Un modèle de composant pour la reconfiguration dynamique de réseaux de capteurs sans fil

Natacha Hoang, Nicolas Belloir, Xavier Detant, Congduc Pham

*LIUPPA
BP 1155
64013 Pau CEDEX
(nom.prenom@univ-pau.fr)

Résumé. Dans ce papier, nous proposons une approche permettant la reconfiguration dynamique des réseaux de capteurs sans fil (WSNs). Cette proposition repose sur une architecture logicielle bâtie autour d'un nouveau système d'exploitation et d'un modèle de composant basé sur Fractal. Ce système d'exploitation, baptisé *Valentine*, permet de conserver à l'exécution la construction par composant, en opposition à une construction monolithique caractéristique des principales solutions existantes dans les WSNs. De plus, étant dédié aux WSNs, il est tout naturellement orienté événement. Il a été généré à partir du canevas logiciel Think. Le modèle de composant permet de réutiliser la flexibilité du modèle abstrait Fractal et d'implémenter le mécanisme de reconfiguration dynamique.

1 Introduction

Les avancées technologiques confirment la présence croissante de l'informatique dans les objets du monde réel : d'après Crnkovic (2006), 98 % du code développé pour les systèmes futurs sera embarqué. De plus en plus d'objets physiques se voient dotés de processeurs et de moyens de communication sans fil leur permettant de traiter de l'information mais également de la transmettre. Cette évolution prend place dans le contexte de l'informatique pervasive, mieux connue sous le nom d'*ubiquitous computing* (voir Weiser (1993)).

Les réseaux de capteurs sans fil[1] entrent dans ce contexte. Ils sont constitués de petits éléments électroniques, appelés capteurs, aux ressources fortement contraintes, mais néanmoins dotés de moyens de communication, de traitement et de sauvegarde de l'information. Ils se différencient des capteurs traditionnels généralement fixes ou peu mobiles par leur mobilité due à la non présence de câblage ainsi que par leur ressources plus réduites. En dépit de cet élément de distinction, ces capteurs sont néanmoins capables d'acquérir des informations sur leur environnement proche, de les traiter et de les distribuer. Cela permet d'envisager alors un large panel de nouvelles applications plus proches du monde réel.

Cependant, l'intégration de tels capteurs dans leur environnement cible n'est pas chose aisée. Les problèmes rencontrés sont généralement liés aux contraintes inhérentes induites par leurs ressources : par exemple la gestion drastique qui doit être faite de la batterie, qui par définition est à capacité fixe et non rechargeable, ou encore les problèmes de gestion de la mémoire. Un certain nombre de travaux ont porté sur le développement d'environnements spécifiques à ces capteurs, pour prendre en compte ces contraintes. Par exemple, Hill et al. (2000) ont proposé TinyOS, le système d'exploitation de référence des WSNs. Il gère au plus prêt la mémoire limitée grâce à de nombreuses optimisations. Cette optimisation a cependant un coût, notamment au niveau de l'architecture du système d'exploitation et du code qui sont généralement monolithiques et par de là même peu souples. Cela rend par exemple les capteurs difficiles à reconfigurer dynamiquement.

D'un autre coté, l'ingénierie logicielle basée composant (voir le livre de Szyperski et al. (2002)) [2], est une approche maintenant reconnue permettant le développement d'applications à la fois flexibles et bien structurées, ayant souvent des capacités de reconfiguration ou d'administration à distance. Dans ce contexte, la reconfiguration dynamique consiste à remplacer un composant logiciel par un autre dans une application en exécution. Cette action peut être nécessaire pour plusieurs raisons. Il peut par exemple être nécessaire de substituer un composant

[1] *WSN* pour *Wireless Sensor Networks*
[2] *CBSE* pour *Component-Based Software Engineering*

mal implémenté, c'est-à-dire ne remplissant pas les fonctionnalités ciblées ou les implémentant d'une mauvaise manière, ou encore d'ajouter de nouvelles fonctionnalités.

Dans le contexte des WSN, la reconfiguration dynamique est un challenge important. Elle est nécessaire à la reconfiguration du réseau ou encore à l'ajout de nouvelles fonctionnalités. Il y a eu d'ailleurs plusieurs proposition pour sa mise en œuvre. De nombreux travaux, notamment en *autonomic computing*, ont identifié la reconfiguration dynamique comme une fonctionnalité fondamentale. L'utilisation de modèles de composant réels est une approche à étudier plus précisément, d'autant plus qu'aucune des approches citées en section 2.1 n'aborde ce type de mise en œuvre dans l'optique de cette caractéristique.

Dans ce contexte, nous présentons notre solution pour pallier le problème de la reconfiguration dynamique dans les WSNs. Cette solution s'appuie sur deux éléments complémentaires : un nouveau système d'exploitation offrant un support modulaire d'une part et un modèle de composant spécialisé pour la reconfiguration dynamique d'autre part. Nous nous concentrons sur la partie système d'exploitation dans la section 2.4. Nous présentons ce modèle de composant en section 3. Nous montrons alors comment nous avons implémenté ce modèle de composant à la section 3.3. Puis nous développons un exemple d'utilisation à la section 5. Finalement, la section 6 présente une conclusion et des directions pour des travaux futurs.

2 Valentine : Un système d'exploitation dynamique et adaptatif pour les WSNs

Nous montrerons dans la section 2.1 que les systèmes d'exploitation couramment utilisés dans les WSNs ne sont pas adaptés pour la reconfiguration dynamique. C'est le cas notamment de TinyOS qui est le système d'exploitation de référence pour les réseaux de capteurs. Nous expliquons en section 2.2 pourquoi il ne répond pas à cette problématique. Puis nous donnons en *Valentine* notre proposition pour pallier ce problème. Cette proposition se base sur l'utilisation d'un générateur d'OS, Think, que nous présentons en section 2.3.

2.1 Travaux connexes

Nous présentons dans cette section un résumé portant sur les principaux systèmes d'exploitation existants dans le contexte des WSNs. Les travaux de Han et al. (2005), de Dunkels et al. (2004) et de Cao et al. (2008) sont basés sur approches basées modules. Dans Han et al. (2005), l'architecture de SOS consiste en des modules chargés dynamiquement s'appuyant sur un noyau compilé statiquement. Dunkels et al. (2004) présentent Contiki, un système d'exploitation léger supportant le chargement et le remplacement dynamique de programmes et de services. Balani et al. (2006) s'appuient sur un noyau SOS et implémente une machine virtuelle au dessus.

