Modèle à Composant pour Plate-forme Autonomique

Pierre Bourret

UNIVERSITÉ DE GRENOBLE

THÈSE

Pour obtenir le grade de

DOCTEUR DE L'UNIVERSITÉ DE GRENOBLE

Spécialité : **Informatique**

Arrêté ministériel : 7 août 2006

Présentée par

Pierre Bourret

Thèse dirigée par **Pr Philippe Lalanda**
et codirigée par **Dr Clément Escoffier**

préparée au sein du **Laboratoire Informatique de Grenoble**
et de l'**École Doctorale MSTII (Mathématiques, Sciences et Technologies de l'Information, Informatique)**

Modèle à Composant pour Plate-forme Autonomique

Thèse soutenue publiquement le **24 octobre 2014**,
devant le jury composé de :

Lydie du Bousquet
Professeur de l'Université Joseph Fourier de Grenoble, Présidente

Éric Gressier-Soudan
Professeur du Conservatoire National des Arts et Métiers de Paris, Rapporteur

Philippe Roose
Maître de Conférences de l'IUT de Bayonne et du Pays Basque, Rapporteur

Nicolas Rempulski
Directeur technique, Ubiquitous Dreams, Examinateur

Philippe Lalanda
Professeur à l'Université Joseph Fourier de Grenoble, Directeur de thèse

Clément Escoffier
Chercheur à l'Université Joseph Fourier de Grenoble, Co-Directeur de thèse

À Jacqueline et Françoise.

Remerciements

Ce travail de thèse n'aurait pu voir le jour sans les nombreuses personnes qui m'ont entouré durant cette longue épreuve. Cette section est là pour eux : tous ceux qui, à leur manière, m'ont ont soutenu, et ont donc participé à l'achèvement de ce travail.

Je tiens avant tout à remercier l'ensemble des membres du jury : un grand merci pour vos relectures attentives, vos corrections minutieuses et vos questions et remarques. Merci à Éric Gressier Soudan et Philippe Roose d'avoir rapporté ce manuscrit avec une délicate attention. Merci à Nicolas Rempulski d'être venu examiner ma soutenance. Merci à Lydie du Bousquet d'avoir présidé ce jury et, malgré des délais très tendus, d'avoir pu relire et corriger minutieusement ce manuscrit.

Je ne remercierai jamais assez mon directeur Philippe Lalanda d'avoir cru en ces travaux de recherche. Merci à toi de m'avoir accordé ta confiance, ton soutien et ton aide durant ces années, et de m'avoir accueilli et hébergé dans ton équipe.

Un énorme merci aussi à Clément Escoffier, mon directeur (littéralement) exécutif. Merci à toi d'être revenu et d'avoir partagé tes connaissances, tes expériences, ton énergie et même ton modèle à composant. Gloire au dynamisme !

Je remercie chaleureusement tous les membres passés et présents de l'équipe ADELE. Merci pour tous ces moments de détente et toutes ces idées partagées ensemble. Merci à Ozan, Jander, Jo, Issac, Walter, Don Gabo, Jian Qi, Mehdi, Didier, Colin, Thibault, Thomas, Yoann, Éric, Bassem, Ada, Kiev, German, Denis, Stéphanie, Vincent, PA, Idrissa, Étienne, João, Jacky, Marc... Merci à Johann de m'avoir trouvé, et ainsi permis de commencer ma grande aventure chez ADELE.

Je remercie également mes amis, qui ont réussi à me supporter durant ces quatre années. Merci à Quentin, Sabri, Clément-les-ciseaux, Ozan, Issac et Eli, Jander, Jo, Walter et Marcia, Mehdi, Don Gabo, Jiana Qi, Morganichou, Lulu, Luis et Marie...

Merci à toute ma famille de m'avoir encouragé sans condition. Merci Mum, Papa, Paul, Nelson, Binbin, Pauline, Bibi, Marion, Nico. Merci à l'oncle Tom de m'avoir initié aux arcanes académiques. Merci Christine et Alain, qui, il y a onze ans, m'avez recueilli chez vous. Je n'aurai jamais pu vivre cette folle aventure grenobloise sans votre soutien.

Enfin, je tiens à remercier du fond du cœur Diana, qui, bien qu'en thèse elle aussi, a réussi à m'apporter tout son amour et son soutien. Merci pour tous ces instants de bonheur. Je n'aurais pas pu traverser ces épreuves sans toi. Gracias mi amor !

Résumé

Ces dernières décennies, les environnements informatiques sont devenus de plus en plus complexes, parsemés de dispositifs miniatures et sophistiqués gérant la mobilité et communiquant sans fil. L'informatique ubiquitaire, telle qu'imaginée par Mark Weiser en 1991, favorise l'intégration transparente de ces environnements avec le monde réel pour offrir de nouveaux types d'applications. La conception de programmes pour environnements ubiquitaires soulève cependant de nombreux défis, en particulier le problème de rendre une application auto-adaptable dans un contexte en constante évolution. Parallèlement, alors que la taille et la complexité de systèmes plus classiques ont explosé, IBM a proposé le concept d'informatique autonomique afin de réduire le fardeau de l'administration de systèmes imposants et largement disséminés.

Cette thèse se base sur une approche où les applications sont conçues sous la forme de composants utilisant et fournissant des services. Un modèle de développement fondé sur une architecture de référence pour la conception d'applications ubiquitaires est proposée, fortement inspiré des recherches dans le domaine de l'informatique autonomique. Dans ce modèle, les applications sont prises en charge par une hiérarchie de gestionnaires autonomiques, qui appuient leurs décisions sur une représentation centrale du système. La mise en œuvre de cette contribution requiert de rendre la couche d'exécution sous-jacente plus réflexive, en vue de supporter de nouveaux types d'adaptations à l'exécution. Nous proposons également un modèle qui décrit le système à l'exécution et reflète sa dynamique de manière uniforme, suivant les principes du style d'architecture *REST*. Les applications reposant sur cette couche d'exécution réflexive et représentées par ce modèle sont qualifiées d'*Autonomic-Ready*.

L'implantation de nos propositions ont été intégrées dans le modèle à composant orienté service *Apache Felix iPOJO*. Le modèle de représentation du système, nommé Everest, est publié en tant que sous-projet d'*OW2 Chameleon*. Ces propositions ont été évaluées et validées par la conception et l'exécution d'une application ubiquitaire sur iCASA, un environnement de développement et de simulation.

Abstract

In the last decades, computing environments have been getting more and more complex, filled with miniaturized and sophisticated devices that can handle mobility and wireless communications. Ubiquitous computing, as envisioned by Mark Weiser in 1991, promotes the seamless integration of those computing environments with the real world in order to offer new kinds of applications. However, writing software for ubiquitous environments raises numerous challenges, mainly the problem of how to make an application adapt itself in an ever changing context. From another perspective, as classical softwares were growing in size and complexity, IBM proposed the concept of autonomic computing to help to contain the burden of administering massive and numerous systems.

This PhD thesis is based on an approach where applications are designed in terms of components using and providing services. A development model based on a reference architecture for the conception of ubiquitous applications is proposed, greatly inspired by researches in the autonomic computing field. In this model, the application is managed by a hierarchy of autonomic managers, that base their decisions on a central representation of the system. The fulfilment of this contribution requires to make the underlying middleware more reflexive, in order to support new kinds of runtime adaptations. We also provide a model that depicts the running system and its dynamics in a uniform way, based on REST principles. Applications relying on this reflexive middleware and represented by this model are what we called Autonomic-Ready.

Implementations of our proposals have been integrated in the Apache Felix iPOJO service-oriented component model. The system representation, named Everest, is provided as a OW2 Chameleon subproject. Validation is based on the iCASA pervasive environment development and simulation environment.

Table des matières

Table des figures

Liste des tableaux

Chapitre 1

Introduction

Dans ce premier chapitre, nous présentons le contexte de notre travail, ainsi que les objectifs précis que nous avons poursuivis. Nous détaillons également la structure de ce document de thèse.

En particulier, nous montrons que les développements informatiques ont considérablement évolué ces dernières années qu'ils abordent de plus en plus le domaine des services pervasifs. Ce domaine présente des exigences liées à la gestion du dynamisme et de l'hétérogénéité très difficiles à atteindre.

Sommaire

1.1 Contexte

Notre environnement contient un nombre sans cesse croissant de systèmes informatiques. Il est aujourd'hui empli d'appareils numériques qui se distinguent par leur forme, leur puissance, leur rôle. Nombre de ces systèmes sont conçus pour nous assister et nous accompagner dans nos activités quotidiennes et même dans nos interactions sociales. Ces nouveaux types de services suscitent dès aujourd'hui de grandes attentes dans des domaines aussi variés que la santé, la domotique, l'énergie, le transport, *etc.* et ce, malgré un manque évident de maturité et donc, en particulier, de fiabilité et de sécurité. En effet, la mise en place de ces services soulève de nombreux problèmes. Il convient de gérer des ressources hétérogènes, dynamiques, dont les services ne sont pas les propriétaires (et qui ont donc des cycles de vie indépendants). Il est également souvent nécessaire de gérer des quantités de données importantes. Eric Schmidt, *CEO* de Google, estimait en 2005 que la masse de données disponible électroniquement atteignait le chiffre étourdissant de 5 millions de téraoctets (seulement 0,004 % étant indexées par Google). Il estimait également que ce chiffre doublait tous les 5 ans [Morand 2013].

La mise en place de services numériques demande la création et la maintenance d'infrastructures d'intégration particulièrement complexes. Il s'agit plus précisément d'intégrer des éléments physiques éventuellement mobiles (capteurs, téléphones, *etc.*) et des systèmes d'information (SI) plus classiques, mais aussi plus lourds et peu flexibles. Ces éléments sont distants, de natures différentes et, bien sûr, caractérisés par des exigences très différentes.

Le domaine des applications ambiantes et ubiquitaires pose aujourd'hui un ensemble de problèmes de recherche non résolus du point de vue des logiciels. Les problèmes présentés ci-dessous sont particulièrement structurants :

- la distribution et la multiplicité des équipements des équipements mis en réseau,
- l'hétérogénéité des équipements qui possèdent des protocoles de communication variés, des langages de programmation distincts et des modèles de données sémantiquement différents,
- la dynamique des environnements qui rend nécessaire l'adaptation des services offerts à l'utilisateur en fonction de son activité et de sa localisation, des capacités et disponibilité des équipements, des caractéristiques ponctuelles de l'espace environnant, *etc.*
- les besoins en autonomie. L'autonomie des applications est un enjeu majeur puisqu'il n y a pas d'administrateur pour mettre en œuvre les adaptations mentionnées précédemment. Les applications doivent donc exhiber des propriétés d'auto-adaptation à leur contexte d'exécution (capacités matérielles, logiciels disponibles, connectivité, présence d'autres équipements, localisation, activité de l'utilisateur, *etc.*) et de sûreté d'exécution (sécurité, autoréparation).

