UNIVERSITÉ DE SHERBROOKE
Faculté de génie
Département de génie électrique et de génie informatique

UN SYSTÈME DE COMPOSANTS DISTRIBUÉ POUR LES RÉSEAUX DE CAPTEURS SANS-FILS

Mémoire de maîtrise
Spécialité : génie électrique

Frédéric SUREAU

Jury : Philippe MABILLEAU (directeur)
Bessam ABDULRAZAK (co-directeur)
Charles-Antoine BRUNET (rapporteur)
Ruben GONZALEZ-RUBIO (correcteur)

Sherbrooke (Québec) Canada Mai 2011

Library and Archives Canada

Published Heritage Branch

395 Wellington Street
Ottawa ON K1A 0N4
Canada

Bibliothèque et Archives Canada

Direction du Patrimoine de l'édition

395, rue Wellington
Ottawa ON K1A 0N4
Canada

Your file Votre référence
ISBN: 978-0-494-83679-8

Our file Notre référence
ISBN: 978-0-494-83679-8

RÉSUMÉ

L'utilisation de réseaux de capteurs sans-fils (RCSF) se développe dans de nombreux domaines où l'informatique doit être intégrée au plus proche de l'environnement. Ce principe appelé informatique *omniprésente* se popularise par des applications dans de multiples domaines, de la domotique à l'étude d'environnements naturels en passant par la régulation des transports ou encore la surveillance de bâtiments à risques.

Si les RCSF présentent de bonnes perspectives pour le domaine de l'informatique omniprésente, le matériel utilisé présente souvent des capacités très limitées et il est souvent compliqué de développer des applications ou de configurer de tels réseaux.

Des travaux récemment réalisés au laboratoire DOMUS amènent la vision d'une informatique omniprésente *autonome* qui permettrait à plusieurs éléments d'un réseau de s'organiser entre eux pour limiter les interventions humaines. Dans cette vision, la reprogrammation dynamique des noeuds est utilisée pour simplifier et alléger le processus de reconfiguration du réseau. Le présent projet s'est donc intéressé à la problématique de la reprogrammation des noeuds du réseau dans une optique future d'informatique omniprésente autonome adaptée aux RCSF.

Le présent projet de maîtrise a permis dans un premier temps de mettre en place un cadriciel de programmation par composants adapté aux ressources contraintes des RCSF. Ce système de programmation par composants (POC) appelé *Nodecom* se place comme une amélioration par rapport aux solutions de POC déjà existantes. En effet, Nodecom présente la première architecture hybride permettant à la fois de programmer en utilisant des composants statiques et à la fois de pouvoir charger de nouveaux composants de manière dynamique. Cette architecture hybride a permis d'alléger l'impact du système de programmation par composants tout en conservant la possibilité de reprogrammer dynamiquement certains composants.

Dans un second temps, le projet a consisté à réaliser un dépôt distribué de composants qui permet à chaque noeud de charger dynamiquement n'importe quel composant publié à travers le réseau. Dans ce dépôt distribué, chaque noeud peut se voir attribuer le rôle de conserver une copie d'un fichier de composant dans sa mémoire locale. Pour ce faire, l'implémentation réalisée repose sur un algorithme de routage par clé inspiré des réseaux pair-à-pair traditionnels et adapté aux contraintes des plateformes utilisées.

Les résultats de l'évaluation de ce système de composants distribué pour les réseaux de capteurs sans-fils sont encourageants puisqu'ils mettent en évidence les faibles besoins en mémoire du système. L'implémentation réalisée dans ce projet se place alors comme un bon support pour les travaux futurs qui chercheront à adapter la vision d'informatique omniprésente autonome au contexte des réseaux de capteurs sans-fils.

Mots-clés : intergiciel, composants, reprogrammation, reconfiguration, réseaux de capteurs sans-fils, informatique omniprésente, autonome

REMERCIEMENTS

Je souhaite adresser mes remerciements les plus sincères aux personnes qui ont contribué de près ou de loin à l'élaboration de ce mémoire.

Mes premiers remerciements vont à Philippe Mabilleau et Bessam Abdulrazak, respectivement directeur et codirecteur de ce projet de maîtrise, pour leur disponibilité, leur sympathie et leurs conseils techniques avisés. Cette expérience fut très formatrice pour moi et je leur en suis grandement reconnaissant.

Merci à Alexandre Malo pour m'avoir éclairé sur beaucoup de points, pour ses nombreuses contributions à mon projet mais aussi pour sa bonne humeur qui a rendu cette collaboration agréable.

Je souhaite également remercier tous les camarades du projet Nodeus ainsi que les collègues du laboratoire DOMUS pour leur accueil, leur aide, pour les soirées et fins de semaines studieuses, ou simplement pour leur ouverture et leur sympathie.

Enfin, je tiens à témoigner toute ma reconnaissance envers mes parents pour leur soutien durant ces deux années passées au Québec. Sans eux l'accomplissement de ce projet n'aurait probablement pas été possible.

TABLE DES MATIÈRES

LISTE DES FIGURES

LISTE DES TABLEAUX

LEXIQUE

Terme technique	Définition
Évolutivité	Principe de conception qui prend en compte les besoins futurs. Représente la capacité à pouvoir étendre les fonctionnalités d'un système ou en modifier le fonctionnement facilement.
Extensibilité ou *scalabilité*	Capacité d'un système à s'adapter à plusieurs échelles. Dans le contexte des réseaux de capteurs, l'échelle peut se quantifier en terme de nombre de noeuds, de bande passante ou de dimension spatiale.
Intergiciel ou *middleware*	Logiciel d'infrastructure, supportant le développement d'applications pour les réseaux distribués.
Cadriciel ou *framework*	Ensemble d'outils ou de composants logiciels réutilisables fournissant les fondations nécessaires au développement d'applications.
Composant logiciel	Unité logicielle qui déclare de manière explicite ses fonctionnalités et ses dépendances envers d'autres fonctionnalités. Les fonctionnalités qu'il propose sont déclarées dans une *interface*, soit une sorte de contrat qu'il doit respecter.
Informatique omniprésente	Informatique présente partout et en tout temps, dans laquelle l'usager interagit avec plusieurs systèmes sans nécessairement en avoir conscience. Les termes d'*informatique diffuse* ou d'*environnement pervasif* sont synonymes.
Informatique autonome	Terme désignant des systèmes informatiques capables de s'auto-organiser de manière à résoudre des problèmes comme la (re)configuration, la réparation, l'optimisation ou encore la protection du système en évitant l'intervention humaine.
Mote	Noeud d'un réseau de capteurs, faible consommation, généralement basé sur un microcontrôleur, une radio sans-fil basse consommation et disposant de quelques capteurs embarqués.

LISTE DES ACRONYMES

Acronyme	Définition
DOMUS	Laboratoire de domotique et d'informatique mobile de l'Université de Sherbrooke.
Nodeus	Réseau de noeuds de capteurs et actionneurs pour les espaces intelligents à l'Université de Sherbrooke.
RCSF	Réseau de Capteurs Sans-Fils
POC	Programmation Orientée Composants
ELF	*Executable and Linkable Format*, Format de fichier exécutable.
DHT	*Distributed Hash Table*, Table de hachage distribuée.
KBR	*Key-Based Routing*, Routage par clé.
DFS	*Distributed File System*, Système de fichiers distribué.
I^2C	*Inter-Integrated Circuit*, Protocole de communication par adresses pour relier des circuits intégrés.
ADC	*Analog to Digital Converter*, Convertisseur Analogique vers Numérique.
IP	*Internet Protocol.* Protocole de la couche réseau utilisé pour Internet.
ROM	*Read-Only Memory.* Mémoire non volatile généralement utilisée pour entreposer le code éxécutable.
RAM	*Random Access Memory.* Mémoire volatile rapide généralement utilisée pour stocker le contexte d'éxécution du système.
OS	*Operating System.* Système d'exploitation.
CPU	*Central Processing Unit.* Composant d'un ordinateur qui éxécute les programmes informatiques. Le temps CPU fait référence au temps d'éxécution d'un programme.

Suite du tableau sur la page suivante.

Acronyme	Définition
XML	*eXtensible Markup Language.* Langage de balisage définit par le W3C et à la base de nombreux autres standards.
PDA	*Personnal Digital Assistant.* Assistant numérique personnel portable.
SQL	*Simple Query Language.* Langage déclaratif pour exprimer des requêtes sur une base de données.

CHAPITRE 1

INTRODUCTION

Les réseaux de capteurs s'affichent comme un domaine novateur de l'informatique permettant de relier la réalité du monde physique au monde virtuel. L'observation de phénomènes physiques trouve son application dans plusieurs terrains, de la domotique à l'étude des environnements naturels en passant par la surveillance de grandes constructions.

Malgré le très grand potentiel de ces réseaux distribués, le matériel utilisé offre des ressources très limitées, principalement à cause de difficultés d'accès et de rareté de l'énergie disponible. Lors de la programmation des noeuds du réseau, la gestion des ressources présente donc une part énorme du travail. C'est pourquoi, beaucoup de recherches aujourd'hui se portent sur des couches d'abstraction logicielles qui permettraient de programmer les réseaux de capteurs plus simplement et en se souciant moins des contraintes bas-niveau. Ces couches d'abstraction sont généralement regroupées sous le terme d'*intergiciel*.

1.1 Mise en contexte et problématique

Le laboratoire DOMUS dans lequel se déroule la présente recherche est un exemple du besoin de systèmes de capteurs distribués dans le monde de la domotique. Le projet Nodeus y a été mis sur pied de manière à résoudre des problèmes liés à l'infrastructure des capteurs utilisée dans le laboratoire. La mise en place de solutions logicielles distribuées se présente alors comme l'un des points clés dans l'élaboration d'un tel réseau de capteurs.

1.1.1 Le laboratoire DOMUS

Le laboratoire DOMUS vise à concevoir et étudier un habitat intelligent dans le but de porter assistance à des personnes atteintes de troubles cognitifs – par exemple, les personnes atteintes de schizophrénie, certaines personnes du troisième âge présentant des démences telles que la maladie d'Alzheimer ou encore les traumatisés crâniens.

Ces personnes peuvent avoir des problèmes d'initiation de la tâche ou de pertes de mémoire qui les empêchent de vivre en autonomie. Les recherches menées dans ce laboratoire visent à améliorer l'autonomie de ces personnes à travers la création d'un habitat intelligent qui leur permettrait de surmonter les problèmes du quotidien auquels elles font face [29,

30]. Le laboratoire dispose d'un appartement reconstitué dans la Faculté des sciences de l'Université de Sherbrooke pour mener à bien ces recherches.

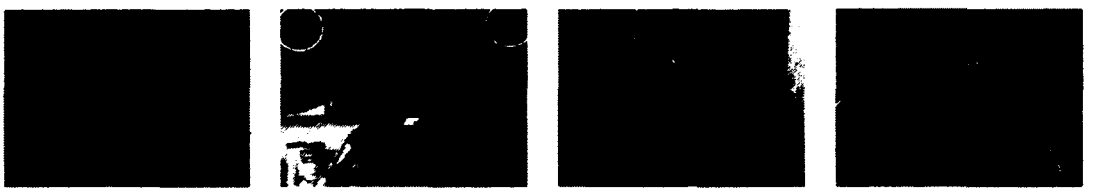
Pour adapter les techniques informatiques au monde réel, des mécanismes de perception et de modification de l'environnement ont été mis en place. La figure 1.1 présente quelques-uns des capteurs actuellement installés dans l'appartement du laboratoire DOMUS, à savoir *a*) les capteurs de mouvement ; *b*) d'ouverture de porte ; *c*) les débimètres ; ou encore *d*) les tapis tactiles. Il est à noter cependant qu'aucune caméra n'est utilisée dans l'appartement pour des raisons de respect de la vie privée et d'acceptation de la technologie par les personnes.

Cette présence de capteurs et actionneurs de manière diffuse dans l'environnement contribue à une vision de l'informatique *omniprésente* dans laquelle les utilisateurs sont constamment en interaction avec un système informatique. Il s'agit donc d'une informatique présente partout et tout le temps avec laquelle les utilisateurs interagissent sans nécessairement en avoir conscience.

L'infrastructure matérielle et l'organisation des capteurs choisis à l'origine dans le laboratoire ont cependant mené à plusieurs problèmes.

Difficulté de maintenance Tous ces capteurs sont actuellement connectés de manière filaire à un ordinateur central qui occupe une pièce entière de l'appartement. Cette organisation pose des difficultés liées au câblage des capteurs, l'emmêlement des fils, la difficulté d'identification de ces capteurs ou encore la complexité d'installation ou de modification des branchements.

Non-réutilisabilité Il est souvent nécessaire d'apporter une certaine intelligence à ces capteurs pour numériser ou prétraîter les données. Il est cependant très coûteux de devoir refaire des conceptions de cartes électroniques pour chaque nouveau besoin. Les concepteurs de ces cartes travaillent souvent pour des besoins différents, et les cartes électroniques réalisées ne sont souvent pas réutilisables.



(a) (b) (c) (d)

Figure 1.1 Exemples de capteurs installés dans l'appartement DOMUS

Faible tolérance aux pannes L'organisation centralisée qui existe actuellement au laboratoire – symbolisée à la figure 1.2a – implique que tout l'appartement dépend d'un seul ordinateur et par conséquent, une panne sur cet ordinateur fait que l'appartement n'est plus fonctionnel jusqu'à la réparation de cette panne.

Coût et installation La centralisation de l'intelligence implique également qu'un seul et même ordinateur pilote l'ensemble des événements de l'appartement. Pour fournir une bonne réactivité à ces événements l'achat de matériel informatique coûteux, volumineux et énergivore est nécessaire. Dans le contexte où l'on recherche des solutions adaptées pour des particuliers, il est inenvisageable d'avoir à utiliser ce genre d'équipements informatiques pour des raisons de prix, d'installation et de consommation électrique.

Perte de qualité Les capteurs étant répartis et connectés par des fils à travers tout l'appartement, des perturbations électromagnétiques pouvaient également altérer les signaux s'ils n'avaient pas été numérisés préalablement, par exemple des signaux sonores noyés dans le bruit.

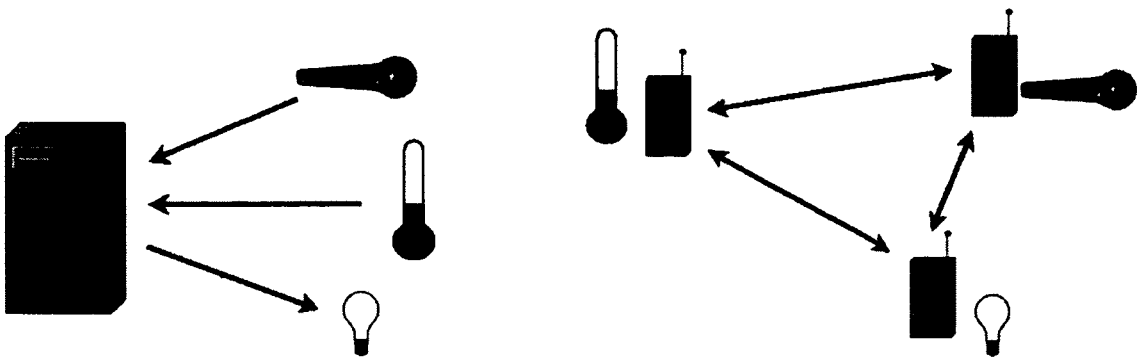
Pour répondre à toutes ces difficultés, le projet Nodeus a été mis en place. Ce projet vise à remplacer la disposition actuelle des capteurs par un réseau distribué de capteurs sans fils ou filaires (RCSF) – le terme anglais de *Wireless Sensor Network* (WSN) est couramment utilisé dans la littérature pour désigner ce concept.

1.1.2 Le projet Nodeus

Le projet Nodeus consiste donc à mettre en place un réseau de capteurs sans-fils pour distribuer la capture et le traitement des données et offrir un système à la fois plus simple à installer, mais également plus robuste.

L'organisation distribuée souhaitée dans le projet Nodeus est symbolisée à la figure 1.2b. Une telle organisation distribuée est plus tolérante aux pannes : si un des noeuds du réseau tombe en panne, un autre noeud peut le remplacer temporairement jusqu'à la réparation de la panne et le reste du système restera fonctionnel.

De plus, puisqu'ils nécessitent moins de puissance de calcul, les noeuds du réseau sont en eux-mêmes très simples et très peu coûteux. Ils consomment également peu d'énergie ce qui permet de les installer facilement avec une batterie de faible taille ou même de puiser directement l'énergie dans l'environnement à partir de vibrations ou des rayons solaires par exemple.



(a) Intelligence unique, puissante et centralisée (b) Intelligences multiples, réduites et distribuées

Figure 1.2 Modes de répartition de l'intelligence

Enfin, un tel réseau permet l'extensibilité du système complet. Il est en effet possible d'ajouter un noeud au réseau et qu'il soit automatiquement reconnu et intégré à une application collaborative.

Cette architecture doit alors répondre à de nombreuses nouvelles contraintes :

Simplicité d'installation Étant donné que les expériences réalisées dans le laboratoire DOMUS se veulent adaptables à tous types d'habitats, l'installation du système doit être la plus simple possible. La technologie sans-fil apporte cette simplicité étant donné qu'elle évite d'avoir à câbler un réseau de capteurs – potentiellement nombreux – dans tout le logement. Les noeuds doivent également être conçus pour être de petite taille et facilement dissimulés dans des prises murales, des interrupteurs, etc.

Faible coût Dans le but de démocratiser ce type d'habitat, les noeuds doivent être de faible coût. Pour ce faire, les noeuds sont construits autour de microcontrôleurs qui sont des unités informatiques embarquées peu dispendieuses.

Basse consommation d'énergie Le déploiement d'un tel réseau entraîne nécessairement des problèmes en terme d'énergie. Il faut que ces équipements électroniques consomment peu d'énergie pour éviter d'avoir à recharger leurs batteries ou leur permettre d'être autoalimentés – reliés à des capteurs photovoltaïques par exemple.

Modularité fonctionnelle Il doit être simple d'adapter les noeuds aux contraintes particulières de chaque domaine. Pour cela, les cartes électroniques doivent proposer une modularité déjà au niveau matériel en séparant les différentes préoccupations – intelligence principale, communication, interface avec l'environnement, de manière à choisir la carte qui convient à chaque contexte d'application.

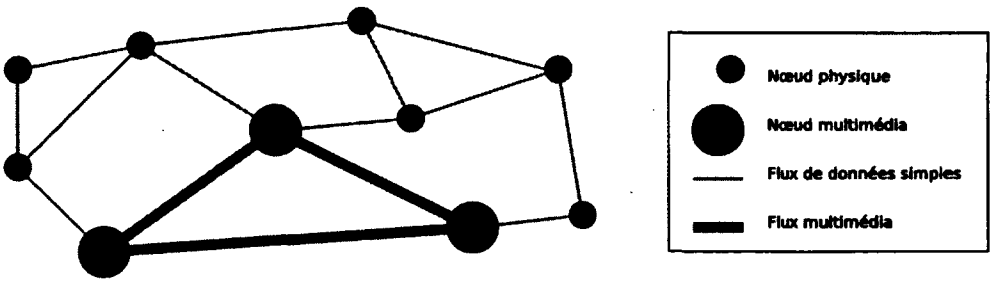


Figure 1.3 Architecture du réseau Nodeus

Puisque différents besoins se manifestent dans les solutions déployées à DOMUS, plusieurs types de noeuds ont été imaginés pour l'architecture de Nodeus ; elle est illustrée à la figure 1.3.

Noeuds physiques Les noeuds physiques sont des cartes électroniques simples, faibles consommatrices en ressources et permettant d'interagir avec l'environnement à l'aide de capteurs ou d'actionneurs.

Noeuds multimédias Les noeuds multimédias sont des noeuds disposant d'une plus grande capacité de calcul et de plus de ressources pour des applications multimédias plus complexes que dans le cas du noeud physique. Ces noeuds serviront notamment à la diffusion d'images vidéos ou de son dans l'appartement.

Les différents noeuds sont ainsi adaptés à des besoins différents selon la puissance de calcul nécessaire et le type d'interfaçage avec l'environnement. Cela apporte une hétérogénéité dans le réseau qu'il faut cependant gérer.

Pour répondre aux contraintes des réseaux de capteurs et simplifier le développement d'applications sur de tels réseaux, des solutions d'*intergiciels* sont mises en oeuvre afin d'en homogénéiser et d'en simplifier la programmation.

1.1.3 La problématique : une informatique omniprésente autonome

Aujourd'hui, les réseaux de capteurs utilisent souvent des applications conçues de manière *ad-hoc*, c'est à dire répondant à des besoins précis et non réutilisables dans d'autres contextes. Les applications se chargent par exemple des communications entre les noeuds ou de l'organisation du réseau. Cette méthode de développement est particulièrement inefficace dans un contexte général, car elle ne factorise pas le travail réalisé. Par ailleurs, la configuration et la maintenance des réseaux n'est souvent pas automatique et nécessite une intervention humaine.

Pour répondre à ce problème, des couches logicielles ont été conçues pour être réutilisables et offrir un support plus générique au développement d'applications pour les réseaux de capteurs. Ces couches logicielles sont généralement regroupées sous le terme d'*intergiciels.*

Une définition plus élaborée du terme d'intergiciel sera proposée dans ce mémoire, néanmoins, les principales fonctionnalités attendues d'un intergiciel sont les suivantes :

- offrir un support à la programmation d'applications ;
- proposer des mécanismes de communication simples qui permettent de s'affranchir de la complexité sous-jacente du réseau ;
- organiser le réseau par des procédés automatiques ou explicites de manière à ce qu'il puisse s'adapter à des modifications de l'environnement ou à de nouveaux contextes.

Les solutions existantes dans la littérature répondent globalement aux deux premiers points, mais présentent beaucoup de lacunes quand à l'auto-organisation du réseau.

Par ailleurs, des travaux récents au laboratoire DOMUS ont amené la vision d'une informatique omniprésente *autonome.* Cette approche propose en effet de combiner les principes de l'informatique omniprésente à ceux de l'informatique autonome. L'informatique autonome cherche à automatiser les processus de configuration, de protection, d'optimisation ou de réparation du système afin de simplifier les travaux de maintenance et de gestion par les humains. Avec cette approche, les systèmes d'informatique omniprésente peuvent être rendus plus simples à installer et à maintenir. L'adaptation de l'approche d'informatique omniprésente autonome au contexte des RCSF est une part importante du concept porté par le projet Nodeus. Dans cette vision d'informatique omniprésente autonome, les noeuds d'un réseau peuvent se reprogrammer de manière individuelle pour se réorganiser.

Les solutions actuelles permettant de reprogrammer les RCSF se focalisent sur la mise à jour du logiciel pendant la phase de développement d'applications. Cette approche est souvent centralisée et la décision de reprogrammer le réseau provient d'un utilisateur derrière un ordinateur et non du réseau lui-même. Le présent projet cherche donc à proposer une solution de reprogrammation décentralisée pour supporter la vision d'informatique omniprésente autonome dans le contexte des RCSF.

1.2 Définition du projet de recherche

Actuellement, les applications et même les intergiciels développés pour les réseaux de capteurs adoptent une vision que l'on peut qualifier de *sense and send* ; c'est-à-dire que

chaque capteur n'a pour seule mission que de capter des informations brutes et les envoyer à un ordinateur central qui les traitera et prendra des décisions en conséquence.

Cette vision traditionnelle des réseaux de capteurs devient de plus en plus obsolète quand des solutions d'auto-organisation ou de reprogrammation des réseaux existent. Les réseaux de capteurs peuvent désormais participer à une vision distribuée des applications en capturant l'information et en la traitant au plus près.

Dans cette perspective omniprésente des RCSF, la reprogrammation dynamique permet de redéfinir le comportement d'un système de manière dynamique, pendant son exécution, pour le mettre à jour par exemple ou pour modifier ses fonctionnalités en fonction du contexte. Dans les principes de l'informatique autonome, cette reprogrammation est décentralisée et permet à chaque noeud de se reconfigurer individuellement.

La question de recherche est donc la suivante :

Comment supporter la mise en place d'une informatique omniprésente autonome avec l'aide de la reprogrammation dynamique des réseaux de capteurs sans-fils ?

Pour répondre à cette question et apporter des solutions à la problématique énoncée, les solutions existantes ont été étudiées. Le paradigme de programmation par composants propose une approche dans laquelle le logiciel est découpé en *composants* indépendants et interchangeables. Ce principe de programmation par composants permet à un système d'être reprogrammé de manière fine, c'est-à-dire composant par composant et non comme un bloc monolithique.

Par ailleurs, la mise en place d'une informatique omniprésente autonome nécessite que chaque noeud d'un RCSF puisse se reconfigurer de manière individuelle. Pour ce faire, un dépôt distribué de composants permettrait à chaque noeud de charger de nouveaux composants depuis ce dépôt. De ce fait, la décision de reprogrammation peut être entièrement décentralisée ce qui participe à la perspective d'informatique omniprésente autonome.

L'approche proposée dans ce projet consiste donc à coupler la programmation orientée composants à des mécanismes de reprogrammation dynamique décentralisés dans le but d'offrir un support pour le développement de solutions omniprésentes autonomes pour les réseaux de capteurs.

1.3 Objectifs du projet de recherche

L'objectif général de cette recherche est donc de **mettre en place une première solution logicielle permettant de supporter la vision d'informatique omniprésente autonome avec l'aide des mécanismes de reprogrammation dynamique décentralisés.**

Cet objectif général se décompose en 3 sous-objectifs spécifiques.

1. **Modèle de programmation** La mise en place d'un système de programmation par composants dynamique est le premier objectif de cette recherche, car de tels systèmes permettent la reprogrammation de manière fine de chaque noeud du réseau.

2. **Auto-organisation du réseau** Un mécanisme de communication auto-organisé inspiré des réseaux pair-à-pair a été réalisé et s'appuie sur le routage par clé pour distribuer l'attribution de responsabilités parmi les noeuds du réseau.

3. **Distribution du logiciel** Une méthode pour la répartition automatique du logiciel parmi les différents noeuds sera proposée. Cette méthode s'appuiera sur le mécanisme de routage par clé du précédent objectif pour réaliser un dépôt distribué permettant à chaque noeud de charger le code de différents composants à partir du réseau.

