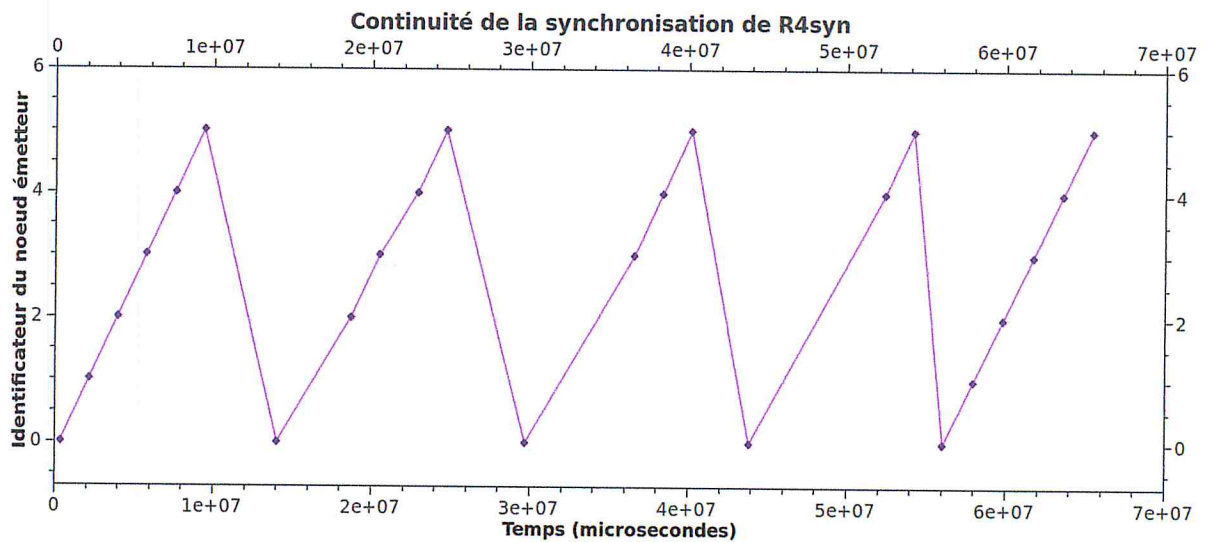


Figure 4.18. Continuité de la synchronisation

4.5.2. Granularité

Dans cette expérience, nous voulons observer le comportement de la précision dans le cas de défaillance d'un ou de plusieurs nœuds, ou dans le cas de perte de messages de synchronisation. Pour cela, nous avons étendu le scénario décrit plus haut par la mesure de précision. Le graphe de la *figure 4.19* est divisé en cinq intervalles. Le premier intervalle $[0, 5]$ représente la précision du protocole R^4_{syn} dans le cas idéal (aucune panne ou perte de messages). La précision dans cet intervalle était comprise entre 2 et 5 microsecondes. Dans le deuxième intervalle $[5, 10]$ où le nœud 1 a été éteint, nous remarquons que la précision s'est un peu détériorée, entre 10 et 13 microsecondes. En général, nous avons constaté que la précision se dégrade par environ 5 microsecondes seulement, à chaque éteignement d'un micro-capteur. Le dernier intervalle montre que la précision est revenue à son échelle normale (entre 2 et 5 microsecondes) après le ré-allumage de tous les micro-capteurs.

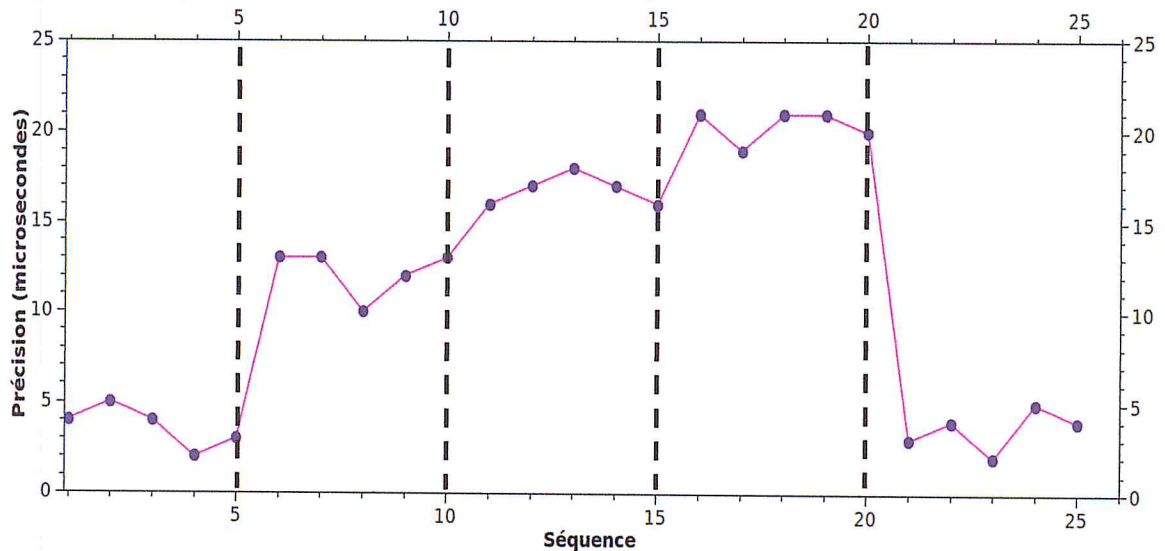


Figure 4.19. Granularité de la synchronisation

5. Application Oscilloscope

Afin de faciliter la visualisation des résultats en temps réel, nous avons utilisé un outil écrit en *Java* nommé « Serial Forwarder » pour récupérer la précision du protocole à partir de la station de base que nous avons réalisé. Pour l’affichage textuel, nous avons écrit un script simple qui permet de récupérer la précision auprès de la station de base et de la sauvegarder sous forme d’un fichier archive (LOG) pour pouvoir l’analyser ultérieurement. Alors que pour l’affichage graphique, nous avons adapté une application *Java* qui existe dans TinyOS nommée «Oscilloscope ». Cette dernière permet d’afficher la précision en temps réel et sous forme graphique. Voici à quoi ressemblent les interfaces de Serial Forwarder et de l’application Oscilloscope (figures 4.20, 4.21).