Pour aborder ces défis, de nouvelles approches logicielles ont été proposées ces dernières années. Les approches à composants [Szyperski 2002], apparues dans les années 1990, permettent la mise en place d'applications modulaires plus faciles à faire évoluer. Les composants ont pour vocation de servir comme éléments de base pour la construction d'applications logicielles. Les approches à composants perçoivent le développement d'applications logicielles comme un assemblage de composants, et gèrent la maintenance et l'évolution d'applications par la personnalisation et le remplacement de composants. Elles se situent à un niveau architectural plus élevé que celui proposé par des objets. Typiquement, un composant peut être constitué de

plusieurs objets pour la réalisation d'une tâche spécifique. Un composant est souvent proposé sous la forme d'une unité de déploiement qui encapsule un certain nombre d'éléments parmi lesquels son implémentation. Un composant possède un ensemble d'interfaces. Ces interfaces définissent les fonctionnalités fournies et requises d'un composant sous la forme d'un contrat spécifique et visible de l'extérieur. Des modèles spécifiques à composants, comme les *EJB*, définissent la structure standard des composants, la manière de réaliser des assemblages et fournissent les mécanismes nécessaires pour la prise en compte de certains aspects non-fonctionnels tels que la distribution, la sécurité ou bien encore les transactions. Les applications basées sur les composants emploient les contrats des composants pour décrire leurs interactions et leurs dépendances fonctionnelles sous un contexte donné avec une vision structurale. Cette représentation explicite de l'architecture élève le niveau d'abstraction par rapport aux assemblages d'objets, puisque la granularité de développement devient plus grande. Néanmoins, il faut comprendre que les dépendances de fonctionnalités des composants reposent sur la granularité des interfaces fournies et requises. Ceci cause un couplage très fort entre certains composants d'une application. En conséquence, le fort couplage contractuel de composants d'une application supporte peu de dynamisme et de flexibilité. En d'autres termes, une fois l'assemblage de composants créé, les changements dans l'architecture sont difficilement gérés au cours de l'exécution de l'application.

Les approches orientées services [Papazoglou 2003] ont été introduites comme un nouveau paradigme pour le développement logiciel durant les années 2000. Ce paradigme utilise la notion de service comme élément de base pour la construction d'applications logicielles. Un service est défini comme une entité logicielle qui peut être utilisée de façon statique ou dynamique pour la réalisation d'une application logicielle. Un consommateur, ou client, sélectionne un service à partir de sa description. Il l'utilise sans avoir connaissance de la technologie sous-jacente nécessaire à son implantation ni de sa plate-forme d'exécution. Inversement, le service ne connaît pas le contexte dans lequel il va être utilisé par un client. Cette indépendance à double sens est une propriété forte des services qui facilite le faible couplage. Chaque service est constitué de deux parties : sa description et son implémentation. Le fournisseur de service définit la syntaxe de l'interface, la sémantique des opérations et les comportements du service dans la description de service. Il peut également décrire certaines propriétés non fonctionnelles telles que la qualité du service, le coût, la localisation, le nombre d'appels autorisés à ce service et par ce service, *etc.* Ces propriétés sont généralement décrites en utilisant des langages fondés sur *XML* et des protocoles standards de l'Internet. Une architecture à service (*SOA* pour *Service-Oriented Architecture*) regroupe un ensemble de services et des mécanismes d'assemblage permettant le développement d'applications basées sur la réutilisation de services. Lors de la sélection d'un service, un contrat est mis en place, de façon tacite ou explicite, entre consommateur et fournisseur. La description du service peut être considérée comme le contrat de base. Cependant, une négociation peut avoir lieu entre l'utilisateur du service et le fournisseur du service. Les contrats de services permettent de réduire le couplage lié aux dépendances lors de la création d'une application par assemblage de services. Ceci améliore aussi le niveau d'abstraction des applications et facilite les évolutions.

Les caractéristiques des applications à services, telles que la substituabilité transparente, le faible couplage, la liaison retardée et la technologie d'implémentation neutre, sont particulièrement intéressantes dans de nouveaux domaines d'applications ayant

de fortes contraintes de dynamisme. L'orientation service est ainsi de plus en plus utilisée dans les applications pervasives, basées sur des réseaux de capteurs. Cette approche permet de gérer également l'arrivée et la disparition de capteurs et l'intégration d'environnements et de plates -formes hétérogènes. Toutefois, la composition de services pour former une application est clairement une tâche complexe, source de nombreuses erreurs potentielles pour plusieurs raisons. En premier lieu, il existe de nombreuses technologies pour décrire, éditer et composer des services. Différents protocoles et mécanismes peuvent être utilisés pour mettre en œuvre une architecture orientée services (*SOA*), tels que les *Web services*, *UPnP* [1], *DPWS* [2], ou *OSGi* (que nous détaillerons dans cette thèse). Deuxièmement, il est très difficile de vérifier la conformité d'une composition de services, incluant les conformités syntaxique et sémantique. Actuellement, les technologies capables de vérifier la conformité d'une composition de services sont seulement émergentes. D'autre part, les développeurs doivent faire face à plusieurs difficultés pour la mise à disposition de la description de service. Aujourd'hui, de nombreux langages servent à la réalisation de la description de services. Le développeur doit connaître les détails techniques des langages et leurs capacités avant de déterminer lequel utiliser. Dans certaines situations où de nombreux services sont disponibles, la stratégie de recherche, sélection et composition de services est difficile à spécifier pour les développeurs. Enfin, la qualité des compositions de services est une problématique critique. Certaines applications distribuées en temps réel désirent faire collaborer plusieurs services pour réaliser une transaction en temps opportun. Mais, le dynamisme des services soulève le défi de la gestion de la qualité des applications distribuées sous la forme de composition de services.

Une approche hybride réunissant les composants et les services a été récemment proposée : il s'agit des composant orientés service. Un composant orienté service est un composant conteneur d'aspects fonctionnels et non-fonctionnels dont les connecteurs suivent le style architectural de l'approche orientée service. Le code fonctionnel est exécuté à l'intérieur d'un conteneur qui va prendre en charge les propriétés non-fonctionnelles. Cette approche permet la simplification de l'écriture des applications orientées service. Le développeur n'a pas à prendre en compte la gestion de la disponibilité dynamique ainsi que le cycle de vie du composant. Ces tâches sont déléguées au code non-fonctionnel. Un modèle des plus utilisé aujourd'hui est iPOJO. Des modèles comme iPOJO facilitent grandement le développement d'application dynamique (ils peuvent être avantageusement complétés par des intergiciels tels que *RoSe* pour traiter les problématiques d'intégration).

Cependant, si ces modèles favorisent une grande dynamicité, leur administration autonomique reste difficile pour un ensemble de raisons détaillées dans cette thèse. Le but de cette thèse est de travailler à ce manque.

1.2 Objectifs de cette thèse

Cette thèse s'inscrit dans la mouvance de l'informatique autonomique et se focalise sur les applications développées avec le modèle à composants à services iPOJO. Ainsi, la problématique de cette thèse est la mise en place de boucle de contrôle autour d'applications reconfigurables dynamiquement. En effet, l'implantation de ces boucles de contrôle demande de nombreuses fonctionnalités, tel que :

1. *Universal Plug and Play*
2. *Devices Profile for Web Services*

- l'introspection de l'état de l'application,
- la connaissance du contexte d'exécution,
- l'accès à des données métiers manipulées par les applications,
- la manipulation de l'application pour l'adapter,
- la manipulation de l'environnement d'exécution de l'application (dans certains cas),
- la protection de l'application lors des reconfigurations,
- le contrôle de l'interruption de service.

Le but de ce travail doctoral est de permettre la réalisation de boucle de contrôle autour d'applications développées avec le modèle iPOJO. Précisément, notre objectif est d'étendre iPOJO et son écosystème d'outils afin de faciliter le développement d'applications autonomiques ; c'est-à-dire capable de s'autogérer. Il s'agit donc essentiellement de fournir aux développeurs d'applications et de gestionnaires autonomiques les moyens d'implanter ou de spécifier aisément les propriétés d'autogestion. Cette solution ne doit pas contraindre le développeur et lui donner une grande flexibilité. De plus l'application administrée ne doit pas nécessairement avoir été développé avec l'idée d'être administrée. Cette contrainte demande de fournir au développeur toutes les informations et leviers pour la gestion autonomique.

De façon plus précise, nous pensons qu'il est nécessaire de fournir aux développeurs de boucles autonomiques :

- une représentation **unifiée** de l'application et de son environnement
- des moyens de reconfigurations fonctionnelles et architecturales
- une représentation unifiée des données métiers
- des mécanismes de notifications afin de réagir aux différents changements

La gestion autonomique d'applications iPOJO demande non seulement d'augmenter iPOJO, mais également de fournir des outils annexes de représentation et d'accès. Le rôle de notre canevas est de fournir toutes les informations et leviers d'action nécessaires à la gestion autonomique. Cette représentation de la connaissance mélange des données techniques et métiers qu'il est nécessaire de lier et de corréler. Il convient néanmoins de noter que iPOJO possède un ensemble de propriétés techniques nécessaire à notre travail.

Bien que notre thèse se focalise sur iPOJO, les concepts abordés ont pour but d'être transposables à d'autres modèles.

1.3 Structure du document

Après cette introduction, le manuscrit de thèse est divisé en deux grandes parties : un état de l'art et la contribution. L'état de l'art comprend deux chapitres :

- le chapitre 2 présente l'informatique pervasive. Les concepts clé de cette nouvelle vision y sont présentés, définis, ainsi que des exemples d'applications. Les caractéristiques des environnements ubiquitaires et les besoins des applications ubiquitaires sont également présentés. Différents travaux existants, qui répondent à certaines des problématiques du développement des applications ubiquitaires, sont finalement exposés.
- le chapitre 3 traite de l'informatique autonomique. Il définit ce nouveau domaine et détaille ses objectifs majeurs. Il décrit également les concepts de base comme les boucles de contrôle et les architectures de référence développées dans ce domaine.

La contribution se divise en trois parties :

- le chapitre 4 expose la problématique, les objectifs et donne une vision d'ensemble de notre approche. Il s'agit de montrer les différents éléments ajoutés à iPOJO pour faciliter l'administration autonomique d'iPOJO.
- le chapitre 5 fournit une description détaillée de notre approche, introduite dans le chapitre précédent.
- le chapitre 6 détaille l'implémentation de notre approche, et distingue clairement les parties d'ors et déjà introduite dans iPOJO et celles qui feront l'objet d'évolutions à venir. Il illustre aussi l'utilisation de notre *framework* par un cas d'application et fournit un ensemble de mesures qualitatives et quantitatives.

Enfin le chapitre 7 synthétise les idées principales de notre proposition. Nous rappelons les points principaux de notre contribution et nous décrivons les perspectives envisagées pour de futurs travaux.

État de l'Art

Chapitre 2

Informatique Ubiquitaire *

Dans ce premier chapitre, nous présentons le domaine de l'informatique ubiquitaire, aussi appelée informatique pervasive. Les concepts clé de ce domaine relativement récent seront définis. Nous présenterons également des exemples d'applications et les domaines de recherche associés. Nous identifierons ensuite les caractéristiques des environnements ubiquitaires et celles des applications informatiques. Nous donnerons alors une introduction aux techniques qui permettent de satisfaire ces besoins, en se basant principalement sur une approche *middleware*. Différents travaux existants, qui répondent à certaines des problématiques du développement des applications ubiquitaires, seront exposés. Enfin, nous montrerons que ces solutions, bien que dédiées au domaine du pervasif, ne résolvent pas les défis soulevés par la nature même de ce nouveau paradigme.

