Table des matières

Liste des figures

Liste des tableaux

Introduction générale :

Depuis quelques décennies, le besoin d'observer et de contrôler des phénomènes physiques tels que la température, la pression, ou encore la luminosité est essentiel pour de nombreuses applications industrielles et scientifiques. Et avec les évolutions dans le domaine des réseaux de capteurs et celui des processeurs, on peut imaginer des réseaux denses, sans fils, ayant pour rôles de récolter des données d'un environnement et de les diffuser au sein d'un réseau. Ce type de réseaux sans fils pourrait avoir plusieurs domaines d'applications. On peut citer à titre d'exemples la surveillance environnementale et de l'habitat, la création d'efficiences industrielles, le contrôle de la circulation, les opérations militaires et de sécurité et même l'amélioration des soins de santé.

La grande réputation de ces systèmes est liée au fait que ces réseaux offrent l'opportunité de développer des applications novatrices liées à la position géographique des éléments du réseau d'une part, et à la conception d'objets communicant entièrement autonomes d'une autre part. En effet, un réseau de capteurs sans fils est un ensemble de nœuds qui communiquent ensembles via une connexion sans fils et qui ont la possibilité de changer de position.

On pourrait prendre pour exemple un réseau de capteurs de température, de battements cardiaques, de taux de glycémie ou de la pression artérielle qui seraient mis en place sur les malades d'un hôpital. Ce réseau aurait pour rôle de contrôler l'évolution de la santé du malade. Le médecin ou l'infirmière peuvent ainsi disposer des informations générées par les capteurs à l'aide de stations de base qui sont les interfaces homme-machine du réseau de capteurs.

Cependant, le caractère autonomie de ces capteurs présente un grand handicap pour ces systèmes. En effet, ces derniers sont généralement alimentés par une source d'énergie irremplaçable. Ainsi, l'épuisement de la source d'énergie d'un nœud engendre la déconnexion de ce dernier du réseau tout entier et parfois la déconnexion d'autres nœuds aussi. D'où s'imposent les techniques de gestion d'énergie et de minimisation de la consommation afin de prolonger la durée de vie du nœud et du réseau tout entier par conséquent.

L'objectif, dans le cadre de ce projet, est de focaliser une étude sur l'autonomie de ces systèmes en effectuant une synthèse des techniques actuelles permettant de prolonger l'autonomie du réseau ou tout au moins des fonctions vitales.

Dans un second temps, nous avons étudié des environnements de simulation de réseaux de capteurs sans fils pour l'estimation des performances. En fait, peu d'environnements supportent des modèles de consommation. Ainsi, nous avons adopté et mis en place l'environnement TinyOS basé sur le système d'exploitation TinyOS et le simulateur TOSSIM. Nous avons étudié, mis en place et utilisé l'extension PowerTOSSIM afin de déduire la consommation au niveau de chaque périphérique du réseau de capteurs.

Par ailleurs, nous avons mis en place une application de validation à travers cet environnement. Cela nous a permis donc d'étudier les points clés du problème d'une part et de donner quelques éléments de réponse sur l'évaluation de la consommation sur ce type d'application d'une autre part.

Ce mémoire est structuré en quatre chapitres :

Le premier chapitre est une introduction pour les réseaux de capteurs sans fils. Il présente leurs domaines d'application, leurs architectures et leurs caractéristiques spécifiques qui font de ces systèmes un domaine en plein essor.

Le second chapitre commence par un rappel sur les parties consommatrices dans un nœud d'un réseau de capteurs. Ensuite, il présente des méthodes d'estimation et de réduction de la consommation dans un nœud.

Le troisième volet de ce rapport débute par une étude comparative des simulateurs de réseaux de capteurs sans fils afin d'en adopter le meilleur répondant à nos besoins. Ensuite, il met l'accent sur l'environnement de simulation de réseaux de capteurs tournant sous TinyOS et les différents outils qui le composent. Cet environnement permet, en effet, la simulation d'un réseau de capteurs sans fils : les liaisons radio entre les nœuds, les modèles radio, les émissions/réceptions, etc. Et le plus important est qu'il permet de modéliser la consommation énergétique.

La dernière partie de ce mémoire aborde l'implémentation d'une application en nesC, tournant sous TinyOS, sa simulation avec TOSSIM et PowerTOSSIM. Et pour valider notre

environnement, nous appliquerons quelques techniques d'optimisation de la consommation et nous illustrerons l'effet de ces optimisations sur la consommation de cette application.

Enfin, nous clôturerons ce mémoire par une conclusion générale et une proposition de perspectives pour ce travail.

Chapitre 1 : Réseaux de capteurs sans fils

1. Introduction :

Les réseaux de capteurs sans fil qui ont été le sujet de plusieurs recherches, sont devenus des équipements clé dans les applications industrielles. En effet, ces systèmes sont en plein essor. Leur fonction est généralement de surveiller leur environnement et de communiquer des données détaillées, pour une grande diversité de secteurs. Parmi les défis que doit surmonter un réseau de capteurs sans fils nous citons :

- ✓ Des sévères conditions de fonctionnement
- ✓ Des nœuds de faibles ressources matérielles faute de la taille très réduite des nœuds
- ✓ Une source d'énergie indispensable et irremplaçable puisque les nœuds sont placés dans des endroits généralement inaccessible par l'homme
- ✓ Une densité énorme du réseau pour compenser les nœuds qui tombent en panne.

Dans ce premier chapitre, nous présentons les réseaux de capteurs sans fils : leurs domaines d'application, leurs caractéristiques uniques ainsi que leurs architectures en vue de bien les connaître et maîtriser ce domaine.

2. Présentation

Un réseau de capteurs sans fils (RCSF) «Wireless Sensors Network : WSN» est un réseau Ad-hoc comportant un grand ensemble de nœuds. Ces nœuds sont conçus pour contrôler en coopération et d'une manière autonome les conditions physiques et environnementales dans différentes parties de l'espace. Les nœuds de ce réseau sont des micro-capteurs capables de communiquer entre eux via une connexion sans fils. La position de ces nœuds n'est pas obligatoirement prédéterminée. Ils sont dispersés aléatoirement dans une zone géographique, appelée champ de capture, qui définit le terrain d'intérêt pour le phénomène capté.

3. Domaines d'application :

La taille de plus en plus réduite des micro-capteurs, le coût de plus en plus faible, La large gamme des types de capteurs disponibles (thermique, optique, vibrations, etc.) ainsi que le support de communication sans fil utilisé, permettent aux réseaux de capteurs d'envahir

plusieurs domaines d'applications. Ils permettent aussi d'étendre les applications existantes et de faciliter la conception d'autres systèmes tels que la surveillance, le contrôle et l'automatisation. Les réseaux de capteurs peuvent se révéler très utiles dans de nombreuses applications lorsqu'il s'agit de collecter et de traiter des informations provenant d'environnements à accès difficiles ou même impossibles. Parmi les domaines où ces réseaux peuvent offrir les meilleures contributions, nous citons les domaines : militaire, environnemental, domestique, sanitaire, sécurité, etc. Comme exemples d'applications potentiels, nous citons :

3.1. Applications militaires :

Comme exemple d'application dans le domaine militaire, on peut penser à un réseau de capteurs déployé sur un endroit stratégique où il est difficile d'y accéder, 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. Par contre, il faut bien noter que les applications dans ce domaine nécessitent un maximum de sécurité. Or la sécurité dans les réseaux de capteurs sans fils n'est pas assurée. En fait, les algorithmes de routage sécurisés demandent plus de ressources en mémoire et traitement que ceux non sécurisé. Dans ce contexte un nouvel axe de recherche se lance : l'optimisation des algorithmes de routages sécurisés pour les réseaux de capteurs sans fils.

Figure 1. *Application militaire [Sentilles 06]*

3.2. Applications environnementales :

Comme applications environnementales des réseaux de capteurs sans fils, nous pouvons penser à des thermo-capteurs dispersés à partir d'un avion sur une forêt qui peuvent signaler un éventuel début d'incendie dans le champ de capture. Nous pouvons aussi penser à des capteurs semés avec les graines dans les champs agricoles. Ainsi, les zones sèches seront facilement identifiées et l'irrigation sera donc plus efficace. Sur les sites industriels, les centrales nucléaires ou dans les pétroliers, des capteurs peuvent être déployés pour détecter des fuites de produits toxiques (gaz, produits chimiques, éléments radioactifs, pétrole, etc.) et alerter les utilisateurs dans un délai suffisamment court pour permettre une intervention efficace. De même pour les volcans et les séismes : des capteurs peuvent être distribués sur la zone à contrôler pour prédire les habitants des régions voisines d'un déclanchement d'un volcan ou d'une éruption, comme illustré à la figure2.

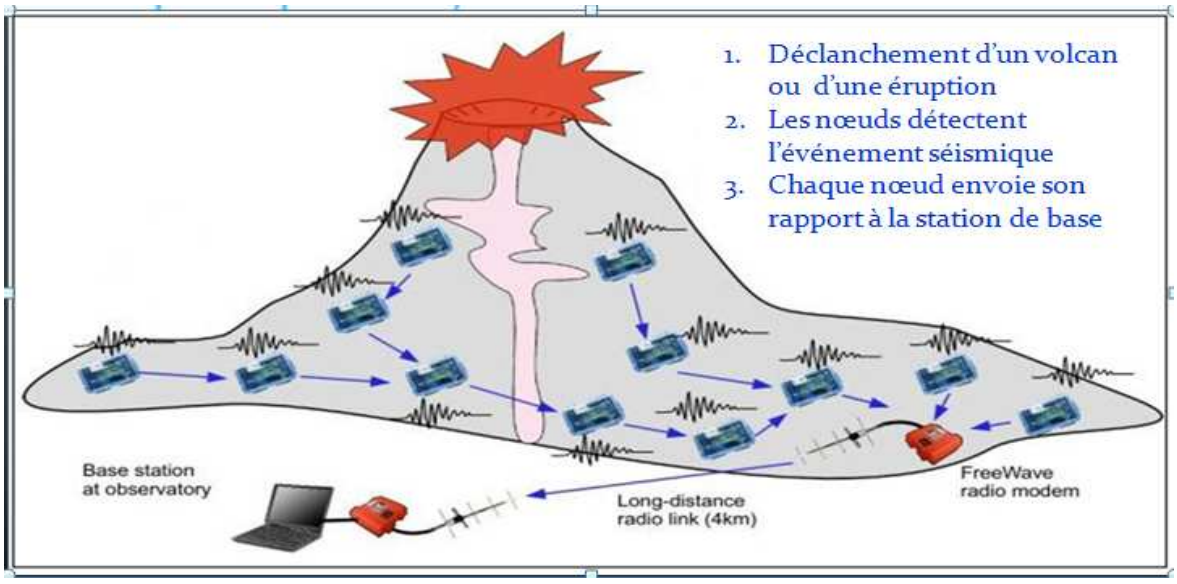


Figure 2. *Déploiement d'un réseau de capteur pour la détection des mouvements séismiques [Welsh et al. 06]*

3.3. Applications médicales :

On pourrait imaginer que dans le futur, la surveillance des fonctions vitales de l'être humain serait possible grâce à des micro-capteurs qui pourront être avalés ou implantés sous la peau. Actuellement, des micro-caméras qui peuvent être avalées existent. Elles sont capables, sans avoir recours à la chirurgie, de transmettre des images de l'intérieur d'un corps humain avec une autonomie de 24 heures. Les auteurs d'une récente étude, présentent des capteurs qui fonctionnent à l'intérieur du corps humain pour traiter certains types de maladies.

Leur projet actuel est de créer une rétine artificielle composée de 100 micro-capteurs pour corriger la vue. D'autres ambitieuses applications biomédicales sont aussi présentées, tel que : la surveillance du niveau de glucose, le monitoring des organes vitaux ou la détection de cancers. L'utilisation des réseaux de capteurs dans le domaine de la médecine pourrait apporter une surveillance permanente des patients et une possibilité de collecter des informations physiologiques de meilleure qualité, facilitant ainsi le diagnostic de quelques maladies.

3.4. Applications liées à la surveillance manufacturière :

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 ou d'une cargaison. 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 réseaux de capteurs, 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. Le système distribué pourra aussi maintenir une température homogène dans les pièces.

4. Caractéristiques d'un nœud d'un WSN :

Afin de remplir toutes les exigences de ces domaines d'application et s'étendre à d'autres domaines, les réseaux de capteurs sans-fil sont sujet de plusieurs travaux de recherches. En effet, des capteurs sont déjà utilisés pour la création de systèmes d'acquisition de données. Et pour mieux mettre en valeur les réseaux de capteurs sans fils, il vaut mieux avant de présenter leurs caractéristiques et architectures, de présenter un réseau « traditionnel, filaire » et la différence entre les deux types de réseaux.

4.1. Un nœud d'un réseau de capteurs traditionnel :

Ces capteurs sont déjà utilisés pour la création de systèmes d'acquisition de données car ils sont capables de détecter des phénomènes dans un environnement proche, de les quantifier

et de transmettre les données ainsi obtenues à d'autres éléments du système en vue de leur traitement. Les éléments mesurés sont des grandeurs physiques telles que la pression, l'humidité, les vibrations, etc. [Hubin 00]

Ces capteurs sont munis d'un élément de détection et de mesure qui détecte le phénomène physique et le transforme en grandeur analogique. Cette dernière sera numérisée via un convertisseur analogique/numérique. En plus, les dispositifs électroniques présents, tel que le CAN, nécessitent une alimentation qui peut être soit le secteur, soit des batteries, etc. Comme dispositifs électroniques, on peut aussi utiliser un conditionneur de signaux et un processeur de signal numérisé : DSP. Le conditionneur de signaux a pour but d'améliorer la qualité des mesures (filtrage de bruit, amplification du signal…). Le processeur de signal numérique permet quant à lui de traiter le signal : d'en extraire des données, de le modifier ou de l'adapter en vue de la transmission ou du stockage. Il faut noter aussi, que si le récepteur des informations est éloigné du capteur, il faut que ce dernier puisse transmettre la donnée numérisée, il peut donc y avoir également un module de communication comme un réseau câblé par exemple.

Ces réseaux sont modélisés par [Hubin 00] de la manière suivante :

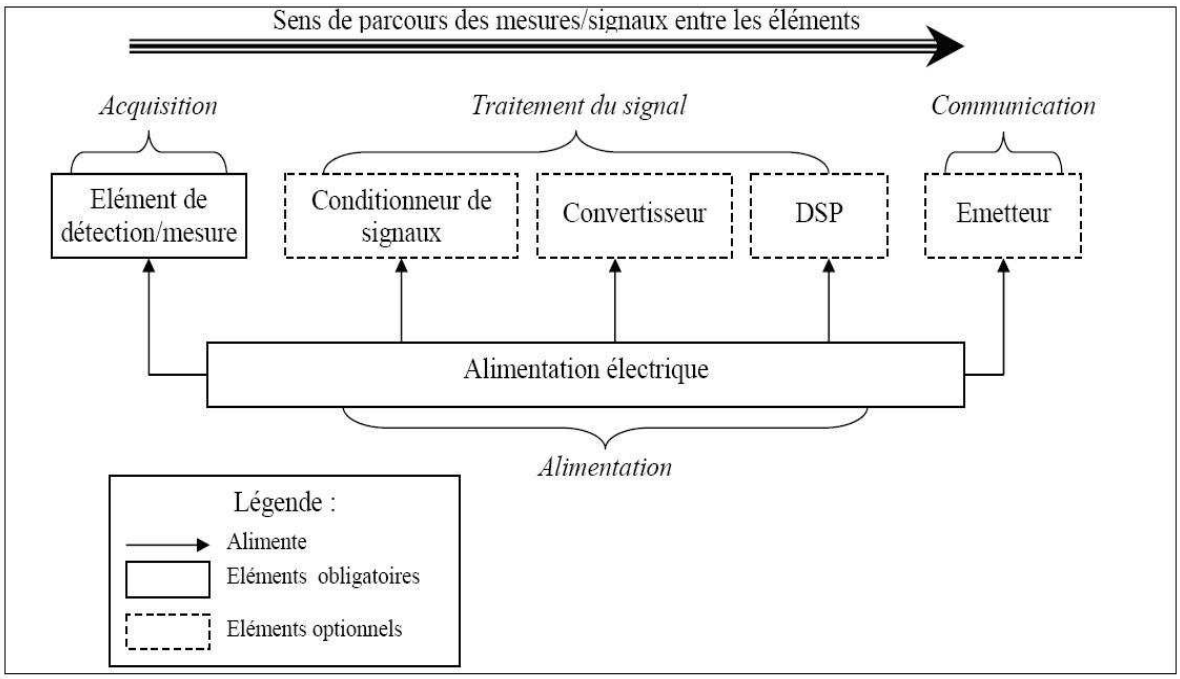


Figure 3. *Schématisation d'un capteur traditionnel*

4.2. Un nœud d'un réseau de capteurs sans fils :

D'une manière analogue au schéma d'un réseau de capteurs traditionnel, [Hubin 00] a modélisé un réseau de capteurs sans fil, et ce de la manière suivante :

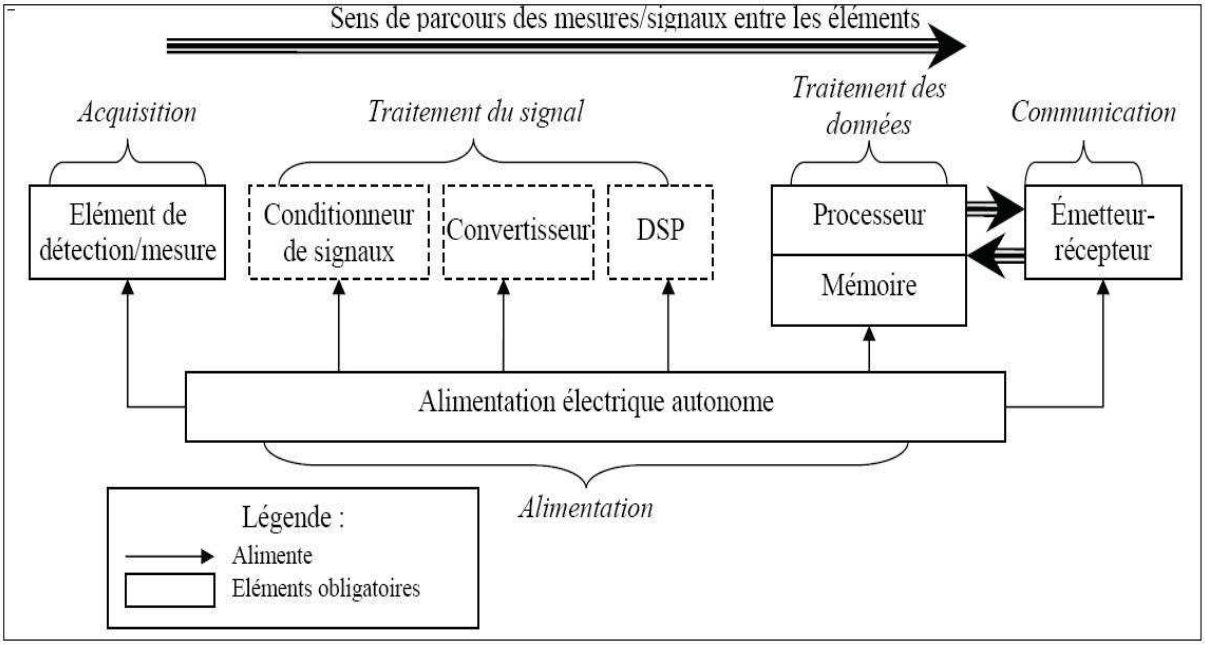


Figure 4. *Schématisation d'un nœud d'un réseau de capteurs sans fils*

Les capteurs présents dans les Wireless Sensor Networks se différencient de ceux précédemment décrits par plusieurs points.

La caractéristique principale qui les différentie est le mode de communication. Alors que les nœuds d'un réseau de capteurs traditionnel sont câblés, les nœuds d'un WSN utilisent un module de communication sans fils. En plus, les premiers se contentent d'envoyer l'information à une unité de traitement, les seconds peuvent aussi en recevoir. En effet, ils ont été conçus afin de fonctionner en réseau pour pouvoir se partager les informations collectées et les informations essentielles au maintient du réseau. Cette transmission est établie via un émetteur-récepteur.

La seconde distinction est due à la diminution des coûts de production et à la miniaturisation de la taille des composants. Bien que ces progrès s'appliquent aux deux types de capteurs décrits précédemment, pour les réseaux de capteurs sans fil, ces progrès représentent une nécessité. En effet, la taille d'un nœud peut varier de quelques centimètres à

la taille d'une poussière. Et ce ci permet d'envisager le déploiement d'un nombre beaucoup plus grand de nœuds à savoir des centaines voire des milliers. Cela permet de créer un réseau beaucoup plus dense où la fiabilité de chaque nœud n'est plus primordiale. En effet, on peut envisager de placer plusieurs nœuds au même endroit pour couvrir la même région. Ce qui permettrait à un nœud défaillant d'être remplacé par un autre.

La troisième différenciation apparaît au niveau de la source d'énergie. En effet, dans le cas des nœuds sans fils, il s'agit d'une alimentation embarquée contrairement à leurs ancêtres où la source d'énergie pouvait être fournie par une source extérieure. Il faut noter ici que la source d'énergie est une contrainte importante pour la durée de vie des capteurs car la capacité des batteries est limitée et que, de plus, il n'est pas toujours possible de les remplacer lorsqu'elles sont déchargées. Ceci est du généralement aux environnements difficilement accessibles et/ou au nombre très grand des nœuds.

La dernière différence se situe entre la phase d'acquisition des données et la phase de communication. En effet, le principal avantage des WSN est que chaque nœud possède sa propre unité de traitement. Cette dernière permet d'unir et traiter les données collectées et surtout de rapprocher le traitement de ces données de leur lieu de détection. Ce qui permet de diminuer la taille des messages à transmettre aux autres éléments du réseau et ainsi de diminuer le temps d'utilisation de l'émetteur-récepteur qui est l'élément consommant le plus d'énergie dans les nœuds. Ceci entre donc dans la politique de la minimisation de consommation.

4.3. Caractéristiques uniques d'un WSN :

D'après ce qu'on vient de voir, un réseau de capteurs sans fil a des caractéristiques uniques qui font de lui un thème de recherche en plein essor. Ces principales caractéristiques sont :

- ✓ Petite gamme de nœuds de capteurs
- ✓ Nœuds de dimension très réduite qui s'approche du mm3
- ✓ Une faible source d'alimentation et une faible capacité de stockage d'énergie.
- ✓ Sévères conditions environnementales
- ✓ Mobilité des nœuds
- ✓ Hétérogénéité des nœuds
- ✓ Les nœuds travaillent sans surveillance

- ✓ Grand risque de panne des nœuds
- ✓ Risque de rupture de communication

5. Architecture d'un réseau de capteurs sans fils :

Le nœud est l'élément principal constituant un réseau de capteurs sans fils. Et c'est lui qui spécifie les caractéristiques du réseau de capteurs sans fils et le différentie d'un réseau de capteurs ordinaire « traditionnel ».