D'autres propositions sont basées sur une mise à jour complète de l'image du système et de son code fonctionnel. Par exemple, TinyOS est un système d'exploitation pour les réseaux de capteurs qui génère à ce titre une image binaire de l'application complète. Cependant, même si TinyOS prétend s'appuyer sur une programmation modulaire basée sur les composants, ces derniers sont noyés dans le code final et perdent ainsi leur modularité. Ainsi, il est impossible de reconfigurer une image. Bhatti et al. (2005) présente Mantis, un système d'exploitation basée sur une image monolithique du système. L'inconvénient de ce type de proposition est qu'elle requiert un redémarrage du système afin de prendre en compte les éventuelles mises à jour.

Dans ce contexte nous proposons *Valentine* : un système d'exploitation basé composant, dynamique et adaptatif pour les WSNs. Nous présentons cet OS dans la section suivante. La Figure 1 présente un tableau récapitulatif de ces différents OS.

2.2 Les limites de TinyOS pour la reconfiguration dynamique

TinyOS est un système d'exploitation conçu pour les systèmes embarqués et, en particulier, pour les WSNs. Dans le but de réduire la taille des systèmes générés, TinyOS effectue des optimisations et impose des restrictions : par exemple, la mémoire est allouée statiquement à la compilation et le modèle d'implémentation fournit via *nesC* ne dispose pas de pointeurs de fonctions ... Par conséquent, l'allocation dynamique n'est pas possible et c'est une des limitations de TinyOS.

D'un point de vue architectural, TinyOS et les applications développées dessus sont construits en utilisant une approche basée composant. Cependant, pour des raisons d'optimisation, les composants, lors de la compilation, sont noyés dans une seule image binaire monolithique. Par conséquent, le concept de composant disparait lorsque l'image du système est générée. Donc, si nous considérons la propriété de reconfiguration et que nous supposons sa

OS \ Modèle de programmation	Image	Module	Composa
TinyOS	X		
SOS		X	
Contiki	X		
Mantis	X		
LiteOS		X	
Valentine			X

FIG. 1 – *Tableau comparatif sur les différents modèles de programmation*

faisabilité, cela implique que si nous voulons modifier l'implémentation d'un composant, comme le protocole de communication par exemple, il est impossible de remplacer seulement ce composant. L'image entière du système présent sur le capteur doit être réinstallée. Cela représente un coût important en terme de consommation d'énergie. En effet, une fois les capteurs déployés dans leur environnement cible, le seul moyen de déployer une nouvelle image est de la transférer via radio, ce qui est fortement énergivore. Les capteurs sont utilisés comme des nœuds relais pour la propagation des données. Il est clair qu'une image entière du système est plus grande qu'une partie de l'image. Ainsi, si nous supposons l'existence d'un mécanisme de reconfiguration pour un capteur fonctionnant sous TinyOS, cela implique de découper cette image en plusieurs paquets afin de les transmettre au capteur cible. Par conséquent, le trafic sur le réseau augmente ainsi que l'activité de chaque nœud relais impliqué dans le transfert des paquets. Chaque nœud impliqué dans le processus de transfert consomme alors plus d'énergie. Or l'énergie est une ressource limitée dans cet type de réseau. Finalement, TinyOS ne fournit pas de mécanisme pour la protection mémoire. Donc les systèmes sont particulièrement vulnérables aux crashs et corruptions de la mémoire.

La remarque précédente montre que TinyOS ne répond pas aux nouveaux besoins des applications futures utilisant des réseaux de capteurs. Par conséquent, il apparaît que pour résoudre le problème de la reconfiguration dynamique dans les WSNs, il est nécessaire de disposer d'un nouvel OS. C'est pourquoi nouys avons proposé *Valentine*, un nouvel OS basé composant, que nous avons présenté précédemment dans Hoang et al. (2008). Nous en décrirons brièvement le concept en section 2.4. Mais avant nous présenterons Think, un générateur d'OS embarqués que nous avons utilisé pour créer *Valentine*.

2.3 Fractal / Think

Nous avons choisi d'utiliser Think, un générateur d'OS, afin de créer *Valentine*. En effet, Think, est un canevas logiciel pour les noyaux de systèmes d'exploitation basés composant (voir Fassino et al. (2002)). Il permet aux concepteurs d'OS d'implémenter tout noyau de système d'exploitation. Il peut également être utilisé pour développer tout système ou application C.

De plus, notre principale motivation réside en le fait que Think est une implémentation C du modèle de composant Fractal. Fractal permet d'implémenter, de déployer et de gérer des systèmes logiciels complexes (voir Bruneton et al. (2004)). Ces objectifs sont à l'origine des caractéristiques principales du modèle de composant Fractal : les composants composites et partagés, les capacités d'introspection, de configuration et de reconfiguration. Le modèle de composant Fractal utilise le principe de conception basée sur la séparation des préoccupations. L'idée est de séparer en différentes parties distinctes de code ou en entités exécutables les différentes préoccupations ou aspects d'une application. En particulier, Fractal utilise trois cas spécifiques de séparation des préoccupations appelés : *la séparation interface / implémentation*, *la programmation orientée composant*, et *l'inversion du contrôle*. Le premier pattern correspond à la séparation de la conception et de l'implémentation. Le second pattern correspond à la séparation de l'implémentation en plusieurs entités composables, plus petites et implémentées séparément, appelées composants. Le dernier pattern concerne la séparation du fonctionnel et du non-fonctionnel

(configuration) : au lieu de trouver et de reconfigurer eux-mêmes les composants et les ressources dont ils ont besoin , les composants Fractal sont configurés et déployés par une entité externe et séparée. Le principe de séparation des préoccupations est aussi appliqué à la structure des composants Fractal. Un composant Fractal est, en effet, composé de deux parties : un contenu qui gère la partie fonctionnelle et une membrane de contrôleurs, qui gère la partie non-fonctionnelle (introspection, configuration,...). Le contenu est conçu à partir d'autres composants Fractal. Les interfaces d'introspection et de configuration qui peuvent être fournies par les contrôleurs permettent aux composants d'être déployés et reconfigurés dynamiquement. Par exemple, le *BindingController* gère les liaisons entre les composants. Malheureusement, les caractéristiques avancées du modèle Fractal possèdent un coût qui n'est pas toujours compatible avec les ressources limitées des environnements contraints.