∗. Ce chapitre a été écrit en collaboration avec Ozan Günalp

Sommaire

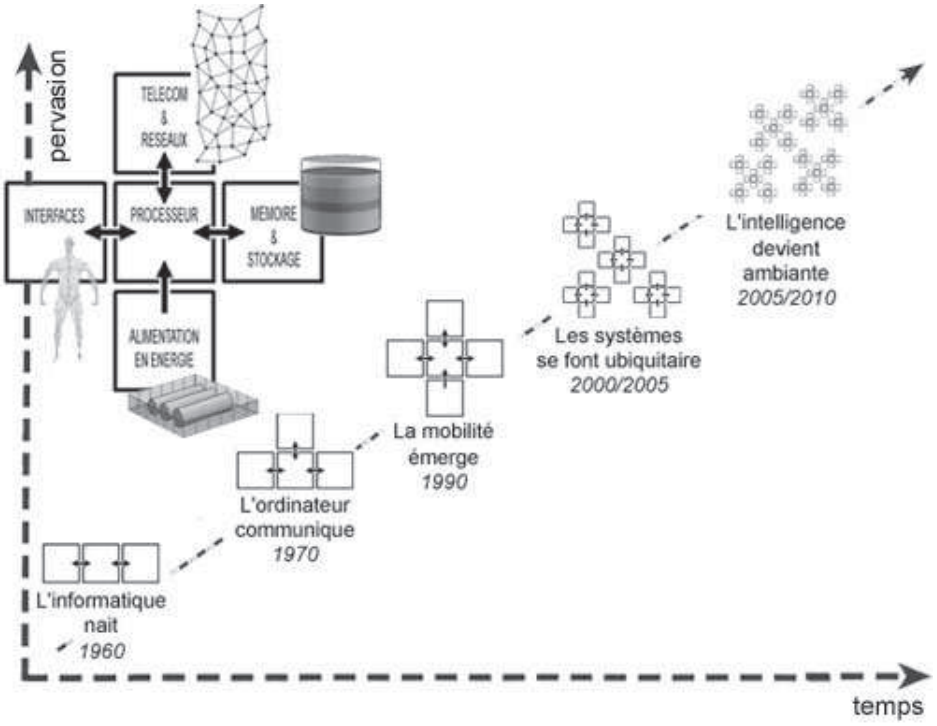

FIGURE 2.1 – Évolution des ordinateurs [1]

2.1 Évolution des environnements informatiques

Notre environnement contient un nombre sans cesse croissant de systèmes informatiques. Il est aujourd'hui parsemé d'appareils numériques qui se distinguent par leur forme, leur puissance, leur rôle. Nombre de ces systèmes sont conçus pour nous assister et nous accompagner dans nos activités quotidiennes et même dans nos interactions sociales. Ce rapport nouveau et très évolué avec les systèmes informatiques peut paraître surprenant si l'on considère qu'il y a à peine cinquante ans, les machines analogiques de l'âge industriel ont commencé à être remplacées par les machines numériques. Le fait est que les systèmes informatiques n'ont cessé d'évoluer depuis lors, grâce aux progrès technologiques permettant de concevoir des appareils toujours plus petits, plus puissants, plus communicants et consommant moins d'énergie. Le rapport humain à l'informatique a lui aussi franchi plusieurs paliers, chacun bouleversant nos manières d'interagir avec les ordinateurs. L'analyse de l'évolution des systèmes qui va suivre s'inspire d'une analyse présentée dans [Weiser 1996] qui débute par ces quelques lignes d'introduction :

> « *The important waves of technological change are those that fundamentally alter the place of technology in our lives. What matters is not technology itself, but its relationship to us.* »

La figure 2.1 ci-dessus illustre les changements technologiques fondamentaux liés à l'informatique. Elle met en évidence quelques étapes cruciales qui ont marqué son évolution. En particulier, les notions de distribution, de mobilité et de multiplication des systèmes informatiques apparaissent clairement comme des marqueurs majeurs.

1. Adapté de [Waldner 2007]

Nous examinons plus précisément dans les sections suivantes ces différentes étapes.

Au début des années 1940, l'informatique centralisée était prédominante et apparaissait comme le seul moyen de construire des systèmes numériques. Ces derniers prenaient la forme d'ordinateurs isolés, pouvant atteindre la taille d'une pièce. Ils étaient composés de processeurs et de mémoire, et étaient administrés en permanence par des experts. Ces experts étaient à la fois les administrateurs, les développeurs et les utilisateurs du matériel et des logiciels. Ces *mainframes*, puisque c'est le nom qu'on donne à ces systèmes centralisés, disposaient de ressources limitées et devaient être partagés entre plusieurs utilisateurs.

Avec les progrès de l'électronique, la signification du terme *mainframe* a évolué. Il est aujourd'hui attaché aux ordinateurs « haut de gamme » avec une puissance de calcul considérable, supportant des applications utilisées simultanément par des milliers de personnes. On considère donc aujourd'hui qu'un système informatique aux ressources limitées et partagées par plusieurs utilisateurs est un *mainframe*, tel que défini dans [Weiser 1996] :

> « *If lots of people share a computer, it is mainframe computing.* »

Dans cette même analyse, Mark Weiser introduit ensuite l'émergence de l'informatique personnelle en ces termes :

> « *In 1984 the number of people using personal computers surpassed the number of people using shared computers. The personal computing relationship is personal, even intimate. You have your computer, it contains your stuff, and you interact directly and deeply with it.* »

L'informatique personnelle, symbolisée par les *PC* (*personal computer*), est devenue le mode d'interaction privilégié entre le monde numérique et l'utilisateur. Encore aujourd'hui, la majorité des systèmes et applications sont basés sur ce mode d'interaction qui place l'utilisateur dans le rôle central : celui d'utiliser de façon proactive un service. Le système étant personnel, la plupart des utilisateurs sont aussi les administrateurs, installant et configurant les logiciels et périphériques conçus par des tiers.

Tandis que les *PC* et *mainframes* disposent de ressources locales limitées, les infrastructures réseaux tels que l'Internet permettent aux ordinateurs d'accéder à des services distants, liant ainsi les systèmes d'information personnels, professionnels et gouvernementaux. Ceci est résumé de la façon suivante par Weiser :

> « *Interestingly, the Internet brings together elements of the mainframe era and the PC era. It is client-server computing on a massive scale, with web clients the PCs and web servers the mainframes.* »

L'accès généralisé à l'Internet permet aujourd'hui la conception d'applications plus complexes, avec plus de valeur ajoutée pour les utilisateurs. Ce concept soulève cependant de nombreux défis pour les concepteurs et les administrateurs de tels systèmes, tels que la prise en compte de la sécurité du système, les communications à distance ou l'intégration d'applications hétérogènes. La tendance initiale qui était de masquer tous ces problèmes aux développeurs est aujourd'hui remise en cause. Nous y reviendrons dans cette thèse.

Au début des années 1990, l'émergence des ordinateurs portables et des technologies sans fil a permis la naissance de l'informatique mobile. L'utilisateur peut ainsi

transporter son ordinateur et toujours accéder aux applications distantes dans un contexte de mobilité. L'intérêt suscité par les *smartphones* et les tablettes tactiles a défini un nouveau type d'ordinateur personnel : puissant, mobile, connecté, communiquant sans fil via une nouvelle génération de réseaux cellulaires (*3G*, *LTE*, *4G* [2]). Ces appareils permettent le développement de services nouveaux. Par exemple, les *smartphones* peuvent être utilisés pour déterminer la position de leur possesseur grâce un système de géolocalisation intégré, ou par triangulation des signaux du réseau cellulaire. L'intégration d'applications mobiles avec les systèmes distribués existants a soulevé de nouveaux problèmes tels que la sensibilité à la position géographique de l'utilisateur, l'économie d'énergie et la gestion efficace et adaptative des ressources [Satyanarayanan 2001].

L'informatique ubiquitaire, aussi appelée informatique pervasive, est la dernière évolution illustrée par la figure 2.1. C'est en fait une vision fusionnant les ordinateurs de nouvelle génération avec l'environnement du monde réel. L'informatique ubiquitaire conçoit un mode d'interaction radicalement différent entre ordinateurs et utilisateurs, au-delà de l'informatique mobile, où les ordinateurs se mélangent aux objets du quotidien et où les utilisateurs utilisent des services et accèdent à des données sans même y penser.

2.2 Informatique ubiquitaire

Dans cette section, nous nous concentrons sur la notion d'informatique ubiquitaire. Nous donnons certaines définitions possibles pour ce nouveau domaine. Nous examinons également la notion de contexte, intimement lié à l'informatique ubiquitaire. Enfin, nous examinons les impacts sociétaux et les travaux de recherche actuels.

2.2.1 Définition

L'informatique ubiquitaire, telle qu'elle a été présentée dans l'article fondateur [Weiser 1991], décrit la prochaine génération d'environnements informatiques qui place l'être humain au centre de l'attention, plutôt que les machines. Mark Weiser et ses collègues, au Xerox PARC [3], envisage un monde rempli de petits dispositifs informatisés intégrés dans les objets quotidiens, et des infrastructures communiquant avec ces appareils, dans le but d'assister les utilisateurs, de la manière la plus transparente et naturelle possible. Les utilisateurs peuvent se focaliser sur leurs tâches, sans s'inquiéter de la manière d'utiliser le système informatique. Ces visions ont inspiré des nombreux chercheurs, ce qui a amené à l'apparition de différents termes tels que *calm computing*, *disappearing computer*, *everyware*, *Internet of things*, *ambient intelligence*, *things that think*. Bien qu'il y ait eu une certaine bataille de concepts autours de ces termes, autant dans la communauté scientifique que dans les médias [Ronzani 2009], tous désignent une fusion des environnements informatiques dans le monde réel.

Dans la suite de cette thèse, ces termes seront traités de façon équivalente ; le terme **informatique ubiquitaire** se réfère au paradigme dans sa globalité. La présence de ces nombreux concepts très liés ne permet pas de donner une définition simple et unique de l'informatique ubiquitaire, mais elle permet de saisir toute la portée de cette vision. On trouve ainsi plusieurs définitions dans la littérature :

2. Générations successives de normes de téléphonies mobiles.
3. *Palo Alto Research Center*, centre de recherche à l'origine des interfaces graphiques.

– « *The most profound technologies are those that disappear, [...] they weave themselves into the fabric of everyday life until they are indistinguishable from it.* »

[Weiser 1991]

– « *We characterized a pervasive computing environment as one saturated with computing and communication capability, yet so gracefully integrated with users that it becomes a technology that disappears* »

[Satyanarayanan 2001]

– « *These devices are intended to react to their environment and coordinate with each other and network services. Furthermore, many devices will be mobile and are expected to dynamically discover other devices at a given location and continue to function even if they are disconnected.* »

[Grimm 2004]

– « *One could describe 'ubiquitous computing' as the prospect of connecting the remaining things in the world to the Internet, in order to provide information on anything, anytime, anywhere. [...] the term 'ubiquitous computing' signifies the omnipresence of tiny, wirelessly interconnected computers that are embedded almost invisibly into just about any kind of everyday object.* »

[Mattern 2001]

– « *The basic idea of this concept [Internet of Things] is the pervasive presence around us of a variety of things or objects — such as Radio-Frequency IDentification (RFID) tags, sensors, actuators, mobile phones, etc. — which, through unique addressing schemes, are able to interact with each other and cooperate with their neighbors to reach common goals.* »

[Atzori 2010]

2.2.2 Propriétés et notion de contexte

Les définitions précédentes possèdent un ensemble de points communs. En particulier, il apparaît que l'informatique ubiquitaire peut être caractérisée par un certain nombre de propriétés essentielles :

1. L'informatique ubiquitaire est invisible grâce à une interaction homme machine discrète, naturelle et transparente. La multiplicité des ressources de calcul et de stockage ne permet pas une interaction directe entre humains et dispositifs électroniques. Au contraire, les interactions doivent rester invisibles la plupart du temps.
2. L'informatique ubiquitaire est intrinsèquement répartie entre des dispositifs mobiles et fixes, ainsi que des services réseaux. Ces dispositifs sont habituellement cachés à l'utilisateur et en constante interaction avec leur environnement

physique et entre eux. Comme pour le point précédent, ces interactions restent majoritairement ignorées par les êtres humains.