Dans cette partie, nous essayerons de présenter l'architecture des WSN en plus de détails. Commençons d'abord par voir l'architecture globale du réseau de capteur sans fils et passons ensuite à l'architecture interne d'un nœud de ce réseau.

5.1. Architecture globale et déploiement d'un réseau de capteurs sans fils :

Le déploiement d'un réseau de capteur sans fils dans un domaine bien déterminé peut être un processus continu. En général, ce déploiement est d'autant divers que le nombre de relations établies, une à une, entre les nœuds et les équipements qu'il faut contrôler. L'architecture de globale sur laquelle se base généralement un réseau de capteur est celle d'un réseau ad-hoc comme indiquer à la figure 5.

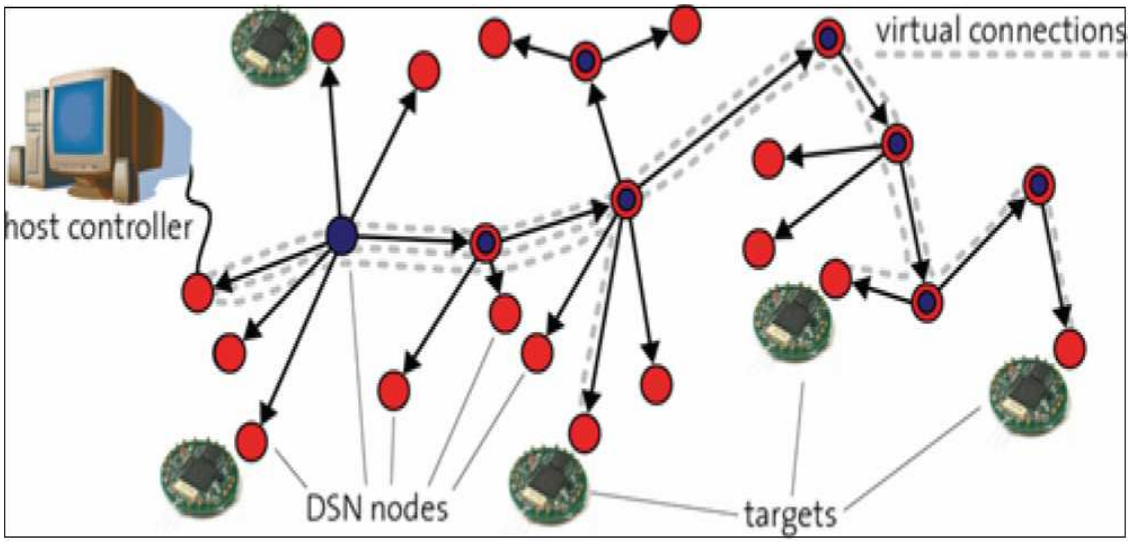


Figure 5. *Déploiement d'un réseau de capteur sans fils [Beutel 06]*

Une fois les nœuds sont déterminés et mis en place, le WSN peut commencer à effectuer ses tâches. Grâce à l'architecture ad-hoc, même les nœuds qui ne se voient pas directement peuvent communiquer par une communication virtuelle via les autres nœuds. En

plus, le fonctionnement d'un réseau de capteurs sans fils est basé sur le concept de coopération. Puisque un nœud unique ne peut transmettre qu'une information simple, donc pour résoudre un problème complexe comme le rapport de la forme, la vitesse et la direction d'un véhicule de 40 Tonnes, se déplaçant dans une surface bien déterminée. Dans ce cas, chaque capteur fournit une information simple : la vitesse par exemple et ensuite, les informations fournies par chaque capteur seront fusionnées pour obtenir un résultat de plus haute sensibilité.

Et il faut noter que les stations de base sont les composants les plus distingués du réseau de capteurs. Elles sont caractérisées par une unité de traitement, une source d'énergie et une unité de communications plus performantes. Elles fonctionnent comme un point d'accès entre les nœuds et l'utilisateur final.

Il faut noter aussi que la mise en place des nœuds dans des endroits dont l'accès est impossible peut être effectuée par jet d'un avion, mais dans ce cas il faut prendre en compte le cas de défaillance des nœuds lors de la mise en place.

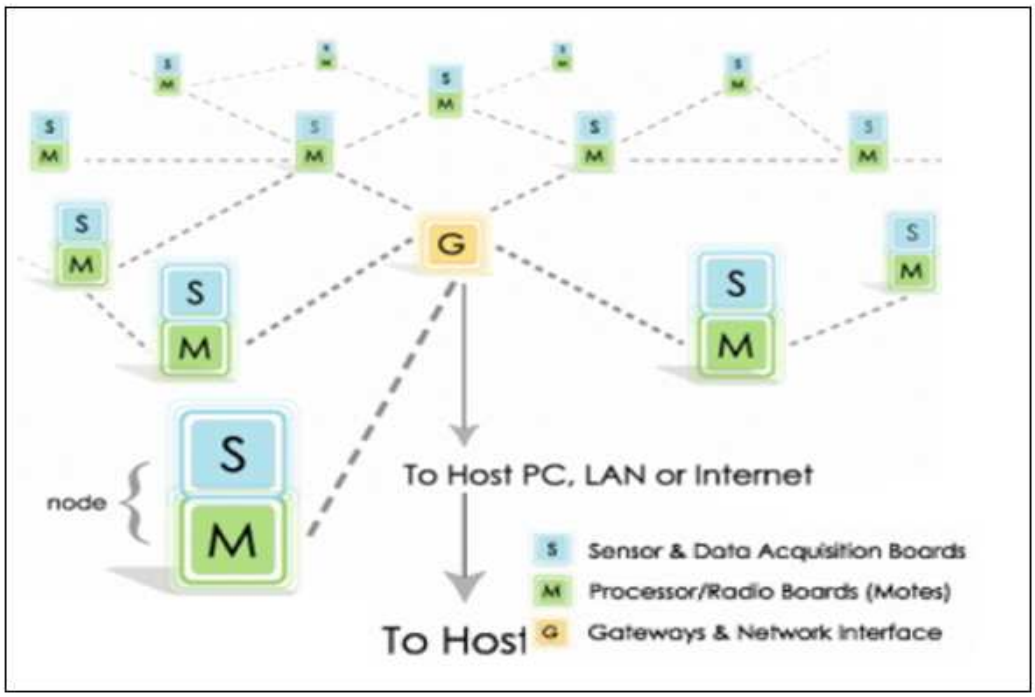


Figure 6. *Architecture globale d'un WSN : la station de base*

5.2. Architecture interne d'un nœud d'un réseau de capteurs sans fils :

Un nœud d'un réseau de capteurs peut être imaginé comme un petit ordinateur extrêmement basé sur des composants et des interfaces. Mais point de vue hardware, et comme indiqué à la figure 4, un nœud peut être décomposé en quatre unités de base :

- ✓ L'unité de capture
- ✓ l'unité de traitement
- ✓ l'unité de transmission
- ✓ l'unité de contrôle d'énergie

En plus de ces quatre unités, un capteur peut contenir, suivant son domaine d'application, des modules supplémentaires. On peut trouver à titre d'exemple un système générateur d'énergie (cellule solaire) ou bien un système de localisation (GPS). On peut aussi trouver des nœuds de taille plus grande, munis d'un système mobilisateur chargé de déplacer le nœud en cas de nécessité.

5.2.1. L'unité de capture :

Cette unité est généralement composée de deux sous unités : le capteur lui-même et un convertisseur Analogique/Numérique. Le capteur est un dispositif qui transforme l'état d'une grandeur physique observée en une grandeur utilisable. En effet, il est responsable de fournir des signaux analogiques, à partir du phénomène observé, au convertisseur Analogique/Numérique. Ce dernier transforme ces signaux en un signal numérique compréhensible par l'unité de traitement.

5.2.2. L'unité de traitement :

L'unité de traitement comprend un processeur associé généralement à une petite unité de stockage et fonctionne à l'aide d'un système d'exploitation spécialement conçu pour les micro-capteurs.

5.2.2.1. Les systèmes d'exploitation :

Les systèmes d'exploitation des nœuds de réseaux de capteurs sans fils ont un usage moins complexe que celui des systèmes d'exploitation en général. Ceci est du aux exigences spéciales des applications ces nœuds et aux contraintes des ressources matérielles pour les réseaux de capteurs sans fils.

Les réseaux de capteurs sans fils utilisent des systèmes d'exploitation embarqués comme eCos ou uC/OS [wiki 07-a]. Il existe d'autres systèmes d'exploitation embarqués qui sont conçus avec le concept temps réel, mais ceci n'est pas nécessairement demandé dans les réseaux de capteurs. Et c'est pour cette raison que la majorité des systèmes d'exploitation des réseaux de capteurs n'ont pas de support temps réel.

Le premier système d'exploitation spécialement conçu pour les réseaux de capteurs est TinyOS que nous détaillerons le fonctionnement dans une partie ultérieure. Il y a aussi d'autres systèmes qui permettent la programmation en C. Les plus connus sont : Contiki, MANTIS et SOS.

Le noyau ***Contiki*** s'appuie sur un fonctionnement événementiel, comme le TinyOS. En plus Contiki contient des protothreads qui fournissent un programme semblable à un processus léger (en anglais thread) mais en plus avec une petite mémoire [Contiki 07]. Contiki est conçu pour supporter le chargement de modules soft via un réseau. Il supporte aussi qu'on charge des fichiers ELF (Executable and Linking Format : un format de fichier informatique binaire utilisé pour l'enregistrement de code compilé) au cours de son d'exécution.

Par contre, le noyau MANTIS qui est un système d'exploitation open source, écrit en C, léger et à faible consommation énergétique, est basé sur un fonctionnement multithreading préemptif [Bhatti et al 05]. Les processus légers (threads) sont similaires aux processus puisqu'ils représentent tous les deux l'exécution d'un ensemble d'instructions du langage machine d'un processeur. Alors que chaque processus possède sa propre mémoire virtuelle, les processus légers appartenant au même processus père partagent une même partie de sa mémoire virtuelle. En fait, les processus légers sont des petits programmes qui sont exécutés ensemble pour accélérer l'exécution.

Semblant à TinyOS et à Contiki, ***SOS*** est un autre système d'exploitation à comportement événementiel. SOS permet aussi de charger des modules et il supporte la gestion dynamique de la mémoire.

5.2.2.2. L'unité de stockage :

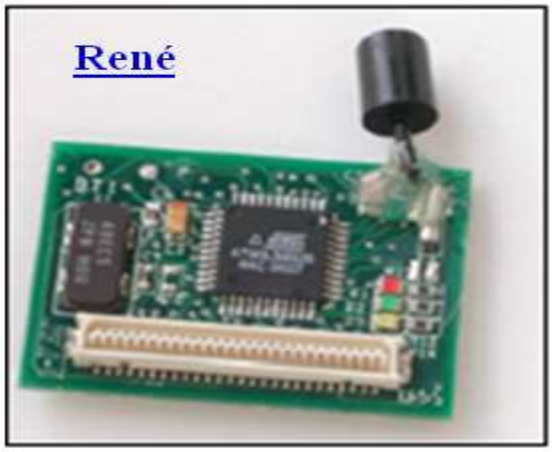
L'unité de stockage est nécessaire à l'implantation et à l'exécution d'un programme logiciel. Celle-ci est généralement de moins de 10ko de RAM et de moins de 100ko de ROM [CES 04]. Afin d'augmenter la taille de la mémoire disponible, il est également possible

d'utiliser des mémoires additionnelles de type Flash ou Eprom par exemple. On peut aussi utiliser une autre mémoire imminente : la Magnetoresistive Random Access Memory (MRAM) [wiki 07-b]. Cette mémoire, en cours de développement, peut présenter un énorme plus pour les WSN. En effet, c'est une RAM non volatile donc les données de cette mémoire sont accessibles directement et elles n'ont pas besoin d'énergie pour être conservées. Donc c'est une mémoire très rapide à l'écriture et à la lecture, non volatile, elle consomme peu d'énergie, peut être effacée et réécrite un nombre de fois illimité, son coût de fabrication est faible et elle ne subit pas les influences des radiations.

Le tableau1 présente l'évolution des capacités matérielles des nœuds tout au long des dernières années et la figure 7 présente quelques nœuds de réseaux de capteurs sans fils.

Tableau 1 . *Evolution des capacités matérielles des nœuds [BV 05]*

Nœud	WeC	Rene	Dot	Mica	Mica2	Mica2dot	Imote	BtNode
Année	1999	2000	2001	2002	2003	2003	2003	2003
Processeur (Mhz)	4				7	4	12	7
Flash (kb)	8	8	16	128	128	128	512	128
RAM (kb)	0.5	0.5	1	4	4	4	64	4
Radio (kBaude)	10	10	10	40	40	40	460	460
Type de radio	RFM				ChipCon		Zeevo BT	Ericson BT
Microcontrôleur	Amtel						ARM	Amtel
Expandable	Non	Oui	Non	Oui	Oui	Oui	Oui	Oui



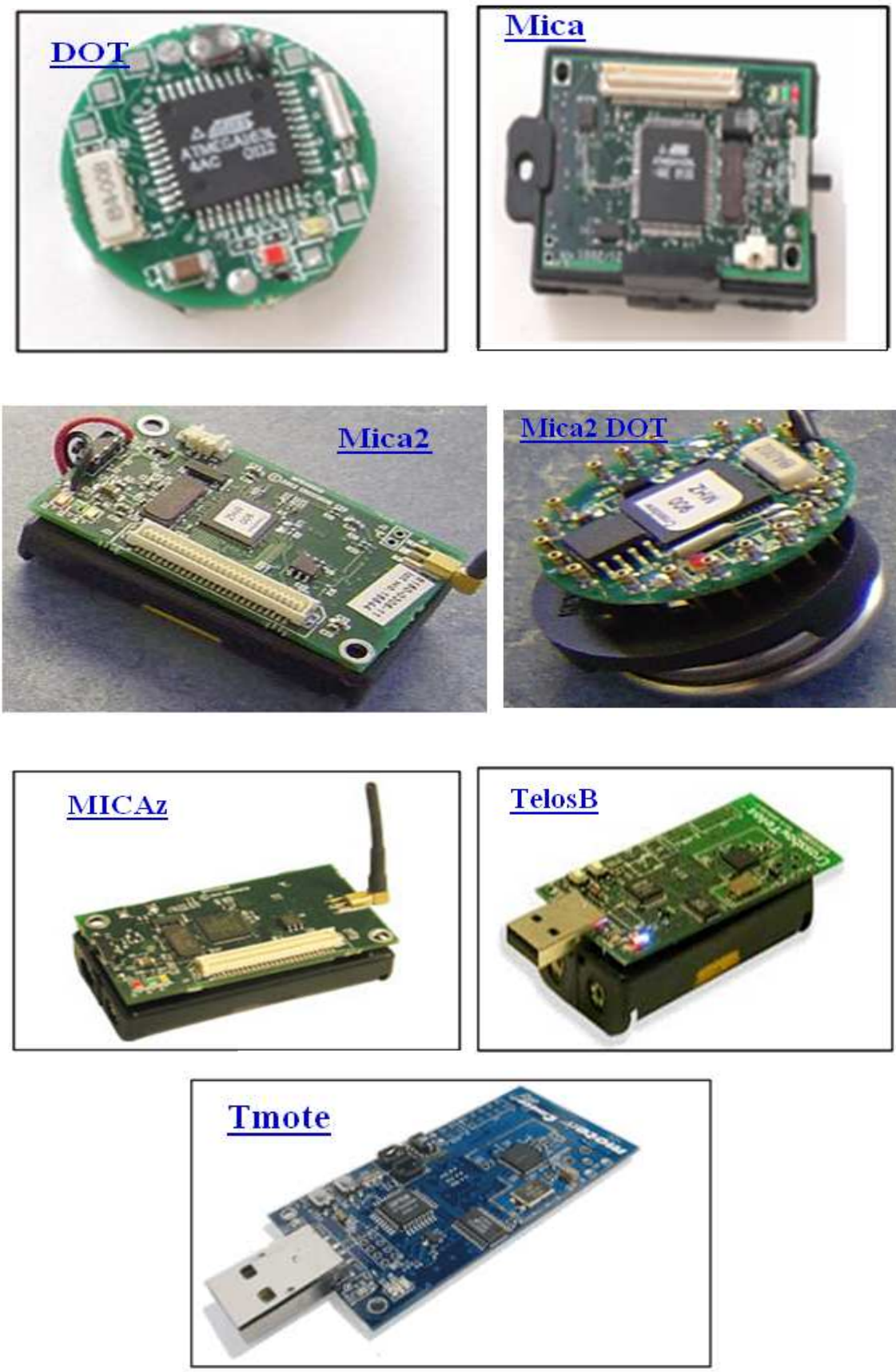


Figure 7. *Présentation de quelques nœuds*

5.2.3. L'unité de transmission :

Le module émetteur/récepteur effectue toutes les émissions et réceptions des données sur un medium sans fil. Cette unité peut être de type :

- ✓ optique
- ✓ radio fréquence.

Une onde radio se distingue d'un rayonnement lumineux par sa fréquence : quelques dizaines de kilohertz ou gigahertz pour la première, quelques centaines de térahertz pour la seconde. Les communications de type optique sont robustes vis-à-vis des interférences électriques. Néanmoins, elles présentent l'inconvénient d'exiger une ligne de vue permanente entre les entités communicantes. Par conséquent, elles ne peuvent pas établir de liaisons à travers des obstacles. Cette communication peut être effectuée grâce soit à des lasers ou des infrarouges.

Les communications de type radio fréquence ne sont pas affectées par les obstacles. Mais ils présentent d'autres défis. En effet, les unités de transmission de type radiofréquence comprennent des circuits de modulation, démodulation, filtrage et multiplexage ; ce qui implique une augmentation de la complexité et du coût de production du micro-capteur. Un autre facteur limitatif dans l'utilisation de cette unité est la portée. En effet, selon les besoins de l'application du WSN, on peut utiliser Wifi : IEEE 802.11 ou Zigbee : IEEE 802.15.4. En plus, cette unité est très consommatrice en énergie. En effet, pour qu'un nœud ait une portée de communication suffisamment grande, il est nécessaire d'utiliser un signal assez puissant. Cependant, l'énergie consommée serait importante. L'autre alternative serait d'utiliser de longues antennes, mais ceci n'est pas possible à cause de la taille réduite des micro-capteurs.

5.3. L'unité de contrôle d'énergie :

Possédant toutes ces unités électroniques, le nœud a nécessairement besoin d'une ressource énergétique (généralement une batterie) pour alimenter tous ses composants. Seulement, faute de la taille très réduite ou l'impossibilité d'accès au capteur, la ressource énergétique dont il dispose est limitée et généralement irremplaçable. L'unité de contrôle d'énergie constitue donc l'un des systèmes les plus importants. Elle est responsable de répartir l'énergie disponible aux autres modules et de réduire les consommations par application de politiques de gestion de l'énergie comme la mise en veille des composants inactifs.

Il faut noter aussi qu'il existe des batteries rechargeables par énergie solaire ou par vibrations… Ce qui permet de prolonger considérablement la durée de vie du nœud et aussi celle du réseau tout entier.

6. Conclusion :

Un réseau de capteurs sans fils est donc un réseau à systèmes embarqués autonomes, à faibles ressources qui collaborent ensemble afin de contrôler le milieu où ils étaient déployés.

En regardant de près ces réseaux, leurs domaines d'application, leurs caractéristiques et surtout leurs architectures, nous pouvons facilement voir que leur grand est handicap localisé au niveau de l'unité de contrôle d'énergie. En effet, l'énergie est la ressource la plus précieuse dans un réseau de capteurs, puisque elle influe directement sur la durée de vie des micro-capteurs et du réseau en entier. C'est pourquoi nous réservons le chapitre suivant à l'étude de la consommation en présentant les méthodes d'estimation et de minimisation de l'énergie dans ces systèmes.

Chapitre 2 : Techniques d'optimisation de la consommation

1. Introduction :

Après avoir présenté les réseaux de capteurs d'une façon générale, nous nous focalisons dans ce chapitre sur le problème de la consommation d'énergie, en termes d'estimation et d'optimisation.

Comme nous venons de voir à la figure 4 du chapitre précédent, un nœud est composé de trois unités principales qui sont alimentées par une quatrième unité : l'unité de contrôle d'énergie. Cette dernière représente la durée de vie du nœud, il faut donc bien gérer son énergie afin de prolonger l'autonomie du système. Et pour étendre l'autonomie de fonctionnement d'un système, seules deux méthodes existent : augmenter la capacité d'énergie embarquée ou diminuer la consommation du système. La première solution est un sujet de recherche relatif au domaine des batteries. Seulement, il est toujours difficile d'augmenter la capacité d'une batterie sans en augmenter le poids, le volume et le prix. La seconde aussi a entraîné plusieurs recherches dans le domaine de l'électronique et de l'informatique.

Dans ce cadre, nous commencerons dans cette partie par voir les parties consommatrices d'énergie dans un réseau de capteurs. Ensuite, nous éclaircirons la différence entre les termes : énergie et puissance. Le reste du chapitre présentera des techniques de minimisation de la consommation existantes. Ces techniques ne s'appliquent pas toutes au même niveau d'abstraction ni au même type de cible technologique. Nous expliquerons seulement les méthodes se rapprochant le plus de notre domaine de réseau de capteurs sans fils. En fait, nous illustrerons les techniques d'estimation et d'optimisation de la consommation au niveau technologique et au niveau technique, ainsi que les techniques hybrides.

2. Distribution de l'énergie dans un réseau de capteurs :

Les unités constituant un nœud du réseau étant électroniques, elles consomment donc toutes de l'énergie mais leurs consommations ne sont pas équitables. En fait, c'est l'unité d'émission réception qui est la plus consommatrice. Mais ceci ne néglige pas la consommation des autres unités.

2.1. L'unité de capture :

Cette unité fournit des signaux analogiques, à partir du phénomène observé, au CAN « convertisseur analogique numérique ». Ce dernier transforme ces signaux en un signal numérique compréhensible par l'unité de traitement. Pour réaliser ces transformations, cette unité a besoin d'énergie. Cette énergie sera proportionnelle au type d'information récoltée. En effet, la puissance consommée sera faible pour des applications simples comme la détection de température ou la lumière. Elle sera plus importante pour des applications acoustiques ou magnétiques. Et cette consommation sera beaucoup plus importante si on va récolter des images ou de la vidéo.

2.2. Unité de transmission :

Cette unité est généralement composée d'un module émetteur récepteur RF. Donc elle nécessite l'énergie pour :

- ✓ La transmission RF (radio fréquence)
- ✓ L'amplificateur de puissance
- ✓ Traitement numérique du signal en bande de base

La consommation de cette unité dépend directement de l'architecture du module RF et du protocole de communication utilisé. En fait, ce dernier est exécuté par l'unité de traitement donc on peut contrôler sa consommation via cette unité. Il faut noter aussi que l'estimation de l'énergie consommée au niveau de cette unité est très difficile à réaliser. En effet, il faut considérer tous les blocs constituants cette unité et qui sont généralement des circuits de modulation, démodulation, filtrage et multiplexage. La complexité réside dans ces circuits car pour estimer leur consommation, il faut connaître leurs architectures au niveau transistor.