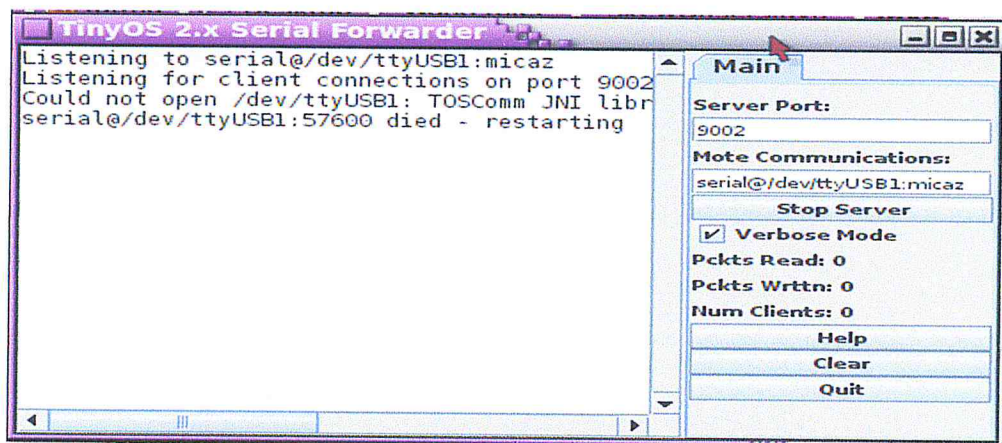


Figure 4.20. Interface de l’outil Serial Forwarder

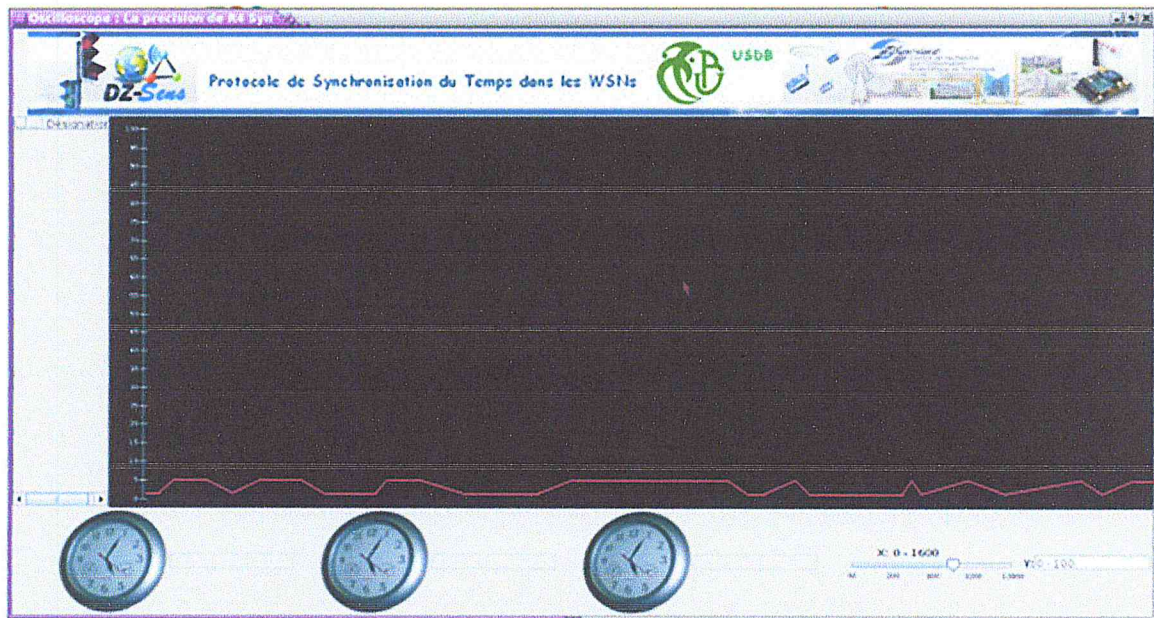


Figure 4.21. Aperçu de l'application oscilloscope

6. Conclusion

Au fil de ce dernier maillon de notre travail, nous avons présenté l'implémentation d'une version tolérante aux fautes des deux modèles de R^4Syn sur une plateforme expérimentale réelle composée de micro-capteurs MICAZ. Nous avons d'abord implémenté le modèle *offset-only* et suite à une série d'expériences réelles, nous sommes arrivées à un résultat très satisfaisant, vu que la précision de la synchronisation a été limitée à quelques microsecondes (entre 1 et 7). La taille du code binaire de ce premier modèle était 17600 octets de ROM et 984 octets de la mémoire RAM. Cependant, la précision se dégrade immédiatement car ce modèle ne permet pas d'estimer le *drift* des horloges. Ensuite, nous avons abordé l'implémentation du modèle *skew/offset* qui a donné des résultats similaires à ceux du premier modèle concernant la précision. De plus, ce fameux modèle a montré une très bonne stabilité vu que la précision est demeurée inférieure à 10 microsecondes pendant plus de trois minutes. La taille du code binaire de ce modèle était 28470 octets de ROM et 1434 octets de RAM. Durant la réalisation du deuxième modèle, nous avons rencontré un problème sérieux qui concerne le débordement de type. Ce problème est pallié par la simplification des estimateurs du *skew* et de l'*offset*.

Pour pouvoir élever la précision avec exactitude, nous avons conçu trois méthodes de tests. La troisième méthode était la plus fiable, mais aussi la plus difficile à concevoir car nous étions obligées de manipulé le niveau le plus bas du matériel afin de programmer des interruptions matérielles.

Nous avons ainsi testé les mécanismes de tolérances aux fautes développés. Ce qui nous a permis de constater que le protocole continuera à s'exécuter et donner une bonne précision malgré la défaillance de quelques nœuds dans le réseau.

*Conclusion Générale
et Perspectives*

Conclusion Générale et Perspectives

Les réseaux de capteurs sans fil (WSN) résultent d'une fusion de deux pôles de l'informatique moderne, les systèmes embarqués et les communications sans fil. Les micro-capteurs sont autonomes, offrent un avantage de coût et de facilité de déploiement, et ils sont capables d'exécuter des tâches très complexes. Des applications diverses et variées, basées sur ces réseaux, sont développées pour améliorer le confort et la sécurité de l'être humain.

La synchronisation temporelle constitue une pièce critique de tout système distribué. Cependant, les réseaux de capteurs sans fil ont une utilisation étendue de ce service pour, par exemple, mesurer le temps de propagation d'un son détecté afin de localiser sa source, ou éliminer les messages redondants qui décrivent des détections dupliquées du même phénomène par des micro-capteurs différents.

En plus des caractéristiques matérielles des micro-capteurs qui compliquent la tâche de la synchronisation dans ce genre de réseaux, plusieurs applications des WSNs visent des objectifs plus stricts que les systèmes distribués traditionnels, concernant la précision et la validité de la synchronisation. Les schémas de synchronisation dédiés aux WSNs doivent minimiser au maximum l'utilisation des ressources, surtout celles énergétiques, tout en respectant les exigences de la synchronisation en terme de précision.

Le travail réalisé dans ce mémoire entre dans le cadre du projet « Gestion du trafic routier » qui se réalise au sein du CERIST. Il consiste à l'implémentation d'un protocole de synchronisation du temps distribué, ainsi qu'au développement de l'aspect tolérance aux fautes des micro-capteurs. Pour cela, nous avons mené une étude sur les réseaux de capteurs et leurs caractéristiques, ainsi que des protocoles de synchronisation du temps dans les WSNs. L'étude englobe aussi le système d'exploitation « TinyOS » et le langage de programmation associé « NesC ». Cette étude nous a permis de développer, implémenter puis tester une variante tolérante aux défaillances du protocole R^4syn sur des micro-capteurs réels de type MICAz.