3. L'informatique ubiquitaire est sensible au contexte afin de pouvoir optimiser son fonctionnement dans l'environnement actuel. Les services rendus par l'informatique sont évolutifs et fondamentalement dépendants de leur contexte d'exécution au sens large.

Ces propriétés fondamentales permettent de clarifier la vision de l'informatique ubiquitaire. Elles dévoilent certains de ses aspects les plus cruciaux, tels que l'invisibilité, la distribution, et la mobilité des dispositifs électroniques, ainsi que la sensibilité au contexte des services mis en œuvre au travers des dispositifs.

Revenons sur la sensibilité au contexte qui est une propriété primordiale des systèmes informatiques ubiquitaires. De façon générale, les systèmes sensibles au contexte sont ceux qui sont « conscients » de leur environnement d'exécution et qui sont capables de réagir pour s'adapter aux changements survenus dans cet environnement [Baldauf 2007]. De tels systèmes ont commencé à apparaître avec l'émergence de l'informatique mobile. Dans ce cas, le contexte était souvent réduit à la localisation géographique de l'appareil mobile. Ces systèmes étaient capables de déterminer avec précision la position de l'utilisateur et d'adapter certains contenus (cartes routières par exemple) au fil de ses déplacements. Bien que la position géographique soit encore une des principales informations de contexte, elle n'est pas nécessairement significative pour tous les types possibles d'applications ubiquitaires. Depuis les prémisses de l'informatique mobile, la définition du contexte et des informations qu'il contient a évolué vers un modèle plus élaboré. Cela est dû au fait que le contexte est un élément fondamental de toute application ubiquitaire.

La sensibilité au contexte a ainsi été mentionnée et définie dès les débuts de l'informatique ubiquitaire par Bill Schilit, Norman Adams et Roy Want [Schilit 1994]. Précisément, ces auteurs ont défini le contexte de la façon suivante :

> « *The location of use, the collection of nearby people, hosts, and accessible devices, as well as to changes to such things over time.* »

Anind Dey et Gregory Abowd [Dey 1999] proposèrent plus tard une nouvelle définition plus précise qui fait aujourd'hui figure de référence :

> « *Context is any information that can be used to characterize the situation of an entity. An entity is a person, place, or object that is considered relevant to the interaction between a user and an application, including the user and applications themselves.* »

À partir des diverses définitions générales de ce qu'est le contexte, une classification a été proposée pour les différents types de contexte [Schilit 1994] :

1. L'**environnement virtuel** inclut toutes les propriétés qui décrivent le système informatique, telles que les ressources utilisées, les périphériques et ressources disponibles, les connexions aux réseaux, la bande passante, *etc.*
2. L'**environnement utilisateur** comprend les informations sur les personnes utilisant le système : position géographique, besoins, activités sociales, personnes avoisinantes, *etc.*
3. L'**environnement physique** décrit quant à lui les attributs physiques de l'environnement dans lequel est placé l'utilisateur, incluant par exemple la température,

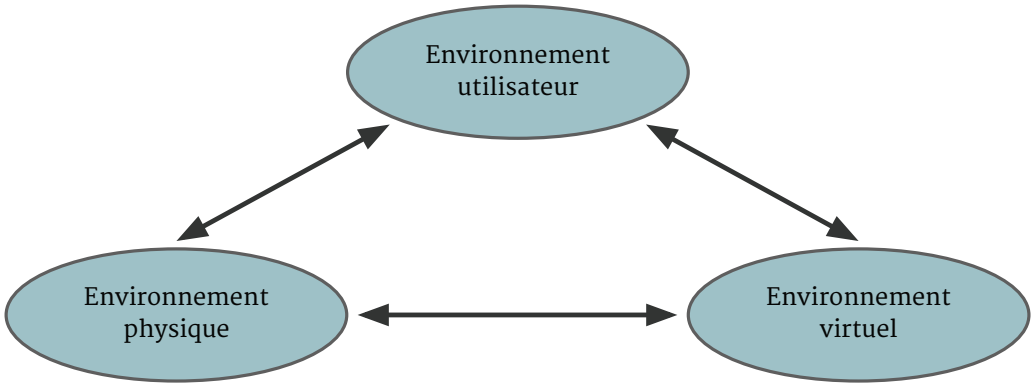

FIGURE 2.2 – Contexte d'un environnement ubiquitaire

la luminosité, le bruit ambiant, *etc.*

Cette classification des différents types de contexte dévoile une caractéristique importante de la sensibilité au contexte dans l'informatique ubiquitaire : cela montre à quel point ce paradigme rassemble et mêle l'environnement virtuel et le monde physique, plaçant l'utilisateur au centre de l'attention. Dans les ères pré-ubiquitaires, l'informatique n'impliquait que des entités virtuelles, conceptualisées par les développeurs de logiciels. Le contexte de fonctionnement de ces systèmes était principalement réduit aux ressources matérielles de l'ordinateur, très souvent abstraites par le système d'exploitation. Néanmoins, l'informatique ubiquitaire n'implique pas seulement une abondance d'appareils informatiques communicants, mais aussi un nombre potentiellement important d'utilisateurs, et une grande diversité et un dynamisme dans les environnements physiques. Les informations relevant de l'environnement des utilisateurs et de l'environnement physique sont aussi importantes que celles de l'environnement virtuel. Un environnement ubiquitaire se compose ainsi de ces trois environnements, comme le montre la figure 2.2.

Les environnements virtuels, utilisateurs et physiques sont en interaction constante entre eux. Par exemple, le fait qu'un utilisateur se déplace provoque des changements dans l'environnement physique, ce qui peut entraîner des changements dans l'environnement virtuel, via des capteurs. Ou encore, un changement dans l'environnement physique, comme l'augmentation de la température intérieure, peut affecter le confort des utilisateurs, mais aussi causer des dommages sur le matériel informatique. En étant sensible aux différents environnements, un système ubiquitaire coordonne les interactions entre ces environnements afin de fournir des services et des informations aux utilisateurs.

La sensibilité au contexte est un concept central pour les systèmes ubiquitaires. Elle est nécessaire pour créer des dispositifs discrets et des applications non-intrusives dans le quotidien de l'utilisateur. Un point important à signaler est que la plupart des travaux de recherche sur la sensibilité au contexte sont appliqués au domaine des environnements ubiquitaires. Ce concept est l'un des plus abondamment traités dans la littérature concernant l'informatique ubiquitaire. Néanmoins le concept d'informatique ubiquitaire ne doit pas être réduit au traitement de la sensibilité au contexte car bien d'autres aspects et besoins doivent être pris en compte. Nous verrons que ces aspects sont très techniques, et ont un impact sur la conception des applications.

2.2.3 Dimensions sociétales et implications

Comme nous l'avons vu précédemment, l'informatique ubiquitaire dépeint un monde empli de dispositifs informatiques et de logiciels communicants. Des êtres humains parcourent ce monde et utilisent ces appareils et applications dans leurs activités quotidiennes. L'interaction avec cet environnement doit être naturelle et transparente pour l'utilisateur, de sorte qu'il puisse se focaliser sur une tâche, un objectif, pendant que le système l'assiste, sans l'envahir. Cette vision a donc des implications considérables non seulement pour nos modes d'interactions avec les machines, mais aussi pour notre vie quotidienne, où monde ubiquitaire et monde physique sont entremêlés. Plusieurs utilisations de cette vision ont d'ores et déjà été définies et (partiellement) mises en place. Nous examinons ici trois cas de figures particulièrement intéressants :

- les *smart spaces*,
- le *M2M*,
- les applications *smartlife*.

Smart spaces

La vision de l'informatique ubiquitaire est donc une idée qui a pu paraître extrêmement futuriste à une époque, mais qui semble réalisable, voire inéluctable de nos jours. Les premiers travaux dans ce domaine se focalisaient principalement sur l'intégration d'ordinateurs miniaturisés dans l'environnement quotidien, et l'exploration de nouveaux types d'interactions homme-machine. Ces travaux ont été les pionniers dans la recherche d'applications de l'informatique ubiquitaire. Ils ont permis de développer des scénarios d'applications s'exécutant dans des environnements équipés de capteurs, actionneurs et dispositifs mobiles, nommés espaces intelligents ou *smart spaces* [Abowd 1998].

Des bureaux sensibles à la position des employés, et redirigeant les appels téléphoniques entrants, des salles de réunion sensibles au contexte qui capturent les activités humaines, des salles de classe équipées de tableaux intelligents et de surfaces interactives, des espaces ouverts augmentés numériquement, ou encore des ordinateurs vestimentaires sont autant d'exemples d'application de l'informatique ubiquitaire. La figure 2.3 illustre un exemple d'habitation aménagée en *smart space*. Bien qu'elles fussent principalement des prototypes hautement expérimentaux, ces applications ont montré quelques-uns des domaines d'applications fondamentaux de l'informatique ubiquitaire.

Systèmes *M2M*

Les systèmes *M2M* (*Machine-to-Machine*) sont basés sur la communication entre machines sans intervention humaine nécessaire. Dans une application *M2M*, les machines communiquent entre elles en utilisant les services fournis par chacune d'entre elles et en échangeant des données. Ces types de systèmes sont largement utilisés dans différents secteurs industriels tels que la surveillance de l'environnement, la logistique, des infrastructures de services publics, *etc.*

L'idée derrière les systèmes *M2M* provient du monde de la télémétrie, littéralement « mesure à distance ». Le concept de télémétrie consiste à utiliser des capteurs et des ordinateurs distants pour collecter des données et les envoyer vers un emplacement central pour les analyser ultérieurement. Les types de machines interagissant

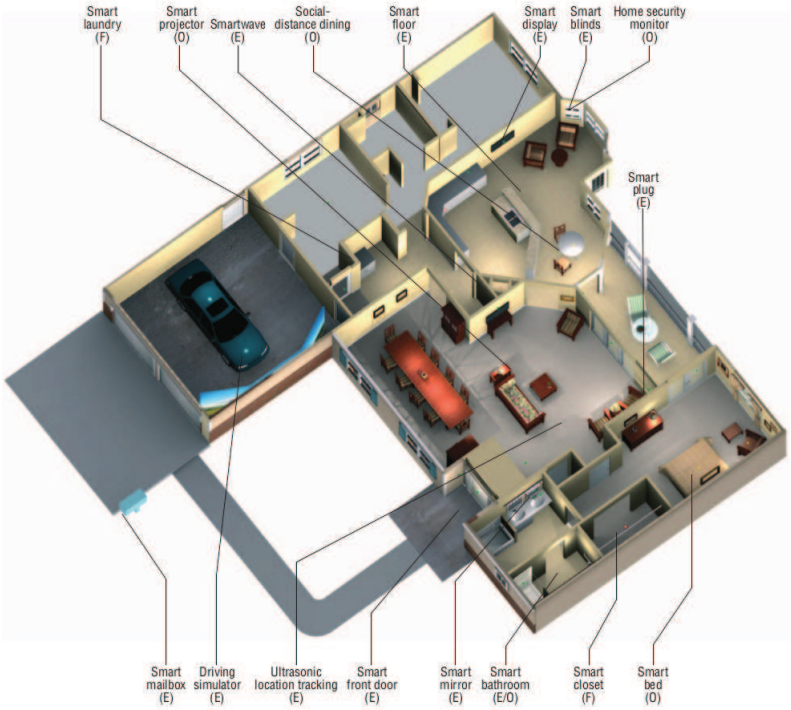

FIGURE 2.3 – Exemple d'environnement de type *smart space* [Helal 2005]

dans un tel système sont très variables : on peut y trouver des capteurs à très faible consommation, des puissants serveurs de collecte ainsi que des nœuds de calculs intensifs.