2.3. Unité de traitement :

Cette unité est très consommatrice en énergie. Elle en a besoin pour fonctionner le système d'exploitation, alimenter l'unité de stockage et faire tourner les applications.

3. La consommation :

3.1. Présentation :

A ce niveau il faut bien identifier le terme consommation. En fait, ce terme regroupe les dissipations de *puissance* et d'*énergie*. Il faut noter aussi que chacun de ces deux paramètres a son importance lors de l'étude de la consommation d'un système.

La consommation de puissance est un paramètre important si l'on s'intéresse à la dissipation thermique dans un système afin de pouvoir dimensionner les circuits de refroidissement (surtout pour les applications embarquées où le système de refroidissement requiert une surface, une masse et un coût non négligeables).

De même, l'énergie est un paramètre à prendre en compte si l'on veut étudier la durée de vie des batteries. Ce paramètre est bien sûr lié à la puissance consommée par l'application mais, également au temps d'exécution. En effet, deux applications peuvent avoir la même puissance dissipée mais, avoir des consommations d'énergie différentes comme l'illustre l'exemple du Tableau 2.

Tableau 2. *Consommation de puissance et d'énergie*

Application	Puissance	Temps d'exécution	Energie dissipée
Algorithme 1	2W	10µs	20µJ
Algorithme 2	2W	25µs	50µJ

Dans ce tableau, nous voyons que l'algorithme 1 et l'algorithme 2 consomment exactement la même puissance ; nous pouvons donc dire que ces algorithmes sont identiques du point de vue de la consommation. Or, les temps d'exécution de ces algorithmes sont sensiblement différents puisque le premier est exécuté en 10µs et que le second l'est en 25µs. Cette différence de temps d'exécution a une grande influence sur l'énergie consommée par l'algorithme puisque le deuxième algorithme consomme 2,5 fois plus d'énergie que le premier à puissance égale. Par suite, nous pouvons déduire que pour les applications embarquées, la consommation de puissance est un paramètre nécessaire mais pas suffisant puisque la durée de vie des batteries dépend de l'énergie consommée.

3.2. Positionnement des techniques de réduction de la consommation :

Avant d'aborder les techniques de réduction de consommation, il faut noter que pour implémenter une application, différentes technologies s'offrent au concepteur. En effet, il peut utiliser une solution de type sur mesure (ASIC full custom), de type pré caractérisé (FPGA), de type processeur (général ou spécialisé), de type IP etc. Et en fonction du choix de la technologie, le modèle de consommation de la cible sera élaboré. En effet, plus le niveau d'abstraction de la cible technologique est bas (c'est à dire proche du niveau transistor) et plus la précision du modèle est importante mais, plus le temps de modélisation augmentera [Laurent 02]. Donc, les techniques abordées dans la suite ne sont pas toutes envisageables pour toutes les technologies.

4. Les techniques de réduction de consommation :

Il existe plusieurs méthodes qui permettent d'obtenir un système à faible consommation. La première consiste à concevoir des composants spécifiques conçus pour consommer le minimum d'énergie : ceci est réalisé à base des techniques matérielles. Alors que la seconde méthode est basée sur des techniques logicielles. Ces dernières consistent à modifier le code afin de réduire la consommation lors de leur exécution. Enfin la dernière méthode consiste à réaliser une collaboration entre le logiciel et le matériel afin d'optimiser la consommation totale du système. Elle s'appuie sur des mécanismes matériels pour diminuer la consommation mais utilise aussi le logiciel pour réaliser une adaptation dynamique de la consommation en fonction de la charge courante du système [PBCHIL 00].

4.1. Consommation au niveau technologique :

La consommation d'un circuit intégré au niveau transistor possède deux composantes : la «consommation statique» et la «consommation dynamique».

4.1.1. La consommation statique :

La consommation statique est due principalement à des courants parasites causés par la formation d'un chemin continue entre l'alimentation et la masse. Il y a quelques années, la puissance consommée liée aux courants de fuite était négligée dans l'étude globale de la consommation puisque elle était quantitativement faible vis-à-vis des autres sources de consommation. Mais avec l'apparition conjointe des nouvelles technologies submicroniques, et des systèmes embarqués, les courants de fuite sont devenus une préoccupation majeure. Ces

courants deviennent critiques quand le circuit passe beaucoup de temps en mode repos, ou bien lorsque son activité dynamique est faible [Turier 00]. Par contre, si tous les blocs sont actifs en permanence, la consommation statique reste faible par rapport à la consommation dynamique pour des technologies supérieures à 130 nm.

4.1.2. La consommation dynamique :

En régime dynamique, il existe deux sources de consommation dominantes : le courant de court-circuit et le courant de commutation. La contribution de ces sources apparaît lors de la commutation des transistors.

✓ Le courant de court-circuit résulte de l'état passant des deux transistors PMOS et NMOS au même instant, pendant un court moment, lors de la commutation d'une porte CMOS. En effet, ceci crée un chemin direct entre l'alimentation et la masse et engendre une dissipation de l'énergie.

✓ L'autre source de consommation dynamique est due à la charge et la décharge et la décharge de la capacité vue en sortie de la porte CMOS lors de la commutation. En fait, comme représenté à la figure8, l'activation d'une des entrées A_i de cette porte provoque le passage d'un courant i_c de l'alimentation V_{dd} vers la charge C_L.

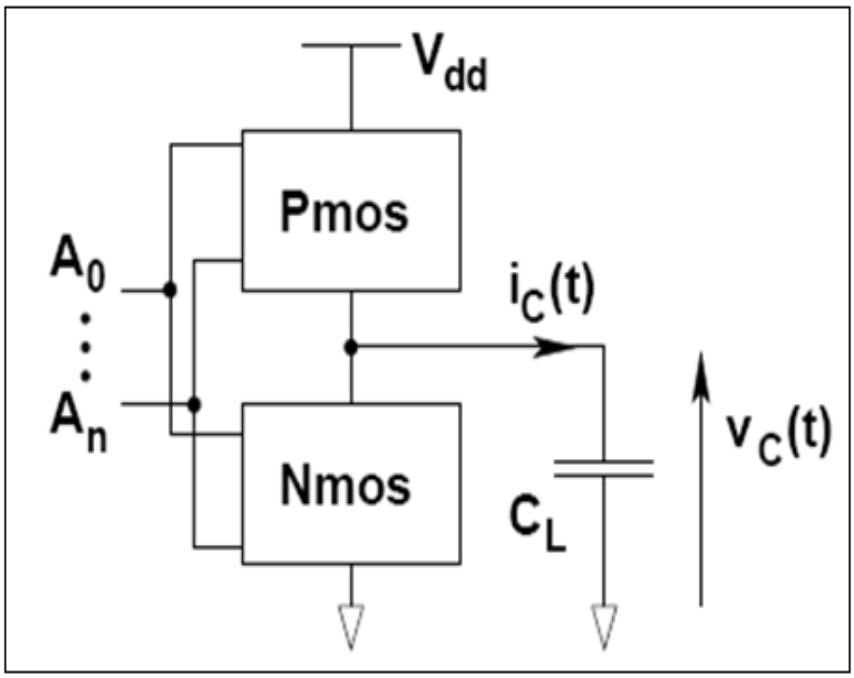


Figure 8. *Activation d'une porte CMOS*

Pendant la charge de la capacité de sortie Cl, l'énergie consommée est :

$$E_s = \int_{t_0}^{t_1} V_{i_c}(t)dt = C_L.V_{dd}^2$$

Afin de décrire ce comportement au niveau système, il faut considérer que pour un cycle d'horloge, la capacité CL est chargée et déchargée successivement un certain nombre de fois, à une fréquence f (fréquence de l'horloge). Alors la puissance dissipée devient :

$$P = E_s.f = C_L.V_{dd}^2.f$$

Et si on considère un taux de commutation du circuit α, alors la puissance moyenne dissipée devient :

$$P_{moy} = \alpha.C_L.V_{dd}^2.f$$

Cette consommation est à sommer pour l'ensemble des portes du circuit.

4.1.3. Réduction de la consommation au niveau technologique :

Le problème de la maîtrise de la consommation est très complexe. De nombreux paramètres sont interdépendants. En effet, la puissance étant proportionnelle au carré de la tension, on peut penser que le plus efficace est de la baisser [Guitton 04]. Or, l'abaissement de ce paramètre conduit aussi à une réduction de la tension seuil : V_{seuil} ce qui provoque une augmentation du courant de fuite et donc la puissance statique. Et comme notre système est, la plus part du temps, en mode veille, il est donc nécessaire de trouver un compromis entre la réduction de la tension d'alimentation et la tension seuil.

D'un autre coté, on peut réduire les capacités de sortie au niveau du layout, au niveau dimensionnement des portes et aussi au niveau architectural [Turier 00].

Il y a aussi la possibilité de tenter de réduire la fréquence d'activation au niveau architectural en partitionnant le circuit en blocs, activer ceux qui sont nécessaire à l'application et mettre le reste en veille.

4.2. Estimation de la consommation au niveau logique :

Comme nous avons vu dans la partie précédente, dans un circuit constitué de portes logiques, pour des technologies supérieures à 90 nm, la principale puissance dissipée est la puissance dynamique. Celle-ci est proportionnelle à l'activité du circuit et donc aux données. D'où, elle est en général difficile à estimer pour un système complexe faute du grand nombre de vecteurs de test nécessaires pour prendre en compte les diverses possibilités de valeurs des données.

En se basant sur l'expression de la puissance moyenne, le terme α est celui qui dépend principalement des données. A partir de vecteurs de test, il y a des outils spécifiques (SPICE, POWERMIL…) qui permettent de simuler l'architecture et d'en déduire les différentes valeurs de α au cours de la simulation. En effet, la trace obtenue permet à ces outils de déduire la consommation. Ce genre de techniques nécessite beaucoup de temps et de mémoire [Guitton 04]. Et pour devancer cet inconvénient, plusieurs améliorations ont été réalisées par des techniques consistant à éliminer les informations redondantes de la trace [WW 99], ou bien encore à réduire l'information par le compactage par blocs des activités de commutation [Benini et al. 97].

Il existe aussi d'autres approches basées sur des méthodes statistiques et probabilistes permettant l'estimation de différents facteurs influant sur la consommation.

4.3. La consommation au niveau logiciel :

Dans les systèmes à base de microprocesseur, la consommation peut être modélisée comme une fonction du logiciel (instruction) s'exécutant sur une plateforme matérielle. De plus, les techniques logicielles interviennent au niveau des outils de développement des applications afin de modifier le code à exécuter dans le but de réduire la consommation induite par l'exécution de ce code.

4.3.1. Estimation de la consommation logicielle :

L'ILPA est souvent considérée comme la première référence dans l'estimation de la consommation au niveau logiciel. En effet, la majorité des techniques d'estimation logicielle de la consommation sont basées sur l'Analyse de la Puissance au Niveau Instruction (ILPA).Cette méthode a été développée initialement par Vivek Tiwari à l'université de Princeton en 1996. Quelques années plus tard, elle a été modifiée par Chakrabarti et al de

l'Université de Tempe. Depuis, elle a été modifiée plusieurs fois afin de l'améliorer et tenir compte des progrès technologiques [Tiwari et al. 96].

Cette méthode est basée sur le fait que tout code (programme) exécuté sur une cible engendre des commutations de transistors et donc une consommation de puissance. Et puisque un code est constitué d'une suite d'instructions, il suffit de connaître la consommation de puissance ou d'énergie de chaque instruction pour pouvoir estimer la consommation du code.

Afin de déterminer la consommation d'une instruction, on répète cette instruction un grand nombre de fois dans un petit programme qu'on exécute en boucle. Ainsi, on peut mesurer le courant moyen consommé comme illustré à la figure 9.

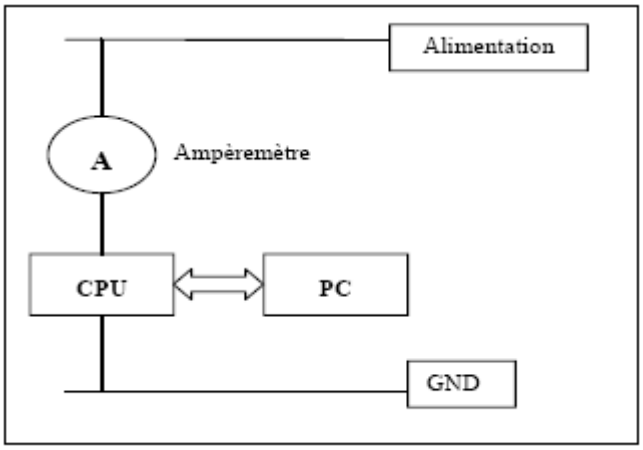


Figure 9. *Principe de mesure de la consommation*

Et de la même façon, on détermine la consommation de chaque instruction. Mais cette valeur mesurée pour chaque instruction ne représente qu'une valeur de base qui doit être modulée en fonction des valeurs d'entrée car la consommation d'une simple instruction MOV peut varier de 14% en fonction du nombre de bits à 1 de la valeur à transférer [Tiwari et al 94]

Il faut noter aussi que la consommation d'un programme n'est pas seulement la somme des consommations des instructions qu'il utilise. En effets, il a été mesuré que la consommation de deux instructions exécutées successivement était toujours supérieure à la somme de leurs consommations respectives. Cette différence, appelée consommation inter-instruction, est due au passage d'une instruction à une autre et elle dépend des deux instructions exécutées.

Il existe également d'autres facteurs pouvant intervenir sur la consommation comme l'ajout due aux ruptures de pipeline ou l'interaction du processeur avec son environnement extérieur (défauts de cache). Une fois tous ces paramètres déterminés, le modèle de consommation d'énergie du processeur est alors créé. L'estimation d'un code peut alors être effectuée en utilisant la formule suivante [Laurent 02].

$$Energie \quad programme \quad = \left(\sum E_i * n_i\right) + E_{ii} + E_{rupture} + E_{dc}$$

- E_i : Energie de l'instruction i.
- n : Nombre d'exécution de l'instruction i.
- E_{ii} : Energie inter-instructions.
- $E_{rupture}$: Energie due aux ruptures de pipeline.
- E_{dc} : Energie due aux défauts de cache.

Cette méthode permet d'obtenir des estimations précises et rapides mais elle est plutôt destinée à des cibles relativement simples à jeu d'instruction faible. D'où, l'apparition d'autres méthodes d'estimation basée sur l'APIA [Laurent 02] soit pour réduire le temps nécessaire à l'élaboration du modèle de consommation du processeur soit pour modéliser des cœurs de processeurs.

4.3.2. Techniques de réduction de la consommation au niveau logiciel :

4.3.2.1. Optimisation du code :

Comme nous venons de voir, l'estimation de la consommation due à l'exécution d'un algorithme constitue une tâche difficile. Mais, il existe toujours une relation entre le nombre d'instructions exécutées et la consommation : plus le nombre d'instruction est grand, plus la consommation est importante. Ainsi, les techniques d'optimisation du code peuvent aider à minimiser cette consommation en réduisant en autre le nombre d'accès à la mémoire externe. Actuellement, la majorité des applications sont écrites en langages « haut niveau » (langage C) puisque ces derniers permettent d'écrire des programmes facilement compréhensibles, portables et dont la génération de code machine pour une architecture donnée est automatique par un compilateur. D'un autre coté, ces langages permettent un développement plus rapide mais une qualité des codes générés moins performante que les langages « bas niveau » (assembleur). Il est toujours possible d'optimiser à la main ces codes générés mais ces derniers ne sont pas généralement très lisibles et la taille importante des applications rend leur

analyse plus complexe [PBCHIL 00]. Par contre, ceci est faisable sur des petits algorithmes, où les opérations de mémoire à mémoire peuvent être remplacées par des opérations de registre à registre, moins coûteuses en énergie.

Mais comme l'optimisation de ces codes manuellement est difficile à mettre en œuvre, on peut utiliser d'autres techniques d'optimisation automatique, intervenant au niveau du générateur de code. Ces techniques ne sont pas aussi performantes que l'optimisation manuelle mais elles sont beaucoup plus simples et sont facilement implémentées sur une chaîne de développement. Parmi les techniques d'optimisation automatiques, on peut citer les techniques d'expansion des fonctions en ligne et de déroulement des boucles.

Fonctions en ligne (inlining and loop) :

La technique des fonctions en ligne consiste à recopier le code d'une fonction à l'endroit où elle est évoquée au lieu d'utiliser une procédure d'appel de fonction. En effet, cette dernière est une opération longue et coûteuse puisqu'elle exige : le stockage de paramètres, la création d'un nouveau contexte (où placer la variable de retour), exécution de la fonction, sauvegarde de la valeur de retour, destruction du contexte et enfin destruction des paramètres. Ainsi, cette technique évite les appels de fonctions et réduit donc le risque de dépassement des capacités du matériel. D'un autre coté, il y a une grande probabilité de travailler avec des instructions sur les registres et non sur des mémoires. On obtient alors un code qui peut être exécuté sans qu'il y ait une surcharge due à l'appel de fonction et surtout moins gourmand en énergie.

Seulement cette technique augmente la taille de code puisque les fonctions en ligne sont recopiées autant de fois dans le code qu'elles sont appelées. Et ainsi, la consommation augmente aussi. En fait, un code de grande taille nécessite une mémoire de grande taille. Et une mémoire de grande taille consomme plus d'énergie. C'est pourquoi, les fonctions mises en ligne sont généralement de petite taille. En effet, il y a un compromis à trouver entre la taille du code et la diminution des appels de fonctions.

Déroulement de boucles (unrolling) :

Les boucles sont basées sur le principe de traitement et de contrôle afin de modifier les compteurs de la boucle et tester la condition de fin. D'une façon analogue aux fonctions en lignes, le déroulement de boucle consiste à recopier plusieurs fois le code correspondant au

traitement, ainsi le temps réservé au contrôle diminue par rapport au temps de traitement global de la boucle. De ce fait, le nombre d'instructions exécutées diminue et donc la consommation diminue. Cette technique a aussi le même inconvénient que l'expansion des fonctions en ligne : augmentation de la taille du code et donc le risque du besoin d'une mémoire plus grande entraînant l'augmentation des accès à la mémoire centrale, donc l'augmentation de la consommation. D'où, la recherche de compromis entre la taille du code et la taille de la mémoire est nécessaire.

Il faut noter que les deux techniques précédentes ne sont pas destinées à réduire la consommation au niveau logiciel mais plutôt à réduire le temps d'exécution des programmes. Mais une fois ce but atteint, on remarque une diminution importante de la consommation.

Les instructions spécifiques :

Il est possible d'optimiser un code selon l'architecture sur laquelle il va s'exécuter. En effet, certains processeurs offrent des instructions très particulières qui peuvent remplacer une ou plusieurs instructions. Nous citons à titre d'exemple l'instruction « LAB » concernant les DSP [Lee 95] qui permet le chargement de deux registres en un seul cycle mais à condition que les deux registres soient dans deux mémoires différentes. Cette instruction remplace la traditionnelle instruction « MOV » et permet un gain d'énergie de 34,5% à 50,55% [Lee 95]. En effet, cette méthode permet des économies significatives au niveau de la consommation. Cependant une telle optimisation ne peut être envisagée que pour une architecture bien déterminée, et ne peut pas garantir la portabilité sur une autre plateforme.

4.3.2.2. Les protocoles :

Il est également intéressant de modifier un protocole de communication pour en diminuer la consommation, par exemple en regroupant les données à émettre afin de diminuer les émissions/réceptions. On peut aussi mettre le récepteur en veille pour minimiser l'écoute du réseau inutile lorsqu'aucune émission ne lui est destinée. Or la modification de ces protocoles de communication ne se limite pas à modifier le système mais affecte des infrastructures du réseau. Donc, cette solution n'est envisageable que pour le développement de nouveaux réseaux. Nous pouvons mentionner ici le cas du « mote on chip » réalisé sur un ASIC en novembre 2006, où la consommation est réduite 5 fois par l'utilisation de nouveaux protocoles par rapport à la technologie ZigBee [Jonson 06].

4.3.2.3. Exécution distante :

L'utilisation de réseaux sans fils peut également permettre de diminuer la consommation la consommation d'un appareil embarqué en déportant certains de ses traitements sur un serveur distant [Rudenko et al. 98]. Ainsi, le système embarqué envoie ses données vers un serveur distant, se met en veille en attendant la récupération du résultat final fourni par le serveur après traitement.

Avec cette méthode, aussi, il y a un compromis à trouver entre le gain en énergie requis par l'exécution distante du traitement et l'ajout de consommation due à la transmission des données tout en tenant compte du risque de collision, des réémission des données et la force du débit.

4.4. Techniques hybrides :

Après avoir présenté quelques techniques d'estimation et réduction de la consommation au niveau technologique et au niveau logiciel, nous présentons maintenant la troisième méthode d'obtention d'un système à faible consommation : les techniques hybrides. Cette méthode consiste à réaliser une collaboration entre le logiciel et le matériel afin d'optimiser la consommation totale du système. En effet, ces mécanismes proviennent des capacités du matériel, mais les décisions d'activer ou non ces mécanismes sont prises par le logiciel puisqu'il peut effectuer des choix plus pertinents [PBCHIL 00]. Parmi ces techniques hybrides, on peut citer : la mise en veille, l'adaptation dynamique de la vitesse de la CPU que nous présenterons dans les paragraphes suivant. Mais il y a encore d'autres méthodes comme l'adaptation dynamique de la tension d'alimentation qui nécessite d'une part, un matériel capable d'ajuster sa fréquence et sa tension à la demande et d'autre part, un contrôle capable de déterminer quelle doit être sa fréquence de fonctionnement courante.

4.4.1. La mise en veille :

4.4.1.1. Présentation :

La mise en veille est un état de fonctionnement implémenté avec le système qui consiste à désactiver le fonctionnement de certains périphériques ou certaines parties des périphériques lorsque le système n'en a pas besoin pendant un certain temps. Cette technique est dite hybride puisque les mécanismes d'endormissement sont implémentés au niveau du matériel et la prise de décision pour la mise en veille s'effectue généralement au niveau du système

d'exploitation. En effet, ce dernier est le mieux placer pour effectuer cette tâche puisqu'il connaît l'ensemble des activités du système et peut donc gérer ces mécanismes d'endormissement en fonction du fonctionnement de tout le système.

4.4.1.2. Compromis gain en énergie et performance :

Généralement les périphériques implémentent des mécanismes pour supporter plusieurs états de fonctionnement : de l'activité totale à la mise en sommeil complète ou encore la déconnexion du système. Et suivant cet état de fonctionnement, la consommation du système varie. En effet, plus l'état de veille est profond, plus la consommation est minime et plus le temps de réveil sera important. L'origine de ce retard peut être de nature physique ou logique.

Le retard physique est généralement dû à une contrainte mécanique. A titre d'exemple, citons le cas d'un disque dur : sa mise en veille revient à arrêter le moteur qui entraîne le disque et qui est très consommateur d'énergie. Donc pour réveiller ce périphérique, il faut remettre les disques en rotation, ce qui demande un certain temps à cause de leurs inerties [LS 98].