Les résultats de l'implémentation montrent une très bonne précision pour les deux modèles d'estimateurs de R^4syn , vu que la précision n'a pas dépassé 7 microsecondes.

Ainsi, le modèle *skew/offset* qui modélise la déviation des horloges a montré une grande stabilité et la précision a subsisté à l'ordre de quelques microsecondes pendant plusieurs minutes.

Ce travail nous a permis d'acquérir des connaissances dans plusieurs domaines d'actualités tels que les réseaux de capteurs sans fil, la synchronisation temporelle dans ce genre de réseaux et les systèmes d'exploitation dédiés aux systèmes embarqués tels que TinyOS. Le développement des protocoles sur des plateformes expérimentales réelles constitue un acquis très important durant la période de notre stage.

Des perspectives de notre travail peuvent être envisagées. L'horloge du microcontrôleur utilisée dans l'implémentation ne continue pas à fonctionner lorsque le micro-capteur passe en état de veille ce qui peut invoquer la dégradation de la synchronisation (si les micro-capteurs ne passent pas en veille en même temps ou dans le cas de l'utilisation de la méthode *moving average* pour la convergence). L'approche VHT (Virtual High-resolution Time) proposée dans [SDS10] peut résoudre ce problème. Cette approche consiste à l'utilisation de l'horloge externe à cristal comme référence pour estimer le temps qui aura été généré par l'horloge interne, après la période de *duty-cycle*.

Ainsi, notre implémentation permet de fournir une synchronisation locale (entre les nœuds-capteurs qui appartiennent au même domaine de diffusion). Néanmoins, et selon le besoin de l'application, une version multi-sauts de ce protocole peut être implémentée [Dje12].

Annexes

Annexe A :
Systeme d'Exploitation
TinyOS

1. Présentation

Les systèmes d'exploitation classiques sont généralement conçus pour un usage générique. leurs objectifs sont la facilitation de l'usage, la rapidité et l'efficacité. Ils sont ainsi conçus en supposant une disponibilité illimitée des ressources. Parmi leurs caractéristiques, nous pouvons citer l'architecture multi-thread qui nécessite une mémoire importante, l'absence de la contrainte d'énergie et la séparation entre espaces noyau/utilisateur. Ces caractéristiques rendent les systèmes d'exploitation classiques inutilisables dans les réseaux de capteurs à cause des limitations matérielles de ces derniers. TinyOS est l'un des systèmes d'exploitation proposés pour s'adapter aux contraintes matérielles des micro-capteurs, c'est pourquoi:

- TinyOS respecte une architecture basée sur une association de composants réduisant la taille du code nécessaire à sa mise en place. Un seul fichier binaire (exécutable) est chargeable en mémoire, il contient les composants des applications et les composants de système d'exploitation nécessaires pour son bon fonctionnement.
- Et pour répondre à la contrainte énergétique très critique dans les réseaux de capteurs, TinyOS propose un fonctionnement évènementiel « event-driven », c'est-à-dire qu'il ne devient actif qu'à l'apparition de certains évènements par exemple l'arrivée d'un message radio.

Chaque composant de TinyOS correspond à un élément matériel : LED, *Timer*, etc. et il peut être réutilisé dans différentes applications.

2. Modèle d'Exécution de TinyOS

TinyOS n'exécute qu'une application à la fois, son modèle d'exécution est basé sur deux types de processus, les tâches « tasks » et les pilotes d'interruption « interrupt handlers ».

a. Tâches

La tâche représente une activité de longue durée. Lors de son invocation, celle-ci est placée dans une file d'attente d'ordre FIFO (First In First Out) non préemptif. TinyOS ne dispose pas de mécanisme de préemption entre les tâches. Une fois lancée, celle-ci s'exécutera entièrement. Ce mode de fonctionnement permet de bannir les opérations pouvant bloquer

le système comme l'inter-blocage ou la famine. Lorsque la file d'attente est vide, TinyOS se met automatiquement en veille, afin d'économiser l'énergie.

b. Pilotes d'Interruption

Appelés aussi traitements des événements matériels. Ils sont exécutés en réponse à une interruption matérielle. Ces derniers ont la capacité de préempter les tâches et les autres pilotes d'interruption. Les commandes et les événements qui sont exécutés par un pilote d'interruption doivent être définis avec le mot clé « **async** », qui signifie que ces fonctions vont être exécutées de façon asynchrone.

3. Gestion de la Mémoire

TinyOS a une empreinte mémoire très faible, il ne prend que 300 à 400 octets de mémoire nécessaire à son installation, dans le cadre d'une distribution minimale [W08]. En plus de cela, il est nécessaire d'avoir 4 Ko de mémoire libre répartie comme suit :

- **La pile** : sert de mémoire temporaire au fonctionnement du système notamment pour l'empilement et le dépilement des variables locales.
- **Les variables globales** : réservent un espace mémoire pour le stockage de valeurs pouvant être accessible depuis des applications différentes.
- **La mémoire libre** : pour le reste du stockage temporaire.

TinyOS ne propose pas d'allocation dynamique ni de pointeur de fonction. Par ailleurs, il n'existe pas de mécanisme de protection de la mémoire sous TinyOS. Ceci rend le système particulièrement vulnérable aux crashes et aux corruptions de la mémoire.

4. Gestion de la Concurrence

Comme les tâches et les pilotes d'interruption peuvent être préemptés par d'autres pilotes d'interruption, des problèmes de concurrence d'accès aux données partagées peuvent se produire. Ces problèmes sont gérés traditionnellement par l'utilisation des sémaphores. TinyOS utilise pour cela le concept « **atomic** » du langage NesC.

a. Mot clé Atomic

Le bloc d'instructions déclaré comme « **atomic** » s'exécute comme si aucun autre calcul ne se fait simultanément ; ce qui signifie une exécution séquentielle sans aucune préemption possible.

Une instruction "atomic" doit être courte, NesC interdit dans une instruction atomique: call, signal, goto, return, break. La syntaxe d'un bloc atomique est la suivante:

```
atomic {ensemble d'instructions}
```

b. Mot clé norace

Le compilateur NesC détecte les problèmes de concurrence et réagit par une erreur de compilation. Lorsqu'il est sûr que le problème ne se pose pas, on peut demander au compilateur via le mot clé **norace** d'accepter une construction apparemment douteuse. Bien entendu il faut utiliser cela avec circonspection.

5. Temps Réel

Lorsqu'un système est dit " temps réel ", il est censé gérer des niveaux de priorité dans ses tâches permettant de respecter des échéances données par son environnement. Ce type de système favorise aussi les préemptions.