2.4 Valentine

Malgré les lacunes de TinyOS pour la reconfiguration dynamique, ce dernier a des caractéristiques utiles pour les WSNs comme par exemple le modèle d'exécution basé événement. En effet, un tel modèle est souvent utilisé dans les systèmes embarqués car il permet, d'une part, de générer une petite empreinte mémoire et, d'autre part, de contrôler plus facilement les activités d'ordonnancement. Partant de ces bonnes pratiques, nous avons proposé *Valentine*, un nouveau système d'exploitation basé composant. Celui-ci est construit à partir de la combinaison des aspects éprouvés de TinyOS avec l'ingénierie logicielle basée composant au travers de Think. Malgré une consommation de ressources notamment mémoire un peu excessive, l'utilisation du modèle proposé par Think semble prometteuse plus particulièrement en vue d'ajouter les aspects dynamiques manquants au modèle TinyOS.

Un capteur a un microprocesseur, plusieurs périphériques d'E/S (réseau, interface série,...) et des périphériques mémoires. Chaque composant système doit donc correspondre à un élément matériel. Dans un premier temps, il a été nécessaire de déterminer comment l'OS utilise ces composants afin de réaliser les fonctionnalités de l'application. Dans un système traditionnel les différentes fonctionnalités sont implémentées par des processus légers, appelés threads. Ces derniers sont alloués au processeur par l'ordonnanceur. Néanmoins, utiliser des threads pour des WSNs semble être une solution quelque peu coûteuse puisque chaque capteur doit sauvegarder une copie du contexte d'exécution en mémoire durant l'ensemble de fonctionnement. De plus, les mécanismes bloquant, tels que les attentes de messages, ne doivent pas être autorisés dans le but de ne pas entraver l'exécution des autres processus. Par conséquent, nous avons choisi d'implementer un mécanisme de queue FIFO[3] telle que celle utilisée par TinyOS. En effet, (i) c'est un algorithme simple à implémenter, (ii)le temps d'activité d'un capteur devant être le plus court possible, les tâches de longue durée sont considérées comme marginales, (iii) comme nous sommes en présence d'un système "mono- utilisateur", si une tâche monopolise le processeur, il s'agit d'une situation acceptable.

Dans un second temps, nous avons dû déterminer le meilleur type de noyau pour notre OS. De façon évidente, un noyau monolithique est trop volumineux pour pouvoir être utilisé sur des capteurs. Un exonoyau est bien adapté car ainsi l'application a directement accès aux composants matériels dont elle a besoin.

Pour finir, la nature des réseaux de capteurs est fortement basée évènement du fait, premièrement des fortes contraintes existantes et deuxièmement du lien très étroit entre le capteur et son environnement proche. Ce sera en effet souvent le capteur qui sera à l'origine de la détection d'un phénomène et qui en avertira le système. Par conséquent, notre OS repose sur un modèle basé évènement fonctionnant de la manière suivante : les composants de plus bas niveau signalent des évènements aux composants de niveau supérieur qui sont capables de les traiter et les composants de niveau supérieur peuvent demander l'exécution de tâches à des composants de niveau inférieur via des appels de fonction. De même, des composants d'un même niveau communiquent également par des appels de fonctions. La Figure 2 schématise ce mécanisme. Du fait de ce côté fortement orienté événement, nous proposons un nouveau modèle de composant. Celui-ci ajoute au concept de composant Fractal une partie orienté événement. Nous décrivons ce modèle de composant dans la section 3.

3 Le modèle de composant proposé

3.1 Description structurelle

Comme expliqué précédemment, les WSNs sont fortement orientés événement. Durant un traitement, comme une agrégation de données, un événement peut survenir. Par exemple, la batterie peut avertir le système qu'elle est trop faible pour continuer à alimenter le capteur. Il est donc impératif d'avoir un mécanisme afin de récupérer

[3]FIFO abréviation couramment employée pour "First In, First Out" signifiant en français "Premier Arrivé, Premier Servi".

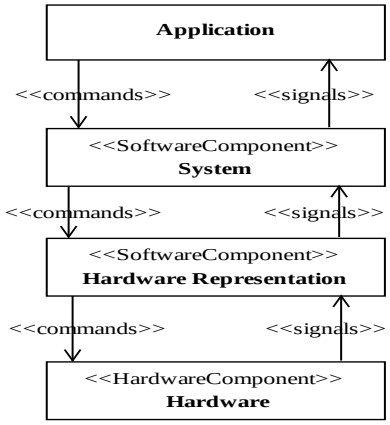

FIG. 2 – *Fonctionnement du système*

ces événements et les traiter. Chaque composant requiert donc la présence d'un gestionnaire d'événements correspondant aux événements nécessaires à son fonctionnement. Par conséquent, nous avons fixé comme règle que chaque composant de notre système devait implémenter deux sous-composants : un gestionnaire d'événements appelé *eventHandler* et une partie fonctionelle appelée *behaviour*. Le sous-composant *eventHandler* contient la liste des événements qui peuvent être signalés ou être à l'origine de l'exécution d'une tâche par le composant. Ces événements sont de deux types : les événements dits classiques qui sont à l'origine du déclenchement d'une tâche et les événements dits de reconfiguration qui sont, comme leur nom l'indique, à l'origine d'une reconfiguration. L'implémentation du gestionnaire d'événements en tant que sous-composant permet de faciliter les éventuelles modifications futures du système. En effet, seulement le gestionnaire et les liens optionnels entre les événements écoutés et le gestionnaire seront modifiés si de nouveaux événements doivent être pris en considération ou si l'ensemble des événements gérés doit être modifié. Le sous-composant *behaviour* est un composant abstrait. Le développeur définira ici le code fonctionnel du composant. La Figure 3 présente ce nouveau modèle de composant. Le composant est composé d'un (ou plusieurs) sous-composant(s) *behaviour* qui dépend des événements reçus. En effet, chaque événement déclenche l'exécution d'une tâche ou d'une reconfiguration. Le code fonctionel de chaque tâche est contenu dans un sous-composant *behaviour*.