Alors que le retard logique est plutôt lié à la mise en sommeil des circuits électroniques, spécialement les mémoires. En effet, à la coupure de courant, une mémoire dynamique perd toutes ses données, y compris les contextes de fonctionnement des périphériques. Sachant que pour réveiller un périphérique il faut restaurer son contexte de fonctionnement, une sauvegarde du contexte par le système d'exploitation est nécessaire avant toute mise en veille. C'est cette sauvegarde avant la mise en veille et la récupération des données au réveil qui cause le retard logiciel.

De point de vue utilisateur, ces retards son considérés comme une diminution des performances. Ainsi une politique d'économie d'énergie maximale peut causer une importante dégradation des performances. A ce niveau, il faut trouver un point optimal pour le système afin de minimiser sa consommation tout en gardant des performances acceptables.

4.4.1.3. Les politiques d'endormissement :

La technique mise en veille est basée sur des délais de garde. Et à chaque périphérique, on affecte un délai fixe. Ce délai est déclenché à la fin de l'utilisation d'un périphérique. Si ce

périphérique est activé avant la fin du délai, ce dernier est ramené à sa valeur initiale. Sinon, le périphérique passe en mode veille.

Cette méthode de délai de garde est simple à mettre en place mais il faut bien choisir les valeurs de ces délais. En effet, les performances de cette méthode dépendent fortement des valeurs de garde fixées pour chaque périphérique [PBCHIL 00]. Le problème majeur de cette méthode vient de la consommation inutile entre la dernière activité du périphérique et sa mise en veille. Pour remédier à ce problème on peut utiliser des algorithmes de prédiction [Benini et al. 98]. Ces algorithmes cherchent à déterminer les périodes d'inactivité du système (instant de fin d'activité et instant de reprise de l'activité). En effet, si l'instant de désactivation prédit d'un périphérique précède le déclanchement du délai de garde et effectivement le périphérique n'est pas en cours d'utilisation par le système, le périphérique passe immédiatement en mode veille.

Grâce à ces prédictions, le système peut se préparer à la mise en veille et la reprise du fonctionnement et ainsi, il peut éliminer les retards causés par le matériel et le logiciel.

4.4.2. Adaptation dynamique de la vitesse du CPU :

Une autre technique hybride pour la minimisation de la consommation est l'adaptation dynamique de la vitesse du processeur. En effet, le fonctionnement de ce dernier n'est pas constant, il est constitué de périodes de traitement qui requièrent une puissance de calcul variable. L'idée donc est d'adapter la puissance de traitement du processeur, au besoin actuel du système. Ainsi, on adapte la consommation du processeur aux besoins du système puisque la première est proportionnelle à la puissance de traitement. La figure 10 illustre bien ce principe. En effet, lorsqu'aucune technique d'adaptation de puissance n'est appliquée, le processeur fonctionne à vitesse maximale, SMAX, et achève les traitements avant l'expiration du temps qui leurs est imparti. En réduisant la vitesse, on met à profit ce temps pour économiser de l'énergie tout en respectant les échéances [Salhiene et al. 03].

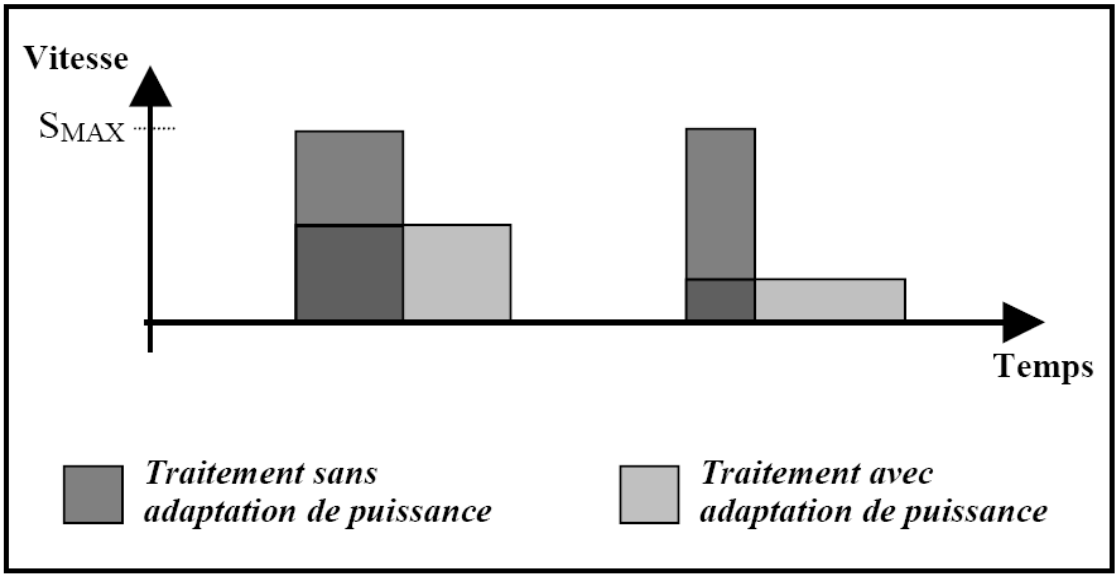


Figure 10. *Illustration de l'adaptation de puissance [Salhiene et al. 03]*

Afin de réaliser ceci, Il existe une méthode appelée adaptation dynamique de la tension d'alimentation (Dynamic Voltage Scaling ou DVS en anglais) qui consiste à agir à la fois sur la fréquence de fonctionnement du circuit et sur sa tension d'alimentation.

Cette méthode est basée sur des algorithmes d'adaptation dynamique des paramètres tension et fréquence. Ces derniers font partie du système d'exploitation et visent à ajuster quand il est nécessaire les paramètres de fonctionnement. En fait ces algorithmes se prévoient la quantité de traitement à effectuer en se basant sur le fonctionnement passé du processeur et déterminent quelle doit être le couple (fréquence, tension).

Au cadre de la DVS, plusieurs algorithmes ont été développés. L'algorithme PAST [Welch et al. 94], par exemple, divise le temps en intervalles réguliers de 10 à 50 millisecondes, considère que l'activité du processeur pendant le prochain intervalle sera identique à celle du précédent puis fait varier la vitesse du processeur en conséquence. Si le temps d'activité du processeur pendant le précédent intervalle était plus important que son temps d'inactivité, le système d'exploitation augmente la vitesse. A l'inverse, si le processeur est resté plus longtemps inactif qu'actif, la vitesse est diminuée.

Les études récentes tiennent en plus compte des contraintes temps réel mais dans ce cas, il y a un compromis à respecter : énergie consommée et tâche inachevée. Au niveau de ces algorithmes, il faut aussi prendre en compte les temps de transition d'une vitesse à une autre,

qu'ils considèrent négligeables, et la variation de la vitesse du processeur, qu'ils estiment pouvoir fixer à n'importe quelle valeur.

5. Conclusion

Avec ce chapitre, nous avons essayé de contourner le problème de la consommation énergétique en termes d'estimation et d'optimisation. Ainsi, nous pouvons maintenant appliquer ces techniques sur les nœuds des réseaux de capteur sans fils en vue d'optimiser leurs consommations et prolonger ainsi leurs durées de vie et celle du réseau tout entier.

Mais afin de valider les performances de ces techniques nous avons besoins d'un environnement qui supporte les réseaux de capteurs sans fils : l'implémentation d'application, la validation du fonctionnement et surtout la détermination de la consommation.

En fait, faute de manque de matériel et comme les réseaux de capteurs sans fils opèrent généralement dans des endroits inaccessible par l'homme, l'utilisation d'un simulateur de réseaux de capteurs sans fils est un impératif.

Dans ce cadre, nous présenterons dans le chapitre suivant les simulateurs de réseaux de capteurs, en particulier ceux qui a été adopté et mise en œuvre afin de nous permettent d'estimer la consommation.

Chapitre 3 : Environnements de simulation de réseaux de capteurs

1. Introduction

Comme beaucoup de technologies, le fonctionnement d'un réseau de capteurs sans fils peut être reproduit de façon virtuelle via un simulateur numérique. Ainsi, on peut prévoir le résultat de fonctionnement de notre système même s'il est inobservable ou difficilement observable pour toutes sortes de raisons : dimension, sécurité, coût, inexistence, etc. En conséquence, l'évolution du domaine des réseaux de capteurs sans fils a été convoyée par l'apparition de plusieurs simulateurs spécifiques à ces systèmes. En vue de mettre en place un tel environnement de simulation, une étude des simulateurs de réseaux de capteurs sans fils est donc mandatée.

Dans ce cadre, nous débuterons ce chapitre par une étude comparative des simulateurs de réseaux de capteurs existants afin de discerner leurs avantages et leurs inconvénients. Parmi ces simulateurs, nous choisirons le meilleur répondant à nos besoins dont nous présenterons les caractéristiques dans un deuxième lieu pour décrire après notre environnement de travail.

2. Les simulateurs de réseaux de capteurs sans fils

Les simulateurs de réseaux de capteurs sont nombreux et divers. Ce qui rend le choix du meilleur simulateur très difficile pour une application générique. En effet, chacun d'eux se différentie des autres par des caractéristiques qui font de lui le meilleur pour une application bien déterminée, dans des conditions bien définies. Cependant, ce même simulateur peut ne pas être le bon simulateur de cette application dans d'autres circonstances ou pour d'autres applications. Nous présentons dans la partie suivante, les simulateurs de réseaux de capteurs sans fils les plus répandus. Pour plus de détails nous présentons en annexe 1 un comparatif des simulateurs de réseaux de capteurs sans fils présent sur le site : http://www.ist-cruise.eu/cruise/sitemap

2.1 Ns-2

Ns ou « network simulator » (encore connu sous le nom de ns-2) est un simulateur de réseau à événements discrets. Il est populaire dans le milieu de la recherche par son caractère extensible, sa nature comme logiciel libre et la disponibilité d'une documentation riche sur

internet. Ce simulateur est plutôt utilisé pour la simulation du routage et de protocoles d'émission/réception et surtout pour la recherche dans les réseaux ad-hoc. En effet, ns supporte plusieurs protocoles de réseaux et permet la simulation de réseaux sans fils et câblés aussi. Mais il faut noter que ns-2 n'est pas un produit fini raffiné, mais c'est plutôt un travail en cours de recherche et de développement. En particulier, plusieurs bugs dans ce logiciel sont en continuelle découverte et matière de correction. En plus, les utilisateurs doivent valider eux-mêmes si leurs simulations ne sont pas concernées par les mauvais fonctionnements de ce simulateur [ns-2].

2.2 OPNET

OPNET (Optimum Network Performance) est un outil de simulation de réseaux très puissant et très complet. OPNET est basé sur une interface graphique intuitive dont l'utilisation et la maitrise sont relativement aisées. En fait, il dispose de trois niveaux hiérarchiques imbriqués : le network domain, le node domain et le process domain.

C'est un simulateur de réseau de capteurs sans fils, orienté objet, écrit en C et C++ avec une bibliothèque très riche mais il n'est pas disponible pour tout le monde. En fait, OPNET n'est disponible que sous une licence commerciale.

2.3 ATEMU

ATEMU est un émulateur de capteurs basé sur un type de capteur existant, le mica2. Il est donc nécessaire de compiler les applications, programmées en nesC, pour ce capteur spécifique afin de faire fonctionner ATEMU. Ce dernier possède une interface graphique nommée xatdb qui permet de visualiser les événements qui se produisent sur chaque capteur et qui simplifie les actions de débogage [BB 06].

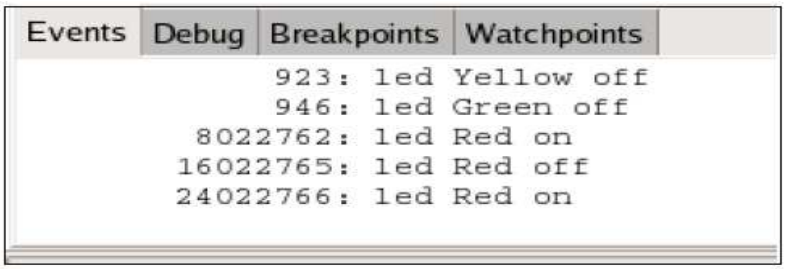


Figure 11. *Fenêtre graphique xatdb à la simulation d'une application avec ATEMU [BB 06]*

Effectivement, ATEMU permet de simuler un réseau de capteurs relativement fidèlement. En effet, via xatdb, ATEMU permet de visualiser les événements se produisant dans un capteur choisi. Mais lorsque le réseau simulé est composé de beaucoup de capteurs, il devient très difficile de visualiser tous les échanges entre les capteurs composant le réseau. En plus, la dernière mise à jour de ce simulateur date de 2004 ce qui laisse penser que les recherches pour ce logiciel n'avaient plus de suite [BB 06].

2.4 TOSSIM

TOSSIM est le simulateur de TinyOS. En effet, c'est un simulateur/émulateur à événements discrets de réseaux de capteurs munis du système d'exploitation TinyOS. Il est répandu par la disponibilité de son code ainsi que sa nature comme étant logiciel libre. TOSSIM peut fonctionner sous Linux ou Windows. Son grand avantage par rapport à beaucoup d'autres simulateurs de réseaux de capteurs est qu'il supporte différents types de nœuds de réseaux de capteurs sans fils. En effet, TOSSIM peut simuler le fonctionnement de mica, imote, mica2, mica2-dot, micaz, telos, telos-hc08, telosa, telosb et tmote. En plus, il peut simuler un nombre très grand de nœuds lors d'une même simulation. Les simulations avec TOSSIM nous donnent une idée sur le fonctionnement du réseau, les émission/réceptions, les liaisons radio entre les nœuds, les messages d'erreurs, etc.

Et ce qui nous intéresse le plus dans ce simulateur est son extension PowerTOSSIM. Cette dernière nous permet de simuler la consommation de tous les nœuds d'un réseau de capteurs sans fils.

2.5 OMNeT++

OMNeT++ est un environnement de simulation à événements discrets. Il est basé sur une architecture orientée composant, dont les modules sont écrits en C++. OMNeT a été principalement conçu pour simuler la communication dans les réseaux mais grâce à son architecture flexible et générique, il a ensuite été utilisé pour beaucoup d'autres applications. Malgré que OMNeT n'est pas un simulateur de réseau proprement dit, il est en train de gagner une large popularité comme étant une plateforme de simulation de réseaux au près de la communauté scientifique ainsi que du domaine industriel [OMNeT].

2.6 Récapitulation

Le tableau 3 récapitule les avantages et les inconvénients de chacun des simulateurs évoqués en dessus. Ce tableau se base sur des critères de comparaison généraux. Nous nous sommes basés sur ces critères pour avoir une idée sur ces simulateurs et leurs fonctionnements. Ainsi nous pouvons choisir celui qui répond à nos besoins c'est à dire un environnement qui supporte un réseau de capteurs sans fils étendu, qui peut nous donner une idée sur la consommation et avec une architecture qui consomme le minimum d'énergie.

Tableau 3. *Comparatif de quelques simulateurs de réseaux de capteurs sans fils*

	Ns-2	OPNET	ATEMU	TOSSIM	OMNeT
Licences	GPL	Commerciale		GPL	Académique et commerciale
Architecture	Orientée objet	Orientée objet	Orientée composant	Orientée composant	Orientée composant
Langage de programmation	C++	C, C++	nesC	nesC	C++/NesC
standard	802.11, Bluetooth…	***802.11, 802.16, mobile IP***	802.15.4	802.15.4	802.11
Extension pour la consommation	non	non	non	***PowerTOSSIM***	non
Nombre de nœuds max		illimité	Un seul	illimité	illimité

Ainsi, à travers ce tableau de comparaison des simulateurs de réseaux de capteurs sans fils, TOSSIM est le meilleur simulateur répondant à nos besoins. En effet, alors que les autres simulateurs ne nous fournissent pas le code source ou ne nous permettent pas de le modifier, TOSSIM et le système d'exploitation TinyOS sont des logiciels libres, gratuits et dont nous pouvons changer le code, en vue de l'améliorer et le rediffuser. Aussi, TinyOS possède un site officiel très riche de documents, de tutoriaux, de publications ainsi que des fichiers d'installation de cet environnement et même des projets réalisés autour de ce simulateur et le système d'exploitation TinyOS : www.tinyos.net.

En plus l'architecture orientée composant de l'environnement TinyOS réduit énormément la taille du code et facilite sa maintenance et son optimisation. Ainsi, en réduisant la taille du code, l'environnement est plus adapté aux faibles ressources des nœuds d réseaux de capteurs sans fils en termes de mémoire et d'énergie.

D'un autre coté, TOSSIM est le seul simulateur qui supporte la simulation de la consommation énergétique en plus des émissions radio et des messages de diffusion entre les nœuds du réseau. TOSSIM peut simuler tout un réseau de capteurs sans fils et surtout la consommation de chaque partie de chaque nœud constituant ce réseau.

Remarque importante :

Il faut noter que la nomination TinyOS désigne d'une part un système d'exploitation conçu spécialement pour les réseaux de capteurs sans fils et qui doit être installé sur chaque nœud du réseau. D'une autre part, TinyOS désigne l'environnement de simulation d'applications de réseaux de capteurs sans fils qui tournent sous le système d'exploitation TinyOS. Cet environnement est formé par le simulateur TOSSIM, le système d'exploitation TinyOS, l'émulateur Cygwin et tout un ensemble d'outils de simulation.

Dans ce cadre, nous présentons dans la partie suivante l'environnement TinyOS sur lequel fonctionne le simulateur TOSSIM.

3. TinyOS

3.1. Présentation

Figure 12. *Logo du système d'exploitation TinyOS [Berkeley 06-a]*

TinyOS est un système d'exploitation libre, conçu pour des applications embarquées fonctionnant en réseau et spécialement pour les réseaux de capteurs sans fils. Il a été initialement développé au sein de l'université de Berkeley en Californie. Le caractère open

source permet à ce système d'être régulièrement enrichie par une multitude d'utilisateurs. En fait, TinyOS a été utilisé pour une douzaine de plateformes de nœuds différents.

TinyOS est caractérisé par une architecture basée sur une association de composants, réduisant ainsi la taille du code nécessaire à sa mise en place. Cette solution est adoptée en vue de respecter les contraintes de mémoires et des ressources dont disposent les nœuds. En plus, plusieurs groupes de recherches et industriels s'en servent pour développer et tester des protocoles, différents algorithmes, des scénarios de déploiement [Berkeley 04].

Cette popularité est due, d'une part, au fait que la bibliothèque de composants de TinyOS est particulièrement complète puisqu'on y retrouve des protocoles réseaux, des pilotes de capteurs et des outils d'acquisition de données. En fait, TinyOS fournit des outils de développement et de simulation répondant aux caractéristiques spéciales des réseaux de capteurs sans fils et à leur diversité. En effet, le domaine de l'embarqué impose de sévères contraintes notamment en ce qui concerne l'espace de stockage et l'espace mémoire alloués au système d'exploitation et aux applications tournant dessus. TinyOS répond à ce problème en générant une très petite empreinte mémoire [Gay et al. 03*], celle-ci correspondant à la fusion du système d'exploitation et de l'application exécutée.

D'une autre part, les capteurs sont souvent à l'origine conçus pour détecter un phénomène et informer le réseau de son existence. Et c'est pour cette raison que TinyOS propose ainsi un modèle de programmation orienté événement ou aussi orientée composants.

3.2. Propriétés du système d'exploitation

TinyOS est basé sur quatre grandes propriétés qui font de lui un système d'exploitation bien adapté aux systèmes à faibles ressources. En fait, TinyOS est basé sur un fonctionnement événementiel. Le capteur est par défaut en état de veille, garantissant une durée de vie maximale ; il ne devient actif qu'à l'apparition de certains événements, par exemple l'arrivée d'un message radio.

Tableau 4. *Caractéristiques de TinyOS*

Propriété	***Valeur pour TinyOS***
Type	Event-driven (événementiel)
Préemptif	Non
Temps réel	Non
Consommation	Réduite

3.2.1 Fonctionnement événementiel

Le fonctionnement d'un système géré par TinyOS est basé sur la gestion des événements déclenchés. Ainsi, l'activation de tâches, leur interruption ou encore la mise en veille du capteur s'effectue à l'apparition d'événements. En effet, ces derniers ont la plus forte priorité. Ce fonctionnement événementiel est l'inverse du fonctionnement temporel (time-driven) où les actions du système sont gérées par une horloge donnée.

3.2.2 Noyau non préemptif

Un système d'exploitation est dit préemptif s'il permet l'interruption d'une tâche en cours d'exécution. TinyOS n'est pas préemptif puisqu'il n'autorise pas ce mécanisme entre les tâches. Par contre, il donne la priorité aux interruptions matérielles. Donc, une tâche ne peut pas interrompre une autre tâche, mais une interruption matérielle peut stopper l'exécution d'une tâche.

3.2.3 Temps réel

Les systèmes informatiques temps réel se différencient des autres systèmes informatiques par la prise en compte de contraintes temporelles dont le respect est aussi important que l'exactitude du résultat, autrement dit le système ne doit pas simplement délivrer des résultats exacts, il doit les délivrer dans des délais imposés [wiki 07-c]. TinyOS ne supporte pas ce concept temps réel car la contrainte du temps n'est pas nécessairement demandée dans les réseaux de capteurs.

3.2.4 Consommation

TinyOS cherche à maximiser la durée de vie du capteur et par suite celle du réseau de capteurs. En effet, il réduit au maximum la consommation en énergie en se mettant par défaut en veille et ne se réveille qu'à l'apparition d'un événement.

3.2 Allocation de la mémoire

La méthode de gestion de la mémoire par un système d'exploitation a une grande importance surtout dans un environnement aussi restreint que celui où doit fonctionner TinyOS. Ce dernier occupe un espace mémoire très faible puisqu'il ne prend que 300 à 400 octets dans sa distribution minimale. En plus de cela, il n'est nécessaire d'avoir que seulement 4 Ko de mémoire RAM libre qui se répartissent de la façon suivante :

- ✓ La pile : c'est une mémoire temporaire qu'on utilise pour l'empilement et le dépilement des variables locales.
- ✓ Les variables globales : c'est un autre espace mémoire dans lequel on stocke les valeurs qui peuvent être utilisés par plusieurs applications.
- ✓ La mémoire libre : on l'utilise pour le reste du stockage temporaire.

Ainsi la mémoire RAM du TinyOS peut être modélisée de la façon suivante :

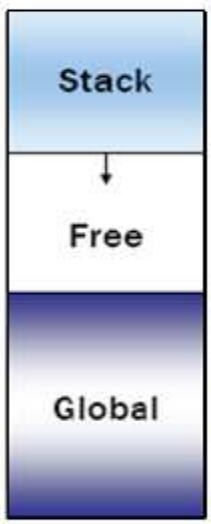


Figure 13. *Modélisation de la RAM d'un nœud avec TinyOS [Berkeley 05]*

La gestion de la mémoire avec TinyOS est caractérisée par une allocation statique de la mémoire et ne supporte pas les pointeurs de fonctions. Ceci allège beaucoup l'implémentation mais rend le système plus vénérable aux « crash » et aux corruptions de la mémoire puisqu'il n'existe pas de mécanismes de protection de la mémoire.