1.4 Contributions originales

Ce projet apporte donc deux contributions scientifiques majeures. D'abord un nouveau système de composants pour les réseaux de capteurs a été réalisé, et ensuite une solution de dépôt distribué pour ces composants a été proposée pour la première fois.

Nodecom est un système de composants pour les réseaux de capteurs sans-fils (RCSF) réalisé durant ce projet de maîtrise. Il s'agit du premier système de composants hybride pour les réseaux de capteurs, c'est-à-dire qu'il permet à la fois de développer des composants statiques, mais également de charger des composants de manière dynamique. Ce type d'architecture a permis de proposer une implémentation plus légère que les systèmes de composants dynamiques existants dans la littérature.

Pour supporter la reprogrammation des composants de manière décentralisée, un dépôt distribué de composants a été réalisé. À la lumière de la revue de littérature, il s'agit du premier projet s'intéressant à un tel dépôt distribué de composants adapté aux contraintes des RCSF.

Pour automatiser la distribution du code à travers le réseau, le dépôt se présente comme un système de fichiers distribué. Les noeuds responsables pour chaque fichier sont choisis automatiquement à l'aide d'un algorithme de routage indirect par clé. Cet algorithme de routage s'inspire des solutions de réseaux pair-à-pair classiques comme Pastry. Cependant, l'implémentation réalisée s'inspire de travaux connexes pour offrir une solution mieux adaptée aux RCSF que les algorithmes traditionnels.

Par ailleurs, l'ordonnanceur par événements du système d'exploitation Contiki [21] a été porté sous forme de composant Nodecom pour permettre l'usage de *protothreads* dans les composants. Cela a permis de simplifier le développement des différents composants de communication dans Nodecom.

Enfin, l'ensemble du code source réalisé dans ce projet est disponible publiquement[1] sous licence libre ce qui est une contribution importante pour les prochaines recherches scientifiques.

1.5 Plan du document

Ce document est une synthèse des travaux réalisés pendant ce projet de maîtrise et se décompose en 4 parties principales.

D'abord la revue de la littérature sera présentée au chapitre 2. Les problématiques générales des réseaux de capteurs y seront présentées. Une classification des solutions logicielles existantes pour le développement d'applications sur les réseaux de capteurs sera ensuite réalisée et les fonctionnalités attendues d'un intergiciel seront identifiées.

Dans le chapitre 3, le système de composants Nodecom sera présenté. L'intérêt d'un système de composants sera étudié. Les choix architecturaux ainsi que l'implémentation de ce système seront présentés. Enfin des idées de travaux futurs seront proposées.

Le chapitre 4 présentera l'algorithme de routage par clé réalisé et adapté aux contraintes des réseaux de capteurs.

Le chapitre 5 présentera enfin le dépôt de composants distribué réalisé dans ce projet de recherche à l'aide de l'algorithme de routage par clé présenté au chapitre précédent.

En conclusion, un récapitulatif du projet et de ses contributions sera présenté et des idées de travaux futurs seront proposées.

[1] `http://nodecom.sourceforge.net/`

CHAPITRE 2

REVUE DE LA LITTÉRATURE

Les réseaux de capteurs sans-fils (RCSF) sont un sujet de recherche actif et une grande partie des enjeux et des contraintes ont déjà été identifiés par le passé. Pour développer du logiciel à destination des réseaux de capteurs, il convient de bien connaître leurs caractéristiques et les plateformes matérielles utilisées. C'est ce qui sera présenté à la section 2.1 de ce chapitre.

Par ailleurs, de plus en plus de recherches se portent aujourd'hui sur la réalisation d'intergiciels, c'est-à-dire des couches d'abstraction logicielles qui permettraient de programmer les noeuds de capteurs plus simplement et en se souciant moins des contraintes bas-niveau. Beaucoup des caractéristiques attendues d'un intergiciel ont été identifiées et seront présentées à la section 2.2.

La section 2.3 proposera une classification des systèmes existants dans la littérature et les limites de ces systèmes seront abordées dans la conclusion de ce chapitre ainsi que les perspectives entrevues pour le domaine.

2.1 Enjeux des réseaux de capteurs

À en juger par les nombreuses conférences et publications traitant du sujet [1], les réseaux de capteurs présentent un grand intérêt pour la communauté de recherche en informatique. L'informatique omniprésente qu'ils incarnent s'affirme comme l'avenir certain du domaine d'après les lois de Bell et de Moore [7, 49] qui nous indiquent que la prochaine génération des systèmes informatiques sera composée de nombreux équipements de faible taille, probablement de natures diverses et hétérogènes.

Par ailleurs, les réseaux de capteurs sont un sujet qui fait beaucoup parler, car ils touchent à de nombreux domaines au delà de l'informatique. Il est possible d'observer des applications dans les domaines de la domotique, les soins à distance [29], la surveillance de constructions (par exemple ponts et bâtiments) [12], la prévention des catastrophes naturelles, l'étude d'environnements naturels [64] et la régulation des transports [14].

[1] DCOSS 2011, `http://www.dcoss.org/`, SenSys 2011, `http://sensys.acm.org/2011/`

Malgré la relative ancienneté des technologies utilisées, le domaine est encore jeune ; les perspectives qu'il offre restent vastes et les sujets de recherche nombreux. Plusieurs domaines sont présents au sein des réseaux de capteurs, que ce soit l'électronique, l'informatique embarquée, la réseautique, les télécommunications, l'intelligence artificielle ou encore le génie logiciel, ce qui en fait une discipline complexe.

2.1.1 Définition et intérêt des réseaux de capteurs

Jusqu'alors, les applications informatiques classiques traitaient des informations virtuelles que l'usager devait fournir à la machine. Le but visé par les réseaux de capteurs est de faire le lien entre ce monde virtuel et la réalité pour que les applications puissent raisonner sur des données du monde réel [33]. Un réseau de capteurs cherche ainsi à capter un phénomène, manipuler les données représentant ce phénomène et réaliser des actions en conséquence [39, 68].

Par ailleurs, la diffusion géographique des noeuds du réseau permet de capter des informations différentes et réparties dans l'environnement. La fusion de toutes ces informations permet de retirer des informations de plus haut niveau. Les capteurs réalisent ensemble un objectif commun qu'ils ne peuvent réaliser seuls [63].

Par exemple, *EnviroTrack* [1] propose une solution de suivi de véhicules militaires par des capteurs de présence disséminés dans l'environnement. Un capteur de présence seul ne permet de détecter que la position d'un véhicule, tandis qu'une fusion des informations de présence permet de déterminer la direction du véhicule et d'anticiper sa trajectoire.

2.1.2 Caractéristiques et contraintes liées

Si les réseaux de capteurs tentent de répondre aux limites de l'informatique traditionnelle, ils le font en apportant un lot de nouvelles contraintes souvent liées à l'environnement. Parmis ces contraintes, celles qui reviennent le plus souvent dans la littérature sont le manque d'accessibilité des équipements, la dimension, le prix et surtout l'énergie disponible pour les alimenter.

La difficulté de conserver l'énergie durablement dans des batteries fait que les plateformes matérielles doivent être dimensionnées de manière à consommer extrêmement peu d'énergie. Certaines méthodes de récolte de l'énergie présente dans l'environnement – comme la récupération de vibrations ou les panneaux photovoltaïques – permettent d'alimenter les noeuds dans une certaine mesure, seulement cette énergie récoltée reste minime. Il convient

alors de proposer des solutions matérielles et logicielles avec une très faible consommation en énergie.

Du fait de la grande quantité de noeuds dispersés dans l'espace, l'accès, et donc la maintenance des réseaux de capteurs est énormément alourdie. Par ailleurs, la nature des environnements fait qu'il n'est pas toujours possible pour des humains d'accéder physiquement aux noeuds [64]. Pour répondre à ce manque d'accessibilité, le plus souvent, des solutions de communication sans-fil sont choisies, telles qu'une radio longue distance (environnements naturels) ou très basse consommation comme IEEE 802.15.4 (domotique). D'un point de vue logiciel, le protocole ZigBee [2] est conçu pour des applications de faible consommation en énergie et est très populaire en domotique, mais l'utilisation d'IP gagne beaucoup d'intérêt dans la communauté [71].

En domotique, un capteur simple doit pouvoir être caché derrière un interrupteur mural, une prise de courant ou encore sous une dalle de plancher [37]. Dans une étude d'habitat naturel, il est possible que les équipements soient plus volumineux, car il y a peu de contraintes sur l'espace. D'autres cas peuvent également exister comme des vêtements intelligents [44] ou des capteurs de pression à l'intérieur de l'oeil pour étudier les glaucomes [43]. Dans ces cas, la dimension des noeuds est réduite à l'extrême. En résumé, la dimension dépend de l'environnement étudié, mais on doit s'attendre à des plateformes de très faible taille, ce qui amène des limitations sur l'énergie disponible et l'accessibilité.

Enfin, la grande quantité de noeuds nécessaires aux applications des réseaux de capteurs nécessite que le prix des équipements soit relativement faible. Dans le cas des habitats intelligents comme le laboratoire DOMUS, le prix du matériel et de l'installation sera très probablement un facteur déterminant pour l'adoption de ces technologies chez les personnes concernées.

2.1.3 Plateformes

Les plateformes matérielles les plus couramment rencontrées dans la littérature pour mettre en place des réseaux de capteurs sont présentées dans le tableau 2.1. Ces plateformes – communément appelées *motes* – ont des caractéristiques communes qui répondent aux besoins des réseaux de capteurs.

Elles sont pour la plupart composées d'un microcontrôleur 8 bits ou 16 bits et de facilités de communication par IEEE 802.15.4. Les deux microcontrôleurs les plus utilisés sont le *MSP430* de *Texas Instruments* et le *Atmel ATMega128L* pour leur faible consommation en

[2] ZigBee Alliance, `http://www.zigbee.org/`

énergie. Typiquement ces micro-contrôleurs disposent d'une fréquence d'horloge de l'ordre de 8 à 25MHz, de 32 à 128ko de mémoire ROM et d'un maximum de 10ko de mémoire RAM.

Tableau 2.1 Plateformes usuelles des réseaux de capteurs

Plateforme	CPU	Fréquence	RAM	ROM
Tmote sky Sentilla node TelosB	MSP430 (TI)	8MHz	10ko	48ko
Mica2 MicaZ	ATMega128L (Atmel)	16MHz	4ko	128ko (code) 512ko (données)
Nodeus physic	MSP430 (TI)	25MHz	10ko	128ko

Les très faibles ressources fournies par ces plateformes impliquent donc des pratiques particulières au niveau logiciel qui compliquent énormément le développement d'applications.

2.2 Nécessité d'un intergiciel

La nature des réseaux de capteurs, à savoir un système distribué avec des ressources extrêmement contraintes et des communications non-fiables, entraîne une complexité de programmation grandissante qui devient alors une barrière au développement d'applications.

De nombreuses recherches ont donc été effectuées sur l'optimisation des méthodes de routage des données, sur des systèmes d'exploitation légers adaptés aux ressources contraintes du matériel [22, 42], notamment concernant les méthodes d'ordonnancement (noyau préemptif *vs.* événementiel), mais jusqu'alors, la plupart des applications pour les réseaux de capteurs restent conçues de manière *ad-hoc*.

Des solutions généralistes ont été proposées sous le terme d'*intergiciel*. Cependant, il convient de définir les caractéristiques que l'on pourrait attendre d'un tel logiciel.

2.2.1 Définition et intérêt d'un intergiciel

Pour répondre aux besoins des programmeurs, plusieurs méthodes d'abstraction du réseau ont donc été imaginées, généralement regroupées sous le terme d'*intergiciel* (ou *middleware* en anglais). La définition de ce qu'est un intergiciel dépend de l'interprétation de chaque auteur [33, 57, 74]. Globalement, tout le monde s'entend sur le fait que l'intergiciel d'un réseau de capteurs est le logiciel d'infrastructure qui relie ensemble le matériel, le réseau, les OS et les applications [31] (voir figure 2.1).

La programmation sur un réseau de capteurs est une discipline complexe et variée par la diversité des connaissances mises en jeu. Les développeurs d'applications ne veulent pas avoir à prendre en considération tous les aspects des réseaux de capteurs dans chacun de leurs programmes. L'intergiciel apparaît alors comme une couche logicielle mise en place pour fournir une passerelle entre les besoins des applications et la réalité technologique du réseau. L'objectif est de développer une interface de programmation expressive et efficace tout en gérant les enjeux des réseaux de capteurs (énergie, bande passante, etc.) [31].

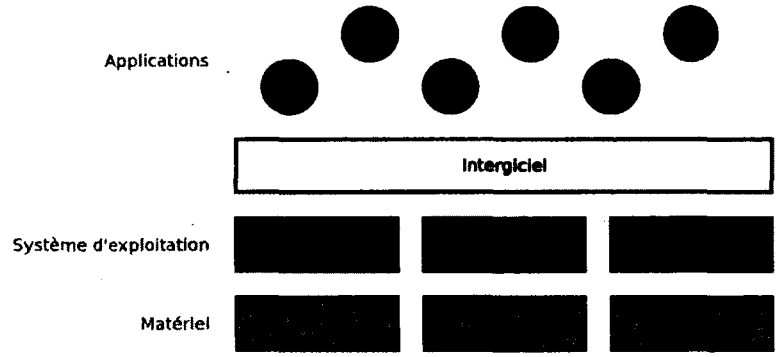


Figure 2.1 Place d'un intergiciel dans une architecture de RCSF

2.2.2 Besoins et attentes d'un intergiciel

Si la définition d'un intergiciel est difficile à résumer en une seule phrase c'est que les rôles qu'on lui associent ne manquent pas. La suite de cet état de l'art tente de définir les besoins associés aux intergiciels des réseaux de capteurs.

Plusieurs attentes liées aux contraintes des réseaux de capteurs sont distinguées – telles que l'efficacité en énergie – mais aussi des attentes au niveau fonctionnel, c'est-à-dire ce que doit proposer l'intergiciel pour remplir pleinement les rôles qu'on lui associe. Enfin, certains besoins se focalisent sur les bonnes pratiques attendues dans la conception d'un tel logiciel.

Contraintes des plateformes

Comme introduit à la section 2.1.2, les réseaux de capteurs apportent leur lot de nouvelles contraintes. L'intergiciel d'un tel réseau se doit d'être adapté à ces contraintes tout en fournissant des fonctionnalités évoluées. Une liste de ces contraintes et des solutions est proposée ici.

Efficacité en énergie L'énergie disponible dans l'environnement pour alimenter les plateformes est limitée, c'est pourquoi, elles sont conçues dès l'origine pour être de

consommation faible, mais si les choix matériels sont déterminant sur la consommation en énergie, la part du logiciel exécuté sur celui-ci n'est pas anodine.

La plupart du temps, les micro-contrôleurs utilisés dans les réseaux de capteurs disposent de modes de gestion de l'énergie. Ceux-ci permettent de contrôler la fréquence d'opération du processeur et d'activer ou non certains périphériques, afin d'adapter la consommation en énergie aux besoins des applications.

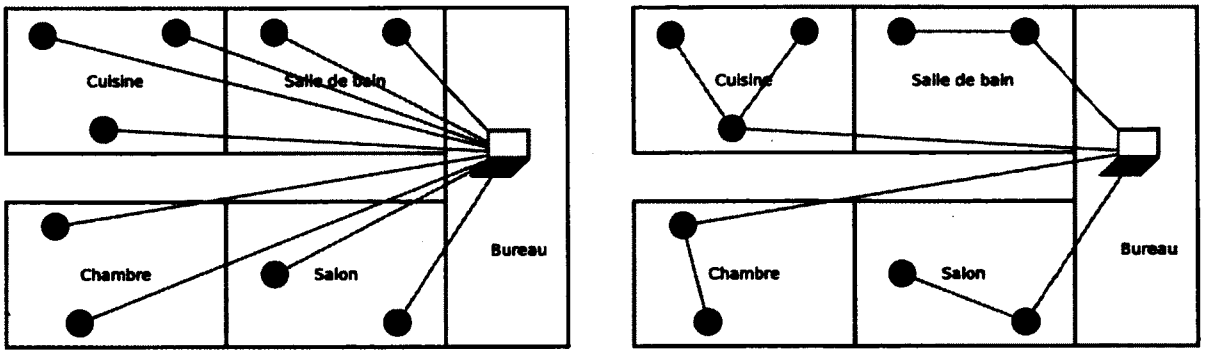
De plus certains périphériques disponibles sur les noeuds des réseaux de capteurs – comme la communication sans-fil, les convertisseurs analogique-numérique, etc. – sont connus pour être très gourmants en énergie. La consommation en énergie des modules de communication radio est considérée par plusieurs auteurs comme étant le point particulièrement critique des réseaux de capteurs [21, 31, 72], il est donc important de prendre en compte ces connaissances pour optimiser le logiciel.

Efficacité en gestion de la bande passante Dans le but de limiter la consommation en énergie de la communication sans-fil, la bande passante mise à disposition sur le réseau est souvent très limitée (par exemple IEEE 802.15.4). Le problème est que la quantité des données mises à disposition par les réseaux de capteurs peut être grande.

Habituellement, il est donc préférable de réaliser l'aggrégation ou la compression des données le plus tôt possible dans les liaisons réseau [74]. Par exemple, dans le cas où l'on souhaite connaître la température moyenne d'un bâtiment, on préférera calculer localement la température moyenne de chaque pièce avant de les transmettre à l'initiateur de la requête, plutôt que de transmettre directement les données de tous les capteurs du réseau (voir figure 2.2).

Il y a aussi une forme de compromis à faire entre l'exactitude des données, la consommation en mémoire et l'utilisation des communications. Pour limiter la diffusion des données, il est possible de stocker des copies locales dans les noeuds. Cela est particulièrement intéressant pour la répartition, sur les noeuds, du logiciel qui change rarement dans le temps [26, 72].

Efficacité en usage de mémoire et de temps processeur De manière générale, les ressources disponibles sur les noeuds de capteurs sont très faibles, qu'il s'agisse d'énergie, de bande passante, ou encore de mémoire. Le logiciel développé pour les réseaux de capteurs doit prendre en compte les faibles ressources disponibles sur les plateformes utilisées.



(a) Pas d'aggrégation des données (b) Aggrégations locales des données

Figure 2.2 Aggrégation des données sur un réseau de capteurs

Les ordonnancements à base d'événements [23, 42] – en opposition à la traditionnelle préemption de tâches – permettent une meilleure gestion des ressources en processeur. En effet, le processeur n'est actif que lorsqu'un événement survient et doit être traité ; le reste du temps, le processeur est endormi. De plus, la consommation en mémoire est allégée par le fait que la pile est partagée entre tous les processus qui ne nécessitent pas de disposer chacun de leur espace mémoire. Plus précisément, l'achèvement d'un traitement suite à un événement ne nécessite pas le stockage de l'environnement de plusieurs processus concurrents ce qui épargne l'espace mémoire. De manière générale, il faudra trouver un compromis entre la facilité de développement, les usages en ressources et les modes de communication [74]. Des mécanismes d'allocation des ressources disponibles, selon certaines politiques (par exemple allocation de bande passante pour une courte durée, de mémoire selon la disponibilité ou de temps processeur) sont également à envisager [46].

Besoins fonctionnels

D'après certains auteurs [33, 57], le but d'un intergiciel est de supporter le développement, la maintenance, le déploiement et l'éxécution d'applications basées sur des capteurs. Au delà de la bonne gestion des ressources décrite précédemment, un intergiciel doit donc fournir aux applications des mécanismes pour répondre à l'ensemble de ces besoins, d'une manière ou d'une autre.

Formalisme haut-niveau Comme un réseau de capteurs sert à capter un phénomène réel, il est nécessaire de disposer d'un formalisme pour représenter les phénomènes réels. L'intergiciel doit donc définir une façon de représenter la réalité. Cette modélisation du réel sera utilisée par les développeurs pour répondre aux besoins de leurs applications.

D'après certains auteurs [57], un intergiciel pour réseau de capteurs doit proposer des mécanismes pour :

1. formuler une requête ;
2. communiquer cette requête aux noeuds ;
3. coordonner les noeuds afin de séparer les tâches et de les distribuer ;
4. fusionner les données des capteurs en une seule réponse à la requête ; et
5. reporter le résultat à l'initiateur.

Distribution du logiciel Contrairement aux réseaux informatiques traditionnels, il est difficile de prévoir quels équipements seront présents ou non à l'avance dans un réseau de capteurs. Les ressources disponibles sont souvent variables avec le temps – en particulier, pour permettre l'extensibilité du réseau et la mobilité des équipements. Le système doit donc utiliser au mieux les ressources matérielles proposées sur le réseau pour répondre aux besoins de toutes les applications, même si ces ressources sont dynamiques.

Afin de supporter le déploiement des applications, l'intergiciel doit proposer un modèle de programmation permettant de distribuer facilement le logiciel sur les noeuds. La répartition automatique des charges en processeur, en mémoire ou en bande passante peut ensuite être envisagée [26, 72].

Reprogrammation Pour supporter la distribution du logiciel sur les différentes plateformes, des mécanismes de reprogrammation dynamique des plateformes peuvent être proposés par l'intergiciel. Si cette fonctionnalité n'est pas indispensable, elle permet d'ouvrir les perspectives des applications possibles sur les réseaux de capteurs, ainsi que des automatismes proposés par l'intergiciel.

Les avantages et inconvénients des systèmes statiques *versus* des systèmes dynamiques, dans le contexte des réseaux de capteurs, est un thème qui a déjà été exploré [21, 54, 55].

Qualité de service Une grande lacune des intergiciels actuels est le manque de mécanismes pour formuler des attentes sur la qualité de service [63]. Puisqu'on traite avec le monde réel, il est important que les informations relatives aux ressources disponibles, à la localisation et au temps soient correctement représentées et accessibles aux applications [57]. La qualité de service permet de respecter des contraintes

sur les services offerts, tels que le temps de réponse, la quantité de bande passante utilisée, la qualité des données, etc.

L'application devrait disposer d'une méthode pour exprimer ses besoins en terme de qualité de service. Beaucoup de ces paramètres ont déjà été identifiés, par exemple dans la spécification *Data Distribution Service* (DDS) de l'*Object Management Group* (OMG) [52].

Le respect de ces contraintes reste un problème ouvert et encore en voie d'exploration, cependant des pistes de réponses ont par exemple été proposées dans les systèmes conscients des ressources (*ressource-aware*), qui attribuent les ressources disponibles aux applications [46]. La qualité de service n'en reste pas moins un point très complexe à gérer du fait qu'il est souvent difficile de quantifier à la fois les ressources disponibles sur un réseau de capteurs et à la fois les ressources requises par les applications.

Mécanismes primitifs Pour mettre en place tous ces services, des mécanismes primitifs sont nécessaires. L'identification et la quantification des ressources disponibles sur les noeuds sont essentielles pour la distribution automatisée du logiciel et des données sur ceux-ci ainsi que pour la gestion de la qualité de service.

Puisqu'un intergiciel cherche à modéliser la réalité, il devrait être informé de l'espace (localisation des noeuds) et du temps (système temps-réel). Les services temps-réel sont généralement fournis par le système d'exploitation ; l'intergiciel devra alors définir clairement les primitives qu'il attend de celui-ci [57]. La localisation peut être réalisée soit par configuration manuelle (noeuds non-mobiles), soit par un algorithme de localisation automatique utilisant les technologies appropriées à l'environnement (GPS, RFID, triangulation RF, etc.) [13].

La conception logicielle des réseaux de capteurs pourrait également aborder d'autres défis tels que l'introspection [16, 24] ou la gestion des préoccupations transversales [24] à travers la programmation orientée aspects par exemple.

Attentes sur la conception

Si un intergiciel doit fournir des fonctionnalités de base et le faire en gérant les ressources réduites que nous proposent les réseaux de capteurs, certains principes dans la conception de celui-ci doivent nécessairement être pris en compte et respectés. Ces principes de conception sont pour la plupart hérités de problématiques générales de l'informatique (extensibilité, ouverture), des réseaux (sécurité, hétérogénéité) et des systèmes embarqués,

cependant, ils prennent une autre dimension dans le contexte des réseaux de capteurs et il est important de s'y intéresser pour la conception d'un intergiciel.

Sécurité La sécurité est un aspect primordial dans le domaine des réseaux de capteurs, car la nature même de ces réseaux les rendent sensibles aux attaques ainsi qu'aux erreurs. La technologie sans-fil qui est souvent utilisée implique que les données échangées sur le réseau peuvent être interceptées ou substituées. Or, les informations captées par les réseaux de capteurs sont souvent d'ordre confidentiel (santé, vie privée, sécurité des personnes et des biens). De plus, les algorithmes de sécurité habituellement utilisés sont très consommateurs en ressources et donc difficiles à mettre en place sur des noeuds de capteurs [31].

Le minimum de la sécurité est de protéger les données par l'encryption et par l'identification des interlocuteurs [39]. Même si ces algorithmes sont encombrants sur les supports visés, la sécurité est primordiale. Par ailleurs, ces mécanismes de sécurité simples ne sont probablement pas suffisants et des études plus approfondies sur le sujet doivent être menées.

Standards L'utilisation de standards est une bonne pratique particulièrement adaptée aux réseaux. Au delà du fait qu'elle permet l'interopérabilité des plateformes, les standards sont généralement de bonne qualité et la gestion de la sécurité y est rarement mise de coté. Malgré cela, les standards couramment utilisés dans les réseaux risquent de poser des problèmes en terme de consommations de ressources. Par exemple, la plupart des standards du *World Wide Web Consortium* (W3C) se basent sur le langage XML. La manipulation d'un tel format de données est très ambitieuse pour les plateformes visées dans les réseaux de capteurs. Il y a donc de quoi améliorer sur les standards et les formats de données utilisés par les réseaux de capteurs, par exemple en s'intéressant à des formats de XML binaires [3] ou de standards adaptés aux faibles ressources des réseaux de capteurs [52].