Nous avons utilisé Think pour implémenter notre modèle de composant. Nous avons donc implémenté un nouveau composant Think, appelé *ValentineComponent*, composé d'un sous-composant *eventHandler* et d'un sous-composant *behaviour*. Les composants génériques Think ont été étendus afin de respecter notre modèle de composant. Cette extension correspond à l'encapsulation des composants génériques Think dans des *ValentineComponent*.

La Figure 4 représente le nouveau composant Think. Dans ce cas, le *ValentineComponent* fournit au moins les mêmes interfaces que le *ThinkGenericComponent* qui est relié aux sous-composants *eventHandler* et *behaviour*. Le *ValentineComponent* peut fournir une interface pour signaler des événements. La gestion des événements est décrite dans la section 3.2. *bc* est le *BindingController*, qui est l'implémentation du contrôleur Fractal du même nom. Ce contrôleur est défini un peu plus loin. Celui-ci gère les liaisons entre composants. Nous l'avons utilisé pour reconfigurer dynamiquement un *ValentineComponent*. Par exemple, si un *ValentineComponent* reçoit l'événement *reconfigure*, le *bc* délie l'ancien composant et va lier le nouveau composant.

Les événements sont générés par les composants ou par le système d'exploitation. Ils sont signalés aux différents composants par le composant système *scheduler*. La Figure 5 montre les liaisons entre un composant *Valentine* et le système. Le *scheduler* est inclus dans le composant *ValentineOS*. Le *UserSpace* est un composant qui contient toutes les applications en exécution sur le capteur. Chaque composant *Application* est composé d'un ou plusieurs *ValentineComponent*.

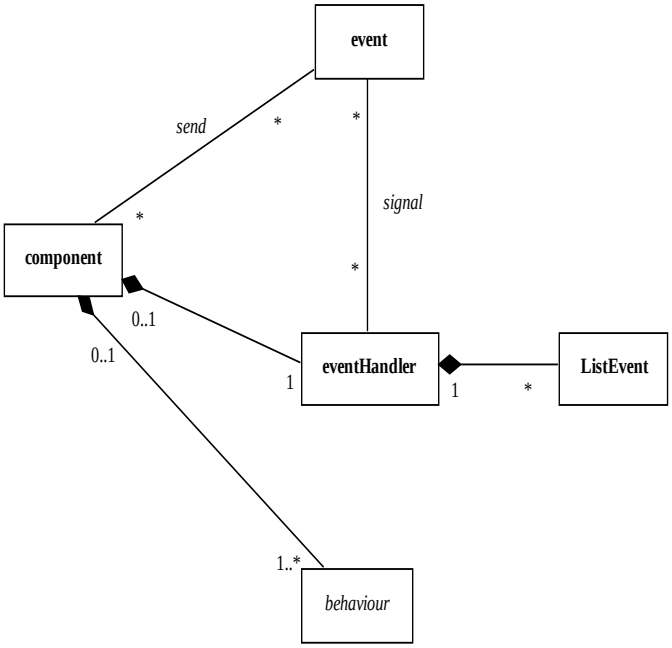

FIG. 3 – *Le modèle de composant abstrait*

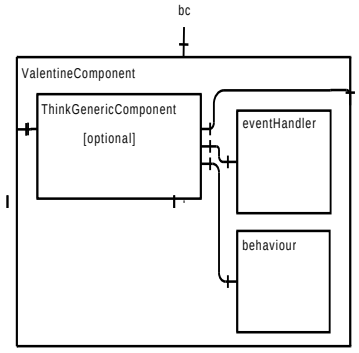

FIG. 4 – *Représentation d'un ValentineComponent*

3.2 Description dynamique

Dans la section précédente, nous avons présenté notre système d'exploitation d'un point vue structurel. Nous allons maintenant montrer comment un composant se comporte lorsqu'il reçoit un événement. La Figure 6 montre le comportement d'un composant à la réception d'un événement.

A la réception d'un événement, le *scheduler* le transmet à tous les composants. Chaque composant vérifie si l'événement appartient à sa liste d'événements par le biais de son sous-composant *eventHandler*. Si l'événement est dans sa liste, le composant place la tâche à exécuter correspondante dans la queue de l'ordonnanceur. Quand la tâche est sélectionnée, l'ordonnanceur envoie un message au composant. Le sous-composant *behaviour* exécute alors la tâche demandée.

La Figure 7 présente le diagramme de séquence d'une reconfiguration dynamique. Si l'événement est un événement *reconfiguration*, le sous-composant *behaviour* appelle le *BindingController* afin de délier l'ancien composant et lier le nouveau composant. Ce mécanisme sera décrit dans la section 4.