Comme nous l'avons vu précédemment, la seule préemption permise dans TinyOS est celle des pilotes d'interruption. A part ça, il ne gère aucun mécanisme de préemption ou de priorité. C'est pourquoi TinyOS n'est pas prévu pour avoir un fonctionnement temps réel.

Pour remédier à ce problème, les commandes écrites dans les composants doivent être de courte durée. Si une commande peut durer longtemps, le programmeur doit utiliser une tâche qui retournera un événement pour indiquer la fin de son exécution, c'est par exemple le cas des commandes d'envoi sur l'interface radio. Les traitements des événements sont par nature de courte durée.

6. Cibles Possibles de TinyOS

Les cibles de TinyOS sont essentiellement des cibles embarquées tels que les micro-capteurs. Toutes ces cibles possèdent une architecture commune basée sur un noyau central autour duquel s'articulent les différentes interfaces d'entrées/sorties, de communication et d'alimentation. (Voir *Figure A.1*)

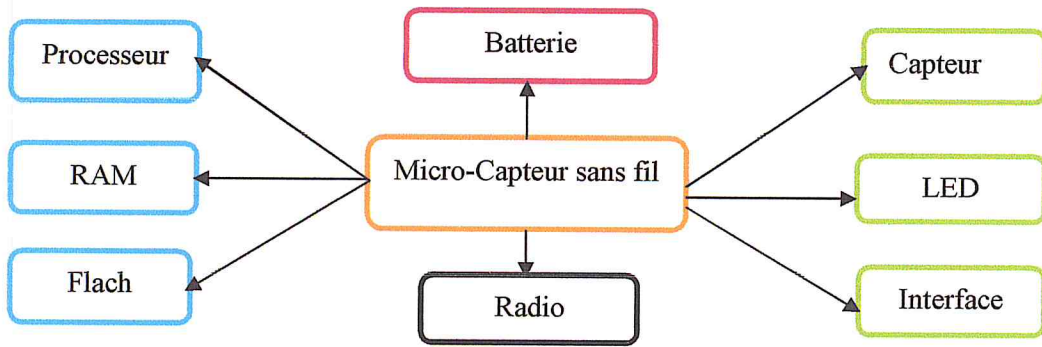


Figure A.1. Présentation d'une cible de TinyOS

Annexe B :
Langage de
Programmation NesC

1. Présentation

NesC est une variante du langage C, orientée composant. Il est construit autour de deux éléments de base qui sont les interfaces et les composants.

1.1. Interface

L'interface est le seul point d'accès aux composants, elle décrit d'une manière abstraite les interactions entre deux composants ; un utilisateur et un fournisseur. La syntaxe d'une interface est la suivante :

```
interface NomInterface{  
    //Déclaration des commandes et des évènements  
}
```

Les interfaces sont bidirectionnelles, vu qu'elles définissent un ensemble de fonctions qui peuvent être soit des commandes ou des évènements.

1.1.1. Commande

Les commandes sont des fonctions qui font des appels de haut vers le bas, c'est-à-dire du composant applicatif vers les composants les plus proches du matériel. La commande doit être implémentée par le composant qui fournit l'interface. La syntaxe de déclaration d'une commande est :

```
command typeDeRetour nomDeLaCommande (déclaration des paramètres) ;
```

L'utilisateur de l'interface peut appeler une commande comme suit :

```
call nomDeL'interface .nomDeLaCommande (valeurs des paramètres) ;
```

1.1.2. Evènement

Les évènements sont des fonctions qui remontent les signaux du bas vers le haut. Contrairement aux commandes, un évènement doit être implémenté par le composant utilisateur de l'interface. La syntaxe de déclaration d'un évènement est :

```
event typeDeRetour nomDeL'evenement (déclaration des paramètres);
```

Le fournisseur de l'interface peut signaler un évènement comme suit :

```
signal nomDeL'interface.nomDeL'evènement (valeur des paramètres);
```

Voici un schéma qui résume les interactions entre les interfaces et les composants d'un système.

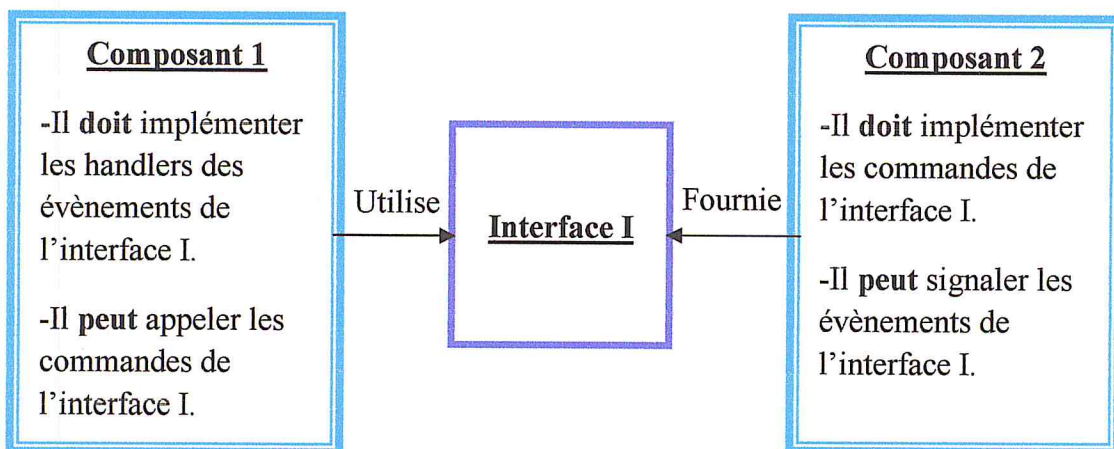


Figure B.1. Association entre les composants et l'interface.

Une même interface peut être utilisée par plusieurs composants. De même, elle peut être fournie par plusieurs composants. Ceci est liée au besoin de réagir aux évènements de l'interface ou encore d'implémenter ses commandes.

- **Interface Générique**

Le langage NesC supporte les interfaces génériques ou paramétrées « generic interfaces ». Une interface générique a la capacité de prendre un ou plusieurs types comme paramètres, cela pour éviter la redondance, c'est-à-dire, créer plusieurs interfaces qui contiennent le

même code dont la seule différence est le type de certaines variables. La syntaxe d'une interface générique est :

```
interface nom_interface < paramètre 1, ..., paramètre n >
```

L'interface du système '*Timer* <precision_tag>' qui se trouve dans `tinyos-2.x/tos/lib/timer`, est un exemple d'une interface générique. La précision voulue est spécifiée lors de son instantiation. Par exemple *Timer* <TMilli> si on veut avoir le temps en millisecondes.

1.2. Composant

NesC possède deux types de composants, les modules et les configurations.