Hétérogénéité La plupart des réseaux de capteurs actuels utilisent des plateformes homogènes en processeur : tous les noeuds sont les mêmes et beaucoup d'intergiciels considèrent ce fait comme une hypothèse de départ [33]. Cependant, les réseaux de capteurs tendent vers plus d'hétérogénéité dans le matériel que les réseaux de *motes* habituels. Certaines architectures hiérarchiques sont mises en place, dans lesquelles certains noeuds ont des capacités de calcul plus fortes que les autres [63]. C'est notamment le cas dans le projet Nodeus. Le système peut également être agrandi en

[3] W3C, XML binary characterization, http://www.w3c.org/TR/2005/NOTE-xbc-characterization-20050331/

intégrant d'autres plateformes au réseau [57]. Ainsi, dans le cadre d'un habitat intelligent, une personne pourrait se déplacer avec un PDA ou une tablette qui devrait alors être intégré au réseau de capteurs.

Par ailleurs, les différentes plateformes permettent des méthodes et langages de programmation différents. Des modèles reproductibles dans différents langages devraient alors être préférés, pour éviter le fort couplage aux technologies informatiques [15, 16, 24].

Ouverture et évolutivité L'évolutivité d'un intergiciel est la faculté de pouvoir ajouter des fonctionnalités à celui-ci [31]. La conception, à travers des méthodes de génie logiciel, doit permettre l'évolutivité des fonctionnalités de l'intergiciel et la réutilisation simple de ce qui existe déjà. Le système doit également être ouvert à la communication avec d'autres systèmes inconnus à l'avance, à travers des protocoles standards. La découverte des services peut être un point de départ pour cette ouverture [30, 39].

Scalabilité ou extensibilité Puisque la dimension du réseau de capteurs n'est pas connue à l'avance, le système doit permettre l'adaptation à différentes tailles de réseaux [63].

Si l'application grandit, le réseau doit s'adapter à cette croissance sans que cela n'affecte les performances. L'intergiciel doit être capable de s'adapter à des conditions changeantes et maintenir des performances acceptables. C'est ce qu'on appelle généralement l'*extensibilité* du réseau – ou *scalabilité* pour éviter la confusion avec l'évolutivité.

En conclusion, les perspectives qu'offrent les intergiciels pour réseaux de capteurs arrivent avec de nombreuses contraintes et questions auxquelles les futures recherches devront répondre. Les besoins sont de différentes natures et les implémentations existantes proposent des manières variées de les prendre en compte. La suite de cet état de l'art tente de classifier les systèmes logiciels existants pour les réseaux de capteurs.

2.3 Approches existantes

La communauté regorge de solutions logicielles pour les réseaux de capteurs, chacune avec son approche et sa façon d'aborder les problématiques du domaine. Cette section de l'état de l'art tente de classifier la vaste étendue des solutions selon ces différentes approches.

2.3.1 Classification proposée

Plusieurs classifications des systèmes logiciels pour les réseaux de capteurs ont déjà été présentées dans la littérature [31, 33, 58, 63].

La classification proposée ici tente d'aborder les différents aspects des intergiciels. Les systèmes présentés dans la littérature sont regroupés selon ces aspects et critiqués de manière à identifier les limites de chacun.

Basiquement, un intergiciel sert à supporter la programmation sur les réseaux de capteurs, les différentes méthodes utilisées seront donc identifiées. Les mécanismes permettant le partage des données des capteurs à travers le réseau seront ensuite abordés. Enfin, les différentes logiques d'organisation utilisées seront également étudiées.

2.3.2 Support à la programmation

Plusieurs paradigmes de programmation ont été étudiés et mis en place pour la programmation des réseaux de capteurs. Le traditionnel système d'exploitation avec son modèle de tâches a été repensé pour l'architecture particulière de tels réseaux. Les concepts de machines virtuelles utilisés depuis longtemps – notamment dans l'univers de Java – ont été adaptées aux ressources contraintes du matériel. Par ailleurs, des approches de génie logiciel, telles que la programmation orientée composants, ont également fait leur apparition dans le monde des réseaux de capteurs.

Toutes ces techniques ont pour but premier d'offrir un modèle de développement pour simplifier la programmation d'applications.

Systèmes d'exploitation Le monde des réseaux de capteurs s'intéresse depuis longtemps à la conception de systèmes d'exploitation adaptés aux faibles ressources que les microcontrôleurs proposent. Les partisans de l'ordonnancement par événements s'opposent aux défenseurs des systèmes multitâches préemptifs.

Les systèmes d'exploitation TinyOS [42] et Contiki [21] proposent un ordonnancement par événement, c'est-à-dire que les processus ne sont éxécutés que lorsqu'ils sont réveillés par des événements. Ce modèle impose souvent de penser l'application en terme de machine à états, puisque le contexte des processus n'est pas sauvé par le noyau [38, 63].

En opposition, dans un système à tâches concurrentes comme μC/OS [40], les processus s'exécutent jusqu'à ce qu'ils soient interrompus par le noyau qui sauvegarde

et restaure le contexte d'exécution. L'avantage des systèmes multitâches est que le programmeur n'a pas à se soucier du contexte de l'application puisqu'il est automatiquement sauvegardé par le noyau, cependant cela engendre une consommation notable en mémoire et en processeur puisque chaque processus dispose de son propre contexte (pile) et que le noyau doit interrompre les processus régulièrement [22].

Les systèmes à événements, même s'ils peuvent être plus complexes à programmer, sont d'une conception plus simple et consomment globalement moins de ressources mémoire et processeur [63]. La limite principale de l'ordonnancement par événements est la capacité du système à traiter un événement avant que le suivant ne survienne. De plus, et d'une façon symétrique à l'introduction de la préemption dans les systèmes à événements, il est possible d'utiliser un noyau temps réel préemptif en mode d'ordonnancement par événements en jouant sur les priorités des tâches.

Des méthodes hybrides tentent cependant de tirer parti du meilleur des deux. Ainsi, Contiki propose un mode d'ordonnancement, en tant que bibliothèque externe au noyau, pour les applications ayant besoins de préemption [22]. Des modèles de concurrence à base d'événements ont également été proposés comme les *fiber* ou les *protothreads* [23] qui sont des sortes de tâches sans contexte associé. Ces modèles permettent de simplifier l'écriture des programmes en conservant la légèreté de l'ordonnancement par événements particulièrement intéressante sur les plateformes visées.

Machines virtuelles Les machines virtuelles telles que Maté [41] ou la *Java Virtual Machine* (JVM) [3] sont des interpréteurs de code machine. Le principe est de garantir la portabilité du code selon la philosophie de Java : *Write once, run anywhere* ce qui signifie que le code machine compilé peut être en théorie directement éxécuté sur n'importe quelle machine virtuelle compatible. Maté est une machine virtuelle à destination du système d'exploitation TinyOS et adaptée aux contraintes des réseaux de capteurs. L'approche des machines virtuelles cible clairement la reprogrammation des systèmes [63] : le code machine peut être optimisé de manière à être réduit en taille et consommer moins de ressources en communication lors du chargement à distance [21]. L'usage de machines virtuelles simplifie également la gestion de la mémoire, c'est-à-dire que le programmeur se soucie très peu des problématiques d'allocation de la mémoire au moment de l'éxécution ou du chargement du code. Il est à noter également que les machines virtuelles permettent de simplifier les processus d'introspection du code, ce qui en code natif est habituellement géré à travers l'utilisation de méta-données [55].

Cependant l'usage de machines virtuelles entraîne une grande surcharge sur la consommation en ressources telles que le temps CPU ou la mémoire. Pour contourner ce problème, des mécanismes sont développés, permettant le chargement dynamique de code natif et cherchant à optimiser ce chargement pour limiter la consommation en ressources de communication [21].

Composants La programmation par composants est un paradigme de programmation très populaire dans le milieu. Basiquement, un composant est une unité logicielle qui déclare de manière explicite ses fonctionnalités et ses dépendances envers d'autres fonctionnalités [32]. Les fonctionnalités qu'il propose sont déclarées dans une *interface*, soit une sorte de contrat qu'il doit respecter (voir figure 2.3). Avec ces contrats explicites, il devient possible de déployer des composants indépendamment les uns des autres. Les choix de composition se font ensuite selon les besoins de chaque système.

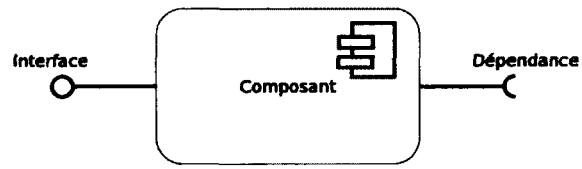


Figure 2.3 Représentation UML d'un composant

Puisqu'au moment de la composition, le programmeur s'intéresse aux contrats et non aux choix d'implémentation, les composants peuvent être préparés d'avance et composés selon les besoins ; cela permet une grande réutilisabilité du code. C'est le principe retenu dans Lorien [54] qui se repose sur le modèle de composants d'OpenCom [16] pour proposer un système d'exploitation modulaire à l'extrême. Par exemple, un composant pour un capteur de température peut implémenter l'interface `icapteur` qui comporte une méthode `recevoir_valeur()`. Cela n'a alors pas d'importance si l'implémentation sous-jacente communique avec le capteur par I^2C ou à travers un ADC. Les autres composants utilisent de manière transparente la fonctionnalité décrite par l'interface. Enfin, les composants peuvent également représenter les unités de code mobile qui simplifient la distribution du code sur le réseau [14].

Comme expliqué précédemment à la section 2.2.2, deux façons de relier les composants sont possibles : soit les composants sont liés de manière statique, soit de manière dynamique. Les liaisons dynamiques offrent plus de perspectives mais consomment généralement plus de ressources [55].

Le NesC [28] est un langage de programmation héritant du C et adapté aux réseaux de capteurs. Il définit nativement une façon de décrire des composants logiciels statiques et des événements synchrones ou asynchrones. Il s'agit d'un langage très populaire dans le domaine et il est à la base du système d'exploitation TinyOS. Les *Real-Time Software Component* (RTSC) [4] de l'*Eclipse foundation* et *Texas Instruments* proposent un modèle de composants statiques comparable à celui du NesC et spécifiquement adapté à la programmation sur microcontrôleurs. Dans un modèle de composants statiques, la liaison entre les composants se fait habituellement au moment de la compilation. Les développeurs de Koala [51] défendent que cette façon de lier les composants engendre moins de consommation en ressources qu'un modèle dynamique.

Les modèles de composants dynamiques – en opposition aux modèles statiques – permettent de lier les composants à l'exécution et donc de lier des composants qui ont été chargés dynamiquement à travers le réseau par exemple. OpenCom [16] est un modèle de composants dynamique implémenté dans plusieurs langages. Même si OpenCom a été identifé comme étant beaucoup plus consommateur en mémoire et en temps CPU qu'un modèle statique [55], il offre des perspectives inégalables avec un système de composants statique, comme l'introspection.

Enfin, une fois les composants écrits, des mécanismes pour la liaison entre ces composants doivent être proposés. Cette macroprogrammation peut s'inspirer de techniques comme les injecteurs de dépendances, très populaires en génie logiciel, comme propose le cadriciel Spring pour Java [5]. Ces systèmes consistent à définir une application en reliant des composants entre eux à travers une description – généralement en format XML. L'intergiciel multimédia NMM [45] propose une méthode similaire, même si elle n'est pas basée sur XML et orientée pour les réseaux multimédias.

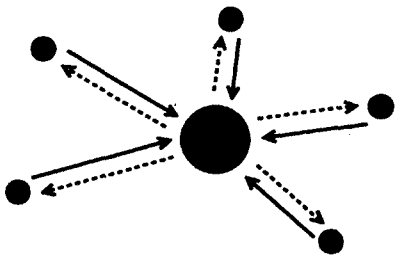
Pour résumer, beaucoup de modèles de programmation ont été hérités des systèmes distribués ou de l'informatique traditionnelle (OS, machines virtuelles, composants). Si certains ont beaucoup été étudiés comme les modes d'ordonnancement ou les systèmes de composants, il reste du travail pour adapter ces modèles aux contraintes spécifiques de réseaux de capteurs.

La programmation orientée composants semble néanmoins répondre à beaucoup des préoccupations tout en restant relativement légère et donc adaptée aux contraintes matérielles.

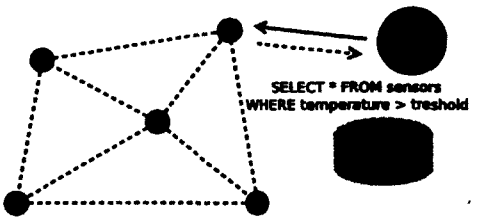
[4] RTSC-Pedia, `http://rtsc.eclipse.org/docs`

[5] Spring reference – the IoC container, `http://static.springsource.org/spring/docs/3.0.x/spring-framework-reference/html/beans.html`

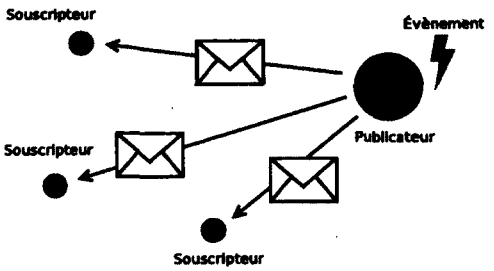
2.3.3 Partage des données



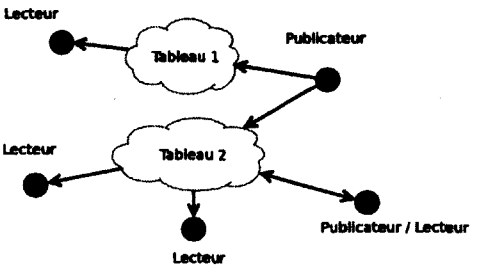
(a) Client/Serveur



(b) Base de données



(c) Publication/Souscription



(d) Tableau noir

Figure 2.4 Illustrations des modes de partage des données

Le rôle principal d'un intergiciel est de faire transiter les données utiles depuis les capteurs jusqu'aux applications voire entre les applications. Ici encore, de nombreux paradigmes hérités des systèmes informatiques habituels ont été explorés.

Client/serveur (figure 2.4a) S'il est un paradigme de communication largement utilisé, il s'agit du modèle client/serveur utilisé dans le Web. Un client envoie des requêtes à un serveur qui, lui, répond à tous ses clients. Ce modèle de communication nécessite de connaître les adresses des serveurs. Ce modèle est criticable puisque les charges sont centralisées sur un seul noeud, le serveur. Il n'apporte donc pas en lui même les qualités de tolérance aux pannes que l'on recherche dans un système distribué. Cependant, les autres modèles de communication ne sont souvent que des surcouches pour abstraire ce modèle de base et dans lesquels tous les noeuds sont à la fois client et serveur.

Bases de données de capteurs (figure 2.4b) Lorsque le réseau de capteurs est vu comme un panier d'informations en provenance de capteurs, des mécanismes de tri de ces informations sont nécessaires. Des principes de bases de données distribuées

ont donc été proposés pour transformer un réseau de capteurs en un puit d'informations de capteurs [47]. Le principe est de définir des contraintes sur les informations que l'on souhaite obtenir dans un langage déclaratif proche du SQL. Il est également possible de commander l'aggrégation des données, comme la moyenne et le maximum.

Il s'agit d'une manière simple de manipuler les réseaux de capteurs, cependant ces systèmes partent du principe que les données manipulées sont scalaires et proposent souvent des mécanismes d'aggrégation limités et difficilement évolutifs. Pour traiter des images ou du son par exemple, le modèle perd toute son expressivité. Ces systèmes ne s'appliquent également pas facilement à des problématiques de découverte du réseau, dans lesquels plusieurs nouvelles plateformes peuvent être intégrées à la tâche. L'abstraction est également restreinte, les applications sont limitées à raisonner sur des données bas-niveau ; un système intermédiaire serait alors nécessaire pour obtenir une représentation plus haut-niveau de la réalité que des données brutes des capteurs. Enfin, les bases de données de capteurs participent à une vision centralisée des applications pour réseaux de capteurs dans laquelle un seul client traite les données fournies par tous les capteurs.

Publication/souscription (figure 2.4c) Pour répondre aux problèmes de dynamicité du réseau et au fait que les données et services disponibles peuvent être très variables dans le temps, le principe de publication/souscription offre une très bonne tolérance aux fautes. Utilisé par l'intergiciel Mires [62] et dans la spécification DDS de l'OMG [52], ce paradigme consiste à séparer logiquement les noeuds producteurs de données (publication), des noeuds qui les consomment (souscription). Les consommateurs souscrivent aux événements publiés par les producteurs de données. L'avantage de cette méthode est que les données ne sont transmises que lorsqu'un producteur a une information à communiquer, ce qui se rapporte à l'approche événementielle populaire dans les réseaux de capteurs pour sa faculté à sauver les ressources.

La souscription peut se faire de manière directe, si l'adresse du producteur est connue, ou de manière indirecte si l'on souhaite soucrire à une donnée sans forcément connaître le ou les producteurs de cette donnée. L'application SCRIBE [11] se base sur l'algorithme de routage par clés proposé par Pastry [20] pour fournir une forme indirecte du modèle de publication/souscription.

Tableau noir (figure 2.4d) La métaphore du tableau noir se réfère aux systèmes permettant d'échanger des informations à travers des espaces virtuels. Les noeuds

peuvent déposer des données dans ces espaces virtuels ou les collecter. Ces espaces peuvent être multiples et séparés logiquement.

Traditionnellement ce type de communication est utilisé par les systèmes d'exploitation pour que des processus puissent s'échanger leurs données. Cependant, ce principe a été étendu aux systèmes distribués sous la forme des espaces de n-uplets (*tuple-spaces*) ou des tables de hachage distribuées – *Distributed Hash Table* en anglais ou DHT.

Contrairement au modèle publication/souscription, le modèle de tableau noir conserve les données tant qu'elles n'ont pas été effacées du tableau. Pour garantir la disponibilité des données de manière fiable, il est donc souvent nécessaire de répliquer ces données sur plusieurs noeuds.

TinyLIME [17] et *TeenyLIME* [12] sont des exemples d'espaces de n-uplets adaptés aux réseaux de capteurs. Ces deux implémentations se basent sur le système d'exploitation TinyOS et proposent de faire communiquer les noeuds du réseau à travers des espaces d'échanges soit une forme de mémoire partagée virtuelle.

Les tables de hachage distribuées sont une autre application de la métaphore du tableau noir. Les données sont distribuées et persistantes de la même manière que les espaces de n-uplets mais chaque donnée est associée à une clé. Il s'agit donc du principe de table de hachage appliqué à un réseau distribué. Ce principe est très utilisé sur Internet pour les échange de fichiers, comme BitTorrent, et a été récemment adapté aux problématiques des réseaux de capteurs [2, 5, 18, 56].

2.3.4 Logiques d'organisation

Si l'intergiciel doit proposer des abstractions pour supporter le développement d'applications et l'échange de données utiles, il doit aussi prendre en compte la réalité technique des réseaux de capteurs et tenter d'optimiser l'usage des ressources. Il doit également souvent mettre en place des mécanismes de découverte et d'auto-organisation pour créer, organiser et maintenir l'état du réseau.

Cela peut se faire au niveau de l'application comme les solutions d'agents mobiles le proposent. L'organisation du réseau peut également se faire au niveau de l'intergiciel ou des couches réseau. Certaines solutions tentent de regrouper les communications entre voisins qui s'organisent entre-eux. D'autres solutions proposent des algorithmes génériques

de routage par clé utilisables pour plusieurs situations. Enfin, certains systèmes proposent d'échanger du code exécutable dans le but de reprogrammer le réseau selon le contexte.

Agents mobiles Les agents mobiles sont des unités logicielles mobiles et autonomes. Un système d'agents mobiles propose généralement des services de déportation, de clonage, de destruction du code, etc. Chaque agent s'occupe alors de répondre à ses objectifs propres de la manière la plus adéquate. Lorsqu'un agent se déplace sur le réseau, le code de cet agent est déplacé ainsi que son contexte d'exécution.

Dans ce sens, un système multi-agents est un système distribué qui entraîne généralement une grande mobilité du code sur le réseau. Cela entraîne de grands besoins en ressources de communication, ce qui est peu adapté au contexte des réseaux de capteurs. La plupart des implémentations connues de systèmes multi-agents sont réalisées sur une base de machine virtuelle [25].

Cependant, une approche de programmation par agents mobiles adaptée aux réseaux de capteurs pourrait permettre de limiter les coûts de déplacement des agents en répliquant le code à travers le réseau par exemple [35].

Regroupement logique et spatial explicite Certaines solutions proposent d'exprimer les rôles de chaque noeud de manière déclarative. *EnviroTrack* [1] est un système cherchant à traquer des véhicules militaires dans des environnements répartis. Chaque noeud prend le rôle de centralisateur de données selon sa proximité au véhicule observé. Cela permet d'optimiser les communications réseau, puisque l'aggrégation des données se fait au plus proche du phénomène observé, et non pas au dernier moment.

Suivant le même principe mais de manière plus générique, les *Generic Roles Assignment* [59] proposent de décrire les rôles que prennent les noeuds à partir de contraintes exprimées sur leurs données. Par exemple, un noeud peut prendre le rôle d'aggréger les données de ses voisins, s'il a une position centrale. Les contraintes sur les données sont exprimées dans la même logique déclarative que pour des bases de données par exemple, même s'il ne s'agit pas ici uniquement de recueillir des données. Le modèle des *Generic Roles Assignment* fournit également une bonne expressivité sur les contraintes en temps et en localisation en plus des contraintes sur les valeurs des capteurs.

Enfin, les *Abstract Regions* [74] proposent de regrouper les noeuds du réseau en régions logiques afin de leur attribuer des tâches.

Globalement ces trois solutions se ressemblent fortement et imposent au développeur d'application de décrire des régions – logiques ou spatiales – de manière explicite pour ensuite pouvoir attribuer des rôles aux noeuds appartenant à ces régions.

Organisation autonome du réseau Certains réseaux par leur architecture proposent intrinsèquement une organisation autonome, distribuée, extensible ou encore tolérante aux pannes.

Les systèmes pair-à-pair répondent souvent à beaucoup d'aspects techniques comme le contrôle décentralisé, l'auto-organisation, l'adaptation ou l'extensibilité. Ils peuvent être définis comme des systèmes distribués collaboratifs, ayant tous des responsabilités identiques [20]. Les versions anciennes des réseaux pair-à-pair reposaient sur des serveurs centralisés pour localiser les pairs. Une deuxième génération de réseaux pair-à-pair inondaient le réseau – principe de *flooding* en anglais – pour découvrir les ressources disponibles, cependant, cette méthode n'est ni efficace, ni déterministe. Les nouveaux réseaux pair-à-pair proposent des solutions de tables de hachage distribuées qui permettent de localiser des ressources à travers le réseau en minimisant le coût – en terme de communications – et en garantissant de trouver la ressource recherchée.

Pastry [20] propose un algorithme de routage par clés – *Key-Based Routing* – présentant des qualités de décentralisation, d'extensibilité ou encore d'auto-organisation. Ce type de routage par clé est un bon support pour le développement de tables de hachage distribuées, de systèmes de publication/souscription [11] ou encore de système de fichiers distribués [61]. Malheureusement, l'implémentation originale de Pastry est réalisée dans le langage Java et inadaptée aux faibles ressources des réseaux de capteurs.

L'utilisation de tables de hachage distribuées dans le contexte des réseaux de capteurs est cependant une perspective assez nouvelle du domaine et quelques systèmes ont été proposés dans la littérature tels que GHT [56], VCP [4, 5], SSR [18] ou encore une adaptation de Pastry appelée ScatterPastry [2].

Reprogrammation dynamique La plupart des travaux de reprogrammation dynamique se basaient traditionnellement sur le principe de machine virtuelle. Le problème de telles solutions est qu'elles présentent un surcoût non négligeable pour les réseaux de capteurs : la place utilisée par une machine virtuelle est de la place en moins pour l'application. C'est pourquoi des solutions de chargement de code natif ont été proposées, notamment pour Contiki [21]. L'implémentation de Contiki suggère d'utiliser

le format ELF – *Executable and Linkable Format* – qui est un format standard utilisé notamment par le compilateur GCC[6]. Le format ELF permet de contenir du code exécutable natif ainsi que des informations de relogement pour permettre de lier le code dans la mémoire exécutable du processeur. Il permet également d'associer des symboles à certaines parties du code exécutable, comme le nom d'une fonction. Le chargement dynamique de code dans Contiki utilise alors ces informations pour lier le code exécutable dans la mémoire de manière dynamique.

L'utilisation du format ELF a néanmoins été identifiée comme inadaptée pour du code natif à destination de microcontrôleurs 8 bits ou 16 bits. C'est pourquoi d'autres formats de fichiers comme CELF [21] ou SELF [19] ont été conçus pour permettre les mêmes fonctionnalités de manière plus efficace en terme de taille de fichier.

Au delà de la faisabilité technique, la reprogrammation dynamique des noeuds laisse entrevoir un nombre important d'applications dans le contexte des réseaux de capteurs. La première application proposée est de permettre de mettre à jour le logiciel à distance pour simplifier la configuration et la maintenance des réseaux de capteurs.

Le système de composants FiGaRo [50] propose une approche automatisée de la mise à jour du logiciel sur les noeuds de capteurs. Chaque composant est versionné et la mise à jour ne se fait que si la nouvelle version est supérieure. De plus, il est possible de sélectionner les noeuds à reprogrammer parmis le réseau à l'aide de techniques similaires aux regroupements logiques. Cependant, ce système propose une approche événementielle de la reprogrammation dans laquelle un noeud ou un ordinateur central doit initier la mise à jour d'un composant. De plus, les liaisons possibles entre composants sont identifiées avec les interfaces uniquement, or, les interfaces sont très peu expressives par rapport au comportement des composants et un composant pourrait alors prendre la place d'un autre alors qu'il ne fournit pas la même fonctionnalité.

D'autres facultés plus ambitieuses que la simple mise à jour du logiciel peuvent par ailleurs être imaginées. Différents algorithmes de répartition du logiciel sur un réseau distribué ont été étudiés ainsi que leurs compromis [72]. Par exemple, la consommation en énergie à l'échelle du réseau peut être optimisée en utilisant un noeud central pour les communications, cependant, cette organisation empêche le réseau de perdurer dans le temps, suite à l'épuisement du noeud central. En opposition, il est

[6] http ://gcc.gnu.org/

possible de répartir les rôles sur différents noeuds, ce qui consommera plus d'énergie globale mais fera durer le réseau plus longtemps.