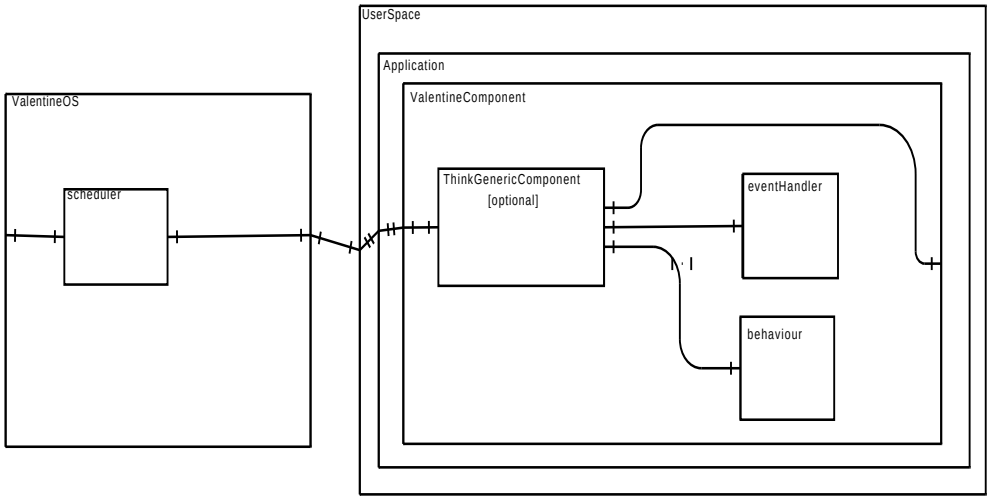

FIG. 5 – *Liens entre un composant Valentine et le système d'exploitation*

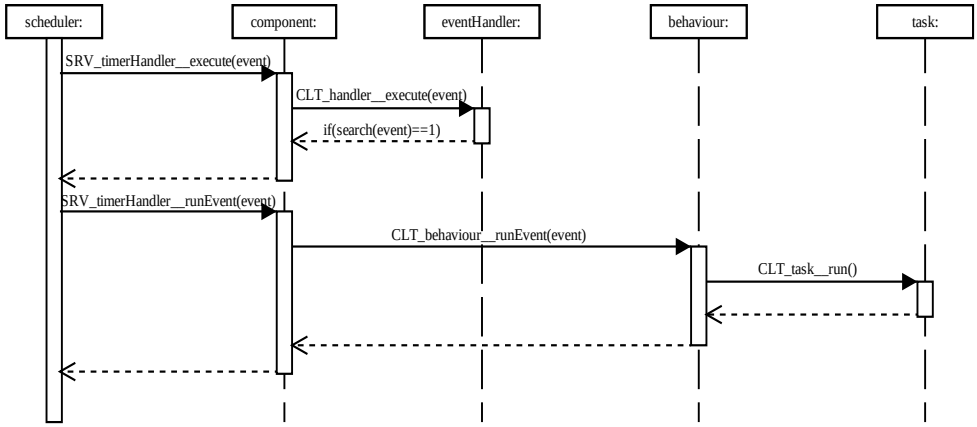

FIG. 6 – *Le diagramme de séquence*

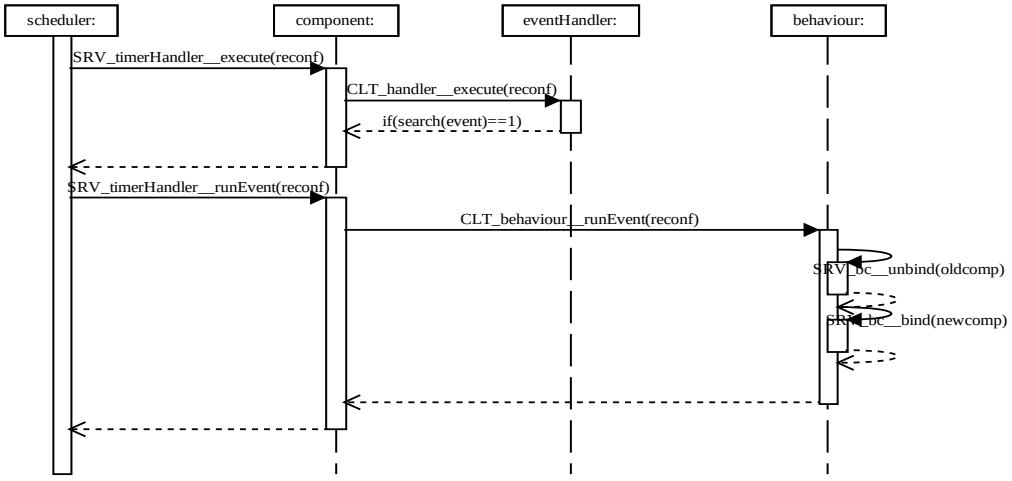

FIG. 7 – *Le diagramme de séquence d'une reconfiguration dynamique*

3.3 Implémentation du modèle de composant

Dans la section précédente, nous avons présenté le modèle de composant d'un point de vue architecture et d'un point de vue dynamique. Nous allons à présent détailler comment nous l'avons implémenté avec Think. Au niveau abstrait, *ValentineComponent* est composé d'un composant *eventHandler*, d'un composant *behaviour* et, dans le cas d'une extension, du composant générique Think étendu. Quand un événement survient, le *ValentineComponent* composite a besoin de rediriger l'événement à ses sous-composants. Pour cela, nous proposons d'ajouter au *ValentineComponent* un nouveau composant appelé *valentineGenericDispatcher*. Ce dernier redirige les événements vers le sous-composant concerné. La Figure 8 montre la nouvelle structure du *ValentineComponent*. Cette nouvelle structure est appelée *astractValentineComponent*.

Le *abstractValentineComponent* est implémenté ainsi :

```
abstract component avr.abstractValentineComponent {
 //the component handler used to receive the scheduler's orders
 provides event.activity.api.ComponentHandler as componentHandler
 //the component is reconfigurable
 provides event.activity.api.Reconfiguration as reconfiguration
 // it need to add events to the scheduler
 requires event.activity.api.SchedulerHandler as schedulerHandler
 //a dispatcher to receive scheduler's orders
 contains dispatcher : avr.abstractComponent
 //an event handler te accept or refuse events
 contains eventHandler = event.activity.lib.eventhandler
 contains behaviour : event.activity.lib.behaviour
 //the componentHandler provided is the dispatcher's one
 binds this.componentHandler to dispatcher.componentHandler
 binds dispatcher.eventHandler to eventHandler.eventhandler
 binds dispatcher.behaviour to behaviour.behaviour
 binds dispatcher.schedulerHandler to this.schedulerHandler
 requires fractal.api.BindingController as bindingController
 provides fractal.api.BindingController as bc in bindingController
 content fractal.lib.bcstring for bindingController
 selfbinds bindingController to bc
 requires fractal.api.ContentController as contentController
 provides fractal.api.ContentController as cc in contentController
 content fractal.lib.cc for contentController
 selfbinds contentController to cc
 requires fractal.api.ComponentIdentity as componentIdentity optional
 requires fractal.api.BindingController as bindingControllerOptional optional }
```

Afin d'implémenter le modèle de composant, nous avons dû créer les composants et interfaces suivants :

- Composants :
  - *eventhandler* : Comme son nom l'indique, c'est le sous-composant qui gère les événements.
  - *behaviour* (abstrait) : Ce composant est abstrait. Le développeur y définit les fonctionnalités du composant.
  - *abstracComponent* : C'est le *ValentineAbstractComponent*. Il contient les sous-composants *eventhandler* et *behaviour*.