Les systèmes *M2M* modernes apportent des améliorations considérables par rapport aux concepts de télémétrie existants. Les technologies de capteurs sans fil offrent une connectivité et une sensibilité améliorées. Les systèmes d'information modernes assistés par des serveurs interconnectés et des grappes de serveurs permettent un traitement rapide de très grandes quantités de données.

Les systèmes M2M impliquent le déploiement à grande échelle de machines et d'applications. Des réseaux de capteurs connectent des nœuds de capteurs fixes ou mobiles qui mesurent différents paramètres de l'environnement et transmettent ces relevés aux systèmes d'information centralisés, équipés de bases de données et/ou de logiciels d'analyse.

Durant leur remontée, les données brutes produites par les capteurs passent éventuellement par une série de transformations et de filtrage, appelée médiation de données. Les données traitées, transférées aux serveurs, seront stockées dans des bases de données pour permettre des requêtes ultérieures. Elles peuvent ensuite être analysées, soit par des humains ou des ordinateurs, afin de créer des rapports et de prendre des décisions.

Les systèmes *M2M* ont de nombreux domaines d'application, comprenant par exemple la surveillance de la qualité de l'eau pour des services publics, la gestion des stocks pour des distributeurs (cf. figure2.4) et des fabricants, ou même la prévision du microclimat pour des régions agricoles.

Les applications *M2M* collectent et traitent de grandes quantités de données potentiellement hétérogènes. Dans le contexte de l'informatique ubiquitaire, elles illustrent un cas particulier d'interaction entre les environnements physiques et virtuels. Créer

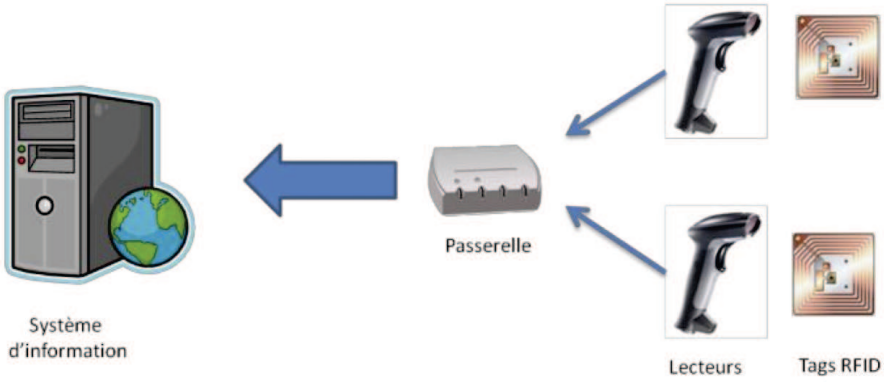

FIGURE 2.4 – Exemple d'application *M2M* [Escoffier 2008]

des infrastructures logicielles et matérielles pour faire face à cette densité de données, en particulier dans des réseaux de capteurs sans fil, peut se révéler extrêmement complexe.

Smartlife : le *M2M* au service des utilisateurs

Durant les années 1990, les systèmes d'information automatisés ont été adoptés par de nombreuses organisations du monde moderne : banques, distributeurs, fabricants, presse, fournisseurs de services publics, organismes gouvernementaux, *etc.* Toutes ces entités ont donc commencé à utiliser ces systèmes pour simplifier la gestion de l'information et optimiser leur fonctionnement. L'information, produite de façon interne comme résultat de processus métiers, ou collectée depuis des sources externes, stockée dans des bases de données et analysée, permet de prendre des décisions vitales par rapport à la gestion des entreprises. En conséquence, l'information, et les systèmes qui les gèrent, deviennent les atouts stratégiques et/ou commerciaux plus importants.

Une décennie plus tard, la diffusion de l'accès à l'Internet a permis à ces organisations d'utiliser le Web afin d'entrer directement en relation avec leurs partenaires (clients, fournisseurs, *etc.*). Les applications Web ont permis à ces personnes d'accéder aux informations stockées dans les systèmes d'informations de ces organisations. Réciproquement, pour les organisations, cela signifiait un nouveau contexte pour collecter des informations et améliorer leurs services, leurs rentabilités et/ou leurs bénéfices. De nouveaux types de commerces « *Internet-only* » ont rapidement émergé, utilisant les services basés et diffusés sur l'Internet, et exploitant les habitudes de leurs utilisateurs, des statistiques d'utilisation, etc. La diffusion de l'utilisation des services basés sur Internet a provoqué une explosion des volumes d'information stockée et analysée dans les systèmes d'information.

En parallèle, les systèmes M2M ont permis aux organisations de diffuser l'information collectée dans les environnements physiques dans le but de surveiller et d'optimiser les processus métiers. Avec l'émergence d'*IPv6* [4], les capteurs et actionneurs intégrés dans les objets physiques sont connectés par les mêmes protocoles qui lient l'Internet. Les systèmes *M2M* et les services basés sur Internet sont devenus des outils pour collecter des informations des environnements et des utilisateurs et communiquer avec eux.

4. *Internet Protocol, version 6*

Un terme qui est récemment apparu dans le domaine de l'informatique ubiquitaire est *smartlife*. Il définit un concept selon lequel les organisations joignent leurs clients et utilisateurs dans leur vie quotidienne en leur fournissant des services utiles et des informations à tout moment, n'importe où. Il comprend différents domaines verticaux de l'informatique ubiquitaire tels que le domaine de la maison, du transport, de la santé, des services publics, *etc.* À partir ces domaines existants, le concept de *smartlife* propose un nouveau modèle d'affaire qui permet aux fournisseurs d'intégrer leurs services dans ces environnements quotidiens avec une approche holistique. La valeur ajoutée de ces nouveaux services est qu'ils profitent de la naturelle sensibilité au contexte de l'environnement ubiquitaire.

Les applications et services intégrés dans un environnement *smartlife* utilisent une connaissance étendue sur les utilisateurs et leurs environnements physiques, et la capacité de réaliser des actions directement sur l'environnement par l'intermédiaire de dispositifs embarqués. L'utilisation des informations de contexte de façon horizontale, entre domaines, permet à une application, par exemple, d'utiliser des informations sur la liste de courses d'un client dans le contexte de la domotique, afin de connaître la fraîcheur des aliments dans le réfrigérateur ; et dans le contexte de la santé pour informer le client sur ses mauvaises habitudes alimentaires.

L'étape suivante pour la poursuite des recherches sur l'informatique ubiquitaire sera la réalisation du concept de *smartlife*. Cela nécessitera d'utiliser les contributions issues des domaines des *smart spaces* et des systèmes *M2M*. Les travaux sur les *smart spaces* vont fournir une interaction plus naturelle avec les utilisateurs et leurs environnements, tandis que ceux concernant les systèmes *M2M* vont apporter des techniques visant à améliorer les interactions entre services métiers. Il y a un besoin évident de développer des infrastructures logicielles et matérielles qui hébergeront et exécuteront ces types d'applications ubiquitaires avec des exigences spécifiques.

2.2.4 Domaines de recherche associés

La recherche sur l'informatique ubiquitaire requiert des efforts collectifs dans des domaines très divers, concernant notamment la microélectronique, l'électronique, les télécommunications, les réseaux, et le logiciel. Dans cette section, nous présentons rapidement quelques enjeux majeurs dans ces domaines.

Électronique : miniaturisation et gestion de l'énergie

Les progrès dans la microélectronique et la conception de circuits imprimés permettent de produire des dispositifs embarqués plus puissants et plus petits qui peuvent mieux s'intégrer dans des environnements physiques. Avec l'utilisation de matériaux et techniques de pointe lors de la production, il est possible de fabriquer des dispositifs à faible coût. Ce faible coût et les technologies de communication sans fil permettent l'utilisation nomade et à grande échelle de dispositifs auto-alimentés. Toutefois, le progrès de ces aspects est freiné par un manque de solutions de stockage énergétique adaptées et efficaces. Les sources d'énergie internes (piles et batteries) constituent la limitation principale des dispositifs actuels à cause de leur taille, poids et durée de vie.

De nombreux efforts sur la création de dispositifs intelligents se focalisent sur l'augmentation de l'autonomie énergétique. Avec des solutions matérielles et logicielles reconfigurables, les dispositifs embarqués peuvent réduire leur consommation d'énergie en optimisant leur fonctionnement, par exemple moins de communication

sans fil signifie moins d'utilisation d'énergie [Druilhe 2013]. De plus, des solutions de capture d'énergie telles que des cellules photovoltaïques, des modules piézoélectriques ou des antennes radioélectriques de récupération d'énergie permettent aux dispositifs de transformer l'énergie environnante en énergie électrique.

En plus de l'optimisation de la consommation énergétique, il y a d'autres défis pour la fabrication de dispositifs intelligents afin qu'ils s'intègrent et interagissent mieux avec les environnements physiques et les utilisateurs. L'utilisation de technologies innovantes comme les nano-matériaux et des matériaux organiques ouvre la voie à la construction de dispositifs petits et précis qui peuvent même être embarqués dans le corps humain. À la lumière de ces progrès, dans un avenir proche, il est prévisible de voir des dispositifs de taille négligeable disposant d'une plus grande autonomie énergétique.

Communication : solutions sans fil et qualité de service

La plupart des dispositifs intelligents récents tirent avantage de la communication sans fil. Les technologies de communication sans fil assurent la connectivité des appareils intelligents avec le monde extérieur, tout en les libérant des contraintes de câblage, et permettant leur mobilité et déploiement à grande échelle. Des solutions de réseaux sans fil sont nécessaires en particulier dans les réseaux de capteurs où des nœuds de capteurs sont répartis dans l'espace et interconnectés par la liaison hertzienne. Des données et des mesures sont transmises d'un nœud à un autre jusqu'à ce qu'elles atteignent une station de base, où elles peuvent être exploitées par les applications et présentées aux utilisateurs. De cette façon, les nœuds de capteurs peuvent être déployés dans des périmètres plus étendus tout en restant connectés à une station de base. Les technologies sans fil soulèvent de nouveaux problèmes en termes de qualité de communication. Les efforts de la communauté de recherche se focalisent sur le développement de solutions matérielles et réseaux pouvant fournir une haute qualité de communication tout en optimisant le fonctionnement et en réduisant la consommation énergétique.

Des solutions réseaux innovantes offrent des topologies adaptatives qui s'auto-optimisent et s'auto-réparent en cas de mauvaises connexions entre les nœuds ou en cas de défaillance d'un ou plusieurs nœuds, basée sur le principe d'auto-stabilisation [Dijkstra 1974]. D'autre part, de nouveaux protocoles de communication au niveau applicatif émergent et s'adaptent à des réseaux à faible bande passante et/ou avec des taux de pertes élevés.

Logiciel : redéfinition des outils de génie logiciel

Avec les progrès de la microélectronique et de la communication sans fil, il devient possible d'exploiter les capacités des dispositifs de communication sans fil intelligents pour développer des applications ubiquitaires. Cependant, des défis majeurs restent à résoudre. Parmi ceux-ci, il en est un qui concerne particulièrement les fournisseurs de services ubiquitaires : la gestion de la haute complexité de développement, d'intégration, de déploiement et d'administration des systèmes et des applications ubiquitaires. La recherche sur le génie logiciel s'efforce d'obtenir des solutions qui facilitent la mise en œuvre et l'exécution d'applications ubiquitaires.