3.3 Le langage de programmation : nesC

Afin de tenir compte des fortes contraintes du domaine d'applications embraquées distribuées, un nouveau langage de programmation nesC a été conçu. Ce dernier est une extension du langage C offrant de nombreuses optimisations pour diminuer l'occupation de l'espace mémoire et en outre, permettant un développement simple, robuste et rapide des applications. En effet, nesC est basé sur des concepts généraux orientés composants. Il permet aussi la décomposition d'une application en modules réutilisables. NesC cible en particulier l'implémentation d'applications pour les réseaux de capteurs. C'est pour cela qu'il offre une réactivité importante vis-à-vis de l'environnement, une gestion de la concurrence et un support de communication. Nous allons présenter ce langage en plus de détails dans le chapitre suivant en exposant l'implémentation d'une application qui tourne sous TinyOS.

3.4 Les plateformes

L'objectif d'une plateforme est de visualiser et commander la simulation du réseau de capteurs avant son déploiement sur le terrain. Dans cette optique, TinyOS est prévu pour fonctionner avec une multitude de plateformes. En effet, TinyOS peut être installé à partir d'un environnement Windows (2000 et XP) ou bien GNU/Linux (Red Hat essentiellement, et d'autres distributions sont également supportées). Actuellement plusieurs versions de TinyOS sont disponibles sur son site officiel www.tinyos.net. Les versions principales sont (v.1.1.0) et (v .1.1 .15). Il y a aussi la version 2 qui vient de sortir (Novembre 2006) mais toujours en stade de test. Seulement cette version n'est pas compatible ni avec TinyDB (système d'exécution des requêtes de TinyOS et à laquelle on consacrera la partie suivante) ni avec PowerTOSSIM.

Afin d'avoir une version stable et qui répond à nos besoins, nous avons installé la version 1.1.0. et nous l'avons ensuite mise à niveau à la version 1.1.15. En plus, cette dernière supporte PowerTOSSIM, l'extension du simulateur TOSSIM et qui permet la modélisation de la consommation. Il faut préciser aussi que cette version stable fonctionne avec la version 1.2a de Cygwin et la version 1.1.2b-1 de nesC [TinyOS].

3.5.1 Cygwin

La principale plateforme de notre environnement est Cygwin. En effet c'est à travers cette plateforme que nous pouvons communiquer avec TinyOS depuis Windows.

Cygwin est une collection de logiciels libres, permettant à différentes versions de Windows de Microsoft d'émuler un système Unix. En fait, il tente de créer un environnement Unix sous Windows, rendant possible l'exécution de ces logiciels après une simple compilation.

Cygwin se compose d'une bibliothèque qui implémente les interfaces de programmation des applications (les API), des outils de développement du GNU (tels que GNU Compiler Collection et GNU Debugger) qui génèrent des tâches de base de développement de logiciel, et de quelques programmes d'application équivalents aux programmes courants des systèmes Unix.

3.5.2 Le simulateur TOSSIM

TOSSIM est le simulateur du système d'exploitation TinyOS et des applications tournant sous cet OS. C'est un simulateur/émulateur à événements discrets de réseaux de capteurs muni de TinyOS. En fait, il permet de reproduire le fonctionnement d'un réseau de capteurs sans fils virtuellement. Ainsi, au lieu de compiler une application TinyOS pour un nœud réel, nous pouvons la compiler via TOSSIM qui s'exécute sur un PC en cas d'inexistence de nœuds réels. Ceci nous permet de déboguer (corriger les erreurs de programmation), tester, et analyser nos algorithmes sur un environnement contrôlable. Le but principal de TOSSIM est donc de simuler fidèlement les applications de TinyOS.

TOSSIM possède une interface graphique appelée ***TinyViz*** qui permet de visualiser de manière intuitive le comportement de chaque capteur au sein du réseau. En plus, il a une extension PowerTOSSIM, qui permet de modéliser la consommation énergétique des différents nœuds du réseau.

3.5.3 TinyViz

TinyViz est une application graphique qui donne une vue instantanée sur le réseau et nous donne un aperçu sur les divers messages émis par les capteurs.

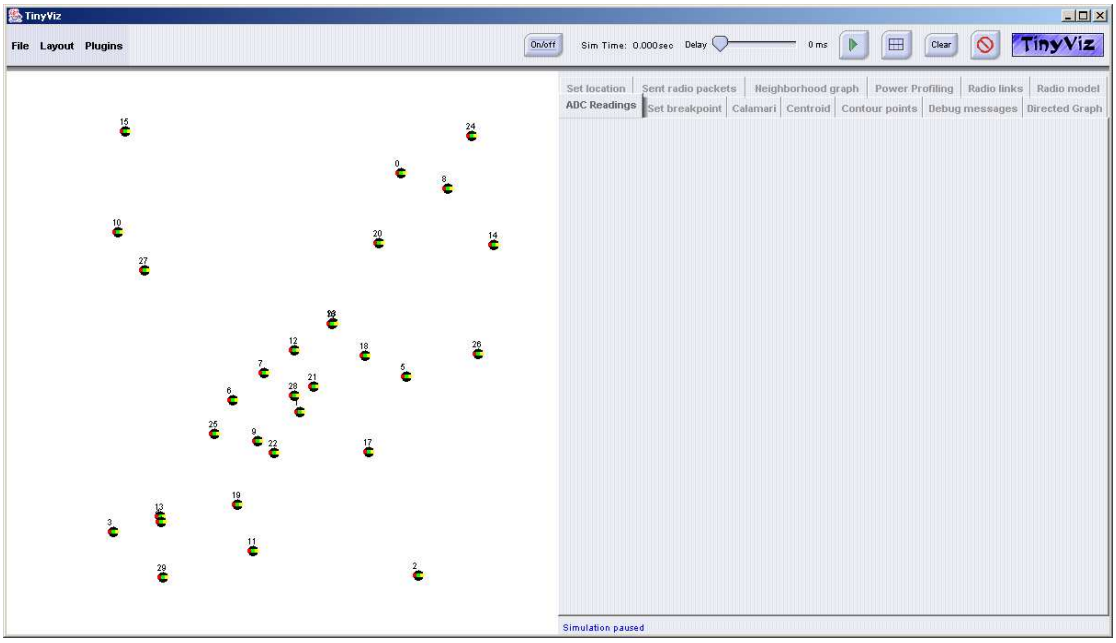


Figure 14. *Fenêtre graphique de TinyViz*

La partie gauche de cette interface montre tous les capteurs appartenant au réseau simulé. Et via les commandes en haut de l'interface à gauche, on peut contrôler la simulation. Le fonctionnement de ces commandes est détaillé comme suit :

- ✓ On/Off : met en marche ou éteint un capteur.
- ✓ Le délai du timer : permet de sélectionner la durée au bout de laquelle se déclenche le Timer
- ✓ Le bouton « Play » : il permet de lancer la simulation ou la mettre en pause.
- ✓ Les grilles : cette commande affiche un quadrillage sur la zone des capteurs afin de pouvoir les situer dans l'espace.
- ✓ « Clear » : il efface tous les messages qui avaient été affichés lors de la simulation.
- ✓ Le bouton de fermeture : il arrête la simulation et ferme la fenêtre.

En dessous de ces commandes, il y a plusieurs onglets contenant chacun un plugin qui permet de visualiser la simulation. Par exemple, le plugin « Radio links » permet de visualiser graphiquement par des flèches, les échanges effectués entre deux capteurs (unicast) ainsi que les messages émis par un capteur à l'ensemble du réseau (broadcaste) [Levis et Lee 03].

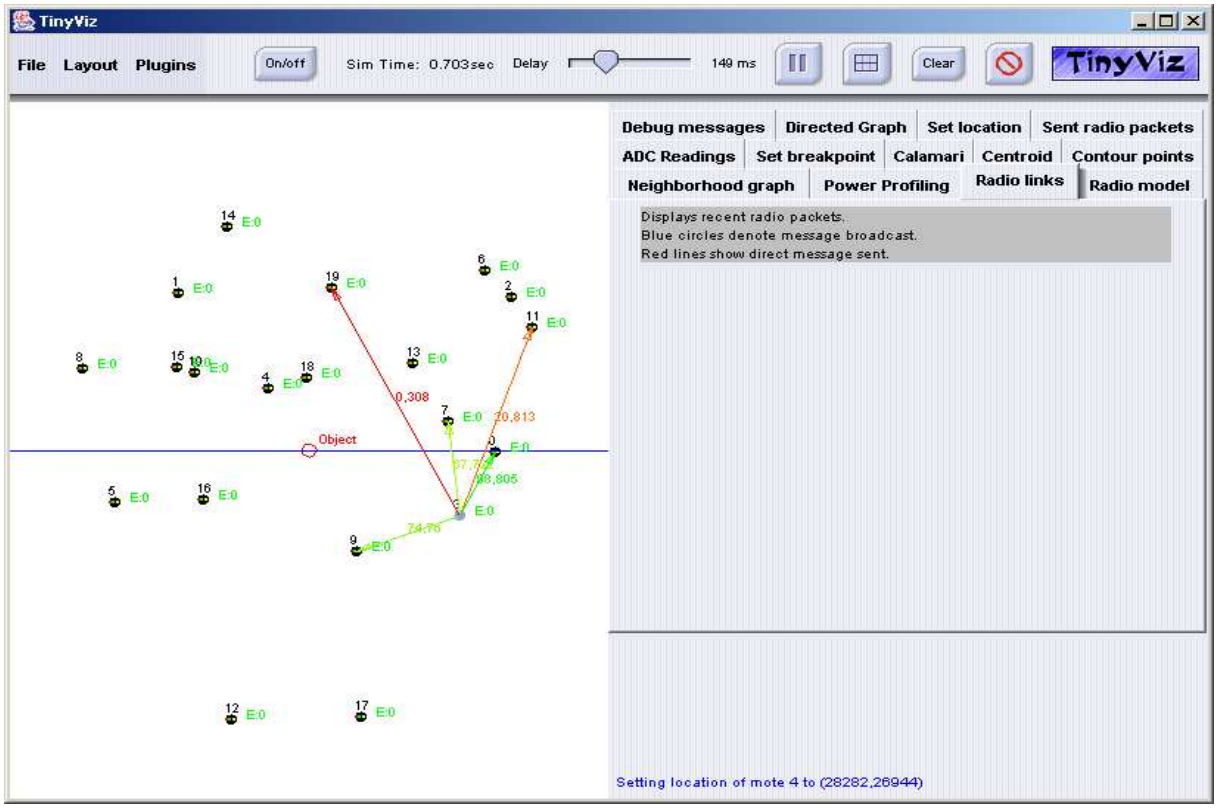


Figure 15. *Visualisation des messages radios (broadcaste, unicast)*

3.5.4 TinyDB

TinyDB est un système d'exécution de requêtes conçu pour extraire des informations d'un réseau de nœuds travaillant avec TinyOS. Seulement, TinyDB n'est pas compatible avec les versions récentes de TinyOS. La dernière combinaison testée était TinyDB 1.1.3 avec TinyOS 1.1.7 (tous les deux libérés en juillet 2004). Certains utilisateurs ont rapporté, cependant, que TinyDB 1.1.3 a fonctionné avec TinyOS 1.1.10 mais nesC exigé est le 1.1.2a plutôt que le nesC 1.1.2b. [Berkeley 06-b].

Lors de nos manipulations avec TinyOS, nous avons pu interfacer TinyDB avec TinyOS1.1.15, Cygwin1.2a et nesC-1.1.2b-1.

TinyDB ne requière pas d'écrire un code en C pour les capteurs comme l'exigent d'autres solutions de traitement de données avec TinyOS. Au lieu de ça, il procure une simple interface, similaire à celle de SQL, pour simplifier l'extraction de données et offrant plus de paramètres comme le taux avec lequel les données devraient être régénérées.

En envoyant une requête, TinyDB collecte ces informations des différents nœuds dans son environnement, les filtres, les associe ensemble et les fait passer à un PC. Ces traitements

sont effectués via à des algorithmes d'exécution en réseau puissants et efficaces. Le but principal de TinyDB est de faciliter la tâche du programmateur en évitant d'écrire du code de bas niveau pour les mécanismes et les interfaces des nœuds du réseau.

Pour utiliser TinyDB, nous devons tout d'abord installer ses composants de TinyOS sur chaque nœud du réseau. TinyDB fournit une simple interface de programmation java (Application Programming Interface ou API) qui permet de définir la manière dont un composant informatique peut communiquer avec un autre. Via cette interface, nous pouvons envoyer des requêtes et collecter des données.

D'un autre coté, TinyDB contient un support pour le simulateur de TinyOS, TOSSIM. Et comme nous n'avons pas de hardware, nous allons recourir à cette solution pour tester TinyDB. Afin d'exécuter ce dernier sur TOSSIM, nous compilons d'une part le code source java et d'une autre part, le code de l'application implémentée sur le nœud. En plus, on doit passer un indicateur spécial pour lancer l'application java de TinyDB. En fait, en travaillant sur l'environnement Cygwin, on peut interfacer TinyDB au simulateur TOSSIM via deux fenêtres Cygwin : la première pour l'exécution de l'application avec TinyOS et la seconde pour lancer l'interface de programmation java.

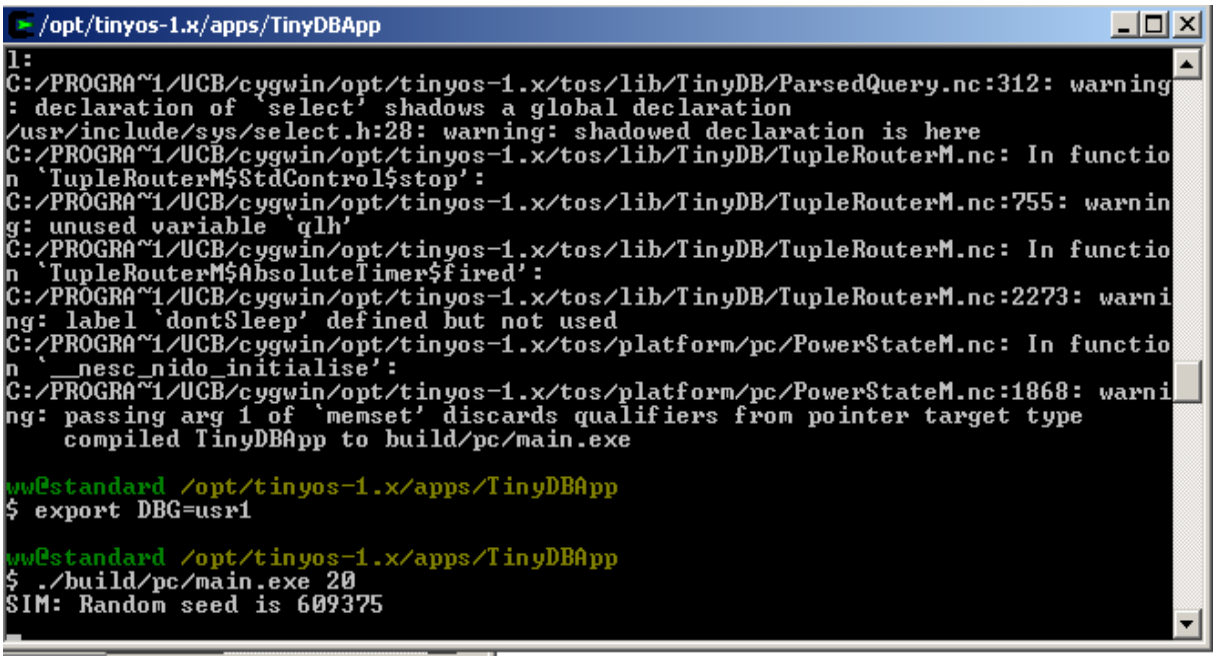


Figure 16. *Lancement de la simulation*

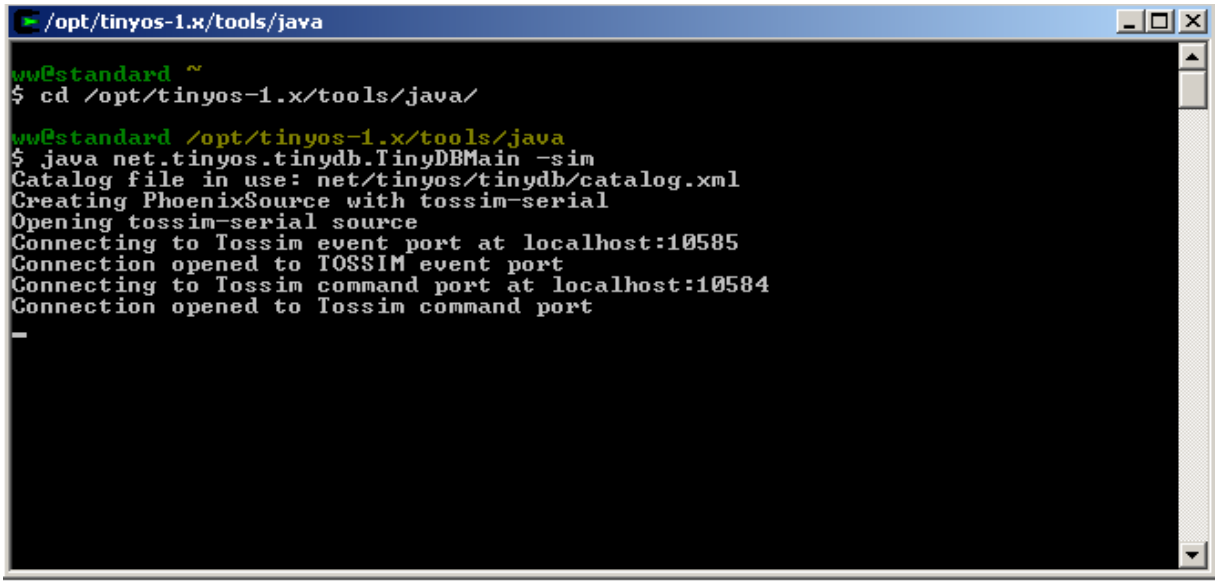


Figure 17. *Lancement de l'interface API*

Ainsi, on obtient l'interface de TinyDB : figure18.

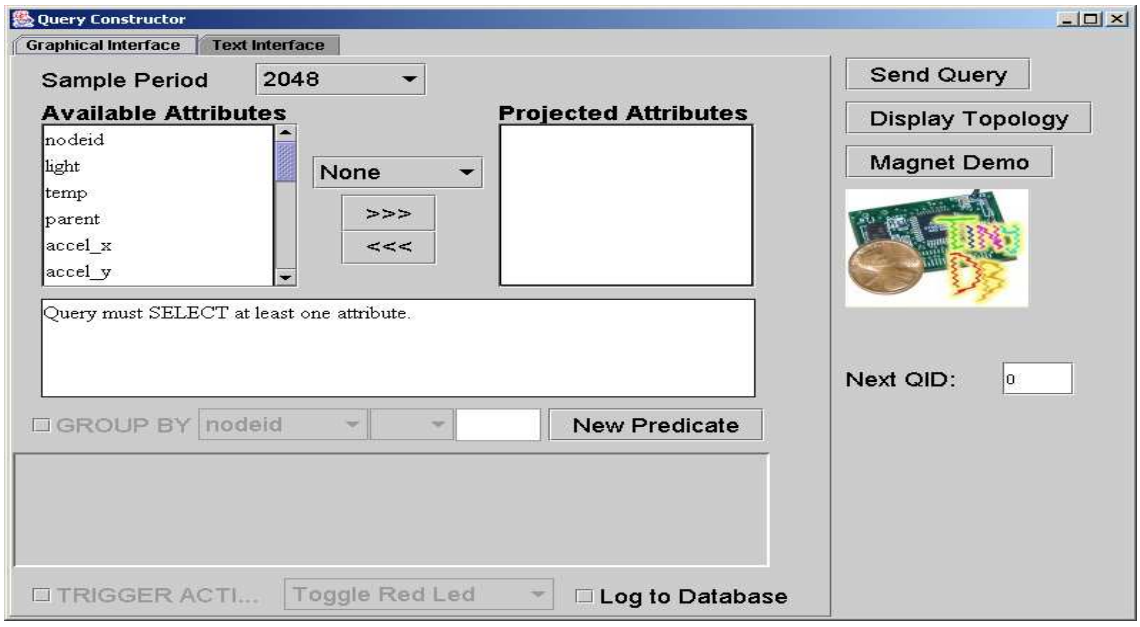


Figure 18. *Interface TinyDB*

Via cette interface, on peut choisir parmi les attributs à gauche (la lumière, le temps…) ceux qu'on veut utiliser pour lancer la requête. En envoyant cette requête, une autre fenêtre figure19, visualisant la lumière, apparait. Et comme réponse à cette requête, une nouvelle fenêtre (figure 20) apparaît en visualisant l'évolution de la lumière en fonction du temps.