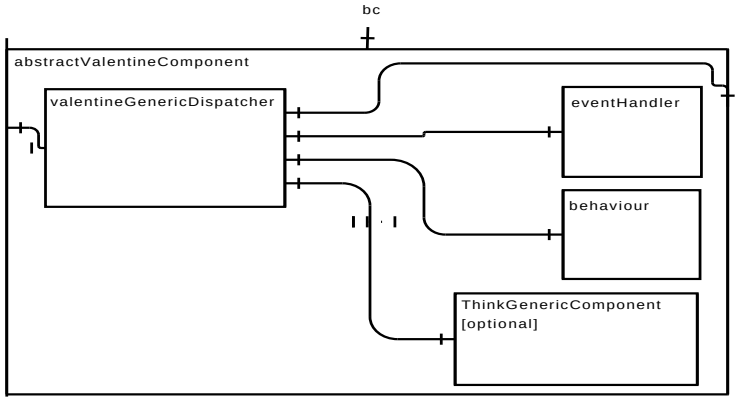

FIG. 8 – *Représentation d'un astractValentineComponent*

– Interfaces :
  – *Behaviour* : Cette interface est utilisée pour interagir avec le sous-composant *behaviour*.
  – *ComponentHandler* : Cette interface permet au *Scheduler* de transmettre les événements aux composants.
  – *EventHandler* : Cette interface est utilisée pour interagir avec le sous-composant *eventhandler*.
  – *Reconfiguration* : Cette interface est utilisée pour effectuer une opération reconfiguration.
  – *SchedulerHandler* : Cette interface lie le composant au composant système *Scheduler*.

Afin d'implémenter notre modèle de composant, nous utilisons les contrôleurs suivant :
– *BindingController* : Ce contrôleur permet de gérer les liaisons entre composants.
– *ContentController* : Ce contrôleur permet de connaître le contenu, i.e. les sous-composants, d'un composant.
– *ComponentIdentity* : Ce contrôleur permet de récupérer l'identité d'un composant.

4 Implémentation d'une reconfiguration dynamique basée sur le modèle de composant

Nous basons notre proposition et son implémentation sur le modèle de composant Fractal et plus précisément sur les interfaces de configuration. Ces interfaces peuvent être fournies par les contrôleurs afin de permettre aux composants d'être déployés et reconfigurés dynamiquement. Nous appliquons le modèle de composant proposé au composant *ValentineComponent*. Ainsi, pour reconfigurer un composant, il faut remplacer son sous-composant *behaviour* par un nouveau. Afin de réaliser cela, nous avons besoin d'accéder aux liaisons de ce dernier. Nous utilisons le *BindingController* qui permet la gestion des liaisons d'un composant.

Soient CC_p le *contentController* du composant parent et BC_p son *bindingController*. Soient CIO_p et BCO_p un *componentIdentity* et un *bindingController* optionnels du composant parent. Enfin soient CI_f et BC_f le *componentIdentity* et le *bindingController* du composant ciblé (la fonctionnalité). Le but est d'avoir accès au *bindingController* de la fonctionnalité depuis le composant parent. Pour cela, on suit la démarche suivante :

1. on utilise CC_p pour obtenir une référence vers CI_f
2. BC_p est utilisé afin de lier CIO_p à CI_f
3. on utilise CIO_p (soit CI_f) pour obtenir une référence vers BC_f
4. on utilise BC_p pour lier BCO_p à BC_f

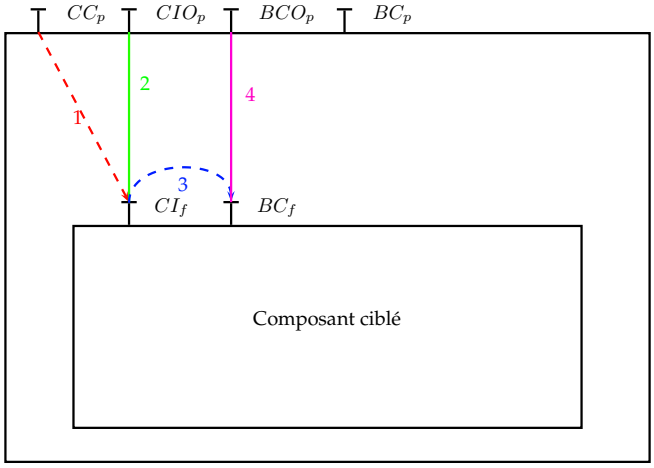

FIG. 9 – *Mécanisme de reconfiguration dynamique*

Une fois (4) réalisé, nous pouvons lier/délier toutes les interfaces du composant *behaviour* et ainsi réaliser la reconfiguration voulue. La séquence que nous venons de décrire est appelée *prise de contrôle*.

Imaginons un composant qui doit faire clignoter une led jaune . Nous souhaitons le reconfigurer pour passer de la led jaune à la verte. L'architecture du composant est donc la suivante :

1. un composant qui ordonne à la led de s'éteindre puis de s'allumer périodiquement afin de la faire clignoter. Nous appellerons ce composant *clignotant*.

2. un composant qui représente la diode jaune et qui permet via son interface de demander à la diode jaune de s'allumer ou de s'éteindre. Il est lié au *clignotant*.
3. un composant équivalent au précédent mais pour la diode verte. Il n'est pas lié et est présent librement dans le composant.

Le but de la reconfiguration est donc de remplacer la liaison du *clignotant* vers la led jaune par une liaison vers la led verte. La procédure à suivre est alors la suivante :

1. prendre le contrôle du *clignotant* (ou de la led jaune) afin de supprimer la liaison.
2. prendre le contrôle de la led verte afin d'obtenir une référence vers son interface de contrôle de led.
3. prendre le contrôle du *clignotant* et y lier l'interface récupérée en 2 .

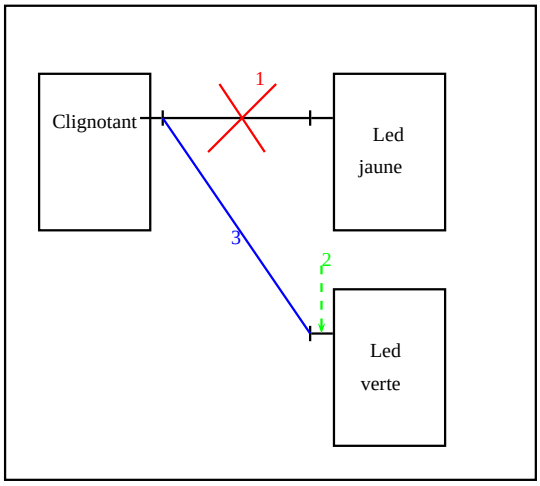

FIG. 10 – *Un exemple simple*

Au final, un composant qui doit être reconfigurable possède quatre interfaces dont deux optionnelles :
– un *contentController* afin de sélectionner les composants internes
– un *bindingController* pour lier les interfaces du composant que l'on veut changer
– un *componentIdentity* optionnel et lié dynamiquement. Il est utilisé pour obtenir les interfaces d'un composant
– un *bindingController* optionnel et lié dynamiquement. Il est utilisé pour (dé)lier un composant.