1.2.1. Module

Les modules constituent la brique élémentaire du langage NesC. Ils peuvent fournir ou utiliser une ou plusieurs interfaces ou encore plusieurs instances de la même interface. Dans ce dernier cas, il faut renommer les interfaces via le mot clé « as » pour pouvoir les distinguer. La syntaxe de création d'un module est la suivante :

```
module nom_module{  
  provides{ //liste des interfaces fournies  
    interface NomInterfaceFournie ;  
  }  
  uses { //liste des interfaces requises  
    interface NomInterfaceUtilisé ;  
  }  
}  
implementation {  
  // La déclaration des variables  
  // L'implémentation des commandes des interfaces fournies et les  
  // handlers des évènements des interfaces utilisées.  
}
```


Les modules permettent aussi d'implémenter les tâches.

1.2.1.1. Tâche en NesC

La tâche est un élément de contrôle indépendant défini par une fonction retournant void et sans arguments, elle a comme syntaxe :

```
task void nomDeTache () { // un grand bloc de calcul }
```

Les tâches sont utilisées pour effectuer la plupart des blocs d'instruction d'une application, surtout ceux qui consomment du temps. Par exemple un travail qui nécessite beaucoup de calculs. Une tâche peut être postée par une commande ou un évènement comme suit :

```
post nomDeTache();
```

1.2.1.2. Configuration

La configuration définit les composants et les interfaces utilisés par l'application, ainsi que les liaisons entre eux. En d'autres termes, la configuration sert à assembler les composants entre eux en connectant les interfaces utilisées par certains composants à celles fournies par d'autres. Dans une application, plusieurs fichiers « configuration » peuvent exister, mais on trouve toujours une configuration de haut niveau « top-level ». Cette configuration a une spécificité puisque elle ne fournit et elle n'utilise aucune interface, elle contient juste les liaisons « wiring » entre les composants. Le composant Main est obligatoirement présent dans la liste des composants à utiliser dans une configuration top-level. En effet, il permet de lancer l'exécution de l'application.

A noter que deux composants ne peuvent pas interagir, c'est-à-dire appeler des commandes ou signaler des évènements, sauf s'ils sont connectés. Voici la syntaxe d'une configuration :

```
configuration nom_configuration {  
    // Déclaration des interfaces fournies et requises quand il ne s'agit pas  
    // d'une configuration de haut niveau  
    provides { Le(s) interface(s) fournie(s) }  
    uses     { Le(s) interface(s) utiliser(s) }  
}  
  
implémentation {  
    // Liste des modules et des configurations utilisés par la configuration  
    components MainC, Module1,..., ModuleX, Config1,..., ConfigY ;  
    // La description des liaisons entre les interfaces et les différents  
    // composants en utilisant les opérateurs qui vont être décrits par la suite.
```

○ Opérateur

Deux types d'opérateurs de connexion peuvent exister dans une configuration :

- **Opérateur « -> » ou « <- »**

L'opérateur -> permet de connecter un utilisateur d'interface à un fournisseur comme suit :

```
composantUtilisateur . interface -> composantFournisseur . interface
```

Et réciproquement, l'opérateur <- permet de connecter un fournisseur d'une interface à un utilisateur dont la syntaxe est la suivante :

```
ComposantFournisseur . interface <- ComposantUtilisateur . interface
```

- **Opérateur « = »**

Les opérateurs `->` et `<-` permettent de connecter des utilisateurs et des fournisseurs concrets qui sont cités après le mot clé « components ». Comme nous l'avons vu, la partie implémentation d'une configuration est réservée à la liaison des composants et ne peut contenir aucune implémentation des fonctions (commandes ou évènements) c'est pour ça, la seule façon pour qu'une configuration puisse utiliser ou fournir des interfaces est l'utilisation d'un composant mandataire « proxy ». La configuration appelle un composant qui implémente son interface fournie / utilisée puis, le connecte à cette interface en utilisant l'opérateur `" = "` grâce à l'instruction suivante:

```
interface = composant . interface
```

1.2.2. Composant Générique

De même que les interfaces, NesC autorise la création des composants génériques. Le composant générique est un composant qui fournit une interface générique. Cette fois-ci, il faut précéder la déclaration des composants par le mot clé « **generic** ». L'entête d'un composant générique est :

```
generic configuration nom_configuration () { } // ou encore :  
generic module nom_module () { }
```

Pour instancier un composant générique, on utilise le mot clé **new** comme suit :

```
new nom_composant ()
```


2. Architecture Globale du Langage NesC

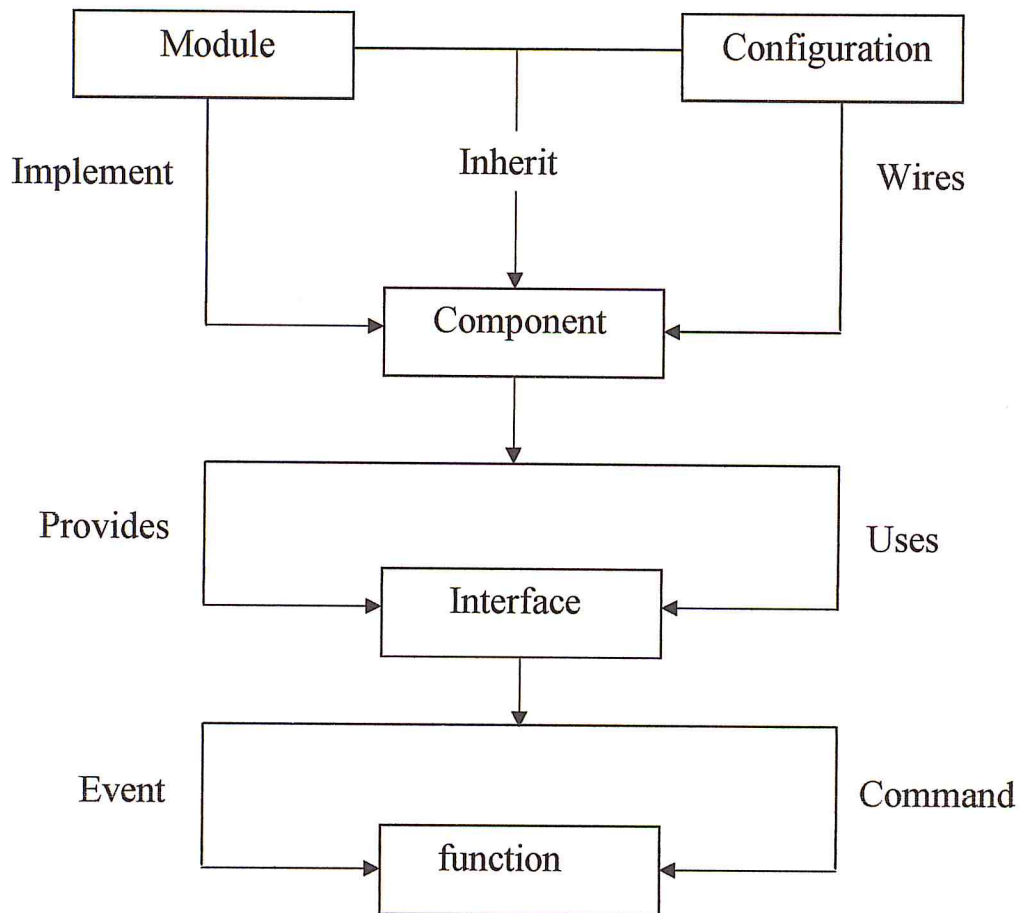


Figure B.2. Architecture du langage NesC.