L'étude de la répartition des composants logiciels OSGi [7] sur un réseau avec l'aide des fonctionnalités de Pastry est proposée par S. Frenot [26]. Lorsqu'une application nécessite un composant, celui-ci est dupliqué sur plusieurs noeuds du réseau. Le réseau conserve donc un historique de l'utilisation des composants et les plus utilisés seront généralement les plus présents dans les différents lieux du réseau. Le système proposé est utilisé pour réaliser un dépôt distribué de composants pour le cadriciel OSGi. Ce système n'est néanmoins pas du tout adapté aux réseaux de capteurs et aucune réalisation comparable ne semble avoir été proposée dans ce contexte.

Enfin, le projet WiSeKit [65] va au delà de la simple reprogrammation des noeuds en étudiant les conditions dans lesquelles le réseau doit être reprogrammé. Ce projet dépasse les études déjà réalisées puisqu'il s'intéresse au *quoi* et au *quand* plutôt qu'au *comment* de la reprogrammation des RCSF, pour proposer une reconfiguration automatique du réseau très proche de la perspective d'informatique omniprésente autonome développée au laboratoire DOMUS.

2.4 Perspectives

Les réseaux de capteurs présentent donc des perspectives ambitieuses impliquant beaucoup de domaines. S'ils permettent d'entrevoir de nombreuses applications possibles, leur mise en place s'avère très complexe et des solutions d'intergiciels sont proposées pour en simplifier la programmation et le déploiement.

Pour simplifier le déploiement et la maintenance des systèmes informatiques omniprésents, l'approche d'informatique omniprésente autonome, développée notamment au laboratoire DOMUS, permet d'entrevoir des systèmes diffus qui s'auto-organisent avec un minimum d'intervention humaine. Ce genre de travaux s'intéresse pour le moment à des machines puissantes permettant d'exécuter des intergiciels haut-niveau comme OSGi par dessus une machine virtuelle Java. Le défi est donc d'adapter les concepts de haut-niveau de l'informatique omniprésente autonome, aux contraintes des réseaux de capteurs sans-fils.

L'utilisation de la reprogrammation dynamique du logiciel peut permettre de simplifier les processus d'auto-organisation des noeuds d'un RCSF en permettant de redéfinir le comportement de chaque noeud. Le principe de reprogrammation des équipements embarqués

[7] Open Service Gateway initiative (OSGi), `http://www.osgi.org/`

est déjà largement utilisé aujourd'hui, comme pour mettre à jour les micrologiciels, notamment avec l'aide des systèmes de *BootStrap Loader* (BSL) qui permettent de charger un nouveau programme au démarrage du processeur. Cependant, ces solutions ne permettent que le déploiement du nouveau programme en un seul bloc monolithique et nécessitent systématiquement le redémarrage du système.

Dans un premier temps, l'utilisation d'un système de programmation par composants dynamique permettrait le chargement dynamique de nouveaux morceaux du logiciel plutôt que du logiciel en entier. Par ailleurs, ce principe de conception permettrait également de reprogrammer le système sans nécessiter de redémarrage de la plateforme. Enfin, le déploiement de composants logiciels permet de limiter la quantité de code propagée sur le réseau et permet donc de réduire la bande passante utilisée pour la reconfiguration du réseau.

Si des systèmes de composants reprogrammables existent déjà, les possibilités qu'ils offrent sont pour le moment sous-estimées. Ces systèmes sont utilisés uniquement pour simplifier la reprogrammation des noeuds pendant la phase de développement et de mise au point des applications. Cependant, les principes de reprogrammation dynamique peuvent présenter un excellent support pour les principes véhiculés dans la notion d'informatique autonome. En effet, la reprogrammation du système peut permettre de reconfigurer, d'optimiser, de protéger ou de réparer le système. Par exemple, si l'un des noeuds du réseau disparait par manque d'énergie, un autre noeud peut prendre sa place en utilisant le logiciel utilisé par le noeud disparu.

Contrairement aux systèmes existants qui proposent une vision centralisée, événementielle et initiée par l'humain de la reprogrammation, le projet Nodeus soutient la vision d'un réseau autonome dans lequel chaque noeud peut se reprogrammer de manière autonome et décentralisée. Pour ce faire, le logiciel à reprogrammer doit être disponible de manière distribuée dans le réseau.

Actuellement, aucune solution adaptée aux RCSF n'existe pour répondre à ce besoin de logiciel distribué, c'est pourquoi une solution décentralisée, permettant à chaque noeud de charger localement un composant publié sur le réseau, serait un bon point de départ à la mise en place de solutions d'informatique autonome.

Pour implémenter une telle solution décentralisée, des solutions de réseaux pair-à-pair peuvent être adaptées puisqu'elles présentent une organisation robuste et auto-adaptative. Cependant, les algorithmes existants sont souvent consommateurs en ressources réseau et donc en énergie et il convient de proposer des solutions prenant en compte cet aspect.

CHAPITRE 3

NODECOM : UN SYSTÈME DE COMPOSANTS LÉGER ET HYBRIDE POUR LES RÉSEAUX DE CAPTEURS SANS-FILS

La première étape de ce projet de maîtrise a consisté à mettre en place une solution de programmation par composants adaptée aux ressources contraintes des réseaux de capteurs sans-fils (RCSF).

Si un certain nombre de systèmes de composants pour les réseaux de capteurs ont déjà été présentés dans des travaux passés, la plupart proposent une vision statique des composants qui sont liés au moment de la compilation et ne peuvent pas être modifiés dynamiquement. Récemment, des systèmes de composants dynamiques ont été proposés, parmis lesquels beaucoup de systèmes basés sur une machine virtuelle. Le système d'exploitation Lorien [54] est un système orienté composants, léger, adapté aux réseaux de capteurs et permettant la reprogrammation dynamique de ses composants en *code natif*. Cependant, l'implémentation de Lorien, si conceptuellement intéressante, présente des difficultés liées à la portabilité sur de nouvelles plateformes et à la consommation en mémoire des méta-informations nécessaires à tous les composants dynamiques.

D'un autre côté, des systèmes comme FiGaRo [50], utilisant le principe des *services* de Contiki [22] pour proposer un système de composants dynamique, nécessite l'utilisation sous-jacente de Contiki et ne participe pas à une vision entièrement orientée composants du système.

Pour ces raisons, le système de composants hybride *Nodecom* a été réalisé en partenariat avec Alexandre Malo. L'implémentation et les choix architecturaux de ce nouveau système de composants pour les réseaux de capteurs ont mené à la rédaction d'un article soumis à la conférence DCOSS 2011. Le présent chapitre est une version adaptée et traduite en français de cet article.

Nodecom est donc un système de composants léger et hybride qui vise les RCSF. Si le domaine a intégré la programmation orientée composants depuis longtemps, les travaux de reprogrammation dynamique ont ouvert de nouvelles perspectives pour le déploiement autonome d'applications diffuses. Cependant, une étude récente comparant les systèmes de

composants statiques et dynamiques laisse à penser qu'une solution hybride pourrait offrir un compromis intéressant entre ces deux solutions [55]. Le système de composants *Nodecom*, réalisé durant ce projet, se présente ainsi comme le premier système de composants hybride pour les RCSF.

3.1 Introduction

Nodecom est un nouveau système de composants adapté aux très faibles ressources disponibles sur les plateformes des RCSF. Son but premier est de réduire la difficulté à développer de nouvelles applications et à les déployer sur les plateformes des RCSF. Avec ce concept de système de composants hybride, de nouvelles perspectives peuvent être entrevues telles que l'informatique omniprésente autonome. En mélangeant les composants statiques et dynamiques dans un seul système, Nodecom minimise l'impact d'un système uniquement dynamique tout en conservant ses avantages.

Les RCSF sont des solutions qui peuvent simplifier et réduire le coût d'installation des capteurs dans l'environnement puisqu'ils évitent d'avoir à filer un réseau entier sur un serveur. La plupart des RCSF d'aujourd'hui se focalisent sur la collecte des données et l'envoi de celles-ci vers un serveur centralisé qui traîte ces données et réagit de manière adéquate. Cependant, les approches de systèmes collaboratifs distribués sont une grande opportunité d'améliorer la fiabilité des RCSF.

Le laboratoire DOMUS a introduit récemment la vision d'informatique omniprésente autonome [30]. Cette vision de l'informatique consiste à coupler les principes de l'informatique omniprésente à l'informatique autonome. L'informatique omniprésente consiste en un ensemble de systèmes communicants répartis dans un environnement et traitant les événements au plus proche de cet environnement. L'informatique autonome consiste en des systèmes qui s'organisent eux-mêmes en évitant au maximum l'intervention humaine. C'est dans la continuité de cette vision que le système Nodecom a été élaboré. L'informatique omniprésente autonome peut être utilisée pour minimiser les coûts de gestion et de maintenance des espaces intelligents. Nodecom repousse les limites de cette vision en étant adapté au matériel extrêmement limité des RCSF. Nodecom a ainsi pu être validé sur une plateforme ne disposant que de 32ko de mémoire Flash et 1ko de mémoire RAM.

Dans cette vision de l'informatique omniprésente autonome, les systèmes sont dynamiques et utilisent la programmation orientée composants (POC). De ce fait, la POC et le chargement dynamique de code devraient être deux sujets de recherche importants pour les RCSF, car ils simplifient la reconfiguration des RCSF et permettent d'adapter et équilibrer

les RCSF en automatisant la propagation du code sur le réseau par exemple. De ce fait ils permettent d'améliorer la tolérance aux fautes et réduisent la complexité d'installation.

3.1.1 La programmation orientée composants appliquée aux RCSF

La POC cherche à diviser le code en composants qui interagissent. Les fonctionnalités des composants sont décrites de manière explicite à travers des *interfaces*. Chaque *type de composant* doit déclarer quelle(s) interface(s) il fournit et quelle(s) interface(s) il requiert. Cette méthode permet de découpler l'implémentation des morceaux de code (types de composants) de leurs spécifications (*interface fournie*) et de leurs dépendances (*interfaces requises*). Idéalement, une application peut être ainsi résumée à un ensemble de composants reliés les uns aux autres.

3.1.2 Le chargement dynamique dans les RCSF

Les récentes avancées sur le chargement dynamique ont fournit de nouvelles possibilités pour simplifier la logique, le déploiement et la reconfiguration des RCSF [21]. La plupart des systèmes de chargement dynamique se basent sur le format ELF (Executable and Linkable Format). Les sections principales d'un fichier ELF sont *bss*, *data*, *text*, *relocation table*, *symbol table* et *string table*. Les data et bss sont les sections qui contiennent les variables globales du programme et sont généralement allouées dans la RAM. La section text contient le code exécutable du programme et est placée dans la ROM. Les tables restantes sont utilisées pour lier le code et le reloger dans un position fixée de la mémoire exécutable.

Le reste de ce chapitre est organisé de la manière suivante. La section 3.2 présente les fonctionnalités principales et les bases qui ont mené à la création de Nodecom. La section 3.3 introduit l'architecture hybride et le méta-modèle de Nodecom. La section 3.4 présente l'implémentation du système. La section 3.5 présente l'évaluation et les performances de notre implémentation. Les systèmes existants et leurs limites seront développés à la section 3.6. Enfin, des propositions de travaux futurs seront apportées.

3.2 Présentation de Nodecom

Nodecom inclut un ensemble de composants statiques qui permettent l'utilisation de composants dynamiques. Cet ensemble de composants est appelé le *noyau* et contient des composants tels qu'un chargeur de fichiers ELF, un système de fichiers ou encore le *ges-*

tionnaire de composants de Nodecom. Ce gestionnaire de composants garantit l'intégrité du *méta-modèle* qui consiste en un ensemble d'informations qu'il utilise pour conserver une trâce des composants existants et de leurs interconnexions.

Les caractéristiques principales de Nodecom sont les suivantes :

Chargement dynamique de composants Nodecom fournit le chargement dynamique de nouveaux types de composants, l'instantiation dynamique et la reconfiguration dynamique des liaisons entre les composants. Il a été conçu spécialement pour l'échange de code sur un RCSF.

Noyau statique léger Un noyau minimal est proposé afin de laisser de la place pour le code applicatif, les fonctionnalités du système d'exploitation et les couches d'abstraction du matériel. Le noyau est composé de composants statiques ce qui le rend plus léger qu'un système entièrement dynamique [55]. Cette architecture hybride sera présentée à la section 3.3.

Facilement portable Nodecom est conçu pour des plateformes extrêmement contraintes qui utilisent des microcontrôleurs. Le code de Nodecom est écrit en C standard et est prévu pour être facilement portable sur de nouvelles plateformes de RCSF.

Utilise des standards À part la portabilité, Nodecom utilise des outils et des formats répandus. Chaque type de composant est distribué sous la forme d'un fichier ELF standard contenant du code natif. Le format ELF est le format par défaut de la célèbre chaîne de développement GCC[1]. Cela permet en particulier d'utiliser l'ensemble des outils de cette suite – les *binutils*.

Les RCSF ont des besoins spécifiques dûs à la faible dimension physique des noeuds, au manque d'accessibilité du matériel par les utilisateurs ou par d'éventuels techniciens et à l'absence d'énergie [48]. Le matériel est conçu pour répondre à ces contraintes, par exemple, une plateforme typique est le noeud Tmote Sky [53] qui fournit seulement 10ko de mémoire RAM et 48ko de mémoire Flash pour un CPU cadencé à 8MHz. Sur ce type de plateforme, l'impact d'un langage interprété ou de couches logicielles haut-niveau est considérable même s'il n'est pas inenvisageable [3]. À partir de ces limitations, un système de composants doit fournir uniquement les fonctionnalités de base nécessaires pour manipuler des composants et devrait avoir une empreinte mémoire très faible pour laisser de l'espace aux applications.

[1] `http://gcc.gnu.org/`

Au delà des besoins spécifiques des RCSF, l'informatique omniprésente a ses propres besoins. Le système doit être à même de se mettre à jour de manière autonome et de s'adapter à de nouvelles conditions. Les nouvelles conditions peuvent par exemple consister en un noeud qui est déplacé, une erreur sur un noeud donné ou une nouvelle personne à observer avec de nouveaux besoins. Un moyen pour mettre en place cette adaptation est de permettre la reconfiguration dynamique du réseau. Un système de composants dynamique peut simplifier ces mécanismes d'adaptation au contexte. Cependant, les systèmes de composants dynamiques ont été démontrés comme étant significativement plus consommateurs en ressources que les systèmes statiques [55].

Les recherches passées ont également prouvé que les communications sont la partie critique dans la consommation d'énergie des RCSF. À partir de ce fait, A. Dunkels a prouvé que les solutions à base de machines virtuelles sont un meilleur choix pour les réseaux hautement dynamiques dans lesquels le code doit migrer souvent [21]. En contrepartie, l'utilisation de code natif s'inscrit plus dans des environnements où l'échange de code est occasionnel. Par exemple dans le contexte des habitats intelligents, la mise à jour du code et l'organisation des processus sont un besoin de dynamicité sporadique [30] et l'utilisation de code natif y est donc plus intéressante.

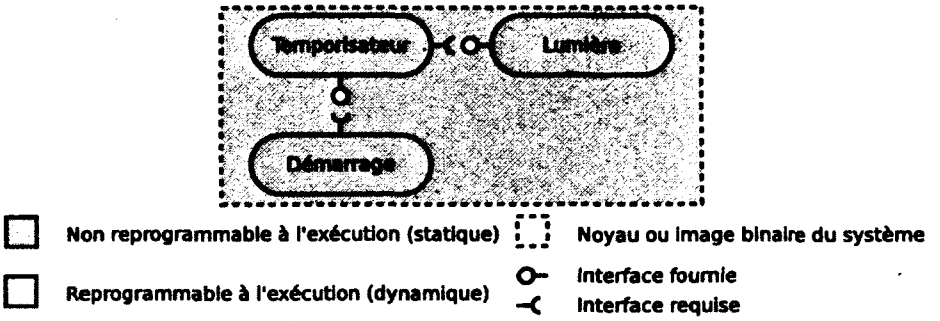
À partir des besoins précédemment identifiés, les hypothèses principales choisies pour la conception de Nodecom sont les suivantes :

- la mémoire des plateformes est extrêmement limitée ainsi que les capacités CPU ;
- le noyau et les composants seront compilés en code natif ;
- et le système est orienté pour des applications dont le dynamisme est occasionnel.

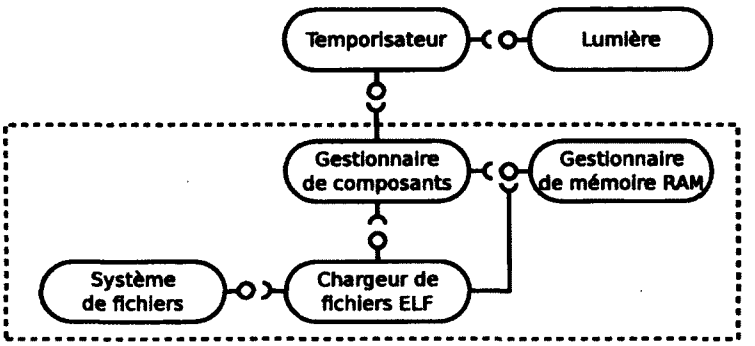
3.3 Une architecture hybride

Nodecom présente une architecture orientée composants hybride et propose un compromis entre les solutions purement dynamiques et purement statiques.

La plupart des solutions de POC existantes sont statiques [63]. La figure 3.1a représente un système statique dans lequel le paradigme est utilisé seulement au moment du développement des applications. Ensuite, une image monolithique du système complet est compilée et téléchargée dans chaque noeud puis exécutée telle quelle. Le code du système n'évolue pas au cours du temps, seules les données traîtées sont dynamiques. C'est l'architecture qui est notamment utilisée dans TinyOS [28].



(a) Système purement statique



(b) Système purement dynamique

Figure 3.1 Architectures classiques de systèmes de POC

Des solutions ont également été présentées pour fournir un système de composants purement dynamique adapté aux RCSF. La figure 3.1b présente un système de composants purement dynamique dans lequel tous les composants peuvent être chargés dynamiquement dans la mémoire exécutable et interconnectés pendant l'éxécution avec d'autres composants du système. Ce type de système peut donc évoluer à travers le temps et le logiciel peut être échangé entre les noeuds de la même manière que des données. C'est notamment l'architecture proposée dans le projet OpenCom [16].

Le paradigme de POC ne doit pas impliquer une surchage trop grande des ressources. Les systèmes de POC statiques ont déjà atteint des empreintes mémoires plus faibles que certains systèmes statiques traditionnels en maximisant la réutilisabilité du code [66]. Les systèmes dynamiques devraient également suivre cette tendance. Cependant, au jour d'aujourd'hui, les systèmes de POC dynamiques impliquent un coût énorme en utilisation des ressources. Pour utiliser un seul composant, plusieurs méta-données doivent être générées et conservées et plusieurs cycles CPU doivent êtres ajoutés [55].

Pour minimiser l'utilisation des ressources des systèmes de POC, Nodecom propose une architecture hybride. Cette architecture permet de faire cohabiter des composants statiques et dynamiques. Des concepts similaires de reprogrammation de bibliothèques dans Contiki

ont été proposés [22], cependant, à notre connaissance, aucune architecture hybride n'a jamais été proposée dans les contexte de la POC et des RCSF.

Pour supporter le chargement de composants dynamiques, Nodecom utilise un gestionnaire de composants. Celui-ci est responsable de charger les types de composants à partir de fichiers ELF exécutables stockés dans le système de fichiers. Cela implique des allocations dynamiques de la mémoire. Il est également responsable d'instancier les types de composants en composants. La figure 3.2 présente le gestionnaire de composants et ses dépendances formant le noyau statique léger de Nodecom. Les composants à l'extérieur du rectangle grisé sont un exemple de composants dynamiques simples pour une application responsable de faire clignoter une diode électro-luminescente (DEL).

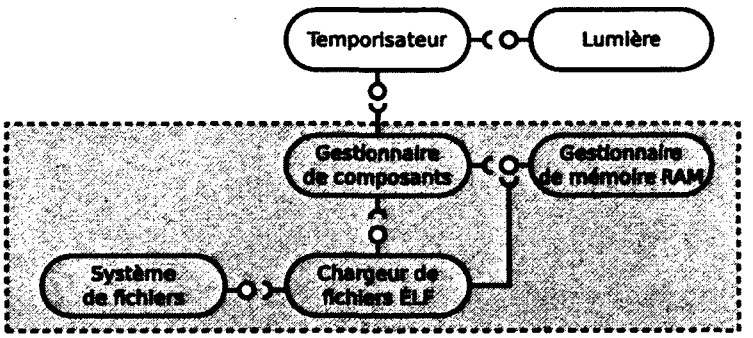


Figure 3.2 Architecture hybride de Nodecom

Les composants constituant le noyau sont coûteux à changer pendant l'éxécution et l'impact d'une architecture purement dynamique n'est pas justifiée pour le type d'application visée. Par exemple, vouloir changer dynamiquement le système de fichiers d'un noeud est une opération très complexe et dangereuse qui nécessite des opérations de transfert de données entre les composants. Ce type d'opérations couteuses est évité si des composants statiques non remplaçables sont utilisés. Les composants du noyau de Nodecom sont les suivants :

Gestionnaire de composants En tant qu'élément principal de Nodecom, le gestionnaire de composants fournit les fonctionnalités de chargement de types de composant, d'instantiation et de liaison des composants. Il est également responsable de mettre à jour le méta-modèle et d'en maintenir l'intégrité.

Chargeur de fichiers ELF Le chargeur de fichiers ELF est utilisé pour charger dynamiquement les fichiers ELF qui sont utilisés pour empaqueter les types de composants. Il se charge de l'édition des liens et du relogement du code dans la mémoire exécutable.

Système de fichiers Le système de fichiers est utilisé pour entreposer les types de composants sous la forme de fichiers ELF. Puisqu'il gère la totalité de la mémoire ROM, il est également utilisé pour allouer de la mémoire exécutable pour le code relogé.

Gestionnaire de mémoire RAM Le gestionnaire de mémoire RAM est utilisé pour allouer de la mémoire volatile. Le chargeur de fichiers ELF s'en sert pour allouer la mémoire volatile nécessaire aux programmes chargés. Le gestionnaire de composants l'utilise pour adapter la taille du méta-modèle selon le nombre de composants chargés.

Contrairement aux systèmes statiques, les systèmes dynamiques nécessitent de conserver des informations pour chacun des composants, telles que le type de composant instancié, l'emplacement du code dans la mémoire, ou encore les interconnexions entre les composants. Ces informations sont appelées méta-données et représentent l'état réel de chaque composant. Dans Nodecom, ces méta-données sont organisées suivant un méta-modèle qui définit quelles informations sont conservées pour chaque composant. Ce méta-modèle consiste en un ensemble de structures C.

Dans un scénario simple de RCSF, un certain nombre de composants peuvent être nécessaires. Comme la consommation en mémoire est une contrainte importante, les méta-données conservées pour chaque composant doivent être minimales. C'est pourquoi le système de composants ne propose que des fonctionnalités réduites. La figure 3.3 présente le méta-modèle utilisé par Nodecom et donc les informations conservées pour chaque type de composant.

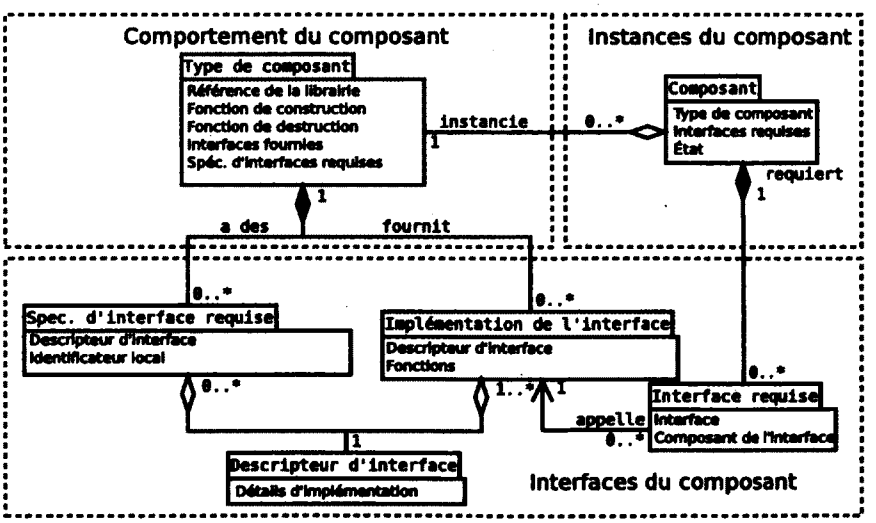


Figure 3.3 Méta-modèle utilisé par Nodecom

Le méta-modèle est divisé en trois parties. La partie *comportementale* contient les informations qui peuvent être réutilisées par chaque instance d'un même type de composant. Le type de composant ne sera ainsi créé qu'une seule fois même s'il est instancié plusieurs fois. La partie *contextuelle* conserve les informations de chaque instance de composant, c'est à dire l'état ou le contexte du composant ainsi que ses interconnexions à un instant donné. La partie *structurelle* fournit les informations qui permettent aux composants d'être liés entre eux à travers des interfaces requises et fournies. L'idée principale derrière cette décomposition du méta-modèle est de réutiliser certaines informations et ainsi éviter de les dupliquer, dans le but de sauver l'utilisation en mémoire.

Chaque type de composant implémente au moins une interface. Les interfaces sont identifiées à l'aide de *descripteurs d'interfaces* qui sont typiquement le nom de l'interface dans une chaîne de caractères. Chaque interface définit les fonctions qu'il faut nécessairement fournir pour l'implémenter. Les types de composant doivent nécessairement fournir des fonctions répondant aux prototypes décrits par leurs interfaces.

Chaque type de composant peut nécessiter les services d'autres composants. Il définit ces services à l'aide du descripteur de l'interface requise. N'importe quel composant implémentant les interfaces requises pourra ensuite lui être connecté pour achever son rôle.