De nombreux travaux se concentrent sur la notion de *middleware*. Comme nous l'avons vu, les *middlewares* se placent entre les systèmes d'exploitation et les applications pour fournir des abstractions et des fonctionnalités de haut niveau, en cachant

diverses complexités du développement d'applications. Ils ont évolué à partir de technologies simples cachant les détails de communication en réseaux aux applications réparties, à partir des blocs logiciels importants qui cachent et gèrent des aspects divers tels que l'hétérogénéité, la mobilité, le traitement de donnés et le passage à l'échelle. Tout au long de leur évolution, les *middlewares* ont adopté des principes du génie logiciel tels que la séparation de préoccupations et la modularité afin de gérer la complexité croissante des applications et faciliter leur programmation. Par ailleurs, des travaux récents sur les middlewares se focalisent sur la surveillance et la gestion d'applications au cours de leur exécution. Nous y reviendrons en détail dans la suite de cette thèse

En complément, des efforts importants sont consentis au niveau des outils supportant l'utilisation des *middlewares*. Ces outils, souvent fondés sur l'automatisation, fournissent des moyens efficaces pour améliorer la productivité et la qualité logicielles. Les outils d'automatisation facilite notamment les étapes de profilage, compilation et *packaging*, et aide à gérer la distribution d'artefacts logiciels. Des outils de gestion de dépendances comme *Apache Maven* [5] servent à gérer des projets complexes avec des dépendances multiples, promouvant ainsi la modularité dans le développement de logiciels.

Les environnements de développement intégrés (*IDE, Integrated Development Environments*) sont des logiciels qui comprennent différentes catégories d'outils tels que des éditeurs, des outils de construction pour la génération d'exécutables, et des débogueurs pour les tests. Par exemple *Eclipse* [6] fournit un environnement extensible qui peut intégrer différents outils pour la modélisation, la programmation, la gestion de dépendances, la construction, la gestion de versions, les tests, *etc.* Grâce à son système de *plugins*, *Eclipse* permet d'intégrer de nouveaux outils et fournit un écosystème pour construire des *IDE* spécifiques à un domaine particulier. Par exemple, *Xtext* [7] permet de créer des langages spécifiques au domaine et des éditeurs de code correspondant basés sur *Eclipse*.

Les outils de développement intégrés facilitent les efforts des architectes et développeurs d'applications pour créer des applications testables et maintenables. En plus de cela, la plupart des outils existants fournissent une base solide pour étendre ces capacités à des domaines spécifiques, dans lesquels il est particulièrement difficile de créer efficacement des applications. Nous verrons cependant que les outils et *middleware existants* ne répondent que partiellement aux défis posés par le domaine de l'informatique ubiquitaire.

2.3 Caractéristiques des environnements ubiquitaires

D'après les différentes définitions de l'informatique ubiquitaire émergent un ensemble de propriétés et de caractéristiques qui, ensemble, définissent ce qu'est un environnement ubiquitaire. Même si différentes visions peuvent se confronter, certaines propriétés essentielles sont évoquées dans la grande majorité des ouvrages traitant du sujet. Cette section énumère les principales caractéristiques permettant de définir avec plus de précision les environnements ubiquitaires et leurs portées.

5. http://maven.apache.org
6. http://www.eclipse.org
7. http://www.eclipse.org/Xtext

2.3.1 Distribution à large échelle

Les services et informations fournis aux utilisateurs dans les environnements ubiquitaires proviennent souvent de divers fournisseurs et sources distants. Certains de ces fournisseurs de services et d'information sont constitués de dispositifs embarqués qui sont dispersés (cachés ou exposés) dans des environnements physiques, tels que des capteurs environnementaux, téléphones portables, appareils électroniques ou actionneurs. Ces dispositifs sont en interaction constante avec les environnements physique et utilisateur – ils sont capables de détecter leur environnement et d'agir sur lui, ils intègrent des interfaces utilisateur pour permettre à ces derniers d'interagir et d'accéder aux informations présentées. Les ressources fournies par ces dispositifs sont accessibles à l'aide des protocoles de communication différents, basés sur des technologies filaires ou non. À cause de limitations en termes de puissance de calcul et de capacité de stockage, les applications qui utilisent les capacités de ces dispositifs et les coordonnent ne s'exécutent pas nécessairement directement dans ces dispositifs. Cette particularité rend les environnements ubiquitaires inévitablement distribués. En général, les ressources disponibles pour l'environnement ubiquitaire ne sont pas limitées à des dispositifs présents dans un environnement physique. Par exemple, dans des applications *M2M*, les mesures collectées par des capteurs sont envoyées — par Internet — à des serveurs distants pour stocker et analyser les données. Ces systèmes d'informations, en général hautement performants en termes de puissance de calcul et espace de stockage, sont physiquement trop éloignés pour être qualifiés d'ubiquitaires. Cependant, leurs ressources peuvent être utilisées dans l'informatique ubiquitaire pour fournir des services et des informations à valeur ajoutée, qui ne serait autrement pas disponibles.

Le nombre croissant de dispositifs de communication et de serveurs apporte des besoins de déploiement à grande échelle, d'installation et de maintenance de composants logiciels et matériels. Les infrastructures informatiques (matérielles et logicielles) dans les environnements ubiquitaires doivent avoir des architectures évolutives pour faire face à la forte densité de dispositifs, services et données. Les concepteurs et développeurs de logiciels doivent prendre en considération la nature distribuée des applications, des services et des dispositifs distants.

2.3.2 Hétérogénéité

Chaque année apparaissent de nombreux nouveaux dispositifs et fournisseurs de services. Les protocoles de communication sont aussi très diversifiés car chaque type de dispositif a des caractéristiques qui lui sont propres et des besoins différents par rapport à la nature de son utilisation. De nombreux groupes de travail focalisent leurs efforts sur la standardisation de protocoles qui ont un usage commun. Néanmoins, les industries fabricant ces dispositifs préfèrent en général utiliser des protocoles propriétaires dédiés au lieu de se tenir à des standards souvent plus généralistes. Ils gardent ainsi leur catalogue de produits privé et fermé afin de maîtriser l'ensemble de leur écosystème et de continuer à vendre des produits.

Dans les environnements ubiquitaires, accéder et utiliser des ressources sur des dispositifs hétérogènes n'est qu'un aspect du problème. Il est également important d'administrer et de configurer les dispositifs présents dans un environnement. Différents fabricants sont susceptibles d'utiliser différents modèles et protocoles de gestion de dispositifs pour représenter l'information sur le dispositif lui-même et réaliser des opérations de maintenance. Une tendance similaire peut être observée dans les

technologies Web pour les modèles de services. Accéder aux services sur Internet et exposer des fonctionnalités comme des services accessibles à distance requiert d'intégrer différents modèles de services et protocoles de communication. En conséquence, les concepteurs d'applications ont un besoin croissant d'intégrer une grande hétérogénéité de protocoles de communication, types de dispositifs et services dans leurs applications, ce qui augmente le niveau de complexité des applications.

2.3.3 Ouverture et autorité plurielle

Dans un environnement ubiquitaire, les ressources informatiques, soit sous forme de dispositifs ou de services, appartiennent généralement à différents acteurs tels que des fabricants de dispositifs, des vendeurs ou des fournisseurs de services. Les applications s'exécutant sur ces environnements doivent rendre ces ressources interopérables. Cela n'est possible que dans un environnement ouvert où chacun de ces dispositifs et services sont conçus pour être accessibles, c'est-à-dire que leurs fonctions doivent être explicitement et ouvertement accessibles depuis d'autres dispositifs ou applications. Bien que l'ouverture soit une condition préalable à la création d'environnements ubiquitaires, de nombreux systèmes sont encore aujourd'hui conçus pour limiter l'ouverture et l'interopérabilité. Les vendeurs de dispositifs peuvent restreindre délibérément l'ouverture et ignorer l'interopérabilité avec des services de vendeurs concurrents afin de préserver leurs parts de marché.

La restriction à l'ouverture implique que l'accès aux ressources de certains appareils ou services peuvent être soumis à des contraintes complexes et/ou des autorisations légales. Dans des environnements ubiquitaires, diverses applications ubiquitaires s'exécutent dans le même environnement. Elles accèdent aux dispositifs disponibles dans l'environnement, partageant en même temps leurs ressources et leurs fonctionnalités. Les applications peuvent éventuellement avoir différents niveaux d'autorisation d'accès à ces ressources. Par exemple, une application certifiée auprès d'un vendeur d'un dispositif peut avoir un accès complet à ses propres dispositifs, tandis qu'une autre application n'aurait qu'un accès limité.

Ce type de restrictions peut servir aux vendeurs de dispositifs ou aux fournisseurs de services pour garder un certain niveau de contrôle sur leurs produits, tout en continuant à contribuer à l'environnement ouvert avec des services disponibles publiquement. La maintenance et la gestion d'un tel environnement, où plusieurs applications ont des accès différents aux ressources partagées et doivent interagir entre elles, sont des tâches complexes.

2.3.4 Dynamisme

L'évolution est une propriété essentielle de tout environnement informatique. Les composants matériels peuvent défaillir suite à des avaries électroniques ou des conditions environnementales hors-normes, et les logiciels présentent souvent des anomalies (des *bugs*) qui nécessitent leur maintenance et leur mise à jour continue. Dans des environnements ouverts, l'évolution, tant logicielle que matérielle, est très fréquente. Chaque appareil, chaque système qui contribue à l'environnement ouvert évolue et change au fil du temps. Les environnements ubiquitaires sont généralement ouverts. Cette ouverture permet aux applications ubiquitaires de découvrir dynamiquement de nouvelles ressources et de les utiliser, lorsque l'état des ressources auparavant disponibles se dégrade et devient inaccessible.

En plus d'être ouverts, les environnements ubiquitaires sont en constante relation avec les environnements physique et utilisateur. En considérant l'évolution permanente de ces environnements, les éléments d'un environnement ubiquitaire sont obligés d'évoluer dynamiquement. Les variations de l'environnement ubiquitaire peuvent être de plusieurs natures, parmi lesquelles on peut trouver :

1. **Disponibilité des services** : dans un environnement ouvert et hétérogène, une ressource peut souvent être indisponible, par exemple parce qu'elle subit une mise à jour ou une maintenance, déclenchée par l'utilisateur, par l'administrateur ou par le système lui-même. Une autre raison qui peut compromettre la disponibilité est liée aux ressources limitées des dispositifs qui contraignent l'accès simultané à leurs services.
2. **Mobilité des utilisateurs** : les utilisateurs se déplacent librement dans leur environnement physique, en intérieur comme en extérieur. Les dispositifs mobiles peuvent changer de localisation en même temps que les utilisateurs qui les portent. Par exemple, des services Web sont accessibles par un smartphone *Wi-Fi* lorsque son utilisateur entre dans une certaine zone, couverte par un routeur *Wi-Fi*. Lorsque l'utilisateur quittera cette zone, le téléphone sortira de zone de couverture *Wi-Fi* et les services Web deviendront donc indisponibles.
3. **Contingence des dispositifs** : les dispositifs qui sont conçus pour être utilisés dans des environnements ubiquitaires sont en général de taille réduite et peu coûteux. Ils sont conçus pour maximiser leur temps d'utilisation avec un minimum de ressources. Dans certains cas, comme les étiquettes *RFID*, ils sont même de coût négligeables ou même jetables. Les fonctionnalités des dispositifs ainsi que leur capacité de communication sont généralement affectées par les propriétés physiques des dispositifs et de leur environnement, telles que la chaleur, des interférences radio ou le niveau de batterie, provoquant des erreurs ou de comportements imprévisibles.
4. **Interaction des utilisateurs** avec l'environnement : l'interaction des utilisateurs avec leur environnement permet aux applications de collecter des informations sur les intentions et les actions de ces utilisateurs. La sensibilité au contexte des applications ubiquitaires doit ainsi dépendre de cette interaction.