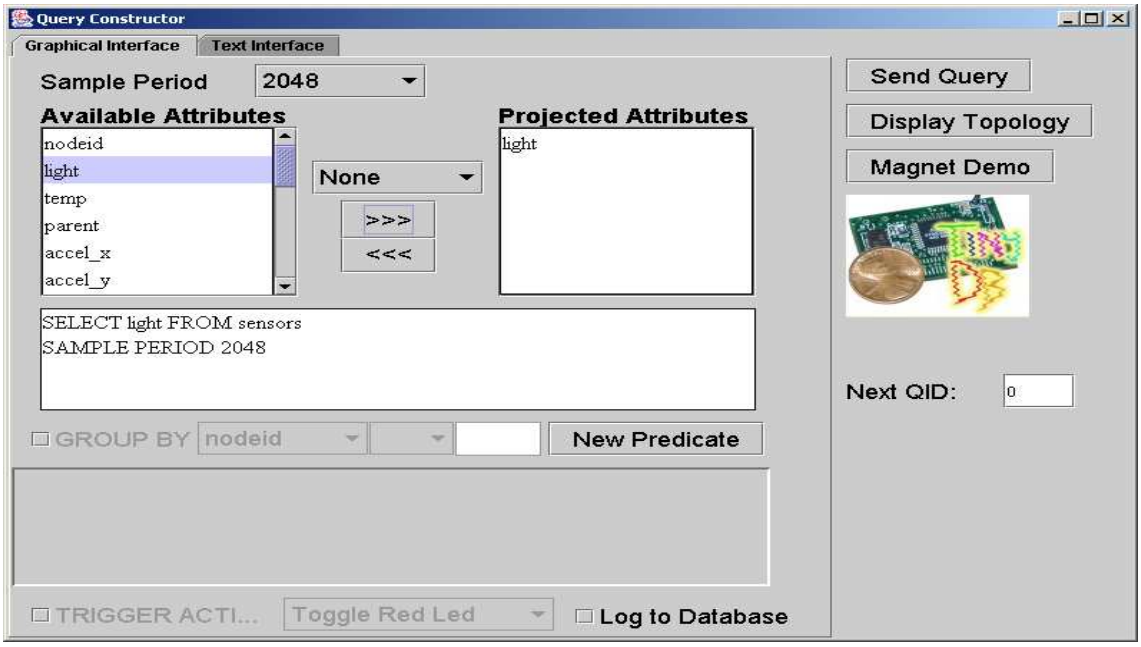


Figure 19. *Envoie d'une requête « light »*

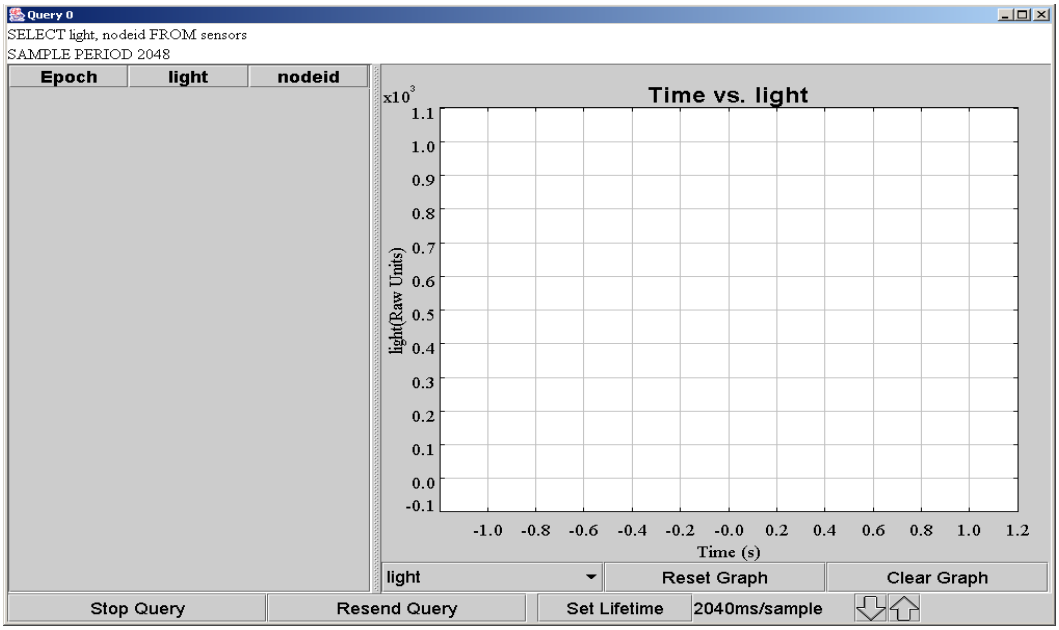


Figure 20. *Réponse à la requête « light »*

En plus de la fenêtre d'envoie de requêtes, il y a aussi une autre fenêtre pour le contrôle des nœuds : figure 21. Cette fenêtre, nous permet d'ajouter d'autres attributs ou modifier les paramètres des attributs…

Figure 21. *Fenêtre de commande des nœuds*

Ainsi, avec TinyDB nous pouvons lancer des requêtes sur notre réseau de capteurs sans fils permettant d'extraire les informations des nœuds de ce réseau.

3.5.5 PowerTOSSIM

La dernière partie de ce chapitre présentera une toute dernière nouveauté de l'environnement TinyOS : PowerTOSSIM. Les travaux de recherche sur cette extension ont débuté depuis 2003 mais le produit fini n'a été disponible qu'en décembre 2006.

3.5.5.1 Présentation

PowerTOSSIM est une extension du simulateur TOSSIM de TinyOS qui permet de modéliser la consommation énergétique d'une application. En fait PowerTOSSIM permet de déterminer la consommation énergétique de chaque périphérique de tous les nœuds d'un réseau de capteurs sans fils munis de TinyOS en se basant sur la formule de l'énergie suivante :

$$E = V * I * t$$

Avec

V : la tension fixe qui alimente le périphérique

I : la quantité de courant consommée

t : la durée d'accès à chaque périphérique pendant le temps de la simulation.

3.5.5.2 Le modèle d'énergie :

Ainsi pour déterminer la puissance, il faut déterminer les trois termes de la formule précédente. Le premier terme V est facile à déterminer c'est la tension fixe qui alimente le capteur. Cette tension peut être déterminée expérimentalement ou tirée directement de la fiche technique du nœud. Mais ce qui est plus compliqué à trouvé est le courant I et le temps t. En effet, le courant I dépend fortement du matériel, c.à.d. du type de nœud utilisé puisque tous les types de capteurs ne consomment pas le courant de la même manière. En plus tous les bocs d'un même nœud ne consomment pas tous la même quantité de courant. Il fallait bien donc établir des modèles d'énergie pour chaque type de nœud. Ce modèle est déterminé expérimentalement via un oscilloscope. En fait, le modèle d'énergie précise exactement la quantité de courant consommé par chaque périphérique du nœud et dans tous les états possibles que peut prendre ce périphérique. Actuellement, Il n'existe qu'un seul modèle d'énergie, établi pour le mica2, et qui est illustré au tableau 5 [Shnayder et al 03].

Tableau 5. *Modèle d'énergie du mica2 [Shnayder et al 03]*

Mode	Current	Mode	Current
CPU		**Radio**	
Active	8.0 mA	Rx	7.0 mA
Idle	3.2 mA	Tx (-20 dBm)	3.7 mA
ADC Noise Reduce	1.0 mA	Tx (-19 dBm)	5.2 mA
Power-down	103 μA	Tx (-15 dBm)	5.4 mA
Power-save	110 μA	Tx (-8 dBm)	6.5 mA
Standby	216 μA	Tx (-5 dBm)	7.1 mA
Extended Standby	223 μA	Tx (0 dBm)	8.5 mA
Internal Oscillator	0.93 mA	Tx (+4 dBm)	11.6 mA
LEDs	2.2 mA	Tx (+6 dBm)	13.8 mA
Sensor board	0.7 mA	Tx (+8 dBm)	17.4 mA
EEPROM access		Tx (+10 dBm)	21.5 mA
Read	6.2 mA		
Read Time	565 μs		
Write	18.4 mA		
Write Time	12.9 ms		

Ce modèle a été établi pour les nœuds de réseaux de capteurs sans fils de type mica2 et les mesures ont été prises via micasb sous une alimentation de 3V. D'après ce tableau, nous pouvons voir la consommation de la CPU sous les 7 modes de fonctionnement du microcontrôleur ATMEL, Atmega128L du mica2. Nous pouvons remarquer aussi dans ce

tableau la consommation importante de l'unité d'émission réception par rapport aux autres unités. Ce modèle d'énergie permet de fournier des informations sur la consommation de l'unité de traitement, l'unité d'émission/réception, l'unité de détection (sensor board), les diodes et l'accès à l'EEPROM.

Remarque :

Nous pouvons remarquer via ce tableau la consommation importante de l'unité d'émission réception par rapport aux autres unités

3.5.5.3 Architecture et implémentation

Une fois ce modèle d'énergie établi, il faut maintenant déterminer le temps d'accès à chaque périphérique en précisant son état de fonctionnement. L'approche adoptée par l'équipe de PowerTOSSIM pour déterminé le temps d'exécution de chaque périphérique de chaque nœud est basée sur l'implémentation d'un composant répondant à cette fonction : *power state.* Power state est un composant de PowerTOSSIM qui détecte l'état de fonctionnement de chaque périphérique de l'application TinyOS et transmet un message pour chaque composant de cette application précisant l'état de fonctionnement du périphérique. Ce message est ensuite combiné avec le modèle d'énergie précédemment établi pour donner une information détaillée sur la consommation énergétique de tous les nœuds du réseau simulé. En effet, PowerTOSSIM permet de déterminer la consommation de chaque périphérique de chaque nœud en milli joule. L'architecture de Power TOSSIM est peut être ainsi représentée par la figure 22.

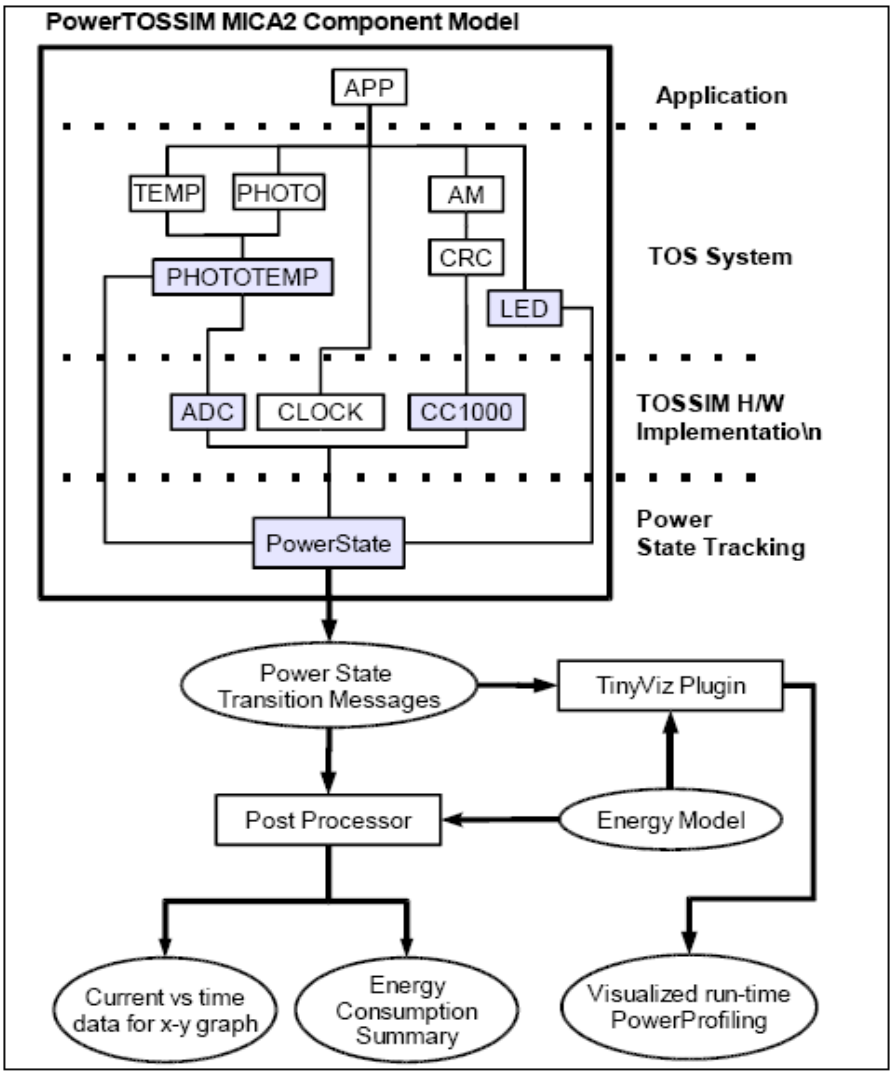


Figure 22. *Architecture de PowerTOSSIM [Shnayder et al 03]*

En haut de cette figure, nous voyons les composants formant le nœud mica2. Ces composants sont liés au composant PowerState qui transmet le message contenant l'état de fonctionnement de chaque composant du mica2. Selon le type d'affichage désiré de la consommation, ce message va être envoyé au composant fixé juste avant la simulation. PowerTOSSIM permet trois types d'affichage :

- ✓ TinyViz : nous permet de visualiser un profil de la consommation énergétique des nœuds du réseau au cours de la simulation.
- ✓ Une consommation en milli joule de chaque périphérique de chaque nœud du réseau pendant une période fixée avant le commencement de la simulation.
- ✓ Une caractéristique du courant en fonction du temps.

La figure 23 présente la simulation de l'application « Blink » pour un réseau de 10 nœuds mica2 pendant 60 secondes. Les résultats présentés sont en milli joule.

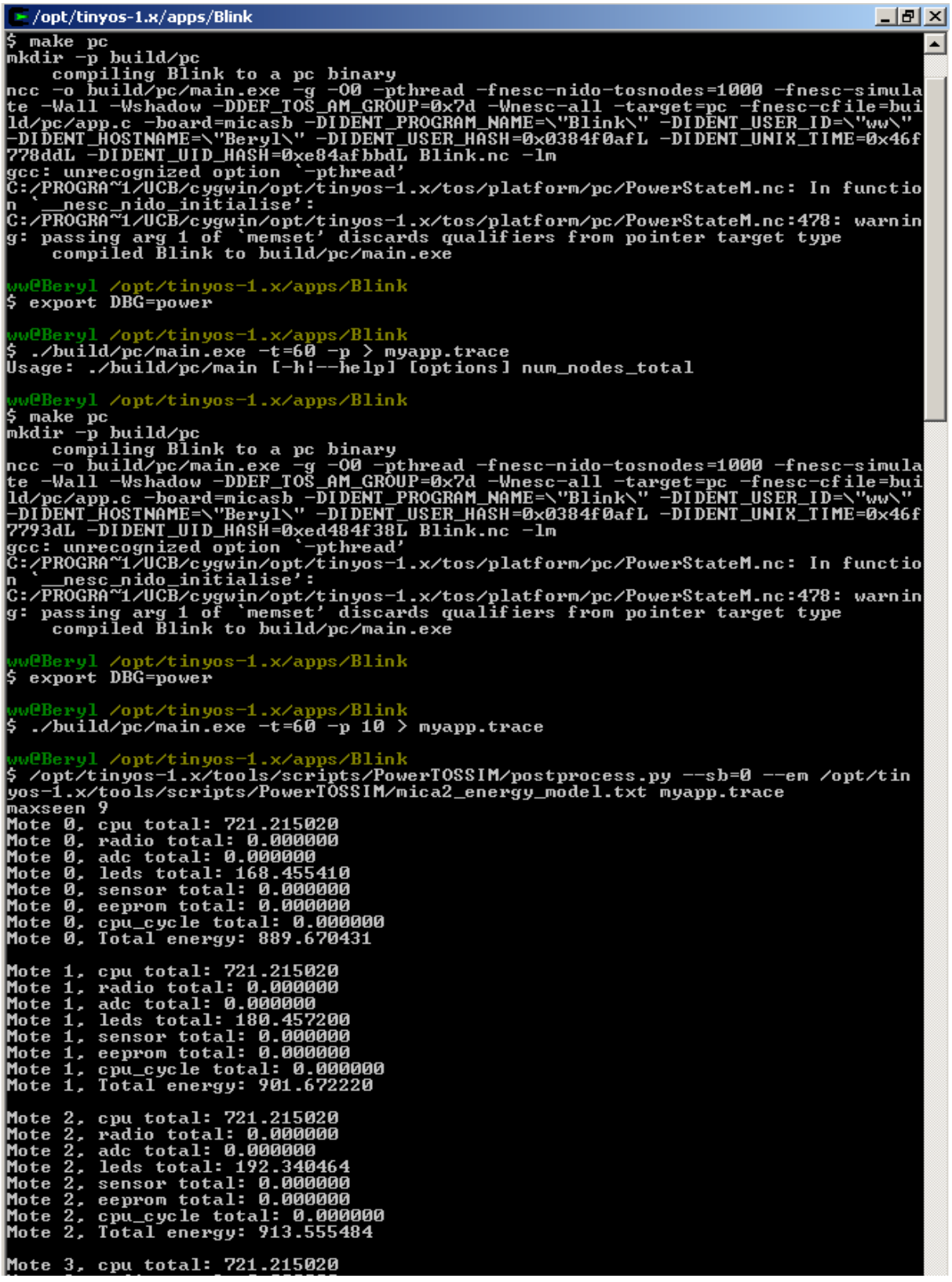


Figure 23. *Simulation de la consommation avec PowerTOSSIM*

4 Conclusion :

Nous avons effectué dans la partie précédente une étude comparative de simulateurs de réseaux de capteurs sans fils afin d'en choisir le meilleur environnement répondant à nos besoins : c.à.d. un environnement qui supporte un réseau de capteurs sans fils étendu, qui consomme le minimum d'énergie et qui peut nous donner une idée sur la consommation. L'environnement répondant à ces critères est TinyOS, pour ce nous avons ensuite présenté cet environnement, son système d'exploitation TinyOS, son simulateurs TOSSIM et les extensions : TinyDB et PowerTOSSIM.

Nous passons dans la partie suivante à l'implémentation d'une application tournant sous ce système d'exploitation et qui servira à la validation de cet environnement. En fait, grâce à cette application nous pouvons appliquer des techniques d'optimisation de la consommation et via PowerTOSSIM nous pouvons voir l'effet de ces techniques sur la consommation. Et dans une seconde étape nous utiliserons cette application et PowerTOSSIM pour voir la variation de la consommation d'un nœud en fonction du nombre des nœuds dans un réseau.

Chapitre 4 : Développement d'une application

1. Introduction :

Afin de valider notre environnement constituée du système d'exploitation TinyOS, son simulateur TOSSIM et PowerTOSSIM, nous avons implémenté une application nommée « MaTemperature ». Cette application répond au cahier de charge suivant :

On distingue deux types de nœuds :

- ✓ Un nœud principal responsable de la récupération de la température des nœuds placés dans sa zone de couverture et du calcul de la moyenne de ces prélèvements.
- ✓ Des nœuds secondaires responsables uniquement de la récupération de la température dans leurs entourages et de l'envoie de cette information au nœud principal.

A travers ce chapitre, nous allons présenter le langage de programmation de cette application et les concepts de base de ce langage. En un second temps nous illustrerons les résultats de la simulation de cette application avec TOSSIM et PowerTOSSIM ainsi que les résultats obtenus suite à l'application de techniques d'optimisation au niveau de code et la caractéristique de variation de la consommation d'un nœud en fonction du nombre du nœuds dans le réseau.

2. Le langage de programmation nesC

Comme nous l'avons mentionné dans le chapitre précédent, nesC est une extension du langage C qui est orientée composant. Présentons dans cette partie ce langage en plus de détails.

2.1. Présentation des concepts de base

Le langage nesC est caractérisé par une architecture basée sur des composants communicants entre eux via des interfaces bidirectionnelles afin de former un exécutable. Ainsi, grâce à cette architecture basée composants, on peut créer une application juste par assemblage des éléments strictement nécessaires soit au niveau système soit au niveau applicatif [Gay et al 03].

Chacun de ces composants représente un élément matériel qui peut être utilisé par n'importe quelle application. En plus, la bibliothèque de composants de TinyOS est particulièrement complète puisqu'on y retrouve des protocoles réseaux, des pilotes de capteurs, des outils d'acquisition de données : des entrées/sorties, des timers, des diodes (LEDs), des convertisseurs (ADC), des potentiomètres (Pot)… Ces derniers couvrent les parties hardware et software d'un capteur.

Il ne reste alors au concepteur que créer ses propres composants, c'est-à-dire ceux répondant aux nécessités de sa propre application et les lier avec ceux fournis par TinyOS. Cet assemblage est réalisé indépendamment de l'implémentation.

2.1.1. Les composants

Le composant est l'élément élémentaire de l'architecture de TinyOS. En effet, chaque composant correspond à un élément matériel (LEDs, Timer…) ou à un concept logiciel (Main, StdControl…) ou aussi à la liaison de composants afin de former une application .Il faut noter qu'un composant peut être réutilisé dans différentes applications et de différentes façons. Un composant fournit et utilise des interfaces.

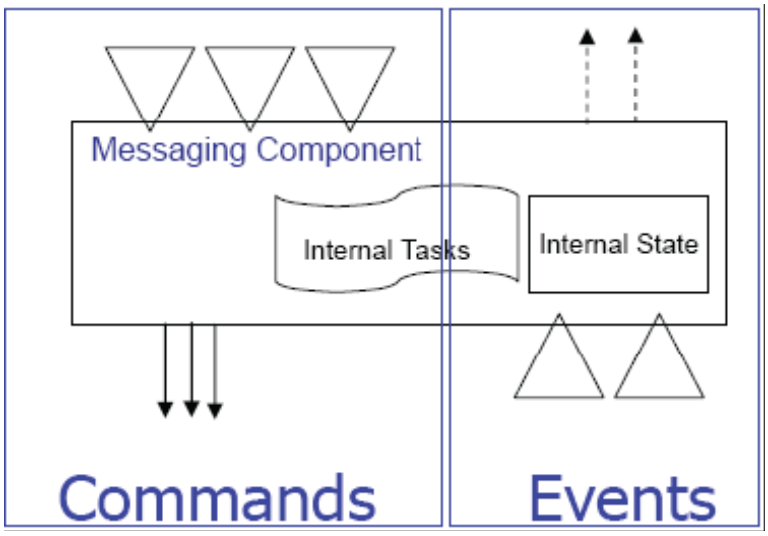


Figure 24. *Schéma d'un composant TinyOS [Hill et al. 05]*

2.1.2. Les interfaces

Les interfaces sont bidirectionnelles et constituent le seul point d'accès aux composants. Une interface contient le descriptif des tâches que le composant est capable de rendre ou

celles dont il a besoin pour fonctionner correctement. On parle alors respectivement d'*interfaces de services fournis* et *d'interfaces de services requis*. En effet, une interface déclare un ensemble de fonctions appelées ***commandes*** que le composant fournisseur d'interface doit implémenter. Et elle déclare aussi un autre ensemble de fonctions appelées ***événements*** que le composant utilisateur doit implémenter. Et par suite, un composant peut être schématisé comme indiquer à la figure 24 où les triangles représentent les gestionnaires d'événements, les triangles inversés le point d'accès des commandes, les flèches pointillées les événements signalés et les flèches pleines les commandes transmises.

Il faut noter aussi que pour qu'un composant appelle une commande d'une interface, il doit implémenter les événements de cette interface. Et un seul composant peut utiliser plusieurs interfaces et plusieurs instances d'une même interface.

2.2. Graphe des composants

Afin de définir quels sont les composants impliqués dans la création de l'application ainsi que la manière dont ils sont reliés, TinyOS utilise un langage de description d'architecture (ADL). En effet, un ADL permet de schématiser une vue globale de l'application. Dans TinyOS, ce schéma est appelé « graphe de composants ».