3. Mots Clés de NesC

Mot clé	Signification
as	Alias qui nous permet de renommer une interface.
Async	Permet d'exécuter les commandes et les événements de façon asynchrone.
Atomic	Garantie l'exécution d'un bloc d'instruction sans préemption.
Call	Permet d'appeler une commande.
Command	Permet de définir une commande.
components	Liste des composants interconnectés entre eux dans une configuration.
configuration	Un composant composite.
Event	Un signal envoyé par une interface.
Implementation	Pour le module, elle contient les variables et le code. Pour la configuration, elle contient les liens entre les composants.
Includes	Permet d'inclure les fichiers header.
Interface	La définition des commandes et des événements.
Module	Un composant de base.
norace	Permet d'éliminer les avertissements "warning" générés par le compilateur.
Post	Permet de lancer une tâche.
Provides	Définir les interfaces fournies par un module ou une configuration.
Signal	Permet d'exécuter un événement.
Task	Permet de définir une tâche.
Uses	Définir les interfaces utilisées par un module ou une configuration.

Tableau B.1. Mots clés de NesC

4. Structure d'une Application NesC

Chaque application NesC contient au moins les trois fichiers suivants : Makefile, un module et une configuration. Tous les fichiers qui contiennent le code NesC portent l'extension `.nc`. Pour réaliser la compilation, les fichiers sources doivent se situer dans un seul répertoire contenant le **makefile**.

makefile est le fichier qui est compilé au premier lieu lors de la compilation du programme. Il contient le nom de la configuration de `top_level`. C'est cette dernière qui fait appel aux composants utilisés par l'application (y compris ceux du système d'exploitation).

- **Exemple d'un fichier makefile**

```
COMPONENT= NomDeLaConfigurationDeHautNiveau
```

```
include $(MAKERULES)
```

5. Compilation et Exécution d'un Code

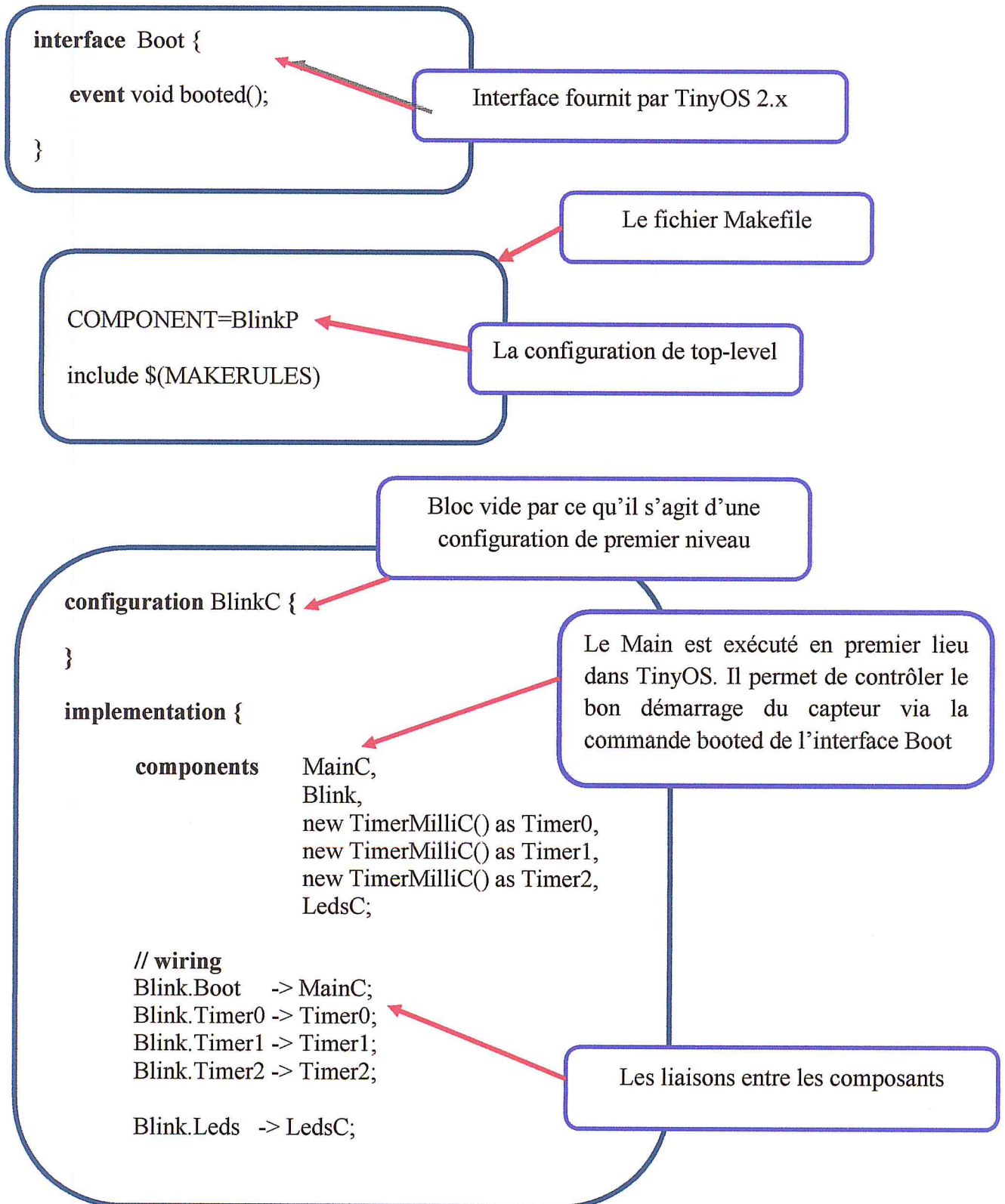
Le compilateur de NesC est appelé `ncc`, traite les fichiers nesC en les convertissant en un fichier C "gigantesque" qui contient l'application et les composants du SE utilisés par cette application. Ensuite un compilateur spécifique à la plateforme cible compile ce fichier C, pour produire un seul exécutable.