À cause du dynamisme dans les environnements ubiquitaires, les applications peuvent ne pas trouver toutes les ressources qui étaient prévues comme nécessaires lors de leur phase de conception.

2.3.5 Autonomie

Un environnement ubiquitaire est, comme nous l'avons vu, un environnement physique augmenté, de manière transparente, par des dispositifs et des systèmes informatiques. Afin de masquer la complexité de cet enchevêtrement entre le monde physique et le monde logique, il est nécessaire que l'environnement dans sa globalité soit aussi autonome que possible. Ce point est particulièrement important pour deux raisons :

1. Dans la vision de l'informatique ubiquitaire, les interactions entre les utilisateurs et les environnements ubiquitaires doivent être naturelles et transparentes. L'utilisateur peut savoir qu'il a affaire à un environnement physique augmenté, mais il ne doit surtout pas avoir à se préoccuper de comment il fonctionne. La propriété

d'autonomie de l'environnement ubiquitaire lui permet de s'adapter face aux événements externes. Cette faculté d'adaptation, de réaction aux changements et même d'anticipation, permet à l'environnement d'offrir une utilisabilité toujours adaptée au contexte, et ceci de manière transparente pour l'utilisateur. En résumé, plus l'environnement est autonome, moins les personnes qui l'utilisent auront à adapter leur utilisation pour interagir avec cet environnement. L'autonomie est donc un facteur favorisant la transparence des interactions entre humains et systèmes ubiquitaires.

2. L'intégration des systèmes ubiquitaires dans les environnements du quotidien permet leur diffusion dans une grande variété de scénarios : *smart-home, smart-office, smart-building, smart-cities*, *etc.* Cette propagation de systèmes informatiques à très large échelle pose le problème de leur maintenance. Comme nous allons le voir, la maintenance de systèmes informatiques est devenue un véritable défi, tant ils sont nombreux, hétérogènes et déjà très largement répandus. L'informatique ubiquitaire accentue encore ce phénomène en disséminant encore plus les systèmes informatiques. L'autonomie de systèmes ubiquitaires est donc indispensable, ceci afin d'alléger la charge de maintenance de ces systèmes.

Le caractère autonomique d'un environnement ubiquitaire permet de diminuer son empreinte sur l'utilisateur en lui fournissant une expérience adaptée au contexte. Cette autonomie aide aussi à réduire la charge d'administration de ces systèmes en réalisant automatiquement des adaptations simples. Le problème général de la maintenance des systèmes et des logiciels est d'ailleurs un sujet de plus en plus préoccupant, et sera traité dans le chapitre suivant.

2.3.6 Synthèse

Les environnements ubiquitaires sont par nature très diffus, hétérogènes, dynamiques et imprévisibles. Une multitude d'entités et d'acteurs différents doivent pouvoir y interagir de manière naturelle et transparente. Ce type d'environnement doit aussi, tel un milieu naturel, montrer des signes d'autonomies, en s'adaptant aux utilisations et aux conditions extérieures. Leurs limites peuvent être précises (par exemple à l'intérieur d'un bâtiment) ou parfois très floues (un parc, un quartier, une ville, *etc.*).

Toutes ces propriétés sont plus ou moins directement héritées de la vision initiale de l'informatique ubiquitaire et de ses diverses interprétations et applications (les *smart-**). Elles sont l'expression de comment est perçu, de l'extérieur, un environnement ubiquitaire : comment il interagit avec des éléments physiques, avec des utilisateurs, et/ou avec d'autres systèmes.

Ces perceptions des qualités ubiquitaires d'un environnement ont, comme nous allons le voir dans la section suivante, des répercussions importantes sur les systèmes ubiquitaires qui les incarnent. Un système informatique est composé à la fois d'éléments tangibles (capteurs, affichage, dispositifs d'acquisition, *etc.*) et d'éléments logiques (données, applications, composants, *etc.*). Il a pour rôle d'orchestrer cet environnement mixte en fonction de son utilisation, et a pour but de fournir une fonctionnalité. Dans le cas d'un système ubiquitaire, la nature de son environnement, dont les propriétés ont été énoncées ci-dessus, vont imposer la gestion de certains aspects primordiaux par le système, et par ses applications.

La section suivante montre comment les systèmes et les applications ubiquitaires doivent se distinguer de systèmes plus traditionnels en prenant en compte ces aspects essentiels qui vont insuffler la caractéristique ubiquitaire à l'ensemble.

2.4 Caractéristiques des applications ubiquitaires

Une application est un logiciel qui réalise des tâches spécifiques pour les utilisateurs. Les applications sont généralement installées au-dessus d'un système logiciel qui exploite et contrôle l'accès aux ressources (le système d'exploitation). Dans le cas d'une application s'exécutant sur un *middleware*, le *middleware* se situe entre l'application et le système logiciel, offrant une manière plus aisée et mieux contrôlée de développer et d'exécuter l'application.

La conception d'une application ainsi que les techniques utilisées pendant sa conception sont strictement liées et parfois limitées par les capacités du système sous-jacent, soit directement le système d'exploitation ou bien un *middleware*. Les caractéristiques des environnements ubiquitaires mentionnées précédemment imposent de nouveaux défis aux techniques existantes utilisées pour la construction d'applications ubiquitaires. Afin de mieux comprendre les caractéristiques qui faciliteront la conception d'applications ubiquitaires, il faut tout d'abord examiner les besoins des applications s'exécutant dans des environnements ubiquitaires. Dans la suite de cette partie, nous présentons des propriétés importantes qui permettent de distinguer les applications ubiquitaires des applications traditionnelles.

2.4.1 Gestion des ressources

Les applications classiques sont conçues pour travailler avec un ensemble de ressources prédéfinies. Exécutées localement ou distribuées entre plusieurs machines distantes, les applications traditionnelles sont fixées à des dispositifs (machines) et sont limitées par les ressources fournies par ces mêmes dispositifs. Toutefois, ces dernières années, divers mouvements ont dévié de ce paradigme. En particulier, l'émergence de dispositifs personnels mobiles a ouvert une nouvelle ère où l'informatique est de plus en plus centrée sur l'homme, et non plus sur la machine. Les applications ubiquitaires se conforment bien à la vision anthropocentrique de l'informatique, où les applications sont associées à des utilisateurs et à des lieux plutôt qu'à des dispositifs. En conséquence, ces applications ont besoins de pouvoir découvrir, administrer et utiliser différents dispositifs et ressources hétérogènes.

Les systèmes traditionnels, tels que les ordinateurs personnels ou les serveurs d'entreprise, exécutent des applications avec un ensemble de ressources prédéfinies, abstraites par les systèmes d'exploitation et/ou les *middlewares*. Le temps processeur, la mémoire, l'espace disque et la bande passante sont des exemples de telles ressources. Ces systèmes utilisent des abstractions pour simplifier l'accès des applications aux ressources. Par exemple, les systèmes d'exploitation abstraient les demandes d'accès au disque par une abstraction du système de fichiers, avec laquelle ils gèrent également les autorisations d'accès. Une ressource est caractérisée par un certain nombre de propriétés qui impactent la façon dont elle peut être utilisée et gérée :

- **Exclusive ou partagée** : une ressource peut être exclusive à une application ou peut être utilisée simultanément par plusieurs applications.
- **Avec ou sans état** (*stateful* ou *stateless*) : une ressource peut avoir un état lié à la/les application(s) qui l'utilise(nt) actuellement.
- **Individuelle ou groupée** : une ressource peut être individuelle ou peut être partie d'un groupe de ressources identiques.

En plus de ces propriétés, une ressource peut avoir d'autres attributs, tels que sa localisation géographique, et peut permettre la modification de sa configuration,

affectant ainsi son comportement. Les propriétés et les configurations sont déclarées en général dans les descripteurs des ressources, qui sont communiqués lors de la découverte de ces ressources.

Les dispositifs présents dans les environnements ubiquitaires sont généralement des ressources de premier ordre pour les applications. Les fonctionnalités des dispositifs peuvent être partagées entre applications, ou des fonctionnalités plus critiques peuvent être exclusives à certaines applications, disposant d'autorisations spécifiques. Un exemple de telles fonctionnalités est la configuration de dispositifs : alors que la plupart des appareils fonctionnent en mode *stateless*, ils deviennent de plus en plus configurables. La capacité de configuration permet d'optimiser les fonctions d'un dispositif en fonction des changements de son état (par exemple, le niveau de batterie). Toutefois, un changement dans la configuration d'un dispositif affecte inévitablement toutes les applications qui l'utilisent. L'hétérogénéité des dispositifs et des protocoles complique la virtualisation de l'accès aux ressources ; ces appareils deviennent, de fait, configurés de manière individuelle.

Dans des environnements avec un nombre important de dispositifs, il est difficile de gérer en même temps les autorisations d'accès, l'utilisation équitable des dispositifs et leurs configurations correctes. Les systèmes d'exploitation ainsi que divers *middlewares* intègrent déjà certaines politiques de gestion pour l'accès aux ressources. Toutefois, la nature dynamique et imprévisible des ressources dans les environnements ubiquitaires requiert de repenser et d'adapter ces politiques.

2.4.2 Orientation donnée

Les applications ubiquitaires offrent des services à valeur ajoutée grâce à l'utilisation de données collectées à partir de diverses sources, incluant généralement des capteurs. Ainsi, il est naturel de s'attendre à ce que, dans une application ubiquitaire, un service dépende non seulement d'autres spécifications de services mais aussi de types de données bien définis, où la méta-information d'une donnée peut être plus importante que son origine. En outre, cette orientation donnée impose un schéma de programmation où un consommateur réagit à un événement contenant des données produites par un fournisseur. Ainsi, un *middleware* ubiquitaire doit permettre de définir des dépendances vers certains types de données et d'assurer que ces dépendances soient satisfaites par des données produites par les services fournisseurs de ces mêmes types de données.

Une approche largement répandue est l'utilisation de mécanismes de médiation [Wiederhold 1992]. Un logiciel de médiation se concentre sur la collecte, transformation, synchronisation de données hétérogènes. Il permet de découpler les sources de données et les consommateurs de ces mêmes données.