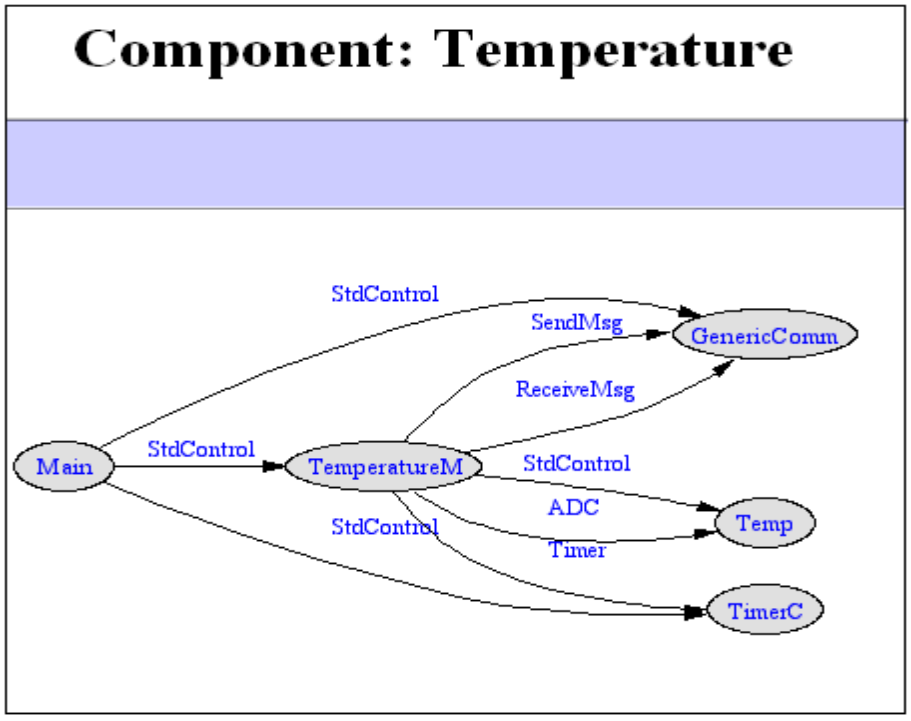


Figure 25. *Graphe de composant dans TinyOS*

La figure 25 représente le graphe de composants de l'application température que nous avons implémenté. Les ellipses représentent les interfaces qui interviennent dans cette application. Les flèches représentent les liaisons entre ces composants. Il faut noter aussi que le sens d'une flèche indique si le composant fournit l'interface ou il l'utilise. En effet, d'après ce graphe, l'interface *SendMsg* par exemple est fournit par le composant *GenericCom* et elle est utilisée par le composant *TemperatureM*. Ainsi, l'interface SendMsg peut implémenter les commandes fournie par le composant GenericCom et peut aussi implémenter les événements de TemperatureM.

3. Implémentation

En pratique, il y a deux types de composant en nesC : les *modules* et les *configurations*.

- Les fichiers *configuration* sont utilisés pour définir les composant que va utiliser l'application et la manière dont ils sont assemblés. En effet, ce fichier relie les composants utilisant une interface par les composants qui la fournissent. Chaque application nesC est caractérisée par un fichier configuration « *top-level* » qui permet de faire le lien entre tous les composants.

```
configuration Temperature {
}
implementation {
components Main, TemperatureM, Temp,TimerC,GenericComm as Comm;
        Main.StdControl -> TemperatureM.StdControl;
        Main.StdControl -> TimerC;
        Main.StdControl -> Comm;
        TemperatureM.Timer -> TimerC.Timer;
        TemperatureM.TempControl -> Temp.StdControl;
        TemperatureM.ADC -> Temp;
        TemperatureM.SendMsg -> Comm.SendMsg;
        TemperatureM.ReceiveMsg -> Comm.ReceiveMsg;
      }
```

Figure 26. *Fichier de configuration de l'application Temperature*

La figure 25 représente le fichier configuration de notre application. En effet, nous avons utilisé les composants : Main, Temp, Timer et GeneriCom. Alors que les lignes suivantes représentent les connexions entre les composants : composant fournisseur d'interface, composant utilisateur de cette interface et l'interface de liaison. Prenons à titre d'exemple la ligne 5. Cette ligne signifie que l'interface StdControl du composant Main est liée au composant Temperature via son interface StdControl.

Il faut noter ici qu'il n'est pas toujours nécessaire de nommer le nom de l'interface avec laquelle communique le composant. En effet, si le composant a une seule instance d'une interface, on peut ne pas indiquer le nom de cette interface : elle sera utilisée par défaut puisque le composant n'a pas d'autres interfaces avec lesquelles il peut communiquer, à l'instar de la ligne 6 et la ligne 7 de la figure 25. Néanmoins, ce n'est pas le cas de la ligne 5 où il faut bien préciser le nom de l'interface avec laquelle le composant communique. En effet, dans le cas des lignes 6 et 7 les composants TimerC et Comm utilisent l'interface StdControl une seule fois, alors que les composants Main et TemperatureM l'utilisent plusieurs fois ; d'où advient la nécessité de la précision. Pour mieux voir cette notion, on peut revoir le graphe des composants de notre application figure 25.

- Le fichier *modules* contient le code d'un composant élémentaire et implémente les interfaces intervenant dans l'application. En fait, le module définit les interfaces utilisées et les interfaces fournies par le module. Ensuite, il implémente les commandes et les évènements selon les besoins et les nécessités de l'application. La figure 26 présente le module TemperatureM de notre application Temperature.

```
module TemperatureM {
 provides {
  interface StdControl;
 }
 uses {
            interface Timer;
            interface StdControl as TempControl;
            interface SendMsg;
            interface ADC;
            interface ReceiveMsg;
         }
 }
implementation {
            uint16_t somme = 0;
            uint16_t nombre = 0;
            int16_t neighbors[MAX_NEIGHBORS];.....
```

Figure 27. Fichier *module de l'application Température*

L'implémentation de composants s'effectue en déclarant des ***tâches***, des ***commandes*** ou des événements. Ces derniers sont des fonctions implémentés dans des composants et qui peuvent être utilisés par d'autre suivant les nécessités.

- Les *commandes* sont des fonctions fournies par l'interface. La figure 28 présente l'implémentation de la commande « start » fournie par l'interface StdControl et utilisée par le composant TimerC. En fait TimerC est un composant générique présent dans la bibliothèque des composants de TinyOS. Timer.start est une fonction déjà implémentée avec le composant Timer et que nous avons ré-implémenté suivant nos besoins : nous avons fixé un compteur qui se re-déclanche après chaque 10secondes.

```
command result_t StdControl.start() {
        call Timer.start(TIMER_REPEAT, 10000);
        return SUCCESS;
        }
```

Figure 28. *Implémentation de la commande start fournie par l'interface StdControl*

- Les *événements* sont des fonctions que l'interface utilise des autres composants. La figure 29 montre l'implémentation de l'événement « fired » fournit par l'interface Timer.

```
event result_t Timer.fired() {
        if(init == 0){
                call ADC.getData();
        }
        else{
                if(TOS_LOCAL_ADDRESS == 1 && init == 1) {
                        *((int16_t *)beacon_packet.data) = TOS_LOCAL_ADDRESS;
                        call SendMsg.send(TOS_BCAST_ADDR, sizeof(int16_t), &beacon_packet);
                        init = 0;
                }
        }
        return SUCCESS;
    }
```

- Alors qu'une *tâche* est un programme formé de blocs d'instructions qui vise à remplir une application bien déterminée.

Une fois les deux fichiers : module et configuration sont prêts, on leurs ajoute un fichier « ***makefile*** » qui permet à l'application d'être compilée. On place le tout dans un même dossier et on l'ajoute au répertoire « apps » de TinyOS. En fait, c'est à partir de c répertoire que toutes les applications son compilée avec Cygwin.

4. La simulation

Une fois notre application compilée, nous pouvons la simuler avec TOSSIM.

4.1. Simulation du fonctionnement

Cette simulation est lancée pour le mode « usr1 ». Le résultat de cette simulation est présenté sur les deux figures suivantes 28 et 29.

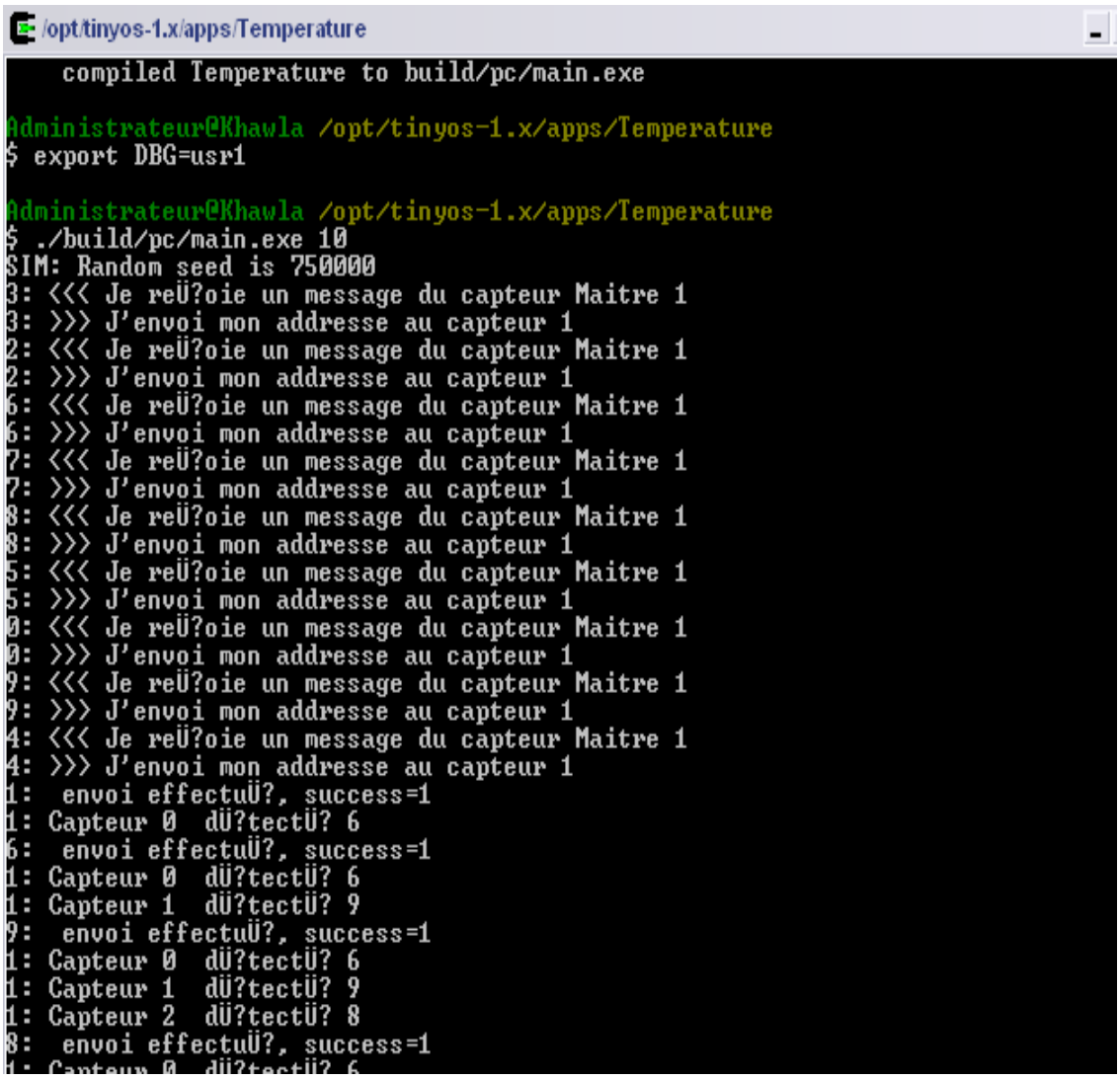


Figure 30. *Simulation de l'application Temperature (1)*

Figure 31. *Simulation de l'application Temperature (2)*

La partie de la simulation représentée par la figure 28 montre le message de diffusion émis par le capteur 1 (nœud principal) aux nœuds présents dans sa zone de couverture et la réception de ce message par les nœuds dans sa zone de couverture. Une fois que tous les

nœuds ont envoyé leurs adresses au nœud 1, ils commencent à capturer la température dans leurs entourages et ils la passent au nœud 1.

Alors que la partie de la simulation représentée à la figure 29 montre bien la réception de la température envoyé par les nœuds du réseau au le capteur 1.Ensuite, ce dernier fait la somme et affiche la moyenne de ces températures.

4.2. Simulation de la consommation

Comme nous l'avons mentionné au chapitre précédent, la consommation d'une application tournant sous TinyOS peut être simulée avec l'extension PowerTOSSIM. Ainsi, nous avons lancé une simulation de notre application avec un réseau de 10 nœuds pour un temps de simulation de 60s.

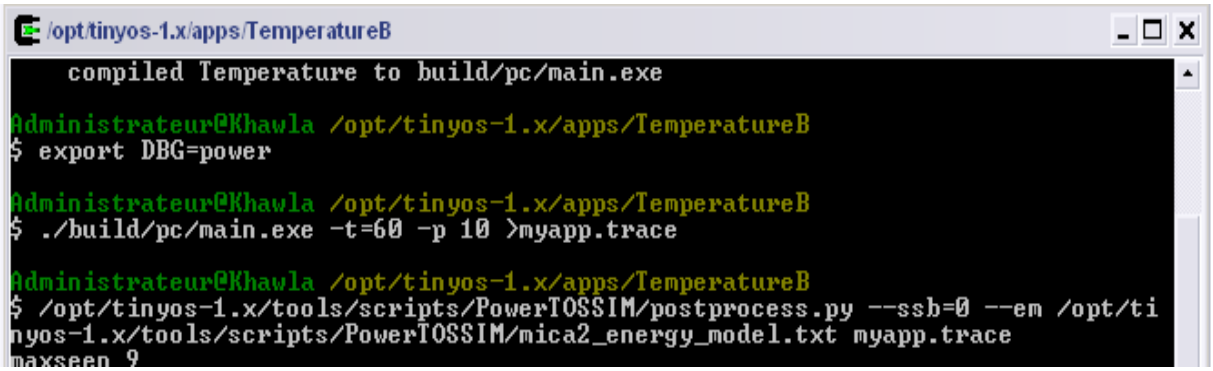


Figure 32. ***Lancement de la simulation de l'application avec PowerTOSSIM***

Les lignes présentées à la figure 30 décrivent la procédure pour lancer une simulation avec PowerTOSSIM [Oh 07]. D'abord, il faut compiler l'application pour TOSSIM avec la commande *« make pc »*. Ensuite, il choisit le type d'affichage qu'on veut visualiser : fonctionnement (usr1), les diodes (led) ou la consommation énergétique (power).Ce choix est effectué via la commande *« export DBG= »*. Après, il faut fixer les paramètres de la simulation :

- ✓ Nombre de nœuds que comporte le réseau
- ✓ Temps de la simulation
- ✓ Nom du fichier où seront stockés les messages contenant l'état de fonctionnement des périphériques du nœud émis par PowerState.

Nous pouvons aussi fixer d'autres paramètres à la simulation comme le type des émission/réception ou aussi la sécurisation des données. Enfin, nous précisons le modèle

d'énergie du nœud utilisé et le fichier contenant les messages envoyé par PowerState pour les combiner ensemble afin de donner la consommation exacte de chaque périphérique du réseau.

Cette simulation avec PowerTOSSIM a donné les résultats illustrés aux figures 33 et 34.

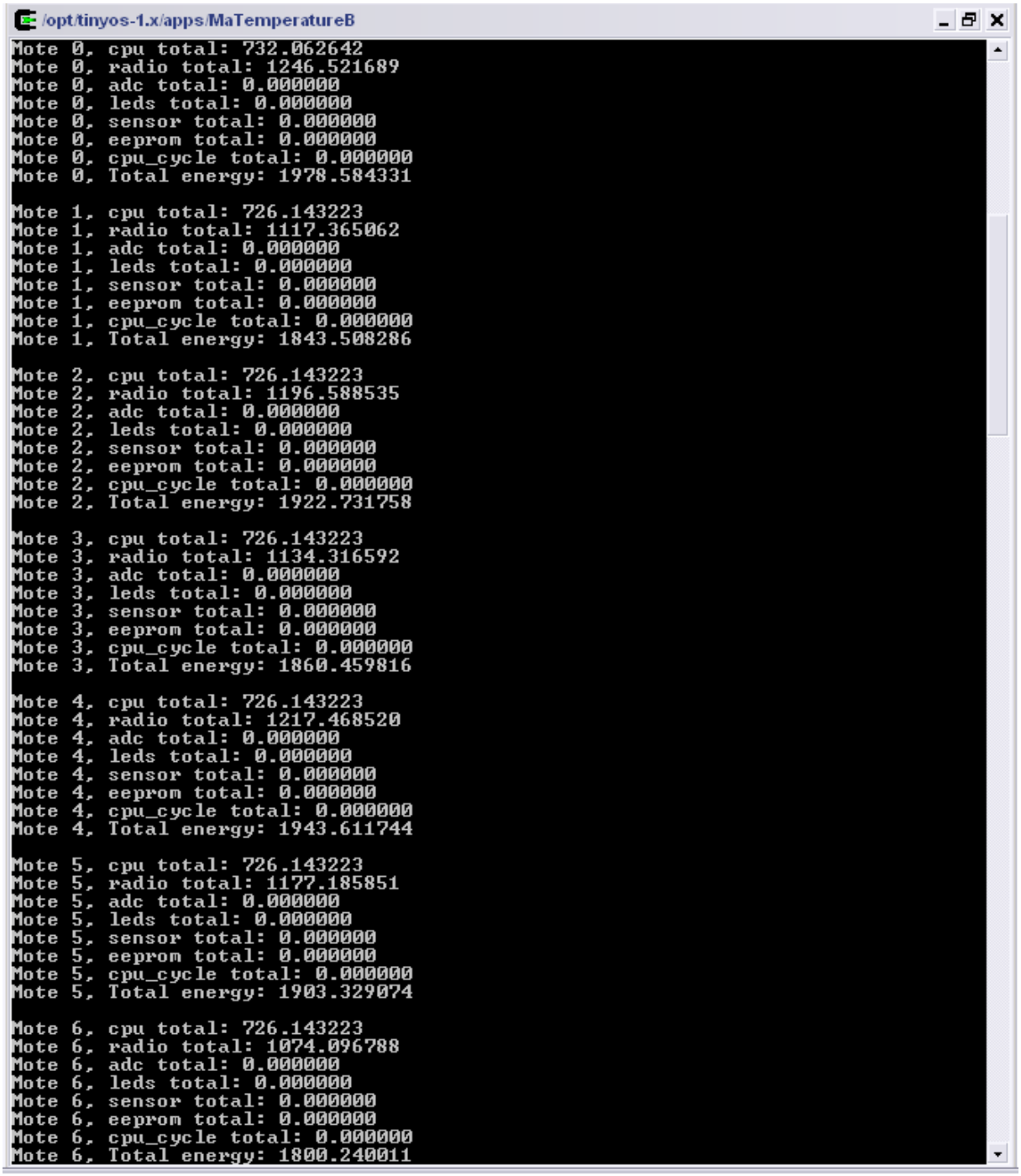


Figure 33. *Résultat de la simulation de notre réseau avec PowerTOSSIM (1)*

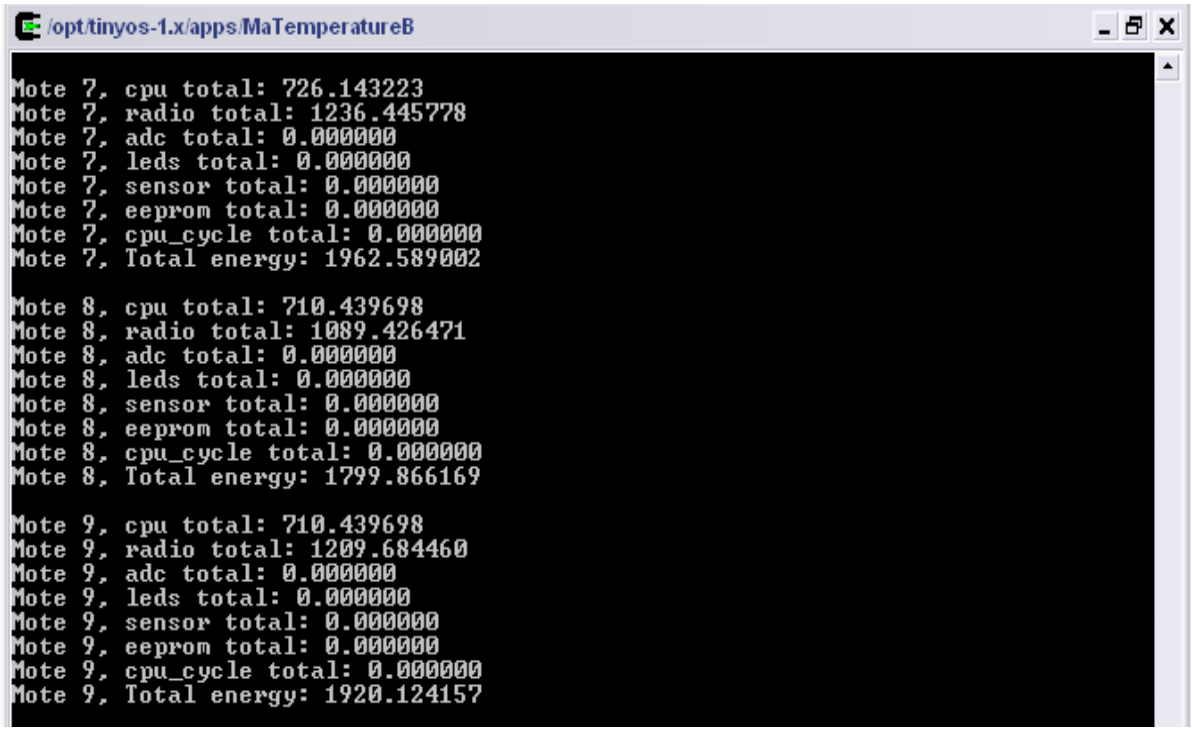


Figure 34. *Résultat de la simulation de notre réseau avec PowerTOSSIM (2)*

Cette figure représente la consommation de chaque périphérique au niveau de chaque nœud en mJ. A titre d'exemple, le nœud 3 a consommé en totalité 1867.7094mJ, dont 726.143165 sont consommés par l'unité centrale de traitement et 1141.558775 sont consommés par l'unité de transmission radio. La consommation des unités leds, adc et sensor est nulle car ces unités ne sont pas utilisés par notre application.

Il faut noter que les résultats donné par PowerTOSSIM varient d'une simulation à une autre sans avoir modifié aucun paramètre de la simulation ou le code de l'application. Mais ces variations ne sont pas très grandes. Ces variations sont dues aux émissions/réceptions. En fait, les nœuds n'émettent pas tous le même nombre de fois pendant une simulation de 60 secondes. D'un autre coté, il y a le cas de la réémission au cas de collision ou autre problème de communication.

Pour ce, dans la suite, toutes les courbes sont des moyennes de 20 simulations successives. La figure 35 représente la moyenne de la consommation de l'unité de traitement pour un réseau de 10 nœuds.