La figure 3.4 présente un exemple des méta-données conservées pour deux composants interconnectés. Ces deux composants sont un temporisateur et une lumière. Le tableau 3.1 présente les adresses hypothétiques des fonctions fournies par les composants utilisés pour l'exemple. Les flèches représentent des pointeurs.

Tableau 3.1 Adresses en mémoire des fonctions fournies par les composants temporisateur et lumière

Fonction	Adresse en mémoire
construct	0x1234
destruct	0x1456
run	0x1678

Adresses des fonctions fournies par le composant temporisateur

Fonction	Adresse en mémoire
construct	0x2123
destruct	0x2345
turnOn	0x2567
turnOff	0x2789
toggle	0x2901

Adresses des fonctions fournies par le composant lumière

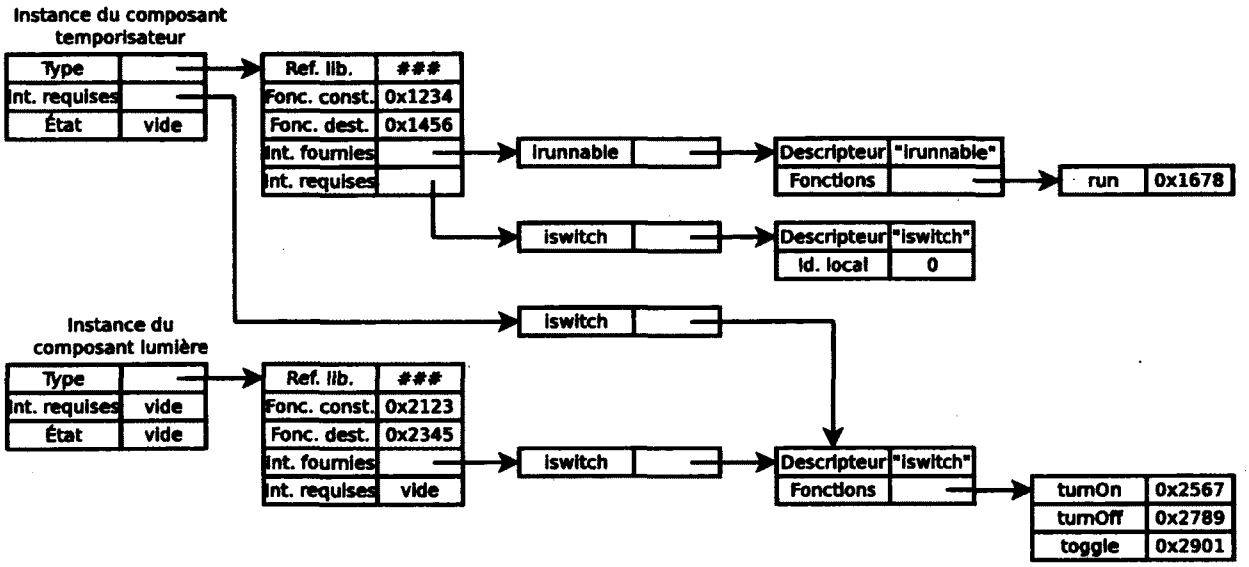


Figure 3.4 Exemple de méta-données pour deux composants temporisateur et lumière inter-connectés

3.4 Implémentation

Le noyau de Nodecom est un ensemble de composants statiques. Les composants principaux sont le chargeur de fichiers ELF et le gestionnaire de composants. Le noyau est également composé d'un système de fichiers et d'un gestionnaire de mémoire RAM pour l'allocation dynamique de mémoire. Cette section détaille les choix d'implémentation de chacun de ces composants mis à part le gestionnaire de RAM. Actuellement, ce composant utilise directement les fonctions *malloc* et *free* de la librairie standard du langage C.

3.4.1 Gestionnaire de composants

Le gestionnaire de composants fournit une API (*Application Programming Interface*) pour manipuler les composants du noeud local. Il est responsable de mettre à jour le méta-modèle et de garantir son intégrité. Chaque composant peut utiliser les fonctionnalités proposées par le gestionnaire de composants. Ces fonctionnalités sont les suivantes :

Charger un type de composant (figure 3.5a)

La procédure de chargement tente de charger un fichier ELF en tant que nouveau type de composant. Cette procédure peut être appelée dynamiquement, ce qui est utilisé pour déplacer du code entre plusieurs plateformes à l'exécution. Le processus de chargement dispose également de son opposé permettant de décharger un type de composant.

Ces processus de chargement et déchargement appelerons respectivement les fonctions `load` et `unload` fournies par le type de composant *après* le chargement ou *avant* le déchargement du type de composant. La fonction `load` est utilisée par le type de composant pour déclarer les interfaces qu'il fournit et les interfaces qu'il requiert.

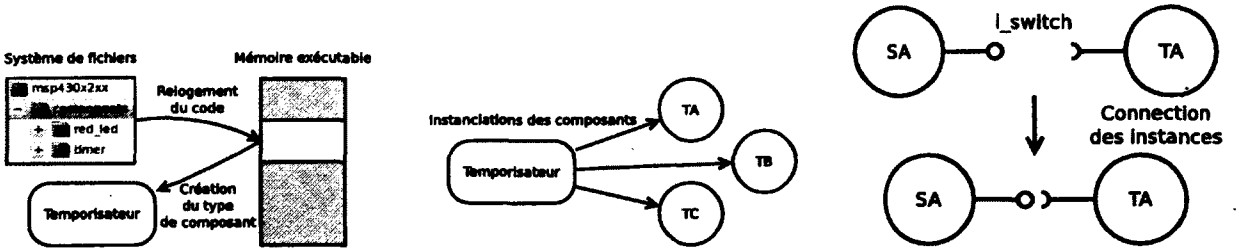
Instancier des composants (figure 3.5b)

Lorsqu'un type de composant est chargé, plusieurs instances peuvent en être créées. Chaque instance dispose de son propre contexte qui est appelé *état* (ou *state*). Cet état contient les variables locales accessibles au composant. Les instances créées peuvent également être détruites avec les fonctionnalités proposées par le gestionnaire de composant.

Lier des composants (figure 3.5c)

Lorsqu'un composant est créé, aucune de ses interfaces n'est encore reliée. Dans le processus de création d'un graphe de composants, les instances peuvent être liées à travers leurs interfaces respectives. Cela se fait en connectant les interfaces requises d'un composant aux interfaces fournies par d'autres composants comme un casse-tête. Les fonctions fournies par ces derniers seront alors accessibles au premier. Ces interconnexions peuvent par ailleurs être modifiées dynamiquement au cours de l'exécution du programme.

Avant de lier deux composants, le gestionnaire de composants s'occupe de vérifier que les interfaces à relier sont bien compatible. Pour ce faire, il utilise les descripteurs d'interface déclarés par chaque composant et vérifie qu'ils sont bien identiques.



(a) Chargement (b) Instanciation (c) Liaison

Figure 3.5 Opérations du gestionnaire de composants

3.4.2 Chargeur de fichiers ELF

Le chargement de types de composants s'inspire fortement du chargement de fichiers ELF de Contiki [21]. Cet exemple typique de chargement dynamique sur un microcontrôleur peut être divisé en quatre parties principales :

1. lire le fichier ELF et identifier les tailles des segments ;

2. allouer de la mémoire volatile (RAM) et exécutable (ROM) ;

3. reloger le code et lier les symboles avec le code déjà existant ;

4. copier le code lié et relogé dans la mémoire exécutable.

Pour les besoins de Nodecom, le chargeur de fichier ELF original de Contiki a été divisé en deux parties. D'abord les fonctionnalités de lecture et de relogement du format ELF ont été placées dans une bibliothèque séparée. Puis, le processus de chargement a été complètement modifié pour être adapté au paradigme de la programmation par composants. En effet, la POC implique des allocations de mémoire différentes et des appels de fonctions à des moments clés du chargement.

L'allocation de mémoire volatile est déjà prise en compte par le gestionnaire de composants à travers l'*état* de chaque instance. Cet état devrait être utilisé en remplacement des segments *bss* et *data* généralement présents dans les fichiers ELF. L'utilisation de l'état du composant réduit ainsi la complexité du chargement dynamique du code, car seule le segment *text* doit être relogé. Cependant, cela implique que l'utilisation de variables globales et statiques ne sont pas autorisées pour la programmation de composants. La mémoire nécessaire à ce type de variables n'est pas allouée par le chargeur. Ainsi, le chargeur de fichiers ELF et la bibliothèque de lecture du format ELF ont pu être réduits de manière significative pour se concentrer seulement sur le segment *text* contenant le code exécutable.

De plus, le relogement du code ainsi que le processus de liaison ne se focalisent maintenant que sur le relogement des symboles locaux. Aucun symbole global n'est lié, ce qui signifie que la table des symboles globaux n'est plus nécessaire. Cela est rendu possible avec l'utilisation d'interfaces entre les composants. Les liens entre composants sont indirects et aucun symbole public n'est visible. Cet avantage pose néanmoins quelques limitations. Il est par exemple impossible d'utiliser les bibliothèques standards du langage C directement. Le programmeur a le choix entre intégrer les fonctions de la bibliothèque standard dans chaque type de composant qui l'utilise, ou de créer un type de composant les intégrant et pouvant être lié aux autres composants.

3.4.3 Système de fichiers

La bibliothèque de lecture des fichiers ELF manipule des fichiers qui sont entreposés dans la mémoire ROM. Un moyen simple de gérer la mémoire ROM est de disposer d'une abstraction telle qu'un système de fichiers. Cependant, la plupart des microcontrôleurs

modernes utilisent de la mémoire Flash en tant que mémoire exécutable. La mémoire Flash présente des contraintes particulières, car il est nécessaire d'effacer des *secteurs* entiers de mémoire avant de pouvoir réécrire dessus. De plus, les opérations d'effaçage de ces secteurs de mémoire sont limitées, car la mémoire s'abîme à chaque effacement. Comme la mémoire est divisée en secteurs, certains secteurs peuvent être usés avant les autres si les opérations d'effacement et d'écriture ne sont pas réparties de manière uniforme sur toute la mémoire. L'usage massif d'un seul secteur userait la mémoire de manière prématurée.

Pour Nodecom, le système de fichiers Coffee [70] a été utilisé. Ce système de fichiers, proposé dans le système d'exploitation Contiki, vise particulièrement les besoins des mémoires Flash. Par ailleurs, il fournit des performances – en terme de rapidité – proches de l'utilisation directe de la mémoire et une utilisation uniforme de la plage d'adresses.

Dans l'implémentation actuelle de Nodecom, la mémoire exécutable est gérée entièrement par le système de fichiers, ce qui implique que l'allocation de mémoire exécutale est réduite à la simple création d'un fichier. Cela implique également, qu'il n'est pas nécessaire de prédire à l'avance la part de mémoire nécessaire pour le code exécutable et la part réservée au système de fichiers.

3.5 Mesures et évaluation

Nodecom a été évalué et comparé à d'autres solutions natives pour les RCSF. Nodecom a ainsi été démontré comme étant un bon compromis prévu par l'architecture hybride.

3.5.1 Mesures générales et évaluation

Seules des solutions en code natif sont étudiées dans cette section. Les machines virtuelles n'ont pas été étudiées, car elles trouvent leur application dans des environnements hautement dynamiques dans lesquels le code doit être déplacé très souvent. Le tableau 3.2 présente une comparaison entre les différentes solutions de POC ou de chargement dynamique. Cette comparaison prend en compte quatre types de systèmes natifs :

- les systèmes de POC statiques (TinyOS[28] et Remora[66])
- les systèmes hybrides ne permettant pas la POC (Contiki[21])
- les systèmes hybrides de POC (Nodecom)
- les systèmes de POC purement dynamiques (Lorien[54])

Tableau 3.2 Comparaison de systèmes natifs pour RCSF

	TinyOS	Remora	Contiki	Nodecom	Lorien
POC	x	x		x	x
Dynamique			x	x	x
ROM	0.43 ko	0.49 ko	5.69 ko	9.32 ko	28 ko
RAM	0.05 ko	0.02 ko	0.02 ko	0.14 ko	3 ko

La POC est un avantage et l'impact en mémoire de celle-ci peut être limité. La consommation en mémoire de Nodecom se place ainsi entre Contiki et Lorien.

Pour évaluer Nodecom, les besoins en ROM et RAM pour chaque partie du noyau ont été mesurés. Les mesures présentées dans le tableau 3.3 correspondent aux valeurs retournées par l'outil *msp430-size*, pour chacun des composants. La totalité du code a été écrit en C et compilé en utilisant la chaîne de développement *mspgcc* (version 4.4.5) avec l'optimisation d'empreinte mémoire activée (-Os).

Tableau 3.3 Empreinte mémoire des composants du noyau de Nodecom

Module	ROM (octets)	RAM (octets)
Gestionnaire de composants	1426	20
Chargeur de fichiers ELF	2782	0
Système de fichiers	5032	124
Gestionnaire de mémoire RAM	80	0
Total	9320	144

Ces mesures ne présentent que la partie statique de l'empreinte mémoire. Comme l'usage de la RAM présente également une partie dynamique, une évaluation de l'usage de la pile a été réalisée pour se faire une idée plus réelle de la consommation en mémoire de Nodecom. Pour ce faire, la mémoire RAM a été remplie avec un motif reconnaissable, comme 0xBEEF, puis des tests unitaires ont été lancés. Le maximum de consommation de la pile a été identifié dans le système de fichiers avec un usage de 430 octets. Si l'on ajoute ce maximum à l'utilisation statique de mémoire présentée au tableau 3.3, Nodecom requière typiquement 880 octets de mémoire RAM pour fonctionner.

3.5.2 Exemple d'utilisation : faire clignoter une lumière

Pour évaluer l'implémentation, un exemple d'application permettant de faire clignoter une DEL est proposé. Cette application très simple est représentée à la figure 3.6. Le principe consiste en un composant *temporisateur* qui bascule régulièrement l'état d'un autre composant *lumière*.

Cet exemple a été implémenté sur la plateforme de développement ez430-RF2500 qui utilise un microcontrôleur MSP430F2274 avec 32ko de mémoire Flash et 1ko de mémoire RAM.

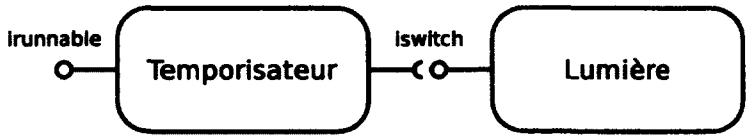


Figure 3.6 Modèle de l'application de clignotement d'une lumière

3.5.3 Impact des composants

Comme il a été mentionné précédemment à la section 3.3, le gestionnaire de composants est responsable de conserver les méta-informations de tous les composants. Ces méta-informations nécessitent beaucoup de mémoire, c'est pourquoi il faut réduire la quantité d'informations conservées au maximum. Le tableau 3.4 présente les besoins en mémoire par type de composant. Ces besoins dépendent du nombre d'instance par type de composant (n_I), du nombre d'interfaces fournies (n_{PI}) et requises (n_{RI}), du nombre total de fonctions fournies à travers toutes les interfaces (n_{PF}) et de la taille de l'état du composant (*state*). Ce tableau peut être résumé en une seule équation qui correspond à la quantité de mémoire nécessaire pour chaque type de composant :

$$18 + 8 \times n_{RI} + 8 \times n_{PI} + 2 \times n_{PF} + n_I \times (6 + state + 6 \times n_{RI})$$

Dans le cas de notre exemple de DEL qui clignote, les méta-informations présentent un usage en mémoire de 86 octets.

Tableau 3.4 Empreinte en mémoire des méta-informations

	Besoins en mémoire (octets)
Par type de composant	$18 + 4 \times n_{RI} + 4 \times n_{PI}$
Par instance de composant	$6 + 2 \times n_{RI} + state$
Par interface requise	$4 + 4 \times n_I$
Par interface fournie	$4 + 2 \times n_{PF}$

Les composants statiques à l'intérieur du noyau nécessitent la même quantité de mémoire que les composants dynamiques puisque le gestionnaire de composants conserve ces informations de la même manière. Cela permet l'interconnexion entre les composants statiques et dynamiques. Cependant, les composants statiques ne nécessitent pas d'informations de relogement ni d'un fichier ELF puisqu'ils sont déjà liés et relogés.

L'implémentation de Nodecom utilise des pointeurs de fonctions pour réaliser les communications entre composants. Cela implique un surplus de cycles d'exécution qui dépendent des processeurs. Ce surplus a été quantifié pour l'architecture MSP430. La figure 3.7 représente le code assembleur généré pour un appel direct de fonction par opposé à un appel à travers un pointeur de fonction. Le surplus est seulement d'une instruction supplémentaire par appel inter-composants. Comme le CPU n'est pas la partie la plus critique des RCSF (comparé aux communications sans-fils ou aux limitations en mémoire par exemple), ce surplus peut être considéré comme négligeable.

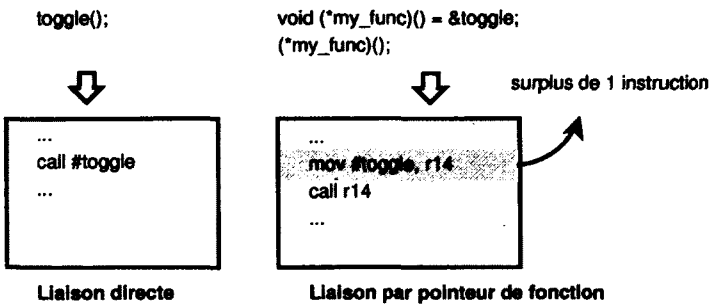


Figure 3.7 Impact de l'utilisation de pointeurs de fonction pour l'architecture MSP430

3.5.4 Chargeur de fichiers ELF

Comme le chargement de fichiers ELF a été repensé pour le paradigme de POC, aucune table des symboles n'est nécessaire pour le processus de liaison du code. Dans Contiki, l'empreinte mémoire de la table des symboles est estimée à 4ko [21]. En fait, ces informations sont stockées sous forme de méta-données dans Nodecom, c'est pourquoi la table des symboles n'a pas à être pré-allouée au moment de la compilation, car elle grandit et se réduit dynamiquement selon les besoins.

Le processus de chargement a également été simplifié puisque seulement les symboles internes sont relogés. L'allocation de mémoire pour les segments *bss* et *data* des programmes chargés est remplacée par l'allocation de l'état de chaque composant réalisée par le gestionnaire de composants. Le processus de chargement de fichiers ELF est ainsi hautement allégé comparé à l'implémentation originale présente dans Contiki.

3.6 Travaux connexes

La POC est une approche très populaire dans le domaine des RCSF. Le choix du langage nesC [28] pour l'implémentation de TinyOS [34] reflète cette tendance, de même que les

travaux récents de la fondation Eclipse et Texas Instruments sur RTSC et les XDCTools[2] qui proposent une nouvelle approche de la programmation sur les microcontrôleurs. L'objectif de la POC est d'apporter de la modularité et de la réutilisabilité dans la conception des systèmes logiciels [27].

Le projet Koala [51] propose également une solution orientée composants qui résout les problèmes de modularité lors du développement de systèmes embarqués très contraints.

Tous ces projets ne sont cependant pas adaptés dans une perspective d'informatique omniprésente autonome, puisqu'ils sont tous statiques. S'ils résolvent les problèmes de modularité au moment de la conception, ils ne fournissent aucun support pour la reprogrammation dynamique ou la reconfiguration dynamique.

OSGi[3] est une spécification de cadriciel bien implantée dans la monde Java et qui propose des fonctionnalités dynamiques. Ses fonctionnalités principales sont la reconfiguration dynamique, la gestion du cycle de vie, ou encore la liaison dynamique des *bundles* (composants). Seulement, OSGi est très orienté pour l'environnement Java et ne cible pas du tout les plateformes très contraintes des RCSF. Dans la même catégorie de systèmes se situent CORBA[4], DCOM[5], PCOM [6] et de nombreux autres.

REMORA [66] suggère une approche intéressante de la POC qui cible particulièrement les RCSF. Il propose un nouveau langage proche du C (avec un traducteur en C) et un cadriciel implémenté par dessus Contiki [22]. REMORA en lui-même est un système de composants statiques mais semble maintenant supporter la reconfiguration dynamique à travers un intergiciel nommé RemoWare. REMORA a été conçu comme une application par dessus Contiki ce qui implique que le système d'exploitation complet est nécessaire pour exécuter le noyau de REMORA. Comparé à un système purement orienté composants, cela implique un coût non négligeable même si REMORA en lui-même est léger.

OpenCOM [16] est le système qui a inspiré à l'origine le développement de Nodecom. Ce système de composants propose la manipulation dynamique de composants pendant l'exécution du système. Lorien [54], basé sur OpenCOM, apporte la vision d'un système complètement orientée composant et fournit un certain nombre de composants réutilisables. Cependant, même si les principes et le modèle d'OpenCOM ont été implémentés dans le langage C, le développement principal est focalisé sur l'implémentation en Java avec des considérations très coûteuses en ressources et plutôt inadaptées à la mémoire

[2] `http://rtsc.eclipse.org/`
[3] `http://www.osgi.org`
[4] `http://www.omg.org/spec/CORBA/3.1/`
[5] `http://msdn.microsoft.com/library/cc201989.aspx`

contrainte des plateformes de RCSF. Ces considérations sont par exemple, l'introspection du code ou la programmation orientée aspects [8]. Le projet OpenCOM se concentre sur plusieurs types de plateformes en même temps, mais il est légitime de penser que les RCSF ont leurs propres besoins.

3.7 Conclusion et travaux futurs

Ce chapitre introduit la vision d'un système de composants minimal pour les RCSF et adapté au matériel utilisé dans de tels réseaux. Un système de composants hybride est proposé et permet une utilisation mixte de composants statiques ou dynamiques de manière à minimiser l'impact sur les performances qu'un système purement dynamique apporte nécessairement. Nodecom devrait permettre de réaliser des applications diffuses pour les habitats intelligents et supporter l'auto-configuration des capteurs dans un tel environnement. Le reste de cette section présente les optimisations possibles de Nodecom.

D'abord, le chargement de fichiers ELF et le système de fichiers ont été identifiés comme les parties principales à améliorer. Le format ELF est en fait inadapté pour les systèmes légers. La plupart des microcontrôleurs ont une architecture 8-bit ou 16-bit et le format ELF est conçu pour supporter les architecture 32-bit. Quelques alternatives comme les formats SELF [19] et CELF [21] peuvent être utilisées en remplacement, mais un format dédié à la POC pourrait apporter des améliorations significatives en contenant directement les méta-données du composant.

Dans Nodecom, la mémoire exécutable est allouée avec le système de fichiers. Cela peut encore être amélioré en exécutant directement le segment *text* du fichier ELF une fois relogé. Cela éviterait de recopier le code et permettrait de sauver un peu d'espace mémoire.

L'utilisation de composants statiques requiert encore l'usage d'un nombre important de méta-données dans le but d'être interfaçable avec les composants dynamiques. Cependant, une solution devrait être envisagée pour limiter les méta-données nécessaires aux composants statiques.

L'allocation de la RAM utilise directement les fonctions *malloc* et *free* de la bibliothèque standard du C. Le processus d'allocation pourrait être amélioré en prenant en considération que les structures C allouées sont principalement celles du méta-modèle. Un algorithme de gestion de la mémoire plus sophistiqué pourrait permettre d'utiliser des blocs de mémoire de taille connu à l'avance et ainsi éviter les problèmes de fragmentation de la mémoire ou de dépassement du *heap* (bassin de mémoire dynamique constitué de blocs de taille fixe).

Le langage C s'avère également être très peu expressif pour la POC et la programmation de composants est rendue difficile. Un langage de POC standard permettrait de simplifier la programmation des composants, comme le nesC [28].

Finalement, à l'échelle du réseau, un certain nombre d'applications qui se reposeraient sur Nodecom peuvent être entrevues. Un système de dépôt distribué permettrait aux types de composants d'être partagés à travers le réseau et accessible depuis chaque noeud. À partir de là, il est possible de déployer certains composants sur le réseau de manière à optimiser certains aspects comme la qualité des communications ou l'efficacité énergétique du système global à l'échelle du réseau.

CHAPITRE 4

AUTO-ORGANISATION DU RÉSEAU À L'AIDE D'UNE TABLE DE HACHAGE DISTRIBUÉE

L'organisation centralisée des capteurs dans le monde de la domotique est vivement remise en question pour des raisons de coût et de complexité dans le processus d'installation et de maintenance. L'utilisation de réseaux de capteurs sans-fils est un début de solution à ce problème du filage, cependant, beaucoup de solutions de RCSF actuelles conservent une logique centralisée dans laquelle, les nombreux capteurs envoient leurs données à un ordinateur central responsable de traiter ces données. C'est pourquoi de plus en plus de solutions de RCSF s'orientent vers des architectures entièrement distribuées dans lesquelles les noeuds du RCSF s'organisent entre eux de manière autonome.

Les réseaux pair-à-pair classiques utilisés pour l'échange de fichiers sur Internet utilisaient historiquement des serveurs centralisés pour localiser les pairs disposant des fichiers recherchés. Cette architecture à-demi centralisée ayant mené à des poursuites judiciaires contre les fournisseurs des serveurs centraux, des solutions totalement distribuées ont été imaginées.

Les tables de hachage distribuées (DHT) ont donc été conçues pour les réseaux pair-à-pair classiques dans le but de distribuer totalement les rôles entre les pairs, de manière à ce que lorsqu'un pair du réseau disparaît (déconnexion du réseau, erreur matérielle, etc.) le reste du réseau puisse néanmoins continuer de fonctionner.

Les DHT utilisées dans les réseaux pair-à-pair classiques présentent de nombreux atouts tels que l'équilibrage des charges, la tolérance aux pannes ou encore l'auto-organisation [26, 60]. Ces atouts sont recherchés également dans le domaine des RCSF, cependant, les DHT classiques étant à la base adaptées aux réseaux filaires et souvent implémentées comme réseau *overlay* par dessus l'Internet, elles présentent également des limitations lorsqu'on souhaite les adapter aux RCSF. C'est pourquoi de nouvelles solutions ont été explorées pour réduire l'impact de l'utilisation de DHT dans les RCSF.