5 Exemple et évaluation

5.1 Exemple : l'application de surveillance *MonitorApp*

L'application de surveillance *MonitorApp* permet de surveiller les changements de l'environnement. Cette application est déployée sur chaque capteur afin d'acquérir la temperature, le taux d'humidité et la pression de leur environnement proche. Plus précisément, chaque capteur mesure ces trois données, calcule leur moyenne et les enregistre. Le capteur transmet les informations à la station de travail pour traitement toutes les heures.

Cette application est composée d'un composant *timerValentine*, un composant *acquisition*, un composant *calculation*, un composant *storage* et un composant *radioValentine*. Le composant *timerValentine* est utilisé pour envoyer deux événements : un événement *mesure* toutes les dix minutes et un événement *send* toutes les heures. Les événements sont transmis à tous les composants à travers le composant *scheduler*. Quand le composant *acquisition* reçoit l'événement *mesure*, il mesure les données demandées, les transmet au composant *calculation* qui calcule la moyenne avant de les transmettre au composant *storage*. Quand un événement *send* survient, le composant *storage* communique ses données au composant *radioValentine* qui les envoie à la station de travail au travers du réseau.

5.2 Application du modèle de composant à cet exemple

La Figure 11 représente une application utilisant notre modèle de composant. Chaque composant contient un composant *behaviour* et un composant *eventHandler*. Le composant *timerValentine* et le composant *radioValentine* sont de type *ValentineComponent*. Le premier contient le composant générique Think *timer* et le second contient le composant générique Think *radio*.

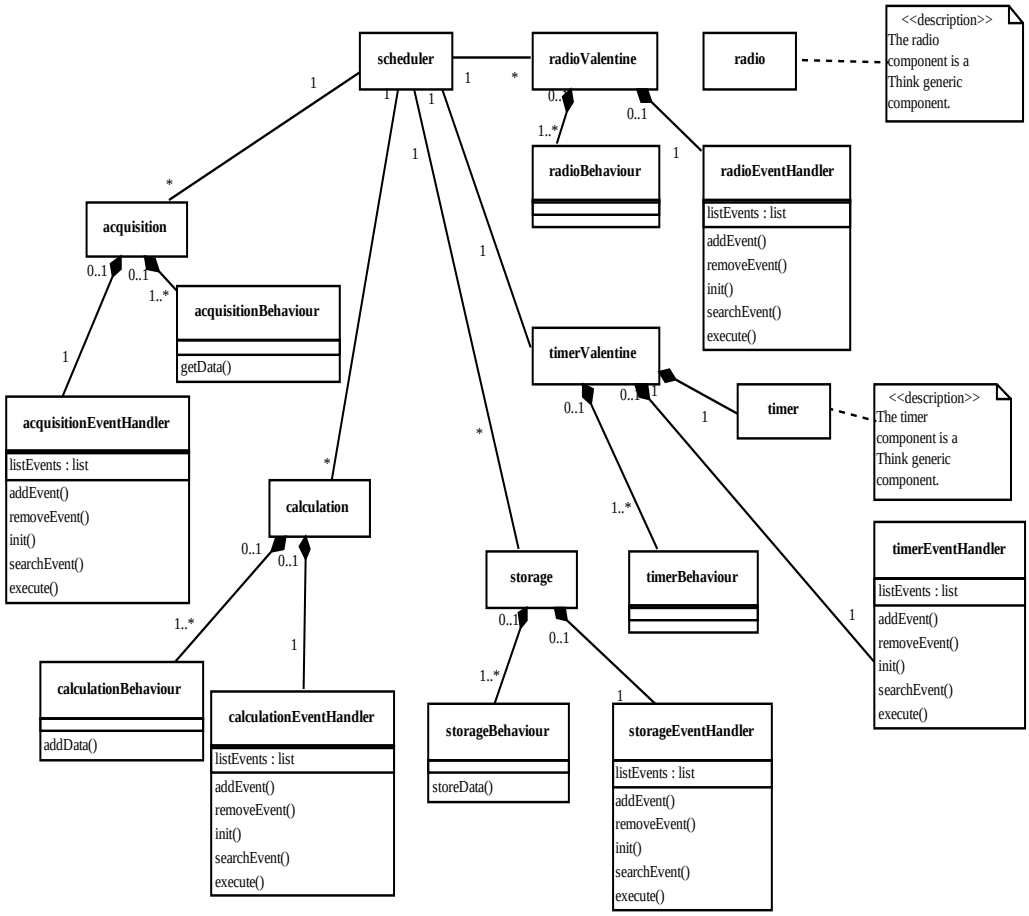

FIG. 11 – *L'application de surveillance*

Quand un événement *mesure* apparaît, le composant *acquire* envoie un événement au *scheduler* qui le place dans sa file. Quand l'événement est séléctionné, le *scheduler* l'envoie au composant *acquire* qui mesure les données demandées et les transmet au composant *calculation* pour calculer la moyenne. Ensuite, ce dernier communique ses résultats au composant *storage* pour sauvegarde. Toutes les heures, sur reception de l'événement *send*, le composant *storage* transmet ses données au composant *radioValentine* qui les envoie à la station de travail. Quand un événement apparaît, il est traité par chaque sous-composant *eventHandler* de chaque composant. Les tâches (lecture de données, calcul de moyenne,...) sont traitées par chaque sous-composant *behaviour* de chaque composant.

5.3 Evaluation

Nous présentons dans cette section les résultats et mesures obtenus. Pour cela, nous avons développé un exemple simple sous TinyOS et sous *Valentine* et nous avons comparé les images obtenus. L'exemple consiste en une application qui, à intervalle régulier, fait clignoter la led verte. L'application est donc composée d'un composant

Timer et d'un composant *Leds*. En implémentant cet exemple avec TinyOS, nous obtenons une image de 4,23 Ko. Cette image contient le système ainsi que l'application.