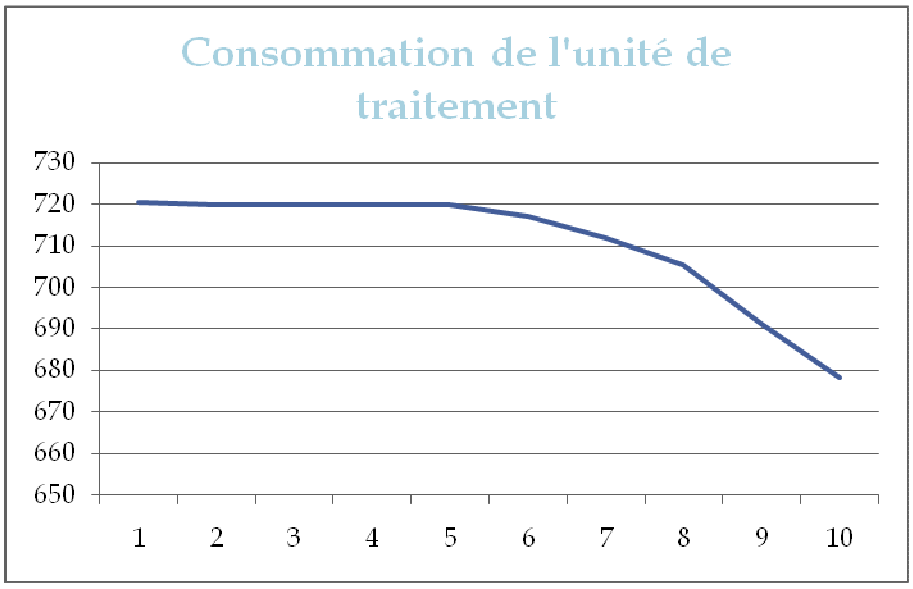


Figure 35. ***Consommation moyenne des unités de traitement d'un WSN de 10 nœuds***

D'un autre coté, nous remarquons que lors de la simulation d'un réseau de capteurs sans fils de 10 nœuds, la consommation des nœuds 9 et 10 est toujours plus faible que les autres nœuds. Nous pensons que ceci est peut être dû au temps de simulation qui ne permet pas à tous les nœuds fonctionner de la même façon. Il faut noter ici que cette variation n'est pas due aux emplacements des nœuds. En fait, nous savons que la puissance d'émission d'un signal varie suivant la distance qui sépare l'émetteur et le récepteur. Cependant le simulateur TOSSIM suppose que tous les nœuds émettent avec la même puissance, donc la distance entre les nœuds n'a aucune influence sur la consommation.

5. Influence du nombre des nœuds

Suite aux simulations précédentes et aux résultats obtenus, nous avons remarqué la variation de la consommation des nœuds d'une simulation à une autre sans que le nombre de nœuds ou le temps de simulation du réseau ne change. Nous avons alors déduit que ceci est dû aux émissions/réceptions non uniformes entre les nœuds. Une question se pose alors : est-ce que le nombre de nœuds dans un réseau influe sur la consommation des nœuds ?

Pour répondre à cette question nous avons tracé la caractéristique de la consommation moyenne d'un réseau en fonction du nombre de nœuds du réseau. Afin d'obtenir cette caractéristique, nous avons lancé des centaines de simulation de l'application «MaTemperatureB» pour différents nombre de nœuds du réseau. La méthode suivie consiste à simuler 10 fois successives notre application. Et pour chaque simulation nous calculons la moyenne de la consommation totale des nœuds du réseau. Ensuite, nous changeons le nombre

de nœuds du réseau à simuler et nous recommençons la même manipulation. Ainsi, nous obtenons pour chaque nombre de nœuds une moyenne de la consommation du réseau. La caractéristique obtenue avec ces moyennes est celle illustrée par la figure37.

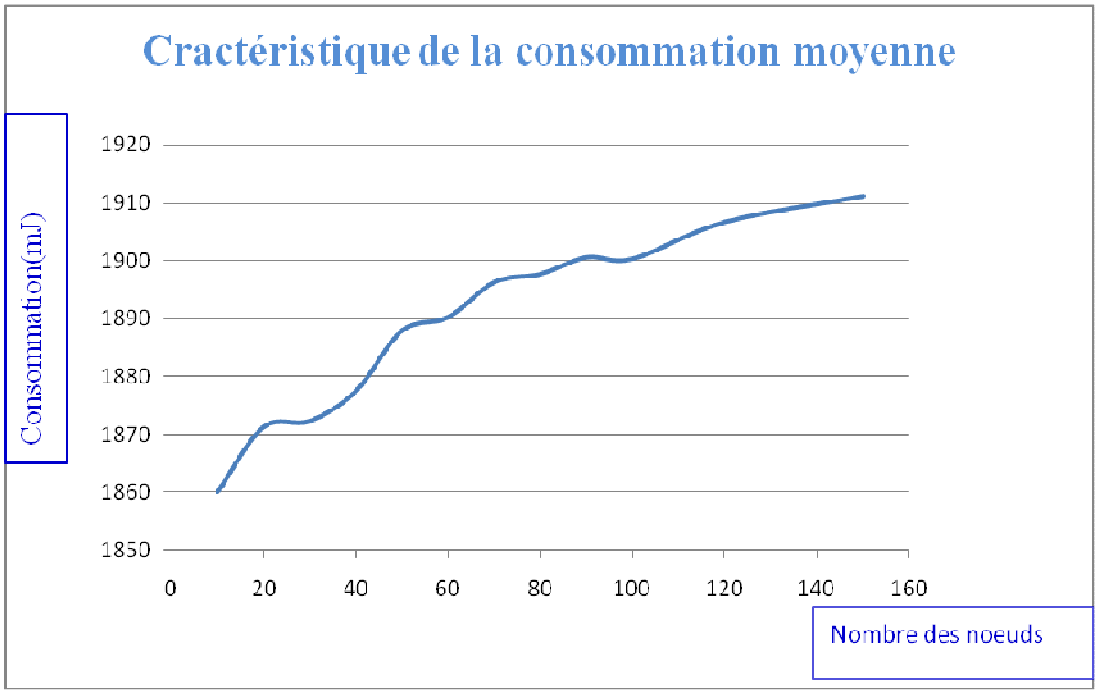


Figure 36. ***Caractéristique de la consommation moyenne du réseau en fonction du nombre des nœuds***

A travers cette caractéristique nous remarquons l'influence du nombre de nœuds sur la consommation moyenne du réseau. La consommation du réseau dépend donc fortement du nombre de nœuds dans le réseau. En effet, plus le nombre de nœuds dans le réseau est grand, plus il y a de risque de collisions et donc risque de retransmission du signal. Sachant que le module émetteur est le plus gourmand en énergie, le nombre de nœuds élevé pénalise la consommation du réseau. Pour ce, il faut bien trouver un compromis entre le nombre de nœuds et leurs consommations. Par ailleurs, le déploiement d'un réseau avec un grand nombre de nœuds garantie une durée de vie plus grande du réseau puisqu'à l'épuisement ou la perte d'un nœud, on peut prévoir son remplacement par d'autres nœuds. En conclusion, en dépit de la surconsommation du réseau faute du nombre des nœuds, le déploiement d'un réseau étendu est une nécessité pour les réseaux de capteurs sans fils. Il faut donc penser à bien gérer la consommation dans les nœuds afin de compenser la surconsommation régie par le nombre étendu des nœuds.

6. Optimisation de la consommation

Il advient en ce moment le problème d'optimisation de la consommation au niveau du nœud : A quel niveau optimiser ? Quelle méthode utiliser ?

Les techniques d'optimisation de la consommation citées dans la littérature se classent en trois niveaux : niveau technologique, niveau logique et des techniques hybrides qui touchent les deux niveaux. Nous avons présenté ces techniques avec plus de détails dans le chapitre deux. Parmi ces techniques nous avons choisi les techniques d'optimisation de la consommation au niveau logique et plus précisément au niveau du code. Ce choix n'est régit que par des contraintes de temps et d'indisponibilité de matériel. En fait, pour travailler au niveau technologique ou avec les techniques hybrides il faut choisir une architecture de nœuds bien déterminée. Le choix de cette architecture demande une étude importante sur l'architecture des nœuds de réseaux de capteurs. Alors que les optimisations au niveau du code peuvent toujours rester valables sur n'importe quelle plateforme.

Dès lors, nous avons choisi d'optimiser la consommation au niveau logiciel et plus précisément, nous avons opté pour des optimisations au niveau du code. Comme indiqué au chapitre deux, l'optimisation du code peut être réalisée soit manuellement soit automatiquement via des techniques intervenant au niveau du générateur de code. Ces dernière sont simples à implémentés et efficace mais elles ne sont pas aussi performantes que l'optimisation manuelle [PBCHIL 00]. Afin de vérifier l'impacte des optimisations manuelles du code sur la consommation nous avons appliqué des techniques d'optimisation du code C sur notre application « Temperature ». Certaines de ces techniques sont appliquées par le module d'optimisation de certains compilateurs mais ce n'est pas toujours le cas. Parmi ces techniques nous citons :

- **Utiliser les registres**

Un bon reflexe quand on a besoin fréquemment d'une certaine valeur est de la sauvegarder dans un des registres du processeur. Ceci est effectué en ajoutant le mot clé ***register*** ainsi :

```
register int index;
```

Ainsi, le compilateur met la variable dans un registre du processeur pour en avoir un accès rapide. En effet, l'accès à une mémoire est plus gourmand en énergie que celui à un registre du processeur. Cependant, ceci n'est pas toujours garanti vu le nombre limité de registres disponible.

- **Inverser les compteurs de boucles**

Une autre optimisation simple et efficace consiste à écrire une boucle « for » avec un compteur décrémenté au lieu d'un compteur incrémenté. En effet, la boucle « for » suivante mène le compilateur à générer une instruction de comparaison pour comparer i et n puis une instruction « branch-on-less-than » pour terminer la boucle.

```
for (i = 1; i < n; i++)
```

Par contre, l'utilisation d'une décrémentation du compteur activera le flag zéro après une suite d'instructions « sub ». Et comme le compteur i est un entier, donc la fin de la boucle serai automatique lorsque i=0. On gagne ainsi en temps d'exécution et en taille de code car on se débarrasse d'une comparaison à chaque décrémentation. On peut alors écrire la boucle de la façon suivante :

```
for( i=10; i--; )
```

Afin de vérifier l'impacte de ces optimisations manuelles du code sur la consommation, nous avons simulé notre application via PowerTOSSIM une vingtaine de fois avant l'utilisation de ces procédés (MaTemperatureE) et une vingtaine de fois après (MaTemperatureF). Nous avons ensuite dégagé la moyenne d'énergie consommée par l'unité de traitement de chaque nœud. Les résultats obtenus sont illustrés à la figure 36.

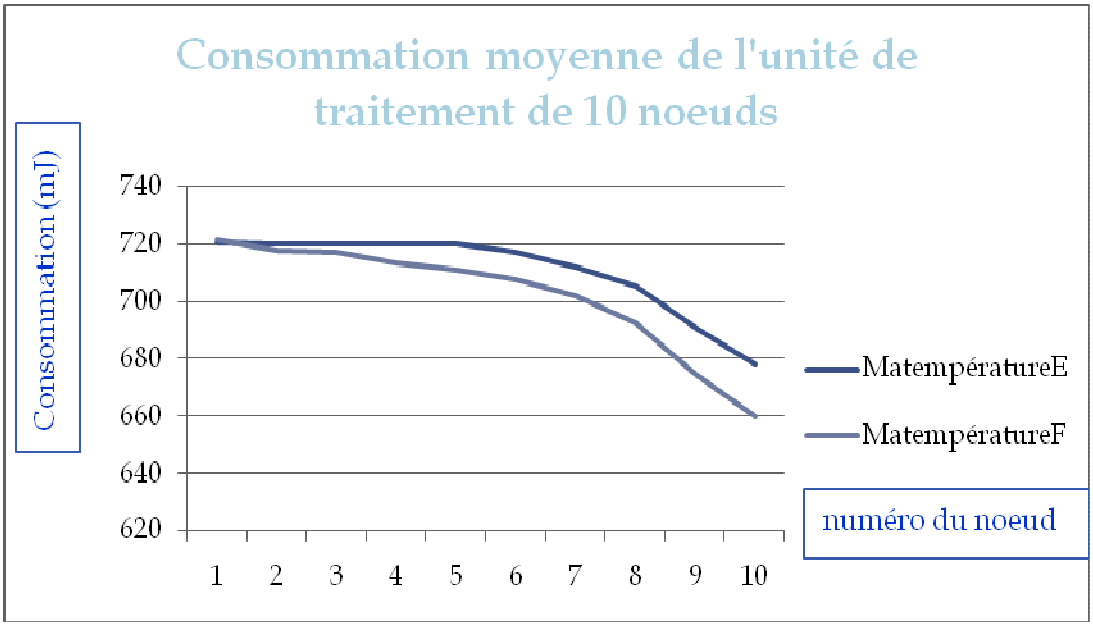


Figure 37. *Courbes comparatives de la consommation moyenne de notre application avant et après optimisation*

7. Conclusion

A travers ce chapitre, nous avons présenté l'implémentation d'une application sous TinyOS. Ensuite, nous avons présenté les résultats obtenus suite à la simulation de cette application avec TOSSIM et TinyOS. En plus, nous avons utilisé cet environnement pour déduire la caractéristique de la consommation d'un réseau en fonction du nombre de nœuds que comporte ce réseau. Ceci nous a permet de dégager des réflexions sur les points de recherche afin d'augmenter la durer de vie d'un réseau de capteurs sans fils.

Enfin nous avons appliqué des techniques d'optimisation de la consommation au niveau du code de notre application.

Conclusion et perspectives

Le présent travail a traité un sujet d'actualité : la consommation dans les réseaux de capteurs sans fils. Au vu de l'état actuel des choses et à notre connaissance, il s'agit d'un sujet en plein essor et beaucoup de travaux de recherche et des industriels montrent un intérêt accru envers cette thématique.

Notre travail visait à étudier en premier lieu les réseaux de capteurs sans fil et leurs gestions de la consommation, chose qui a été bien acquise.

Dans un second temps, nous avons étudié des simulateurs de réseaux de capteurs sans fils. Cette étude avait pour but d'adopter le meilleur simulateur répondant à nos besoins : c.à.d. un environnement qui supporte un réseau de capteurs sans fils étendu, qui consomme le minimum d'énergie et qui peut nous donner une idée sur la consommation. Notre étude a aboutit au fait que TinyOS est le meilleur environnement répondant à ces besoins par :

- ✓ Son architecture orientée composant et qui minimise la taille du code nécessaire à son implémentation et par suite la consommation énergétique.
- ✓ Son système d'exploitation TinyOS conçu spécialement pour les réseaux de capteurs sans fils. Son architecture orientée composants représente son point fort puisque elle simplifie sa maintenance, son amélioration et surtout l'implémentation d'applications tournant dessous.
- ✓ Son simulateur TOSSIM permettant la reprise virtuelle du fonctionnement d'un réseau de capteurs sans fils pour une douzaine de plateformes une douzaine de plateformes : mica, imote, mica2, mica2-dot, micaz, telos…
- ✓ Son extension PowerTOSSIM qui permet la simulation de la consommation au niveau de chaque périphérique des nœuds d'un réseau de capteurs sans fils.

Dans un second temps, nous avons implémenté une application tournant sous cet environnement. Cette application a été ensuite simulée avec TOSSIM et PowerTOSSIM afin de montrer le fonctionnement de cette application ainsi que sa consommation.

Notre contribution ne s'est pas pour autant arrêtée à ce stade, nous avons utilisé l'application que nous avons implémenté en nesC pour étudier des points clés du problème de

la consommation d'une part, et de donner quelques éléments de réponse sur l'évaluation de la consommation sur ce type d'application d'une autre part. En effet, nous avons caractérisé la consommation moyenne d'un réseau de capteurs sans fils en fonction du nombre des nœuds. En plus, nous avons appliqué des techniques d'optimisation de la consommation au niveau du code de notre application. Ceci nous a permet de dégager des réflexions sur les points de recherche afin d'augmenter la durée de vie d'un réseau de capteurs sans fils.

Par ailleurs, cette étude n'est que le commencement d'un travail énorme visant à optimiser la consommation énergétique des nœuds d'un réseau de capteurs sans fils.

Comme travaux perspectives, nous envisageons d'apporter des modifications à l'environnement de simulation TinyOS, et plus précisément son extension PowerTOSSIM afin de l'améliorer pour qu'elle supporte plus de plateformes et plus de services.

D'un autre coté, nous envisageons aussi d'optimiser la consommation avec d'autres techniques et sur d'autres niveaux, notamment, appliquer les techniques hybrides à savoir : l'adaptation dynamique de la vitesse du processeur et l'adaptation dynamique de la tension d'alimentation.

Bibliographie :

[BB 06]: Mathieu BADET et Willy BONNEAU « *Réseaux de capteurs : Mise en place d'une plateforme de test et d'expérimentation* », Juin 2006

[Benini et al. 97]: Luca Benini, Giovanni De Micheli, Enrico Macii, Massimo, Poncino, Riccardo Scarsi. «*Fast power estimation for deterministic input streams*». (IEEE/ACM International Conference on Computer Aided Design) pp.494-501, November 9-13 1997, SanJose, California.

[Benini et al. 98] : Luca Benini et Giovanni Di Micheli. «*Dynamic Power Management, Design techniques and CAD tools*». Kluwer Academics Publishers, 1998

[Berkeley 04] : UC Berkeley «*TinyOS Mission Statement*», 2004. http://www.tinyos.net/special/mission

[Berkeley 05]: UC Berkeley «*TinyOS/nesC Overview*». In a Smart Dust Training Seminar CD, San Jose, February 9-10, 2005.

[Berkeley 06-a] : www.tinyos.net

[Berkeley 06-b] : http://www.tinyos.net/faq.html

[Beutel 06] : Jan Beutel «*Deployment, Test and Validation of Sensor Networks*» Computer Engineering and Networks Lab, ETH Zurich, 06/11/2006

[Bhatti et al 05] : Shah Bhatti, James Carlson, Hui Dai, Jing Deng, Jeff Rose, Anmol Sheth, Brian Shucker, Charles Gruenwald, Adam Torgerson, Richard Han *«MANTIS OS: An Embedded Multithreaded Operating System for Wireless Micro Sensor Platforms»*, ACMKluwer Mobile Networks & Applications (MONET) Journal, Special Issue on Wireless Sensor Networks, August 2005

[BV 05] : Bouillaguet Mathieu et Valero Mathieu «*OS pour réseaux de capteurs:TinyOS et Zigbee*»,2005

[CES 04] : David Culler, Deborah Estrin et Mani Srivastava. «*Overview of Sensor Networks*». In IEEE Computer, vol. 37, no. 8, pp 41–49, Ôut 2004.

[Contiki 07]: http://www.sics.se/contiki/about-contiki.html

[Gay et al 03]: David Gay, Philip Levis, David Culler, Eric Brewer *«nesC 1.1 Language Reference Manual»*, May 2003

[Gay et al. 03*] : David Gay, Philip Levis, Robert von Behren, Matt Welsh, Eric Brewer and David Culler. «The nesC Language: A Holistic Approach to Networked Embedded Systems». In Proceedings of Programming Language Design and Implementation (PLDI), June 2003.

[Guitton 04] : Patricia GUITTON, « Estimation et Optimisation de la Consommation lors de la conception globale des systèmes autonomes », 14 Octobre 2004

[Hill et al. 05]: Jason Hill, Robert Szewczyk, Alec Woo, Seth Hollar, David Culler, Kristofer Pister «*System Architecture Directions for Networked Sensors*,www.tinyos.net/papers/tos.pdf, 2005

[Hubin 00] : Michel HUBIN « *Traité sur les capteurs et la conception instrumentale*» 2000 http://perso.orange.fr/xcotton/electron/coursetdocs.htm

[Jonson 06]: R. Colin Johnson «Mote-on-chip' rolls for sensor nets» http://www.eetimes.eu/195300011

[Laurent 02] : Johann LAURENT «*Estimation de la consommation dans la conception système des applications embarquées temps réels*», le 9 décembre 2002

[Lee 95]: Mike Tien-Chien Lee and Vivek Tiwari. «*A memory allocation technique for low-power embedded DSP software*» In proceedings 1995 IEEE Symposium on low-power Electronics, Octobre 1995.

[Levis et Lee 03] : Philip Levis and Nelson Lee *«TOSSIM: A Simulator for TinyOS Networks*» pal@cs.berkeley.edu, le 17 Septembre, 2003

[LS 98]: Jacob R. Lorch et Alan Jay Smith «Software Strategies for Portable Computer Energy Management», 24 Février 1998

[ns-2] : http://www.isi.edu.nsam/ns/

[Oh 07] Jisu Oh: *«TinyOS Tutorial: CS580S Sensor Networks and Systems*» Dept. of Computer Science, SUNY-Binghamton, le 7 Février 2007

[OMNeT] : www.omnetpp.org

[PBCHIL 00] : Frédéric Parain, Michel Banâtre, Gilbert Cabillic, Teresa Higuera, Valérie Issarny et Jean-Philippe Lesot « *techniques de réduction de la consommation dans les systèmes temps-réels* », Mai 2000

[Rudenko et al. 98]: Alexey Rudenko, Peter Reither, Gerald J. Popek, and Geoffrey H. Kuenning. «*Saving portable computer battery power through remote process execution* ». Mobile Computing and Communication review, 2(1): 19-26, Jannuary 1998;

[Salhiene et al. 03] : Mohammed Es Salhiene, Laurent Fesquet et Marc Renaudin « *Adaptation dynamique de la puissance des systèmes embarqués : Les systèmes asynchrones surclassent les systèmes synchrones* », 2003

[Sentilles 06] : Séverine Sentilles *«Architecture logicielle pour capteurs sans-fil en réseau »* Juin 2006

[Shnayder et al 03] : Victor Shnayder, Mark Hempstead, Borrong Chen, Geoff Werner Allen, and Matt Welsh, «*Simulating the Power Consumption of Large Scale Sensor Network Applications*» Division of Engineering and Applied Sciences, Harvard University 2003

[TinyOS]: http://bingweb.binghamton.edu/%7Ejoh3/CS580S/tinyos_installation.html

[Tiwari et al 94]: Vivek Tiwari, Sharad Malik, Andrew Wolf. Power analysis of embedded software: «*A fist step toward software power minimization*». Dept. of Electrical Engineering. Pinceton University, 1994

[Tiwari et al. 96]: V. Tiwari, S. Malik, A. Wolfe «*Instruction Level Power Analysis and Optimization of Software*» In Journal of VLSI Signal Processing 1996 Kluwer Academic Publishers.

[Turier 00] : Arnaud TURIER, « *Etude, conception et caractérisation de mémoires CMOS, faible consommation, faible tension en technologie submicronique*», le 13 d´ecembre 2000

[Welch et al. 94] : Weiser M. Welch, B. Demers, A. Shenker. *"Scheduling for reduced CPU energy*", USENIX Symposium on Operating Systems Design and Implementation, pp. 13–25, 1994.

[Welsh et al. 06]: Matt Welsh, Geoff Werner-Allen et Konrad Lorincz, «*Monitoring Volcanic Eruptions with a Wireless Sensor Network*», School of Engineering and Applied Sciences, Harvard University, 3 novembre 2006

[WW 99]: Zhao Wu et Waine Wolf, «*Iterative Cache simulation of embedded CPUs with trace stripping*», in Proc. IEEE Int'l Workshop on Hardware/Software Codesign (CODES), May 1999.

[wiki 07-a] : http://en.wikipedia.org/wiki/Wireless_sensor_network#Operating_systems

[wiki 07-b] : http://en.wikipedia.org/wiki/Magnetoresistive_random-access_memory

[wiki 07-c] : http://fr.wikipedia.org/wiki/Syst%C3%A8me_temps_r%C3%A9el