L'implémentation réalisée consiste en un algorithme de routage par clé dérivé de Pastry et ayant été adapté aux RCSF. L'empreinte mémoire de cette implémentation a été évaluée.

4.1 Principe du routage par clé

Les tables de hachages distribuées (DHT) sont une manière de choisir automatiquement un noeud responsable pour une ressource donnée. Pour ce faire, les DHT se reposent souvent sur des algorithmes de routage par clé. Pastry est un exemple d'algorithme de routage par clé utilisé pour les réseaux pair-à-pair sous la forme d'*overlay* par dessus Internet. Son fonctionnement sera expliqué, car il est à la base de nombreux autres systèmes. Enfin, les solutions de DHT classiques ont été adaptées aux fortes contraintes des RCSF.

4.1.1 Routage par clé

Le principe de routage par clé est à la base de la plupart des implémentations de DHT. Chaque noeud du réseau se voit attribuer un identifiant numérique unique. Lorsqu'un message doit être transmis sur le réseau, une clé est associée à ce message. Le choix de la clé associée à un message est laissé à l'application. Le message est alors routé vers le noeud dont l'identifiant est le plus proche de la clé du message. La proximité entre un identifiant et une clé correspond à une distance numérique déterminée.

Dans un réseau classique, un message est routé directement à son destinataire qui est connu à l'avance. Si le noeud destinataire n'existe pas ou a disparu, le message est perdu et ne sera pas traité par un autre noeud. Au contraire dans un réseau utilisant le routage par clé, il existe nécessairement un noeud destinataire pour le message. Ce principe est également appelé *routage indirect.*

4.1.2 Exemple d'application

Ce principe de routage par clé est utilisé dans plusieurs types d'applications. Des échanges de type publication/souscription peuvent par exemple être implémentés. Voici les étapes principales d'une communication de type publication/souscription avec un routage par clé :

1. Un noeud A souhaitant connaître la température du salon envoie un message `subscribe("temperature salon")`. La chaîne `"temperature salon"` est transformée en clé T avec une fonction de hachage et le message `SUBSCRIBE` est donc routé à la clé T.

2. Le noeud destinataire est le noeud C dont la clé est la plus proche de T. Il sauvegarde l'adresse du noeud A comme étant souscripteur aux publications sur le sujet `"temperature salon"`.

3. Un troisième noeud B souhaite désormais publier la température du salon acquise par son capteur. Il envoie donc un message `publish("temperature salon", 24)`. Le message `PUBLISH` est envoyé au noeud C car sa clé est la plus proche de T.

4. Le noeud destinataire est donc le noeud C qui sait que A est intéressé par ce type d'information. Il transfert alors le message `PUBLISH` au noeud A.

Cela permet de choisir de manière automatique le noeud responsable de transférer les informations de température du salon. L'organisation est donc autonome et distribuée. Pour rendre le routage par clé efficace, des tables de routage sont généralement maintenues pour localiser le noeud destinataire rapidement.

4.1.3 Pastry

Pastry [60] est un algorithme de routage par clé utilisé pour des applications pair-à-pair sur Internet. Il a été montré comme étant un support efficace et adapté pour de nombreuses applications, parmis lesquelles SCRIBE [11] qui propose un mécanisme de communication de type publication/souscription ou encore PAST [61] qui présente un système de fichiers distribué et sécurisé.

Chaque noeud d'un réseau Pastry se voit attribuer un identifiant unique (*nodeId*). Lorsqu'un message et une clé sont fournis à Pastry, celui-ci transmet le message au noeud dont le *nodeId* est le plus proche de la clé.

Pastry se présente comme un réseau *overlay* sur Internet, permettant de faire communiquer plus de 100 000 noeuds. Pour ce faire, il se repose sur l'utilisation d'une table de routage et d'une table des voisins logiques et permet ainsi de router les messages en un nombre de sauts d'ordre $O(logN)$ avec N le nombre de noeuds potentiels sur le réseau.

Pastry est un algorithme générique permettant de choisir la taille des clés et des identifiants et donc le nombre de noeuds potentiels du réseau. Dans la suite de cette section, pour simplifier la compréhension, la configuration de base de Pastry sera utilisée. Chaque clé/identifiant est composé de 128 bits, c'est-à-dire 32 chiffres hexadécimaux. La granularité est fixée à 4 bits, c'est à dire 1 chiffre hexadécimal. Il s'agit de la configuration de base de Pastry et celle-ci implique d'autres configurations présentées à la table 4.1. Il est cependant important de remarquer que le nombre de lignes de la table de routage correspond au nombre de chiffres hexadécimaux dans les clés/identifiants. De même, le nombre de colonnes de la table de routage correspond au nombre de valeurs possibles pour un chiffre hexadécimal (moins une).

Tableau 4.1 Paramètres de configuration de Pastry

Paramètre	Valeur théorique	Valeur choisie
Nombre de noeuds potentiels	N	2^{128}
Granularité	b	4
Taille des clés/identifiants (en bits)	$log_2 N$	128
Nombre de colonnes de la table de routage	$2^b - 1$	15
Nombre de lignes de la table de routage	$log_{2^b} N$	32
Nombre de lignes de la table des voisins logiques	2^b ou 2×2^b	16

Tables de routage

Pour garantir le nombre de sauts, Pastry entretien une table de routage et une table des voisins logiques. Chacune de ces tables contient une paire identifiant/adresse qui associe l'identifiant d'un noeud à son adresse IP.

Tout d'abord, la table des voisins logiques contient les 8 voisins logiques dont l'identifiant est supérieur à l'identifiant local et les 8 voisins logiques dont l'identifiant est inférieur à l'identifiant local. Ces 16 voisins logiques sont les noeuds du réseau dont l'identifiant est le plus proche numériquement de l'identifiant local et ne correspondent a priori pas aux voisins géographiques.

La table de routage est une table à deux dimensions contenant 15 colonnes et 32 lignes. Chacune des colonnes pour une ligne n se réfèrent à un noeud dont les n premiers chiffres de l'identifiant sont les mêmes que ceux de l'identifiant local, mais dont le chiffre à la place $n+1$ a l'une des 15 autres valeurs possibles que celle de l'identifiant local du noeud.

Le tableau 4.2 représente la table de routage d'un noeud hypothétique dont l'identifiant commencerait par 31F4. Chaque cellule dans la table de routage contient l'adresse réseau d'un autre noeud du réseau. Plus les noeuds enregistrés dans cette table ont un identifiant proche du noeud courant, plus ils sont bas dans le tableau jusqu'à n'avoir qu'un seul chiffre de différent. En pratique, toutes les cellules de la table de routage ne sont cependant pas toujours remplies.

Tableau 4.2 Exemple de table de routage

	0	1	2	3	4	...	F
0	0...	1...	2...		4...	...	F...
1	30...		32...	33...	34...	...	3F...
2	310...	311...	312...	313...	314...	...	
3	31F0...	31F1...	31F2...	31F3...		...	31FF...
...	...	...	...	...	...	...	...

Algorithme

L'algorithme de routage est tiré directement de l'article présentant Pastry [60] et est détaillé dans l'algorithme 1. Ce code est exécuté lorsqu'un message arrive avec la clé D au noeud dont l'identifiant est A. Les notations utilisées sont les suivantes :

D : clé du message à router
A : identifiant local
R : table de routage
L : table des voisins logiques
R_l^i : la cellule de la table de routage à la ligne l et à la colonne i.
L_i : la cellule de la table des voisins logiques à la position i avec $-8 \leq i \leq 8$. Une valeur négative (respectivement positive) indique que l'identifiant est inférieur (respectivement supérieur) à l'identifiant local.
D_l : la valeur du $l$ième chiffre de la clé D.
$shl(A, B)$: le nombre de chiffre dans le préfixe commun aux clés A et B.
$|A - B|$: la distance numérique entre les clés A et B.

Algorithme 1 Algorithme de routage de Pastry [60]

if $L_{-8} \leq D \leq L_8$ **then** {D est dans le rayon des voisins logiques}
  trouver L_i tel que $|D - L_i|$ est minimal
  if $L_i == A$ **then**
    délivrer le message au noeud local
  else
    transférer le message au noeud L_i
  end if
else
  {Essayer avec la table de routage}
  $l = shl(D, A)$;
  if $R_l^{D_l} \neq null$ **then**
    transférer le message au noeud $R_l^{D_l}$
  else {Cas rares}
    transférer le message à T tel que
    $T \in L \cup R$ et que
    $shl(T, D) \geq l$ et que
    $|T - D| < |A - D|$
  end if
end if

Discussion

Pastry est donc un algorithme de routage convergent qui garantit que le noeud destinataire obtiendra le message en un nombre de sauts maximum d'ordre $O(logN)$. Ce déterminisme et cette extensibilité à un grand nombre de noeuds se font néanmoins au détriment de la mémoire utilisée.

Sur un réseau IPv4, les adresses ont une longueur de 32 bits. Ajoutée à la longueur des identifiants de 128 bits, un seul élement de la table de routage consomme 160 bits soit 20 octets. Dans la configuration décrite de Pastry, la table de routage contient $15 \times 32 = 480$ éléments. La table des voisins logiques contient quant à elle 16 éléments. La quantité de mémoire nécessaire pour les tables de routage de Pastry est donc $(480 + 16) \times 20 = 9920$ octets. Si ce nombre est acceptable dans le contexte des ordinateurs de bureau, il s'agit de la quantité maximum de mémoire RAM disponible sur les plateformes de RCSF.

Pastry, par ailleurs, ne tient compte que de la proximité logique des noeuds. Ainsi, si l'algorithme permet de localiser un noeud en $log_{16}N$ sauts dans l'*overlay*, le nombre de sauts physiques est bien plus grand. Le réseau peut ainsi être parcouru plusieurs fois de long en large avant de délivrer le message au noeud final. La figure 4.1 présente un exemple dans lequel 4 sauts dans l'*overlay* traversent 2 fois le réseau physique en réalité.

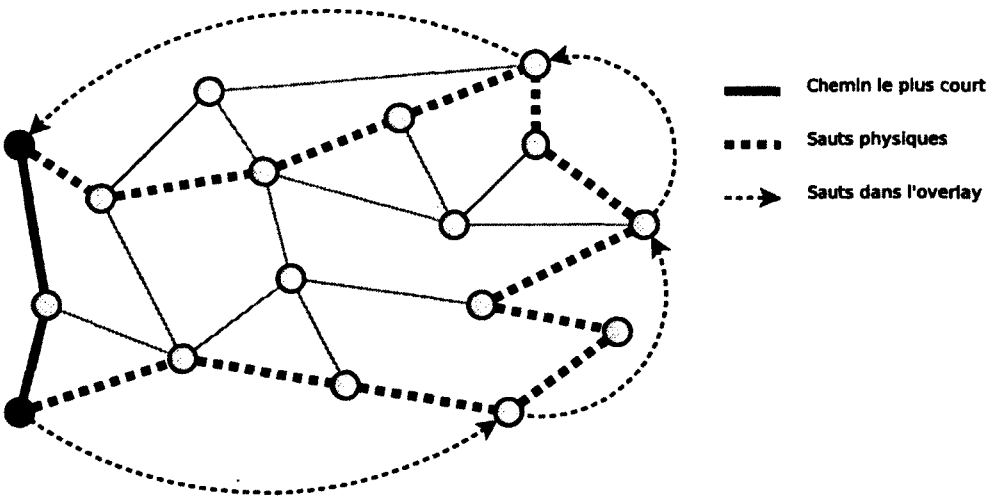


Figure 4.1 Routage dans l'overlay et routage physique [75]

Par ailleurs l'implémentation publique FreePastry est réalisée dans le langage Java et n'est donc pas utilisable directement sur un noeud de RCSF.

Pour pallier à ces difficultés, plusieurs implémentations ont été proposées et adaptées aux ressources très contraintes des RCSF.

4.2 Travaux connexes

Pastry n'est donc pas utilisable directement sur des plateformes de RCSF pour trois raisons principales :

1. Les tables de routage sont trop volumineuses.
2. L'organisation du réseau overlay ne tient pas compte de la proximité physique des noeuds et engendre un grand surcoût en ressources réseau.
3. L'utilisation d'un overlay est peu adaptée aux réseaux mobiles puisque les algorithmes de routage généralement utilisés pour la couche réseau, comme AODV, réalisent des *broadcasts* pour découvrir les autres noeuds, autant profiter alors de ces *broadcasts* pour envoyer la clé recherchée au même moment.
4. La maintenance des tables de routage nécessite l'échange régulier de messages vérifiant que les noeuds voisins sont encore en vie.

Par ailleurs, certaines fonctionnalités recherchées dans Pastry peuvent être limitées dans le contexte des RCSF. En effet, l'extensibilité du réseau se limite souvent à un millier de noeuds tout au plus dans le domaine de la domotique, contrairement à un réseau pair-à-pair classique qui cherche à faire communiquer des centaines de milliers de noeuds.

De plus, Pastry profite du fait que deux noeuds qui sont proches dans l'*overlay* sont très probablement éloignés géographiquement. De ce fait, les données peuvent être répliquées sur des voisins logiques sans être soumises aux mêmes conditions de propriété, ou de juridiction. Dans le contexte des RCSF, cette caractéristique n'est pas recherchée et entraine un surcoût notable en ressources réseau puisqu'elle empêche de tenir compte de la proximité physique des noeuds.

Plusieurs implémentations ont donc été proposées pour tenter de résoudre ces problèmes. Ces nouvelles implémentations utilisent plusieurs principes pour limiter les impacts négatifs des algorithmes pour ordinateurs fixes tout en conservant leurs avantages :

1. réduire la taille des clés/identifiants utilisés (ScatterPastry [2]) ;
2. tenir compte de la proximité physique des noeuds dans le choix des identifiants (GHT [56], MADPastry [75]) ;
3. intégrer le routage par clé à la couche réseau (MADPastry, ScatterPastry, GHT) ;
4. réduire la maintenance des tables de routage (MADPastry).

Réduction de la taille des clés/identifiants

Puisque les RCSF présentent un nombre bien plus réduit de noeuds par rapport aux réseaux pair-à-pair classiques, la taille des clés/identifiants peut être réduite tout en conservant une faible probabilité de collisions.

ScatterPastry [2] est une implémentation de Pastry pour les RCSF. Il propose de réduire la taille du réseau et la taille des clés/identifiants pour réduire l'empreinte mémoire de l'utilisation d'un tel algorithme sur les RCSF. La taille des clés y est réduite à 16 bits.

Identifiants géographiques

Les algorithmes de routage par clé se basent sur la proximité numérique des clés et ignorent souvent la proximité physique des noeuds. De ce fait, les messages peuvent parcourir des distances bien plus grandes que le chemin le plus court.

Pour connaître la localité approximative d'un noeud, deux solutions sont envisageables. D'abord, une table de routage pour les voisins physiques pourrait être maintenue, malgré que cette solution a un coût notable en mémoire. La seconde solution est d'intégrer l'information de localité dans les identifiants de chaque noeud. L'algorithme de routage peut alors optimiser ses choix en intégrant la gestion de proximité des noeuds. Cette information de localité n'a cependant pas besoin d'être précise.

GHT [56] utilise le principe de routage par clé. Les clés correspondent à des positions physiques de type (x, y) et l'identifiant d'un noeud correspond à sa position géographique. Cela implique que chaque noeud connaisse sa position géographique.

MADPastry [75] est organisé en *clusters*, c'est-à-dire en zones du réseau. Tous les noeuds d'un *cluster* partagent un même préfixe pour leur identifiant – par exemple le premier chiffre hexadécimal. Le routage avec la table de routage n'est utilisé que pour les sauts entre *clusters*. La table de routage originale de Pastry est donc réduite à une seule ligne dans le cas ou le préfixe est d'un seul chiffre.

Le routage à l'intérieur d'un même *cluster* se fait ensuite de proche en proche à l'aide de la table des voisins logiques. Même si algorithmiquement la localisation d'un noeud dans un *cluster* est loin d'être optimale ($O(n)$ avec n le nombre de noeuds dans un *cluster*), celle-ci reste raisonnable si l'on considère le faible nombre de noeuds par *cluster*.

De plus, comme MADPastry est un algorithme intégré à la couche réseau, les autres noeuds peuvent écouter les communications (sans-fils) et intercepter les messages s'ils leurs sont destinés.

Algorithme intégré à la couche réseau

L'utilisation d'un *overlay* implique de maintienir des tables de routage dans la couche application. Cependant, cela ne sert à rien si la couche réseau sous-jacente réalise des broadcasts systématiques pour découvrir les routes. En effet, quitte à réaliser des broadcasts systématiques, autant transmettre la clé du message dans ces broadcasts à la manière du *flooding* et ne pas maintenir de table de routage dans la couche application. Ce problème n'apparaît pas dans des réseaux filaires classiques puisqu'ils ne sont pas mobiles et s'appuient sur des tables de routage qui sont créées une bonne fois pour toute.

Pour contourner ce problème, MADPastry combine les techniques utilisées par Pastry et AODV et ramène donc Pastry dans la couche réseau. MADPastry étant ainsi intégré dans la couche réseau, chaque noeud qui se trouve sur le chemin d'un message peut décider d'intercepter le message s'il est lui-même plus proche de la clé ou s'il connait un autre noeud qui est plus proche de la clé. Cela permet d'éviter aux messages de parcourir le réseau de long en large avant de localiser le noeud de destination.

ScatterPastry propose également d'intégrer le routage par clés à la couche réseau, mais ne décrit pas les avantages de cette technique.

D'autres algorithmes comme SSR [18] et VRR [9] cherchent également à intégrer les concepts des *overlays* de types DHT à l'intérieur de la couche réseau.

Maintenance des tables de routage

La maintenance des tables de routage se fait habituellement par l'envoi régulier de messages pour s'assurer que les voisins sont toujours en vie. Cette maintenance explicite est bien sûr très couteuse en communications.

Dans MADPastry, la maintenance n'est effectuée que pour les deux voisins logiques les plus proches. En effet, ces deux voisins doivent nécessairement être à jour pour garantir la convergence de l'algorithme de routage. Le reste des tables n'est pas mis à jour explicitement mais seulement lorsqu'un noeud n'est pas joignable ou que l'on reçoit des nouvelles informations sur un noeud existant.

Discussion

MADPastry est une adaptation de Pastry pour les réseaux mobiles *ad-hoc* (ou MANETs) et n'est pas adaptée directement aux RCSF. Par exemple, elle ne cherche pas à réduire la taille des clés et identifiants et nécessite de ce fait une grande quantité de mémoire.

De plus, l'algorithme de routage de MADPastry a seulement été simulé avec le simulateur ns-2 et aucune implémentation ne semble exister.

ScatterPastry est une solution visant directement les RCSF et une implémentation en C existe pour le projet ScatterWeb. Seulement, le code source de cette implémentation est introuvable et les auteurs du projet sont injoignables.

Pour le présent projet, une nouvelle implémentation dans la ligne de Pastry a donc été réalisée en s'inspirant directement des solutions existantes dans la littérature.

4.3 Implémentation

Comme la plateforme mise en place au laboratoire repose sur des radios XBee compatibles avec le protocole ZigBee, la couche réseau est déjà implémentée dans la puce du XBee et n'est donc pas modifiable. Cela permet cependant une interopérabilité avec les équipements ZigBee, contrairement à une couche réseau non standard.

Cette première contrainte empêche d'intégrer le routage par clé à la couche réseau et impose l'implémentation d'un tel algorithme en tant qu'*overlay*. Néanmoins, il est possible de tirer parti des solutions déjà développées pour réduire le coût de l'implémentation.

4.3.1 Diagramme de composants

L'implémentation de Pastry proposée dans ce projet se repose sur Nodecom, le système de programmation orientée composants pour RCSF présenté précédemment. La figure 4.2 décrit les composants implémentés et leur organisation.

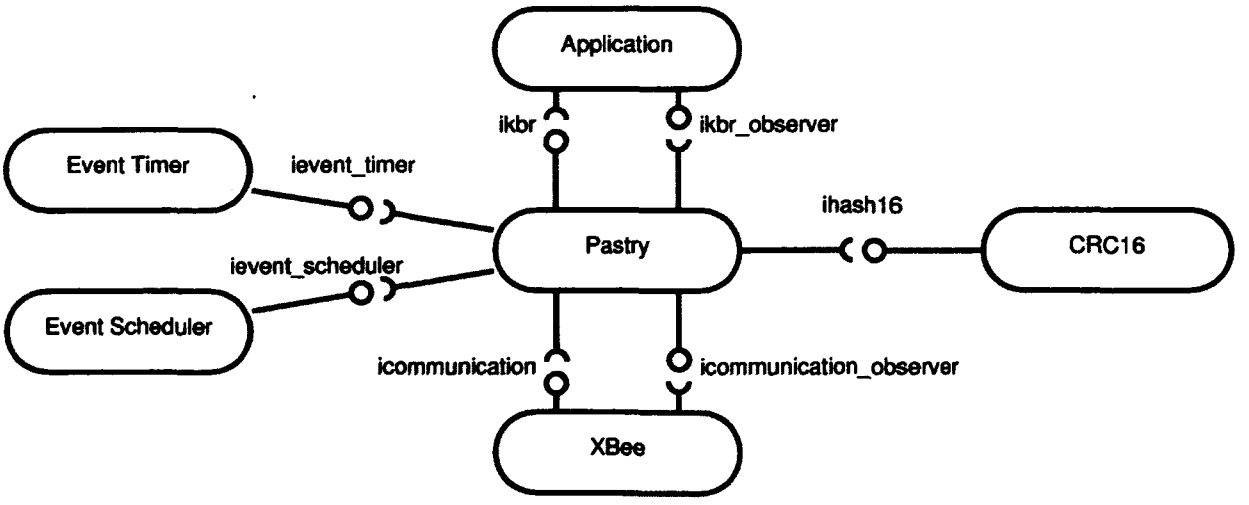


Figure 4.2 Diagramme de composants de l'algorithme de routage par clé

Le composant Pastry se repose sur 4 autres composants. Tout d'abord, il nécessite un composant de communication – en l'occurence le composant XBee – afin de pouvoir communiquer avec les autres noeuds du réseau. Une fonction de hachage en 16 bits est également nécessaire à la génération des clés – ici la fonction de hachage est la fonction CRC 16 bits (voir section 4.3.2). Enfin, le composant Pastry dispose de *protothreads* [23] à ordonnancer, c'est pourquoi il dépend également d'un ordonnanceur et d'un temporisateur.

Le composant réalisé propose par ailleurs l'interface `ikbr` (KBR = *Key-Based Routing*) qui permet d'envoyer des messages routés par clé sur le réseau. L'interface `ikbr_observer` doit être fournie par le composant qui recevra les messages entrants de Pastry. Voici une version simplifiée de la déclaration de ces deux interfaces :

```
DECLARE_INTERFACE(ikbr) {
    void (*initialize) (struct nodecom_component * component);
    void (*route) (struct nodecom_component * component, kbr_key_t key,
        uint8_t * message, size_t length);
    void (*get_local_key) (struct nodecom_component * component,
       kbr_key_t * key);
};

DECLARE_INTERFACE(ikbr_observer) {
    void (*deliver) (struct nodecom_component * component, kbr_key_t
       key, uint8_t * message, size_t length);
    void (*forward) (struct nodecom_component * component, kbr_key_t *
       next_id, kbr_key_t key, uint8_t * message, size_t length);
};
```

4.3.2 Réduction de la taille des clés

L'utilisation des XBee impose de conserver les adresses des noeuds au format IEEE pour communiquer en *unicast*. Une adresse IEEE – aussi souvent appelée adresse MAC – a une longueur de 64 bits soit 8 octets. Un réseau ZigBee associe des adresses de 16 bits ou 2 octets à chaque noeud. L'utilisation de l'adressage ZigBee permet d'améliorer les performances de routage sur le réseau. En effet, si l'adresse ZigBee de destination n'est pas connue, une découverte de la route sera réalisée avec des *broadcasts* (algorithme AODV). Cependant, les XBee imposent d'utiliser au moins l'adressage en 64 bits. Pour réduire l'empreinte mémoire, l'implémentation proposée utilise donc uniquement l'adresse 64 bits.

Puisque le nombre de noeuds d'un RCSF ne dépassera probablement jamais quelques centaines de noeuds, il est raisonnable de penser qu'une longueur de 16 bits pour les clés

et identifiants sera suffisante. Une clé de 16 bits permet en effet d'associer un identifiant à 65 536 noeuds.

Pour associer un identifiant à chaque noeud, une fonction de hachage est appliquée sur l'adresse MAC. Cette fonction de hachage retourne un identifiant de 16 bits avec une faible probabilité de collision des identifiants. Dans l'implémentation d'origine de Pastry, la fonction de hachage utilise l'algorithme SHA-1. Si des algorithmes de hachage sécuritaires tels que MD5 ou SHA-1 ont déjà été implémentés pour des microcontrôleurs, leur empreinte en mémoire et leur utilisation du processeur sont conséquentes. C'est pourquoi un simple algorithme de CRC 16 bits a été utilisé pour la génération des identifiants. Cet algorithme à la base utilisé pour générer des sommes de contrôle peut également servir de fonction de hachage peu sécuritaire. Néanmoins, cela suffit dans un réseau de petite taille.

De plus, le microcontrôleur MSP430 qui est utilisé dans ce projet dispose d'un périphérique interne permettant de calculer le CRC plus rapidement.

Puisque le nombre de noeuds du réseau a été réduit, le nombre de lignes de la table de routage peut également être réduit drastiquement. Ainsi, pour des identifiants de 16 bits, la table de routage est réduite à $log_{16}(65536) = 4$ lignes et 15 colonnes. La table des voisins logiques reste inchangée par rapport à l'implémentation d'origine de Pastry.

4.4 Évaluation

Parmis les principales critiques apportées à l'implémentation d'origine de Pastry, la consommation en mémoire des tables de routage est le point le plus important. En réduisant les clés à une longueur de 16 bits, l'empreinte des tables de routage a été grandement diminuée.

Enfin, dans ce projet, Pastry a été réalisé sous forme de composant dans Nodecom. Cela a permis l'utilisation de code natif – contrairement à du code interpreté – et l'empreinte mémoire de ce code a été mesurée.