En ce qui concerne l'évaluation de l'exemple implémenté sous *Valentine*, nous avons choisi de procéder en plusieurs étapes. Dans un premier temps nous avons mesurer la taille de l'image du système d'exploitation seul. Comme nous l'avons dit précédemment, Think est gourmand en terme de consommation de ressources et notamment en ce qui concerne les ressources mémoires. Par exemple, l'image de *Valentine* pèse déjà 48,3 Ko, c'est-à-dire 11 fois plus que l'image obtenu pour l'exemple avec TinyOS. Dans un second temps, nous avons implémenté l'exemple sans les parties modèle de composant et reconfiguration dynamique. L'image qui en résulte mesure 120,9 Ko. Ensuite, nous avons appliqué le modèle de composant que nous proposons. Nous obtenons alors une image de 189,4 Ko. Pour terminer, l'implémentation de la propriété ainsi que la mise en œuvre du code de reconfiguration dynamique coûte seulement 0,5 Ko. L'image totale mesure alors 189,9 Ko. La Figure 12 présente un tableau récapitulant les tailles des différentes images.

Description	Taille (Ko)	Total (Ko)
TinyOS + Application	6,35	6,35
Valentine	48,3	48,3
Application	72,6	84,6
Modèle de composant	68,5	189,4
Reconfiguration dynamique	0,5	189,9

FIG. 12 – *Tableau récapitulatif des tailles des images*

6 Conclusions et travaux futurs

Dans ce papier, nous avons traité de la reconfiguration dynamique dans le cadre des réseaux de capteurs sans fil. Nous avons mis en évidence que TinyOS ne permettait pas cette fonctionnalité à cause du fait que son architecture à base de composant est délayée dans le processus d'optimisation. Pour palier cette limitation, nous avons présenté notre approche. Celle-ci est basée sur deux contributions. D'une part, nous avons développé un nouveau système d'exploitation appelé *Valentine*. Celui-ci a été généré en utilisant Think, un générateur de systèmes d'exploitation basés composants. D'autre part, nous avons développé un modèle de composant basé sur Fractal spécifique à la reconfiguration dynamique et déployé sur *Valentine*. Cette double contribution nous permet d'administrer à la fois les composants fonctionnels déployés sur notre système mais aussi les composants même du système.

Notre système d'exploitation et notre modèle de composants sont aujourd'hui fonctionnels. Afin de finir d'outiller notre approche, il nous reste à mettre en place un protocole de reconfiguration prenant en compte les particularité des réseaux de capteurs, notamment en terme d'économie d'énergie et donc économe en communication. L'étape suivante portera sur l'évaluation de notre travail en terme de performance, de vitesse de reconfiguration et de réduction de consommation des ressources.

Références

Balani, R., C. Han, R. K. Rengaswamy, et I. Tsigkogiannis (2006). Multilevel Software Reconfiguration for Sensor Networks. *ACM Conference on Embedded Systems Software (EMSOFT)*.

Bhatti, S., J. Carlson, H. Dai, J. Deng, J. Rose, A. Sheth, B. Shucker, C. Gruenwald, A. Torgerson, et R. Han (2005). MANTIS OS : An Embedded Multithreaded Operating System for Wireless Micro Sensor Platforms. Volume 10, pp. 563–579.

Bruneton, E., T. Coupaye, et J.-B. Stefani (2004). The Fractal Component Model Specification. Draft, version 2.0-3, ObjectWeb Consortium. `http://fractal.objectweb.org`.

Cao, Q., T. Abdelzaher, J. Stankovic, , et T. He (2008). The liteos operating system : Towards unix-like abstractions for wireless sensor networks. In *In Proceedings of the 7th International Conference on Information Processing in Sensor Networks (ACM/IEEE IPSN)*.

Crnkovic, I. (2006). Component-Based Software Engineering for Embedded Systems. In J.-P. Babau, J. Champeau, et S. Gérard (Eds.), *From MDD Concepts to Experiments and Illustrations*, Chapter 5, pp. 71–90. London, UK : ISTE.

Dunkels, A., B. Grönvall, et T. Voigt (2004). Contiki - a Lightweight and Flexible Operating System for Tiny Networked Sensors. In *Proceedings of the First IEEE Workshop on Embedded Networked Sensors (Emnets-I)*, Tampa, Florida, USA.

Fassino, J.-P., J.-B. Stefani, J. Lawall, et G. Muller (2002). THINK : A Software Framework for Component-based Operating System Kernels. In *In the USENIX Annual Technical Conference*, Monterey, CA, USA, pp. 73–86.

Han, C.-C., R. Kumar, R. Shea, E. Kohler, et M. Srivastava (2005). A Dynamic Operating System for Sensor Nodes. In *Proceedings of the 3rd international conference on Mobile systems, applications, and services*, Seattle, Washington, USA.

Hill, J., R. Szewczyk, A. Woo, S. Hollar, D. Culler, et K. Pister (2000). System Architecture Directions for Network Sensors. In *Proceedings of Ninth International Conference ASPLOS*, Cambridge, MA, USA.

Hoang, N., N. Belloir, C.-D. Pham, et S. Sentilles (2008). Valentine : A Dynamic and Adaptive Operating System for Wireless Sensor Networks. In *Proceedings of the 1st IEEE International Workshop on Component-Based Design of Resource-Constrained Systems*, Turku, Finland.

Szyperski, C., D. Gruntz, et S. Murer (2002). *Component Software – Beyond Object-Oriented Programming* (2nd ed.). ACM Press. New York, NY : Addison-Wesley.

Weiser, M. (1993). Some Computer Science Issues in Ubiquitous Computing. *Communications of the ACM 36*(7), 75 – 84.

Summary

In this paper, we present an approach allowing dynamic reconfiguration in wireless sensor networks (WSNs). This proposition is based on a software engineering build around a new operating system and a componnt model. This operating system, called *Valentine*, allows to keep the building by component at runtime, in opposition to monolithic contruction used by the main existing solutions in WSNs. It generated through the Think framework. The component model allows to reuse the Fractal abtract component model and to implement the mechanism of dynamic reconfiguration.