Voici la commande qui permet de compiler une application sous la plateforme souhaitée. Le résultat de la compilation est la génération d'un exécutable « `main.exe` » dans le même répertoire que l'application.

```
make plate-forme
```


6. Exemple Illustratif d'une Application en NesC

Le programme ci-dessous permet de faire clignoter les trois LEDs de capteur.



```
module BlinkP {  
  uses { interface Timer<TMilli> as Timer0;  
         interface Timer<TMilli> as Timer1;  
         interface Timer<TMilli> as Timer2;  
         interface Leds;  
         interface Boot;  
  }  
}
```

Déclaration des interfaces utilisées par le module Blink

```
implementation {  
  event void Boot.booted(){  
    call Timer0.startPeriodic(1000);  
    call Timer1.startPeriodic(1000);  
    call Timer2.startPeriodic(1000);  
  }
```

Permet d'initialiser le composant, cette commande est exécutée au démarrage du capteur.

```
  event void Timer0.fired(){  
    call Leds.led0Toggle();  
  }
```

Permet de lancer le Timer numéro 1 en mode continu pour une période répétitive d'une seconde.

```
  event void Timer1.fired(){  
    call Leds.led1On();  
  }
```

Elle s'exécute à l'expiration du Timer 0, après une seconde de démarrage de capteur

```
  event void Timer2.fired(){  
    call Leds.led2Off();  
  }
```

La première commande permet d'allumer la led verte, et la deuxième permet d'allumer la jaune.

Annexe C :
Simulateur Aurora


```
Ln -s build/micaz/main.exe main.elf  
Ou mv build/micaz/main.exe main.elf
```

La commande qui permet de simuler les programmes est la suivante :

```
java avrora.Main -platform=micaz -simulation=sensor-network -seconds=10  
-nodecount=2 -topology=static -monitors=c-print,leds,energy -  
VariableName=ComposantC__debugbuf1 -topology-file=fichierTopologie.def  
main.elf
```

On peut changer les valeurs des variables de cette commande selon le besoin. Voici la signification de chaque variable :

- **platform** : permet de spécifier la plate-forme des micro-capteurs utilisés.
- **seconds** : permet de préciser la durée de simulation souhaitée en secondes.
- **Nodecount** : permet de préciser le nombre de nœuds qu'on veut simuler. Si le nombre de nœuds est supérieur à 1, il faut créer un fichier qui décrit la topologie du réseau.
- **topology** : peut prendre deux valeurs : static pour une topologie de réseau statique, et dynamic pour une topologie dynamique.
- **monitors** : il existe plusieurs moniteurs dans avrora, par exemple : energie qui permet d'analyser l'énergie consommée, leds qui permet d'afficher les états des leds de chaque capteur "on" ou "off", et c-print qui permet d'afficher les messages dans le shell (Débogage). Pour pouvoir afficher des messages via Avrora, on doit copier le fichier AvroraPrint.h dans le dossier de l'application, ou encore ajouter un chemin dans le fichier Makefile qui pointe sur : /opt/Avrora/src/avrora/monitors. Par ailleurs, la limite d'Avrora est qu'il permet de n'afficher que des messages provenant d'un et un seul composant.

- **VariableName :** permet de spécifier le nom du composant qui contient les messages à afficher. Le nom de ce composant doit être suivi par le mot clé « __debugbuf1 ».
- **Topology-file:** contient le nom du fichier qui décrit la topologie du réseau.

Annexe D :
Procédure
d'Installation

1. Procédure d'Installation de TinyOS sous Linux

Pour installer le système d'exploitation TinyOS sous Linux, il faut suivre les étapes suivantes [W10, W11] :

1. se connecter à Internet.
2. Ouvrir le Shell.
3. Ouvrir le fichier `/etc/apt/sources.list` en mode root via la commande suivante :

```
sudo nano/etc/apt/sources.list
```

4. S'assurer que l'installateur de paquet trouve les urls de tinyOS et les mette dans un dépôt, par l'ajout du texte suivant à la fin du fichier : **deb <http://hinrg.cs.jhu.edu/tinyos> lucid main**

5. Mettre à jour les informations sur votre système via la commande suivante :

```
sudo apt-get update
```

6. Installer Java6 via la commande suivante :

```
sudo apt-get install sun-java6-jdk
```

7. Installer le compilateur via la commande suivante :

```
sudo apt-get install g++ g++-3.4 gperf swig sun-java5-jdk graphviz alien fakeroot
```

8. Installer TinyOS par la commande suivante :

```
sudo apt-get install tinyos-2.1.0
```

9. Ajouter la ligne suivante à la fin du fichier caché `.bashrc` qui se trouve dans votre dossier de départ par exemple `/home/mekahlia`. Pour afficher les fichiers cachés, il suffit d'appuyer sur **Ctrl+H**.

```
source ~/.bash_tinyos
```

2. Procédure d'Installation d'Avrora sous Linux

1. Se connecter à internet.
2. Ouvrir le Shell.
3. Installer Java6 via la commande suivante :

```
sudo apt-get install sun-java6-jdk
```

4. En utilisant le browser, télécharger Avrora Beta [1.6.0.zip] ou la version que vous voulez à partir de ce lien : <http://compilers.cs.ucla.edu/avrora/>
5. Décompresser le dans le répertoire /opt/
6. Ajouter la ligne suivante à la fin du fichier caché .bashrc

```
export CLASSPATH=/opt/avrora/bin:$CLASSPATH
```

3. Injection du Code dans le Micro-capteur

Pour la plateforme micaz, il faut connecter le micro-capteur au programmeur « programming board ». Par contre, les capteurs qui contiennent une interface USB, tel que telosB, peuvent être connecté directement à l'ordinateur sans programmeur. La commande suivante permet d'injecter le code et d'identifier le micro-capteur :

```
make micaz install.1
```

Pour utiliser le port USB et afin d'afficher les messages envoyés par les micro-capteurs, au niveau de l'invité de commande, il faut installer jni comme suit :

```
sudo tos-install-jni
```

La commande qui permet d'afficher les messages est la suivante :

```
java net.tinyos.tools.PrintfClient -comm serial@/dev/ttyUSB1:micaz
```


*Références
Bibliographiques et
Webographiques*

Références Bibliographiques et Webographiques

- [ASC02] : I.F. Akyildiz, W. Su, Y. Sankarasubramaniam, E. Cayirci. "Wireless sensor networks: a survey", *Computer Networks* vol.38, pp. 393–422. August 2002.
- [ASS02] : Ian F. Akyildiz, Weilian su, Yogesh Sankarasubramaniam, and Erdal Cayirci, "A Survey on sensor networks", *IEEE Communications Magazina*, vol. 40, num. 8, pp. 102-114. August 2002.
- [CON03] : H. Cam, S. Ozdemir, P. Nair, and D. Muthuavinashippan, "ESPDA: Energy Efficient and Secure Pattern Based Data Aggregation for Wireless Sensor Networks" in *Proceedings of IEEE Sensor*, pp. 732 – 736. Toronto, Canada. 2003.
- [Dap08] : Dazhi Chen and Pramod K. Varshney. "Qos support in wireless sensor networks: A survey". Department of EECS, Syracuse University Syracuse, NY, U.S.A 13244 Journals. 2008.
- [Dje11]: Djamel Djenouri, "Distributed Receiver/Receiver synchronization in sensor networks: New solution and join offset/skew for Gaussian delays". 6th International Conference Wireless Algorithms, Systems, and Applications (WASA'11), Chengdu, China. LNCS vol.6843, pp.13-24, Springer Verlags. August 2011.
- [Dje12]: Djamel Djenouri, "R⁴ Syn Relative Referenceless Receiver/Receiver Time Synchronization in Wireless Sensor Networks". *IEEE Signal Processing Letters*, vol 19, num4: pp 175-178. April 2012.
- [EGE02] : Jeremy Elson, Lewis Girod and Deborah Estrin "Fine-grained

Références Bibliographiques et Webographiques

network time synchronization using reference broadcasts". In 5th USENIX Symposium on Operation system Design and Implementation (OSDI'02). December 2002.