4.4.1 Empreinte mémoire des tables de routage

La tableau 4.3 compare les empreintes mémoires des tables de routage des différentes implémentations d'algorithme de routage par clé.

Dans l'implémentation proposée, chaque paire identifiant/adresse nécessite donc 2+8 = 10 octets de mémoire contre 20 octets dans l'implémentation d'origine de Pastry. Ce simple changement des identifiants permet de réduire par 13 l'empreinte en mémoire.

Comme les clés sont désormais composées de 4 chiffres hexadécimaux, la table de routage ne contient plus que 4 lignes. La mémoire nécessaire pour les tables de routage est alors réduite à $(4 \times 15+16) \times 10 = 760$ octets contre environ 10 kilo-octets dans l'implémentation d'origine.

Tableau 4.3 Empreinte mémoire des tables de routage

Implémentation	FreePastry	Nodeus Pastry
Taille des clés	128 bits	16 bits
Taille des adresses	32 bits	64 bits
Table de routage (octets)	9600	600
Table des voisins logiques (octets)	320	160
Total de l'empreinte mémoire (octets)	9920	760

4.4.2 Empreinte mémoire de l'algorithme

L'implémentation a été réalisée dans le langage C pour un microcontrôleur MSP430. Le compilateur *mspgcc* version 4.4.5 a été utilisé avec l'option d'optimisation de la taille du code (-Os).

Le tableau 4.4 présente l'empreinte mémoire du composant Pastry implémenté ainsi que l'empreinte de ses dépendances. L'empreinte du code est mesurée avec l'utilitaire *msp430-size* et correspond à la quantité de mémoire ROM nécessaire. La taille de l'état du composant est calculée à partir de l'état déclaré pour chaque composant et correspond à la mémoire RAM allouée pour chaque composant.

Tableau 4.4 Empreinte mémoire de l'algorithme et ses dépendances

Composant	Code (octets)	État (octets)
Nodeus Pastry	2916	772
Event scheduler	776	210
Event timer	824	12
CRC16	66	0
XBee	1834	182
Interruption	140	0
Timer	408	2
UART	596	2
Total	7560	1180

4.5 Travaux futurs

L'implémentation de l'algorithme de routage par clé de ce projet n'a été réalisée que pour supporter le développement d'un dépôt distribué de composants. Les fonctionnalités principales du routage par clé ont donc été réalisées et l'implémentation proposée est une preuve de la viabilité de telles solutions dans le contexte des RCSF. Cependant, le composant réalisé présente encore de nombreuses lacunes qu'il serait possible de combler dans des travaux futurs.

Tout d'abord, les tables de routage sont actuellement allouées de manière fixe indépendamment du nombre d'entrées non nulles. Puisque les RCSF sont parfois de faible taille, la quantité de mémoire nécessaire aux tables de routage pourrait être réduite en allouant la mémoire dynamiquement selon le nombre d'entrées utiles.

Par ailleurs, l'algorithme proposé repose directement sur le principe de routage par clé proposé par Pastry. La notion de proximité des noeuds n'est donc pas du tout intégrée dans le choix des clés. Aucune table des voisins physiques n'est non plus entretenue. Cette prise en compte de la proximité des noeuds permettrait d'améliorer grandement la consommation en énergie d'un tel algorithme. Le regroupement en clusters proposé par MADPastry s'adapterait bien aux contraintes des RCSF et permettrait de réduire encore plus la taille des tables de routage et donc la consommation en mémoire vive. L'appartenance à un cluster pourrait être déterminée par un simple broadcast initial auquel les noeuds voisins répondraient par l'identifiant de leur cluster. Le noeud souhaitant joindre le réseau pourrait alors choisir le cluster auquel il appartient d'après ce sondage des voisins.

Enfin, l'algorithme réalisé dans ce projet utilise une architecture de type *overlay*, car les noeuds de Nodeus utilisent des radios XBee dont la couche réseau n'est pas modifiable. Les projets futurs pourraient s'intéresser à l'intégration d'un algorithme de routage par clé à la couche réseau tout en respectant le standard ZigBee qui permet l'interopérabilité avec de nombreux équipements domotiques bon marché.

4.6 Conclusion

En conclusion, le projet réalisé propose l'utilisation des méthodes de routage par clé pour supporter le développement d'applications distribuées à l'échelle des RCSF. Les techniques utilisées traditionnellement dans les réseaux d'ordinateurs fixes ne sont pas adaptables telles quelles aux RCSF, c'est pourquoi de nouvelles implémentations adaptées aux nouvelles contraintes doivent être proposées.

L'algorithme de routage par clé détaillé dans ce chapitre, présente une nouvelle implémentation de l'algorithme de Pastry légèrement adaptée aux contraintes des RCSF, principalement en terme de mémoire utilisée. Cet algorithme a permis la mise en place d'une solution de dépôt de composants distribuée. L'aspect *distribué* du dépôt présenté au chapitre suivant tire toute son origine dans le présent algorithme de routage par clé qui permet d'attribuer des rôles de manière distribuée à des noeuds du réseau inconnus à l'avance.

Cette distribution du code sur le réseau permettra de mettre en place, dans le futur, la vision d'informatique omniprésente autonome proposée par le laboratoire DOMUS à l'échelle des RCSF.

CHAPITRE 5

UN DÉPÔT DISTRIBUÉ DE COMPOSANTS

La présence de capteurs et d'actionneurs répartis dans l'habitat apporte une connaissance précise de l'environnement qui permet de mettre en place une informatique omniprésente. Les précédents travaux de recherche au laboratoire DOMUS ont amené la vision d'une informatique omniprésente *autonome* dans laquelle les différents équipements informatiques s'auto-organisent. Cela implique des concepts comme l'auto-(re)configuration, l'auto-réparation, l'auto-optimisation ou encore l'auto-protection des noeuds du réseau.

Dans un tel contexte d'informatique omniprésente autonome, chaque noeud doit pouvoir prendre plusieurs rôles de manière dynamique dépendamment du contexte de l'environnement. Cette faculté de reconfiguration dynamique en fonction du contexte est appelée *adaptation* du réseau. L'informatique omniprésente autonome utilise la reprogrammation dynamique des noeuds du réseau comme moyen pour soutenir et élargir les capacités d'adaptation du réseau.

En effet, si avec des infrastructures puissantes il est possible que chaque noeud soit programmé à l'avance pour tous les rôles qu'il aurait à prendre, cela est inenvisageable sur des plateformes de RCSF car la mémoire disponible dans chaque noeud est beaucoup trop faible pour contenir l'ensemble des fonctionnalités nécessaires. Chaque noeud peut prendre un seul rôle et doit être reprogrammé pour en prendre un autre.

Pour ce faire, le système de composants Nodecom – présenté au chapitre 3 – propose déjà la reprogrammation des noeuds de manière fine, c'est-à-dire la reprogrammation de certaines parties seulement du code exécutable. Cette reprogrammation dynamique fine est donc un bon support pour l'adaptation dynamique et la reconfiguration du réseau.

Beaucoup des systèmes existants qui permettent la reprogrammation dynamique des noeuds sur un réseau de capteurs sans-fils cherchent à reprogrammer les noeuds d'une manière centralisée. En effet, la plupart des systèmes existants dans le contexte des RCSF proposent des solutions de reprogrammation des noeuds du réseau à partir d'un ordinateur central. Si ce genre de solution est tout à fait adapté pour la phase de développement ou de mise à jour de tels réseaux, elle implique l'intervention d'utilisateurs humains ou du moins d'une entité centralisée qui décide de la réorganisation du réseau.

Dans une vision d'informatique omniprésente autonome, l'intervention humaine est réduite au minimum et les noeuds du réseau décident d'eux-mêmes les conditions d'adaptation du réseau.

La reconfiguration des RCSF dans une optique d'informatique omniprésente autonome, implique trois aspects principaux :

1. un formalisme permettant la reprogrammation de manière fine du système local (système de composants dynamique) ;
2. un mécanisme permettant la reprogrammation des noeuds de manière distribuée (chargement de code à partir d'un dépôt distribué) ;
3. l'identification de *quand* et *quoi* reprogrammer sur le réseau.

Le présent projet de recherche s'attache aux deux premiers aspects et laisse le troisième point comme un travail futur d'adaptation des principes de l'informatique omniprésente autonome au contexte des RCSF.

La solution développée ici propose un entrepôt distribué de composants. Cet entrepôt distribué permet de répartir le code de chaque composant à travers les noeuds du réseau. De ce fait, chaque noeud peut décider de charger un composant dont le code n'est pas disponible localement mais présent ailleurs sur le réseau.

Pour ce faire, le dépôt de composants proposé ici s'inspire des réseaux pair-à-pair. Chaque noeud partage une partie de sa mémoire avec les autres noeuds dans le but de créer un entrepôt virtuel contenant tous les composants disponibles. Avec ce dépôt distribué, les noeuds peuvent se reconfigurer de manière autonome et ne nécessitent ni de disposer localement des fichiers de composants, ni de questionner un serveur centralisé.

5.1 Travaux connexes

À la lumière des systèmes étudiés dans la revue de littérature, aucun système de dépôt distribué de composants n'existe actuellement dans le contexte des réseaux de capteurs sans-fils. Aucun système de fichiers distribué réel n'a par ailleurs été conçu pour le contexte des RCSF.

Certains systèmes proposent des abstractions du réseau sous forme de système de fichiers distribué.

Tilak et al. proposent une solution qui repose sur le protocole Styx du système Plan 9 pour accéder aux données du RCSF [69]. Chaque noeud du réseau est représenté sous la forme d'un dossier. Les capteurs ou actioneurs sont représentés sous la forme de fichiers accessibles en lecture ou en écriture. Le système présenté dans cet article n'a pas été implémenté mais simulé avec le simulateur ns-2. L'article ne présente donc pas de résultats quantifiables quant à la mémoire nécessaire, l'usage du CPU ou encore l'utilisation de la bande passante du réseau. L'article fait par ailleurs état des limitations du protocole Styx dans le contexte des RCSF. Enfin, le système proposé ne présente aucune solution en terme de découverte des autres noeuds et de recherche de fichiers à travers le réseau.

Dans la même catégorie de systèmes, LiteOS [10] propose une abstraction du réseau sous la forme d'un système de fichier UNIX. Ce système est accessible depuis un ordinateur centralisé et s'intègre au système de fichier UNIX (montage d'un répertoire).

Ces deux systèmes proposent une représentation du RCSF sous forme de système de fichiers mais ne présentent pas de réelles solutions de distribution permettant le stockage et l'échange de fichiers entre les noeuds.

L'intergiciel COMiS [36] propose la découverte et l'interconnexion des composants dans un RCSF. Les composants sont considérés comme des services distants mais ne sont pas déplaçables entre les noeuds. La découverte des composants – ou services – sur le réseau se fait par *flooding*, en une succession de broadcasts ce qui est très coûteux en ressources réseau et donc en énergie.

Wang et al. [73] s'intéressent à des noeuds dont le but spécifique serait le stockage des données et propose une solution pour leur attribuer une place stratégique au sein du RCSF. Ces noeuds peuvent être multiples et répartis dans l'environnement. Même si cette solution ne propose pas de mécanisme direct pour l'échange de fichiers, l'usage de noeuds de stockage pourrait augmenter la capacité de stockage d'un RCSF. Il faudrait cependant pour cela gérer l'hétérogénéité des noeuds en terme de capacité de stockage.

FiGaRo [50] présente un système de composants reprogrammables de manière fine et dynamique. Ce projet s'est intéressé à la problématique de la reconfiguration du réseau à l'aide de la reprogrammation dynamique. Les noeuds à mettre à jour ou à reconfigurer peuvent être regroupés en régions logiques – types de noeuds, capteurs disponibles, etc. Cependant, l'approche proposée est événementielle dans le sens où la reconfiguration est effectuée ponctuellement après une prise de décision centralisée. Dans le contexte d'une informatique omniprésente autonome, la décision de reconfiguration et de reprogrammation devrait être distribuée elle aussi. De plus, le remplacement des composants est réalisé selon

les interfaces fournies, or l'utilisation des interfaces pour décrire un composant manque de précision. Une interface décrit les méthodes pour accéder à un composant, mais ne décrit pas le comportement du composant.

Thoelen et al. [67] proposent une solution pour prendre en compte le comportement des composants et permettre ainsi la réorganisation des RCSF sans utiliser uniquement les interfaces. Cependant, ce système est lui-aussi centralisé et utilise une interface de contrôle sur PC pour initier la reconfiguration. Si cette interface centralisée utilise un dépôt de composants, la forme et les caractéristiques de ce dépôt n'ont pas été abordées dans l'article.

5.2 Présentation du dépôt de composants

Dans Nodecom, les types de composants se présentent sous la forme de fichiers au format ELF. Distribuer le code des composants à travers le réseau revient donc à distribuer des fichiers. Si beaucoup de systèmes de fichiers distribués ont été imaginés dans les années passées, la plupart des solutions existantes sont conçues pour Internet ou pour des réseaux d'entreprises et s'intéressent à la distribution de fichiers volumineux.

Dans notre contexte de RCSF, les fichiers nécessitant une présence sur le réseau sont de faible taille, en moyenne quelques kilo-octets. C'est pourquoi les systèmes de fichiers distribués conçus pour Internet et utilisant souvent des principes de fragmentation des fichiers ne sont pas adaptés aux RCSF et complexifient inutilement le processus. Les principes d'anonymité des utilisateurs est également inutile dans le contexte des RCSF, car il n'y a pas de considérations de droits d'auteurs à prendre en compte.

5.3 Implémentation du dépôt de composants

Comme le code des composants chargeables dynamiquement se présente sous la forme de fichiers au format ELF, la réalisation d'un dépôt distribué de ce code revient à un système de fichiers distribués.

5.3.1 Principe

Le système de fichiers distribués présenté ici, cherche à équilibrer la répartition des fichiers parmis les noeuds du réseau. Pour ce faire, chaque fichier du système de fichiers est associé à un noeud responsable. L'association d'un fichier à un noeud est réalisée grâce au routage

par clé de Pastry. En effet, lorsqu'un fichier est ajouté au système de fichiers, le nom de ce fichier est transformé en clé grâce à une fonction de hachage et un premier message (FPUT) est routé à cette clé pour avertir le noeud responsable du fichier, qu'il va recevoir un fichier et qu'il en est responsable.

L'envoi du fichier est ensuite réalisé grâce à un protocole simple décrit à la section 5.3.3.

5.3.2 Diagramme de composants

L'implémentation du dépôt proposée dans ce projet se présente comme un composant Nodecom. Il repose sur l'implémentation de Pastry proposée au chapitre 4. La figure 5.1 décrit les composants implémentés et leur organisation.

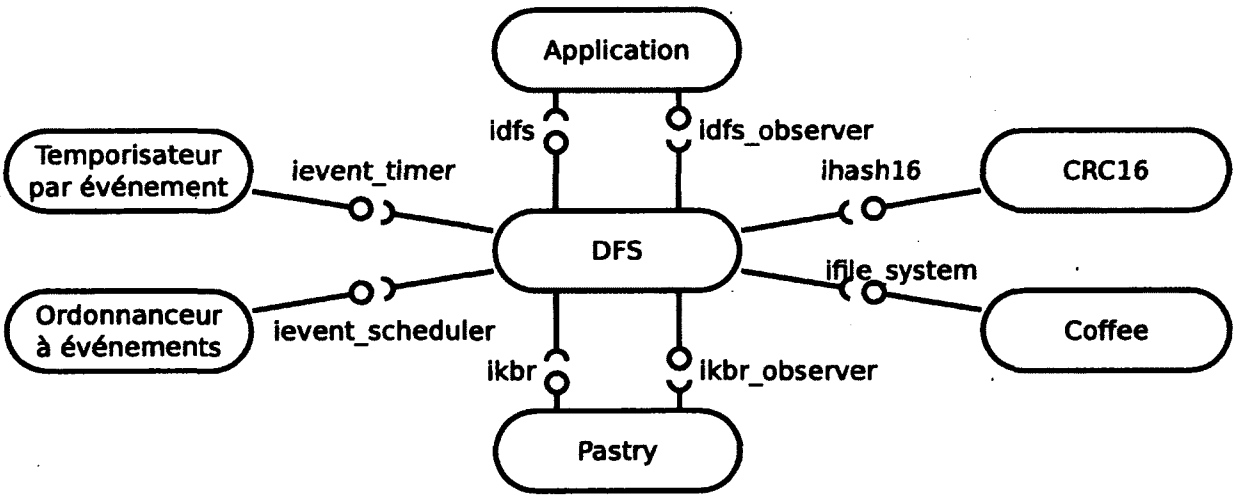


Figure 5.1 Diagramme de composants du dépôt distribué de composants

Le composant du dépôt (DFS = *Distributed File System*) présente 5 dépendances. Il repose principalement sur l'algorithme de routage par clé réalisé précédemment. Il nécessite également un ordonnanceur et un temporisateur pour utiliser ses propres *protothreads*. La fonction de hachage du CRC permet de transformer les noms des fichiers en clés. Enfin, il utilise le système de fichiers du noeud pour conserver localement les fichiers disponibles pour les autres noeuds et pour le noeud local.

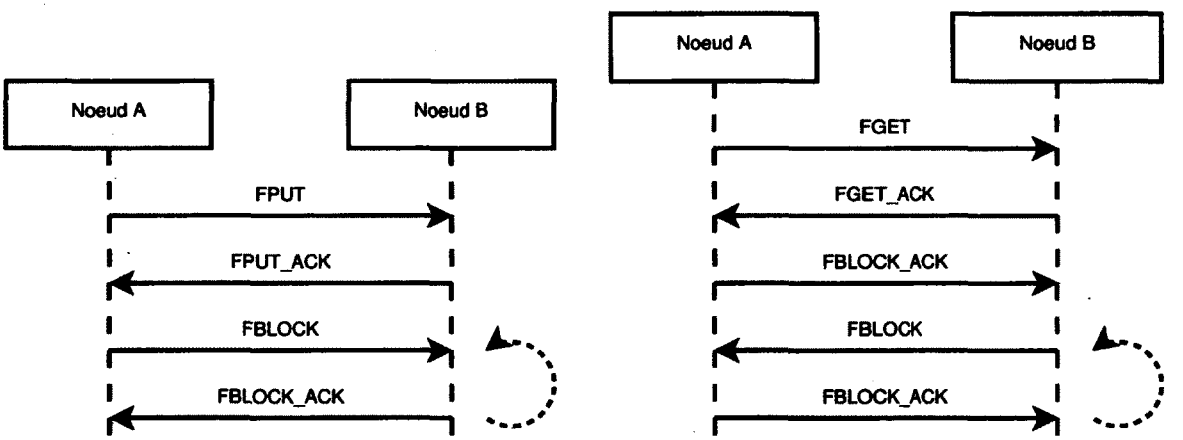
Le composant DFS propose l'interface `idfs` qui permet d'ajouter et de récupérer des fichiers sur le réseau. L'interface `idfs_observer` doit être fournie par le composant qui utilisera ce système de fichiers distribué et permet de notifier l'arrivée d'un fichier. Ces deux interfaces sont les suivantes :

```
DECLARE_INTERFACE(idfs) {
    void (*initialize) (struct nodecom_component * component);
```

```
    int (*put_file) (struct nodecom_component * component, char *
        filename);
    int (*get_file) (struct nodecom_component * component, char *
        filename);
};

DECLARE_INTERFACE(idfs_observer) {
    void (*notify_received_file) (struct nodecom_component * component,
        char * filename);
    void (*notify_sent_file) (struct nodecom_component * component,
        char * filename);
};
```

5.3.3 Protocole d'échange de fichier



(a) Envoi d'un fichier (b) Téléchargement d'un fichier

Figure 5.2 Échanges de fichiers entre deux noeuds

Le protocole utilisé pour échanger les fichiers a été imaginé pour les besoins spécifiques de cette implémentation et est d'une conception très simple. Deux types d'échanges sont possibles : l'envoi d'un fichier ou le téléchargement d'un fichier.

L'envoi d'un fichier (voir figure 5.2a) commence par une demande d'envoi `FPUT` à laquelle le noeud distant répond avec un acquittement `FPUT_ACK`. Le fichier est alors échangé en blocs de taille fixe avec l'aide de messages `FBLOCK`. Chaque bloc reçu est acquitté par un message `FBLOCK_ACK`. Si l'acquittement n'est pas reçu, le bloc est renvoyé.

Les blocs envoyés sont d'une longueur de 123 octets. En effet, les messages envoyés avec le XBee sont d'une longueur de 128 octets. L'algorithme de routage par clés réserve 3 octets

pour son en-tête. Le message `FBLOCK` utilise 2 octets pour son en-tête ce qui laisse 123 octets pour l'envoi des données utiles.

La réception d'un fichier (voir figure 5.2b) est semblable. La demande de téléchargement débute par un message `FGET`, acquitté par le destinataire par un message `FGET_ACK`. Enfin, le téléchargement est démarré par l'initiateur par l'envoi d'un premier message `FBLOCK_ACK`.

Le premier message d'un échange de fichiers est routé avec l'algorithme de routage par clés vers le noeud responsable de ce fichier. Pour ce faire, le nom du fichier est transformé en clé de 16 bits avec l'algorithme CRC16 utilisé comme fonction de hachage. Le noeud dont l'identifiant est le plus proche de cette clé sera le destinataire du premier message de l'échange. Les deux noeuds créent ensemble une connexion pour s'échanger les messages suivants.

5.4 Évaluation

Dans le contexte des RCSF, aucun travail similaire au présent projet n'a été réalisé ; il est donc difficile de comparer le système réalisé à d'autres. Cependant, une évaluation de l'empreinte mémoire a été réalisée ainsi que le coût en terme de communication d'un transfert de fichier.

5.4.1 Empreinte mémoire

L'empreinte mémoire du système de fichiers distribué a été évaluée ainsi que celle de ses dépendances. Cette évaluation a été réalisée avec l'utilitaire *msp430-size*. Les composants sont compilés avec le compilateur *mspgcc* (version 4.4.5) et l'option d'optimisation en taille (-Os).

Le tableau 5.1 présente les résultats des mesures réalisées sur l'implémentation du dépôt de composants distribué ainsi que ses dépendances.

L'empreinte mémoire du noyau de Nodecom a également été précisée dans le tableau 5.1. Cela permet d'avoir une idée de l'empreinte totale du système de composants distribué. Sur la plateforme utilisée pour le projet Nodeus, le microcontrôleur MSP430F5529 dispose de 128 ko de mémoire Flash et 10 ko de mémoire RAM. Le système ici présenté utilise donc 16% de la mémoire ROM et 15% de la RAM, laissant l'espace restant pour le développement d'applications.

Tableau 5.1 Empreinte mémoire du dépôt et ses dépendances

Composant	Code (octets)	État (octets)
DFS	3840	210
Coffee	5032	124
Nodeus Pastry	2916	772
Event scheduler	776	210
Event timer	824	12
CRC16	66	0
XBee	1834	182
Interruption	140	0
Timer	408	2
UART	596	2
Sous-total du DFS	16432	1514
Gestionnaire de composants	1426	20
Chargeur de fichiers ELF	2782	0
Gestionnaire de mémoire RAM	80	0
Sous-total du noyau de Nodecom	4288	20
Total	20720	1534

5.4.2 Coût en communication

L'envoi d'un fichier à un autre noeud impose que plusieurs messages soient échangés entre les pairs du réseau. Ces messages présentent un coût supplémentaire dans le processus d'échange du fichier. Ce coût supplémentaire est détaillé dans le tableau 5.2. La taille du nom du fichier est $taille_{nom}$ et la taille de chaque bloc du fichier envoyé est $taille_{bloc}$. Ces tailles sont exprimées en octets.

Tableau 5.2 Taille des messages pour l'envoi ou la réception d'un fichier

Message	Taille (octets)
FGET	$4 + taille_{nom}$
FGET ACK	8
FPUT	$6 + taille_{nom}$
FPUT ACK	6
FBLOCK	$2 + taille_{bloc}$
FBLOCK ACK	2

Pour l'envoi en blocs de 123 octets d'un fichier moyen de 3000 octets et dont le nom est de 10 octets, cela correspond à 3122 octets envoyés. Le surcoût d'un envoi est donc de $122/3000 = 4\%$. Le coût de la réception de ce même fichier est sensiblement équivalent.

Si l'on néglige le temps de traitement des messages, le coût en terme de temps est également proportionnel à la quantité de données envoyées.

Cependant, comme le dépôt utilise un algorithme de routage par clé (présenté au chapitre précédent), l'impact de cet algorithme est également à prendre en compte dans le système global.

5.5 Travaux futurs

Le dépôt de composants distribué présenté dans ce chapitre est la première implémentation de ce concept dans le contexte des RCSF. Le composant réalisé pourrait bénéficier d'un certain nombre d'améliorations.

D'abord, l'association d'un fichier à un seul noeud du réseau présente une grande limitation. En effet, si le noeud responsable d'un fichier disparaît, le fichier disparaît également. Une évolution possible face à ce problème serait de répliquer un même fichier sur plusieurs noeuds du réseau.

Une autre évolution possible serait de réaliser des vérifications des fichiers échangés. Pour le moment aucune vérification, telle qu'une simple somme de contrôle, n'est effectuée.

Ensuite, le système réalisé ne présente pas d'arborescence des fichiers. Les fichiers distribués sur le réseau ne sont identifiés que par leur nom et aucune notion de répertoire n'est actuellement supportée.

De même, la gestion de numéros de version des composants pourrait être utile pour les processus de mise à jour des composants.

Actuellement, le RCSF mis en place à DOMUS est un réseau homogène dans le sens où tous les noeuds sont identiques. Le microcontrôleur utilisé est le MSP430 et les radios sont des radios XBee compatibles ZigBee. Cette homogénéité du réseau simplifie grandement la reprogrammation des noeuds mais limite fortement les possibilités qu'un réseau hétérogène apporterait. Dans un réseau hétérogène, les ressources des noeuds pourraient être différentes et – comme le projet Nodeus le prévoit déjà – certains noeuds pourraient être utilisés pour la gestion du multimédia tandis que d'autres seraient simplement utiles pour traiter les données des capteurs. Enfin, certains noeuds pourraient présenter une capacité de stockage plus grande que les autres et permettraient d'entreposer un nombre plus grand de fichiers.