[FMT10] : Federico Ferrari, Andreas Meier, and Lothar Thiele "Secondis: An Adaptive Dissemination Protocol for Synchronizing Wireless Sensor Networks", IEEE Secon. 2010.

[GIG07] : C. F. Garcia-Hernandez, P. H. Ibargnengoytia-Gonzalez, J. Garciahernandez and J.A. Perez-Diaz: "Wireless Sensor Networks And Applications: A Survey", IJCSNS International Journal of Computer Science and Network Security, vol.7, num.3, pp. 264-273. 2007

[GKS03] : Ganeriwal, S., Kumar, R., and Srivastava, M. B, "Timing-sync protocol for sensor networks ". In Proceedings of the 1st international conference on Embedded networked sensor systems. SenSys '03, 138–149. 2003.

[GLC03]: David Gay, Philip Levis, David Culler, Eric Brewer, "nesC 1.1 Language Reference Manual ", May 2003.

[HSW00] : J. Hill, R. Szewczyk, A. Woo, S. Hollar, D. Culler, and K. Pister, "System architecture directions for networked sensors". ACM SIGPLAN, Notices, vol. 35, num. 11, pp. 93–104. 2000.

[HWi03]: Hologer K., Willig A, "A short survey of wireless sensor networks", Technical university Berlin, Telecommunication Networks Group. October 2003.

Références Bibliographiques et Webographiques

- [Lam78]: Leslie Lamport, "Time, Clocks, and the Ordering of Events in a Distributed System", Communications of the ACM, Volume 21, Number7, PP 558-565. July 1978.
- [LeG09] : Philip Levis, David Gay, "TinyOS programming". Cambridge university press, 978-0-511-50730-4, London, 2009.
- [LLC03] : Philip Levis, Nelson Lee, Matt Welsh and David Culler, "TOSSIM: Accurate and Scalable Simulation of Entire TinyOS Applications", In Proceedings of the First ACM Conference on Embedded Networked Sensor Systems. 2003.
- [Mil06] : David L. Mills "Computer Network Time Synchronization: the Network Time Protocol". CRC Press. 26-Apr-06.
- [MJS02] : Alan Mainwaring, Joseph Polastre, Robert Szewczyk, David Culler, and John Anderson, "Wireless sensor networks for habitat monitoring", In ACM International Workshop on Wireless Sensor Networks and Applications (WSNA'02), Atlanta, GA, September 2002.
- [MKS04] : Maróti Miklós, Kusy Branislav, Simon Gyula, and Lédeczi Ákos. "The Flooding Time Synchronization Protocol". In SenSys. pp 39–49. 2004.
- [NCS07]: Noh, K.-L., Chaudhari, Q. M., Serpedin, E., and Suter, B. W.. "Novel clock phase offset and skew estimation using two-way timing message exchanges for wireless sensor networks". IEEE Transactions on Communications 55, 4, 766–777, 2007.

Références Bibliographiques et Webographiques

- [NDS04] : F. Nekoogar, F. Dowla, and A. Spiridon, "Self organization of wireless sensor networks using ultra-wideband radios". Technical Report UCRL-CONF-205469, Atlanta, GA, United States, September 2004.
- [PPi02] : Papoulis, A. and Pillai, S. U. "Probability, Random Variables and Stochastic Processes 4th Ed". McGraw Hill Higher Education. 2002.
- [SBK05] : Sundararaman, B., Buy, U., and Kshemkalyani, A. D, "Clock synchronization for wireless sensor networks: a survey. Ad hoc Networks" 3, 3, 281–323. 2005.
- [SDS10]: T. Schmid, P. Dutta, and M. Srivastava. High-resolution, low-power time synchronization an oxymoron no more. In ACM/IEEE IPSN, 2010.
- [SDZ07] : Kazem Sohraby, Daniel Minoli, Taieb Znati, "WIRELESS SENSOR NETWORKS: Technology, Protocols, and Applications", A John Wiley & Sons, INC., publication. 2007.
- [SiY04] : Fikret Sivrikaya and BulentYener. "Time Synchronization in Sensor Networks: A Survey", IEEE Network Magazin, vol. 18, num. 4, pp.45-50, July/August 2004.
- [STG07] : Cory Sharp, Martin Turon, David Gay, "Timers", Core Working Group, TEP: 102, Created: 22-Sep-2004, Modified: 11-06-2007.

Références Bibliographiques et Webographiques

- [TBL05] : Titzer, Ben L. and Lee, Daniel K. and Palsberg, Jens, "Avrora: scalable sensor network simulation with precise timing", Proceedings of the 4th international symposium on Information processing in sensor networks, IPSN '05, Los Angeles, California, 2005.
- [TLG07] : Gilman Tolle, Philip Levis, and David Gay SIDs, "Source and Sink Independent Drivers", Core Working Group, TEP: 114, Created: 30-Oct-2005, Modified: 10-01-2007.
- [WLC08] : Bachar Wehbi, Anis Laouiti and Ana Cavalli "Efficient time synchronization mechanism for wireless multi Hop networks". 19th IEEE International Symposium on Personal, Indoor and Mobile Radio Communications. PIMRC 2008. 15-18 Sept. 2008.
- [W01] : <http://www.memsic.com/support/documentation/wireless-sensornetworks/category/7-datasheets.html>
- [W02] : <http://www.cmt-gmbh.de/Mica2dot.pdf>
- [W03] : <http://www.atmel.com/devices/atmega128.aspx>
- [W04] : <http://www.atmel.com/devices/atmega1281.aspx>
- [W05] : <http://moodle.utc.fr/>
- [W06] : <http://www.tinyos.net>
- [W07] : <http://docs.tinyos.net/tinywiki>
- [W08] : <http://www.techno-science.net/?onglet=glossaire&definition=11588>
- [W09] : nsc.sourceforge.net
- [W10] : <http://www.5secondfuse.com/tinyos/install.html>
- [W11] : <http://www.tinyos.net/tinyos-2.x/doc/html/install-tinyos.html>

Références Bibliographiques et Webographiques

- [W12] : <http://compilers.cs.ucla.edu/avrora/>
- [W13] : <http://www.omnetpp.org/doc/omnetpp/manual/usman.html>
- [W14] : <http://arduino.cc/en/Main/ArduinoBoardUno>