5.6 Conclusion

En conclusion, l'entrepôt de composants distribué proposé dans ce chapitre repose sur l'algorithme de routage par clé présenté au chapitre précédent et vise à répartir les fichiers contenant le code des composants – au format ELF – à travers l'ensemble des noeuds dans une logique pair-à-pair.

S'il présente encore quelques limitations – abordées dans la section 5.5 – cet entrepôt de composants distribué est la première implémentation du genre dans le contexte des RCSF.

L'utilisation d'un tel dépôt distribué de composants permet d'apporter les fonctionnalités de reprogrammation des noeuds dans le contexte des RCSF et est un bon support pour les travaux futurs qui visent à intégrer l'informatique omniprésente autonome aux RCSF. Les concepts de reconfiguration autonome, d'auto-réparation, d'auto-optimisation et d'auto-protection des noeuds pourront alors être développés.

CHAPITRE 6

CONCLUSION

La présente recherche propose des solutions pour supporter la reconfiguration dynamique des réseaux de capteurs utilisant des composants logiciels et une organisation pair-à-pair autonome. Dans la vision d'informatique omniprésente autonome développée récemment au laboratoire DOMUS, cette reconfiguration est réalisée avec l'aide de la reprogrammation de composants logiciels sur les noeuds sans nécessiter de redémarrage.

Avec la programmation par composants, les composants logiciels créés sont réutilisables dans différents contextes et modulables à volonté. Cependant l'utilisation de composants statiques empêche la reprogrammation dynamique du système et n'apporte cette modularité qu'avant la phase de compilation.

L'utilisation de composants chargeables dynamiquement permet de mettre en place une reprogrammation à granularité fine pendant l'exécution du système et évite d'avoir à reprogrammer l'ensemble du système. Le paradigme de programmation orientée composants (POC) simplifie le processus de remplacement de modules logiciels en déclarant explicitement leurs spécifications fonctionnelles et leurs dépendances. Le fonctionnement interne des composants est dissimulé derrière des interfaces qui décrivent les méthodes d'accès de chaque composant.

Suite à une évaluation récente de l'impact des systèmes de composants statiques et dynamiques, les systèmes dynamiques ont été identifiés comme étant très consommateurs en ressources par rapport aux systèmes statiques. La solution proposée à cette limitation est une architecture hybride permettant à la fois l'usage de composants statiques et à la fois l'usage de composants dynamiques. Le concepteur du système peut alors choisir quels composants sont liés statiquement et lesquels sont remplaçables de manière dynamique.

Le système de composants Nodecom réalisé dans ce projet est une implémentation de cette architecture et présente une empreinte en mémoire très limitée qui se rapproche de solutions non-orientées composants telles que le système d'exploitation Contiki. L'architecture hybride mise en place dans ce projet s'affiche donc comme un bon compromis entre les solutions purement statiques et les solutions purement dynamiques.

Dans un deuxième temps, les solutions de routage par clé utilisées dans les réseaux pair-à-pair traditionnels pour mettre en place des tables de hachage distribuées ont été étudiées et adaptée de manière simple aux contraintes des réseaux de capteurs sans-fils (RCSF).

L'utilisation d'un tel algorithme de routage par clé a permis de mettre en place une certaine forme d'auto-organisation dans les communications à travers le RCSF. Les rôles peuvent être distribués selon les noeuds présents sur le réseau avec une bonne tolérance aux failles.

Cet algorithme de routage a été utilisé pour mettre en place un dépôt distribué de composants. Les solutions actuelles de reprogrammation des RCSF ne s'adaptent pas à une vision d'informatique omniprésente autonome puisqu'elles sont toutes centralisées. La mise en place d'un dépôt distribué de fichiers de composants permet à chaque noeud de se reconfigurer seul en chargeant n'importe quel type de composant présent sur le réseau. La décision de reconfiguration d'un noeud est donc localisée au noeud et non-plus centralisée, et les notions d'informatique omniprésente autonome peuvent désormais commencer à être élucidées.

6.1 Contributions

Les principales contributions de ce projet de maîtrise sont tout d'abord la mise en place pour la première fois d'un système de programmation par composants pour les RCSF qui présente une architecture hybride sur le plan de la dynamicité des composants. Cette architecture a été démontrée comme étant bénéfique pour l'impact en mémoire.

Ensuite, un système de routage par clé a été étudié et adapté au contexte des RCSF. Ce travail est une première approche et révèle que l'usage de tels algorithmes de routage indirect peuvent répondre simplement aux problématiques des RCSF s'ils sont adaptés à leurs contraintes.

Enfin la création d'un dépôt distribué de composants est le premier travail de ce type dans le contexte des RCSF et se présente comme une bonne première approche dans la perspective de faire évoluer une vision d'informatique omniprésente autonome adaptée aux RCSF.

6.2 Travaux futurs

Ce projet est donc concluant et présente de bonnes ouvertures pour de futurs travaux. Parmi les idées pour les futurs évolutions de ce projet, le système de composants Nodecom

nécessite encore un certain nombre d'améliorations pour être utilisable simplement. Si de nombreuses macros ont été réalisée pour tenter de simplifier l'écriture des composants, le langage C s'avère être très peu expressif pour la programmation orientée composants et un langage dédié à ce paradigme serait sans-doute plus simple à utiliser. Par ailleurs, l'utilisation du format de fichiers ELF pour encapsuler les composants dynamiques présente quelques limitations et l'utilisation d'un format de fichiers dédié aux composants permettrait de réduire la taille des fichiers et indirectement de réduire la quantité d'énergie nécessaire pour leur propagation. L'impact des composants statiques est également un point qui pourrait être amélioré dans Nodecom et permettrait d'augmenter encore plus l'avantage d'une architecture hybride.

Par ailleurs, les composants réalisés ont souvent besoin d'une fonction d'initialisation pour configurer leur état en fonction des dépendances connectées. Pour ce faire il serait bon d'avoir une gestion de cycle de vie des composants plus évoluée. Chaque composant serait averti quand ses dépendances sont connectées et pourrait agir en conséquence par exemple.

L'implémentation de l'algorithme de routage par clé réalisée pour ce projet présente encore des limitations malgré l'impact en mémoire largement réduit. L'architecture sous forme d'*overlay* présente des limitations évoquées dans le chapitre concerné et le manque d'expressivité des identifiants de chaque noeud sur leur localité empêche à l'algorithme d'optimiser le routage par rapport à l'organisation réelle des noeuds dans l'espace. Une manière d'optimiser l'algorithme réalisé serait de mettre une solution se rapprochant des principes utilisés dans l'algorithme MADPastry et mis en évidence au chapitre 4.

Le dépôt de composants distribué présente quant à lui encore des limitations, car un seul noeud est responsable d'un fichier donné ce qui n'exploite pas le potentiel de tolérance aux failles de l'algorithme de routage par clé à sa pleine capacité. Afin de renforcer la tolérance aux failles d'un tel système de fichiers distribué, une certaine forme de réplication des données sur plusieurs noeuds pourrait être mise en place.

Les problématiques étudiées ici se sont limitées à l'aspect de la reprogrammation des noeuds et à la distribution du code à travers le réseau. Dans une vision plus large pour la suite de ce projet, il faudrait s'intéresser aux conditions dans lesquelles cette reconfiguration des noeuds prend place. Peu de travaux s'intéressent actuellement au processus de reconfiguration des RCSF malgré que ce sujet semble représenter un bon avenir pour le domaine. Les conditions d'adaptation et le contexte de ces reconfigurations ont été abordées dans le projet WiSeKit [65] et ces travaux devraient intéresser les succeseurs au présent projet.

Avec les solutions développées durant ce projet, les capteurs de l'appartement intelligent pourront dans l'avenir s'organiser de manière autonome sans nécessiter d'intervention humaine. Le réseau de capteurs Nodeus sera donc une avancée considérable sur le plan de l'installation et de la maintenance des infrastructures domotiques ce qui permettra, entre autres, de démocratiser les solutions déjà réalisées dans le domaine des gérontechnologies.

LISTE DES RÉFÉRENCES

[1] Abdelzaher, T., Blum, B., Cao, Q., Chen, Y., Evans, D., George, J., George, S., Gu, L., He, T., Krishnamurthy, S., Luo, L., Son, S., Stankovic, J., Stoleru, R. et Wood, A. (2004). EnviroTrack : towards an environmental computing paradigm for distributed sensor networks. IEEE Computer Society, p. 582–589.

[2] Al-Mamou, A. et Labiod, H. (2007). ScatterPastry : an overlay routing using a DHT over wireless sensor networks. Dans *The 2007 International Conference on Intelligent Pervasive Computing, 2007.* p. 274–279.

[3] Aslam, F., Fennell, L., Schindelhauer, C., Thiemann, P., Ernst, G., Haussmann, E., Rührup, S., Uzmi, Z., Rajaraman, R., Moscibroda, T., Dunkels, A. et Scaglione, A. (2010). Optimized java binary and virtual machine for tiny motes. Dans *Distributed Computing in Sensor Systems*, Lecture Notes in Computer Science, *volume 6131.* Springer Berlin / Heidelberg, p. 15–30.

[4] Awad, A., German, R. et Dressler, F. (2008). P2P-based routing and data management using the virtual cord protocol (VCP). *Proceedings of the 9th ACM international symposium on Mobile ad hoc networking and computing*, p. 443–444.

[5] Awad, A., Sommer, C., German, R. et Dressler, F. (2008). Virtual cord protocol (VCP) : a flexible DHT-like routing service for sensor networks. Dans *5th IEEE International Conference on Mobile Ad Hoc and Sensor Systems, 2008. MASS 2008.* p. 133–142.

[6] Becker, C., Handte, M., Schiele, G. et Rothermel, K. (2004). PCOM - a component system for pervasive computing. IEEE Computer Society, p. 67–76.

[7] Bell, G. (2008). Bell's law for the birth and death of computer classes. *Communications of the ACM*, volume 51, numéro 1, p. 86–94.

[8] Bencomo, N., Blair, G. S., Coulson, G. et Batista, T. (2005). Towards a metamodelling approach to configurable middleware. Dans *2nd ECOOP2005 Workshop on Reflection, AOP and Meta-Data for Software Evolution, Glasgow, Scotland (July 2005).* p. 73–82.

[9] Caesar, M., Castro, M., Nightingale, E. B., O'Shea, G. et Rowstron, A. (2006). Virtual ring routing : network routing inspired by DHTs. *ACM SIGCOMM Computer Communication Review*, volume 36, p. 351–362.

[10] Cao, Q., Abdelzaher, T., Stankovic, J. et He, T. (2008). The LiteOS operating system : Towards Unix-Like abstractions for wireless sensor networks. IEEE Computer Society, p. 233–244.

[11] Castro, M., Druschel, P., Kermarrec, A. M. et Rowstron, A. I. (2002). SCRIBE : a large-scale and decentralized application-level multicast infrastructure. *IEEE Journal on Selected Areas in Communications*, volume 20, numéro 8, p. 1489–1499.

[12] Ceriotti, M., Mottola, L., Picco, G. P., Murphy, A. L., Guna, S., Corra, M., Pozzi, M., Zonta, D. et Zanon, P. (2009). Monitoring heritage buildings with wireless sensor networks : The torre aquila deployment. Dans *Proceedings of the 2009 International Conference on Information Processing in Sensor Networks.* p. 277–288.

[13] Chakroun, O., Abdulrazak, B., Chiazzaro, M. et Frikha, M. (2010). Indoor and outdoor localization architecture for pervasive environment. Dans *Aging Friendly Technology for Health and Independence*, Lecture Notes in Computer Science, *volume 6159.* Springer Berlin / Heidelberg, p. 242–245.

[14] Costa, P., Coulson, G., Gold, R., Lad, M., Mascolo, C., Mottola, L., Picco, G. P., Sivaharan, T., Weerasinghe, N. et Zachariadis, S. (2007). The RUNES middleware for networked embedded systems and its application in a disaster management scenario. IEEE Computer Society, p. 69–78.

[15] Costa, P., Coulson, G., Mascolo, C., Picco, G. P. et Zachariadis, S. (2005). The RUNES middleware : A reconfigurable component-based approach to networked embedded systems. Dans *IEEE 16th International Symposium on Personal, Indoor and Mobile Radio Communications, 2005. PIMRC 2005.* volume 2. p. 806–810.

[16] Coulson, G., Blair, G., Grace, P., Taiani, F., Joolia, A., Lee, K., Ueyama, J. et Sivaharan, T. (2008). A generic component model for building systems software. *ACM Transactions on Computer Systems (TOCS)*, volume 26, numéro 1, p. 1–42.

[17] Curino, C., Giani, M., Giorgetta, M., Giusti, R., Murphy, A. L. et Picco, G. P. (2005). TINYLIME : bridging mobile and sensor networks through middleware, p. 61—72.

[18] Di, P. et Fuhrmann, T. (2009). Using link-layer broadcast to improve scalable source routing. *Proceedings of the 2009 International Conference on Wireless Communications and Mobile Computing : Connecting the World Wirelessly*, p. 466–471.

[19] Dong, W., Chen, C., Liu, X., Bu, J. et Liu, Y. (2009). Dynamic linking and loading in networked embedded systems. Dans *Proceedings of the 6th International Conference on Mobile Adhoc and Sensor Systems.* IEEE Computer Society, Los Alamitos, CA, USA, p. 554–562.

[20] Druschel, P. et Rowstron, A. (2001). PAST : a large-scale, persistent peer-to-peer storage utility. *Workshop on Hot Topics in Operating Systems.*, p. 75–80.

[21] Dunkels, A., Finne, N., Eriksson, J. et Voigt, T. (2006). Run-time dynamic linking for reprogramming wireless sensor networks. ACM, Boulder, Colorado, USA, p. 15–28.

[22] Dunkels, A., Gronvall, B. et Voigt, T. (2004). Contiki - a lightweight and flexible operating system for tiny networked sensors. IEEE Computer Society, p. 455–462.

[23] Dunkels, A., Schmidt, O., Voigt, T. et Ali, M. (2006). Protothreads : simplifying event-driven programming of memory-constrained embedded systems. ACM, Boulder, Colorado, USA, p. 29–42.

[24] Eliassen, F., Goebel, V., Kristensen, T., Plagemann, T., Andersen, A., Rafaelsen, H. O., Yu, W., Blair, G., Costa, F., Coulson, G., Saikoski, K. B. et Hansen, O. (1999).

Next generation middleware : Requirements, architecture, and prototypes. *Future Trends of Distributed Computing Systems, IEEE International Workshop*, p. 60–65.

[25] Fok, C., Roman, G. et Lu, C. (2005). Mobile agent middleware for sensor networks : an application case study. IEEE Press, Los Angeles, California, p. 382–387.

[26] Frenot, S. (2004). Gestion du deploiement de composants sur reseau P2P. *DECOR04 (2004)*, p. 113–124.

[27] Gay, D., Levis, P. et Culler, D. (2005). Software design patterns for tinyos. Dans *Proceedings of the 2005 ACM SIGPLAN/SIGBED conference on Languages, compilers, and tools for embedded systems.* LCTES '05. ACM, New York, NY, USA, p. 40–49.

[28] Gay, D., Levis, P., von Behren, R., Welsh, M., Brewer, E. et Culler, D. (2003). The nesC language : A holistic approach to networked embedded systems. ACM, San Diego, California, USA, p. 1–11.

[29] Giroux, S., Bauchet, J., Pigot, H., Lussier-Desrochers, D. et Lachappelle, Y. (2008). Pervasive behavior tracking for cognitive assistance. ACM, Athens, Greece, p. 1–7.

[30] Gouin-Vallerand, C., Abdulrazak, B., Giroux, S. et Mokhtari, M. (2008). Toward autonomic pervasive computing. Dans *Proceedings of the 10th International Conference on Information Integration and Web-based Applications & Services.* iiWAS '08. ACM, New York, NY, USA, p. 673–676.

[31] Hadim, S. et Mohamed, N. (2006). Middleware : Middleware challenges and approaches for wireless sensor networks. *IEEE Distributed Systems Online*, volume 7, numéro 3, p. 1–23.

[32] Heineman, G. T. et Councill, W. T. (2001). *Component-Based Software Engineering : Putting the Pieces Together*, 1re édition. Addison-Wesley Professional.

[33] Henricksen, K. et Robinson, R. (2006). A survey of middleware for sensor networks : state-of-the-art and future directions. ACM, Melbourne, Australia, p. 60–65.

[34] Hill, J., Szewczyk, R., Woo, A., Hollar, S., Culler, D. et Pister, K. (2000). System architecture directions for networked sensors. *SIGPLAN Not.*, volume 35, numéro 11, p. 93–104.

[35] Ibrahim, M. et Mabilleau, P. (2009). Mobile agent platform for embedded systems. Dans *Parallel and Distributed Computing and Networks*, ACTA Press. p. 69–73.

[36] Janakiram, D., Venkateswarlu, R. et Nitin, S. (2005). Comis : Component oriented middleware for sensor networks. Dans *proceedings of 14th IEEE Workshop on Local Area and Metropolitan Networks (LANMAN).* p. 1–9.

[37] Kaddoura, Y., King, J. et Helal, A. S. (2005). Cost-precision tradeoffs in unencumbered floor-based indoor location tracking. Dans *From smart homes to smart care : ICOST 2005, 3rd International Conference on Smart Homes and Health Telematics.* p. 75–82.

[38] Kasten, O. et Romer, K. (2005). Beyond event handlers : programming wireless sensors with attributed state machines. *IPSN 2005 Fourth International Symposium on Information Processing in Sensor Networks*, volume 2005, numéro C, p. 45–52.

[39] Kuorilehto, M., Hännikäinen, M. et Hämäläinen, T. D. (2005). A survey of application distribution in wireless sensor networks. *EURASIP Journal on Wireless Communications and Networking*, volume 2005, numéro 5, p. 774–788.

[40] Labrosse, J. J. (2002). *MicroC/OS-II : The Real Time Kernel*, 2e édition. Newnes.

[41] Levis, P. et Culler, D. (2002). Maté : a tiny virtual machine for sensor networks. ACM, San Jose, California, p. 85–95.

[42] Levis, P., Madden, S., Polastre, J., Szewczyk, R., Whitehouse, K., Woo, A., Gay, D., Hill, J., Welsh, M., Brewer, E. *et al.* (2005). Tinyos : An operating system for sensor networks. *Ambient Intelligence*, volume 35, p. 115–148.

[43] Liu, J. H. K. et Weinreb, R. N. (2011). Monitoring intraocular pressure for 24 h. *British Journal of Ophthalmology*, volume 95, numéro 5, p. 599–600.

[44] Lo, B., Thiemjarus, S., King, R. et Yang, G. Z. (2005). Body sensor network–a wireless sensor platform for pervasive healthcare monitoring. Dans *The 3rd International Conference on Pervasive Computing.* volume 13. p. 77–80.

[45] Lohse, M., Winter, F., Repplinger, M. et Slusallek, P. (2008). Network-integrated multimedia middleware (NMM). ACM, Vancouver, British Columbia, Canada, p. 1081–1084.

[46] Lorincz, K., rong Chen, B., Waterman, J., Werner-Allen, G. et Welsh, M. (2008). Resource aware programming in the pixie OS. ACM, Raleigh, NC, USA, p. 211–224.

[47] Madden, S. R., Franklin, M. J., Hellerstein, J. M. et Hong, W. (2005). TinyDB : an acquisitional query processing system for sensor networks. *ACM Transactions on Database Systems (TODS)*, volume 30, numéro 1, p. 122–173.

[48] Mainwaring, A., Culler, D., Polastre, J., Szewczyk, R. et Anderson, J. (2002). Wireless sensor networks for habitat monitoring. ACM, Atlanta, Georgia, USA, p. 88–97.

[49] Moore, G. E. (1998). Cramming more components onto integrated circuits. *Proceedings of the IEEE*, volume 86, numéro 1, p. 82–85.

[50] Mottola, L., Picco, G. et Sheikh, A. A. (2008). Figaro : Fine-grained software reconfiguration for wireless sensor networks. *Wireless Sensor Networks*, p. 286–304.

[51] Ommering, R., Linden, F., Kramer, J. et Magee, J. (2000). The koala component model. *IEEE Computer*, volume 78, p. 85–98.

[52] Pardo-Castellote, G., Farabaugh, B. et Warren, R. (2003). An introduction to dds and data-centric communications. Dans *Proceeding of the 23rd International Conference on Distributed Computing Systems Workshops.*

[53] Polastre, J., Hui, J., Levis, P., Zhao, J., Culler, D., Shenker, S. et Stoica, I. (2005). A unifying link abstraction for wireless sensor networks. ACM, San Diego, California, USA, p. 76–89.

[54] Porter, B. et Coulson, G. (2009). Lorien : a pure dynamic component-based operating system for wireless sensor networks. ACM, Urbana Champaign, Illinois, p. 7–12.

[55] Porter, B., Roedig, U., Taiani, F. et Coulson, G. (2010). A comparison of static and dynamic component models for wireless sensor networks. Dans *Proceedings of the The First International Workshop on Networks of Cooperating Objects (CONET2010).* p. 1–11.

[56] Ratnasamy, S., Karp, B., Yin, L., Yu, F., Estrin, D., Govindan, R. et Shenker, S. (2002). GHT : a geographic hash table for data-centric storage. *Proceedings of the 1st ACM international workshop on Wireless sensor networks and applications*, p. 78–87.

[57] Römer, K., Kasten, O. et Mattern, F. (2002). Middleware challenges for wireless sensor networks. *ACM SIGMOBILE Mobile Computing and Communications Review*, volume 6, numéro 4, p. 59–61.

[58] Römer, K., Mattern, F., Dübendorfer, T. et Senn, J. (2002). *Infrastructure for virtual counterparts of real world objects.* Citeseer.

[59] Römer, K. et Zurich, E. T. H. (2004). Programming paradigms and middleware for sensor networks. *2. Fachgesprach der GI/ITG Fachgruppe KuVS" Drahtlose Sensornetze" 26./27. Februar 2004 Universitat Karlsruhe (TH)*, p. 49–54.

[60] Rowstron, A. et Druschel, P. (2001). Storage management and caching in PAST, a large-scale, persistent peer-to-peer storage utility. *ACM SIGOPS Operating Systems Review*, volume 35, p. 188–201.

[61] Rowstron, A., Druschel, P., Rowstron, A. et Druschel, P. (2001). Pastry : Scalable, decentralized object location and routing for large-scale peer-to-peer systems. *IFIP/ACM International Conference on Distributed Systems Platforms (Middleware), Heidelberg, Germany*, p. 329–350.

[62] Souto, E., aes, G. G., Vasconcelos, G., Vieira, M., Rosa, N., Ferraz, C. et Kelner, J. (2005). Mires : a publish/subscribe middleware for sensor networks. *Personal Ubiquitous Computing*, volume 10, numéro 1, p. 37–44.

[63] Sugihara, R. et Gupta, R. K. (2008). Programming models for sensor networks : A survey. *ACM Transactions on Sensor Networks (TOSN)*, volume 4, numéro 2, p. 8–34.

[64] Szewczyk, R., Osterweil, E., Polastre, J., Hamilton, M., Mainwaring, A. et Estrin, D. (2004). Habitat monitoring with sensor networks. *Communications of the ACM*, volume 47, numéro 6, p. 34–40.

[65] Taherkordi, A., Le-Trung, Q., Rouvoy, R. et Eliassen, F. (2009). WiSeKit : a distributed middleware to support application-level adaptation in sensor networks. Dans *Distributed Applications and Interoperable Systems.* p. 44–58.

[66] Taherkordi, A., Loiret, F., Abdolrazaghi, A., Rouvoy, R., Le-Trung, Q. et Eliassen, F. (2010). Programming sensor networks using remora component model. *Distributed Computing in Sensor Systems*, p. 45–62.

[67] Thoelen, K., Matthys, N., Horré, W., Huygens, C., Joosen, W., Hughes, D., Fang, L. et Guan, S. (2010). Supporting reconfiguration and re-use through self-describing component interfaces. Dans *Proceedings of the 5th International Workshop on Middleware Tools, Services and Run-Time Support for Sensor Networks*. MidSens '10. ACM, New York, NY, USA, p. 29–34.

[68] Tilak, S., Abu-Ghazaleh, N. B. et Heinzelman, W. (2002). A taxonomy of wireless micro-sensor network models. *ACM SIGMOBILE Mobile Computing and Communications Review*, volume 6, numéro 2, p. 28–36.

[69] Tilak, S., Pisupati, B., Chiu, K., Brown, G. et Abu-Ghazaleh, N. (2005). A file system abstraction for sense and respond systems. Dans *Proceedings of the 2005 workshop on End-to-end, sense-and-respond systems, applications and services*. p. 1–6.

[70] Tsiftes, N., Dunkels, A., He, Z. et Voigt, T. (2009). Enabling large-scale storage in sensor networks with the coffee file system. Dans *Proceedings of the 2009 International Conference on Information Processing in Sensor Networks*. IPSN '09. IEEE Computer Society, Washington, DC, USA, p. 349–360.

[71] Vasseur, J. et Dunkels, A. (2010). Why IP for smart objects ? Dans *Interconnecting Smart Objects with IP : The Next Internet*. Morgan Kaufmann, p. 29–38.

[72] Vincze, Z., Vass, D., Vida, R., Vidács, A. et Telcs, A. (2006). Adaptive sink mobility in event-driven multi-hop wireless sensor networks. ACM, Nice, France, p. 13–22.

[73] Wang, Y., Fu, X. et Huang, L. (2009). A storage node placement algorithm in wireless sensor networks. Dans *Joint Workshop on Frontier of Computer Science and Technology, Japan-China*. IEEE Computer Society, Los Alamitos, CA, USA, p. 267–271.

[74] Welsh, M. et Mainland, G. (2004). Programming sensor networks using abstract regions. USENIX Association, San Francisco, California, p. 3–16.

[75] Zahn, T. et Schiller, J. (2005). MADPastry : a DHT substrate for practicably sized MANETs. Dans *Proceedings of Workshop on Applications and Services in Wireless Networks*. p. 1–25.