2.4.3 Notion de contexte

Comme mentionné précédemment, la sensibilité au contexte est une des propriétés fondamentales de l'informatique ubiquitaire. Le contexte peut représenter toute information pertinente du point de vue de l'application et il peut être séparé en trois groupes : contexte utilisateur, contexte physique et contexte d'exécution. La notion de contexte n'est pas nouvelle. En effet, il est déjà présent dans des applications depuis une longue période, car les développeurs ont eu besoin d'informations sur l'état du système sous-jacent — logiciel ou matériel. Le besoin de modéliser le contexte est devenu plus évident avec des logiciels s'exécutant sur des machines virtuelles,

comme *Java*. Même si le principe *WORA* (*Write Once, Run Anywhere*) réduit les efforts nécessaires pour programmer des logiciels multi-plateformes, différentes architectures matérielles peuvent présenter des comportements différents, ce que les développeurs doivent prendre en compte dans leur application. Un exemple simple pour accéder au contexte de l'environnement virtuel sous-jacent est la liste les propriétés du système dans *Java*. Il permet aux développeurs d'accéder aux informations statiques primitives telles que la version du système d'exploitation, l'architecture processeur du système, *etc.*

Déterminer le contexte utilisateur dans le code de l'application est une chose déjà plus ardue. Dans les applications traditionnelles, telles les applications *Web*, les utilisateurs changent de contexte plus souvent que d'environnement virtuel sur lequel les applications s'exécutent. Dans ce cas-là, un contexte utilisateur peut être le navigateur utilisé pour accéder la page *Web*, l'historique des visites, les cookies, *etc.* L'*API* populaire de *Servlet* qui a été introduit durant les premières années de *Java* fournit des mécanismes standards pour représenter une requête *HTTP* à un serveur. Elle permet aux développeurs, côté serveur, d'accéder aux informations sur la requête, et de construire ainsi un profil utilisateur qui représente son contexte.

Pour les applications ubiquitaires, en plus du contexte virtuel et utilisateur, le contexte physique doit aussi être considéré. Les applications ubiquitaires requièrent de transformer les données brutes transmises par des dispositifs (des mesures de capteurs, des indicateurs d'autres appareils) en des mesures plus significatives sur l'état de l'environnement physique. Déterminer des contextes utilisateur complexes (par exemple le comportement, l'humeur ou l'intention des utilisateurs) et des contextes virtuels dynamiques (par exemple, la disponibilité des ressources, des mesures de performance) est plus difficile, en comparaison avec les applications traditionnelles. Dans de nombreux cas, le contenu du contexte dépend d'un point de vue spécifique à une application particulière. En conséquence, les applications intègrent des services fournisseurs de contexte qui sont responsables de transformer l'information brute, provenant de sources différentes et éventuellement hétérogènes, en information de contexte [Huebscher 2004]. Pour des raisons de sensibilité au contexte, les *middlewares* doivent utiliser des mécanismes pour inspecter le contexte d'exécution virtuel. De plus, fournir des mécanismes pour construire des modèles de contexte réduira non seulement le temps de développement des applications, mais permettra aussi la sensibilité au contexte lors de l'exécution.

2.4.4 Adaptabilité

La sensibilité au contexte requiert que les applications ubiquitaires s'adaptent en permanence suite aux variations du contexte. Une application ubiquitaire doit continuer à satisfaire les exigences des utilisateurs face à des dispositifs contingents, à des modules logiciels défaillants et en règle générale à l'évolution constante du contexte. Pour ce faire, elle doit être sensible à son contexte et suffisamment flexible pour être capable d'appliquer les configurations nécessaires et changer son comportement. Une telle adaptation doit être réalisée de façon autonome afin de rassurer l'utilisateur et de réaliser et d'être ainsi conforme à la vision de l'informatique ubiquitaire.

La plupart des applications traditionnelles sont développées pour satisfaire un ensemble figé de besoins. Cependant, dans des environnements ubiquitaires, à cause du contexte dynamique, les variations dans les besoins ne peuvent pas être considérées par des applications statiques. De ce fait, les applications complètement définies à

la conception et codées statiquement ne sont pas adaptées à des environnements de type ubiquitaire. Les applications doivent donc être développées et exécutées en considérant les possibles variations des besoins. D'une part, à la conception, les développeurs doivent spécifier et développer le système en fournissant différentes configurations possibles des applications. D'autre part, à l'exécution, les applications doivent être composées de manière flexible, permettant la reconfiguration dynamique, c'est-à-dire de passer dynamiquement d'une configuration à une autre.

Un exemple typique dans des environnements ubiquitaires mobiles est le cas où la disponibilité d'un certain dispositif déclenche le changement : selon la localisation de l'utilisateur, l'application peut choisir d'afficher l'interface utilisateur sur un écran haute définition ou sur un dispositif portable, optimisant la quantité d'information présentée à l'utilisateur.

En outre, des options de variabilité différentes peuvent exister dans une application. Ces options doivent être coordonnées pour fournir un fonctionnement optimal dans un contexte donné. En raison des changements rapides dans les environnements ubiquitaires et du manque d'administrateurs humains, le besoin d'approches autonomiques émerge afin de guider les adaptations dynamiques à l'exécution.

2.4.5 Sécurité

Comme évoqué précédemment, des mécanismes de sécurité sont nécessaires pour contrôler l'accès des applications aux ressources. Des protocoles d'authentification, d'autorisation et de gestion de comptes utilisateurs peuvent être implémentés dans différentes couches du système ubiquitaire, y compris la couche *middleware*, afin de contrôler l'accès aux ressources, d'assurer les politiques d'autorisation, de contrôler l'utilisation des ressources, *etc.*

Un autre aspect important des environnements ubiquitaires concerne la vie privée de leurs utilisateurs. Les informations collectées par divers dispositifs peuvent contenir des informations privées sur la vie des utilisateurs, ou peuvent permettre de les déduire. En présence de plusieurs applications et dispositifs de propriétaires différents, les *middlewares* requièrent de préserver la confidentialité des utilisateurs.

Un des principaux défis pour établir des mécanismes de sécurité dans les environnements ubiquitaires est de déterminer l'identité de l'utilisateur. En général, il n'est pas possible de demander aux utilisateurs de s'identifier, comme dans les pages *Web*, et en conséquence les applications doivent être authentifiées avec différentes sources d'authentification (par exemple, propriétaire de la plate-forme, propriétaire de l'application) et gérer en permanence la sécurisation des communications.

2.4.6 Synthèse

On se rend compte que le développement d'applications pervasives soulève des défis particulièrement difficiles, bien plus exigeants que ceux rencontrés pour des applications plus « classiques ». Cela demande de très fortes compétences de la part des développeurs, largement au-delà de ce que l'on rencontre usuellement.

Nous pensons dès lors qu'il est nécessaire, inévitable de fournir des environnements de programmation et d'exécution (des *middlewares*) dédiés à l'informatique ubiquitaire. Le but de ces *middlewares* est d'abstraire un certain nombre de problèmes tels que ceux mentionnés précédemment : gestion de l'adaptabilité, gestion des données, gestion de la sécurité, *etc.* De nombreux travaux en ce sens ont déjà été menés, avec un succès relatif.

De nombreux chercheurs se sont penchés sur la création d'environnement de développement et d'exécution pour les applications ubiquitaires. Nous explorons cet aspect dans les sections à venir.

2.5 Conception des applications ubiquitaires

La production d'une application est décomposée en plusieurs tâches élémentaires aux caractéristiques bien distinctes. Chaque phase demande des compétences différentes, et ainsi plusieurs spécialistes sont maintenant nécessaires afin de conduire un projet informatique conséquent à terme. Les phases majeures sont les suivantes :

1. **Analyse des besoins** : phase primordiale de tout projet logiciel, elle s'attache à définir quels en sont les objectifs essentiels et secondaires. Comme dans les projets non-informatiques, elle peut être appuyée par une étude de marché et/ou de faisabilité (prototypage) afin de cibler quels sont les besoins fondamentaux et/ou réalisables. Dans les cycles de vie traditionnels du génie logiciel, cette phase est première et unique ; les cycles de vie plus récents, itératifs puis agiles, découpent cette phase en plusieurs parties relativement indépendantes.
2. **Conception** : cette phase se concentre sur la création du squelette de logiciel : son architecture. Le projet est ainsi être découpé en plusieurs parties (modules, ou composants), et les spécifications de chaque partie sont établies, ainsi que les relations entre ces parties. Cette phase du cycle de vie est déterminante car le choix d'une architecture adaptée (aux besoins) facilite grandement la réalisation du projet, autant en termes de coûts, de temps, et de maintenances futures.
3. **Développement** : le développement est la phase qui consiste à réaliser les composants ou modules logiciels établis lors de la conception. Ces composants doivent être conformes aux spécifications énoncées lors de la conception. Les différents composants peuvent le plus souvent être développés parallèlement et de manière indépendantes.
4. **Déploiement** : le déploiement est l'action d'installer et de configurer le logiciel afin de le faire fonctionner sur les machines cibles. Selon les machines ciblées par le projet logiciel (machines personnelles, serveurs d'entreprises, *etc.*), cette phase peut être comprise dans le cycle de vie (livraison avec installation sur site) ou laissée à l'appréciation de l'utilisateur (logiciels grands publics).
5. **Exécution** : c'est la phase de fonctionnement nominal du logiciel, et donc la phase la plus importante, puisqu'elle incarne l'aboutissement du projet. Il faut cependant noter que cette phase est instable : un logiciel peut cesser de fonctionner, ou nécessiter des évolutions. Ces problèmes doivent être traités parallèlement à l'exécution du logiciel, dans la phase de maintenance.
6. **Maintenance** : la maintenance s'attache à conserver le logiciel dans un état d'exécution optimal. Des dysfonctionnements peuvent survenir, à cause d'une erreur dans la phase de développement (*bug*) par exemple ou d'un cas d'utilisation non prévu. Dans ce cas, une fois l'avarie signalée, une équipe de maintenance va devoir corriger le problème et proposer un correctif et une mise à jour de l'application.
7. **Test** : transversalement aux phases du cycle de vie énoncées ci-dessus, des tests peuvent être effectués afin de vérifier la conformité du logiciel par rapport aux besoins, et de garantir la qualité du logiciel. Différents types de tests permettent

de valider les différentes phases du cycle de vie, par exemple les test unitaires permettent de valider les composants issus de la phase de développement, tandis que les tests d'intégration valident l'assemblage de ces composants, suivant l'architecture définie dans la phase de conception.

D'après Meir Lehman [Lehman 1980], la maintenance des projets informatiques est la phase la plus coûteuse du cycle de vie du logiciel, en grande partie à cause de la durée de vie du logiciel qui peut s'étendre sur plusieurs décennies. Les sociétés éditrices de logiciels doivent en effet disposer en permanence d'une main d'œuvre qualifiée pour effectuer des retouches sur le logiciel, voire y ajouter de nouvelles fonctionnalités. L'impact de dysfonctionnements logiciels est d'ailleurs d'autant plus important que le nombre d'utilisateurs (conscients ou non) touchés est grand. Les éditeurs, qui distribuent de plus en plus systématiquement leurs produits gratuitement et/ou librement, accompagnés du code source, facturent à leurs clients le support technique de ces produits (centre d'assistance, correction rapide de *bugs*, intervention sur site, *etc.*) car c'est cette phase de maintenance qui leur coûte le plus cher.

Bien entendu, la correcte mise en œuvre des précédentes phases du cycle du logiciel, ainsi qu'une validation minutieuse ont une influence sur la quantité de travail qui devra être fournie durant la phase de maintenance. Cependant, le domaine de l'informatique ubiquitaire est par nature très dynamique, et les évolutions de l'environnement d'exécution du logiciel (environnement physique, besoins utilisateurs, interactions avec d'autres applications) sont impossibles à déterminer à l'avance. Il en résulte que les besoins doivent en permanence être réévalués, et le programme adapté à ces nouveaux besoins. La phase de maintenance est donc considérablement alourdie, et la nécessité d'un cycle de vie réactif, permettant des itérations très rapides, se fait ressentir.

Afin de faciliter la fabrication de logiciel, et ce tout au long du cycle de vie de l'application, le génie logiciel offre de nombreux méthodes et outils : compilateurs et interpréteurs de code, bibliothèques partagées, outils de gestion des exigences, outils de tests, mesures de métriques, *etc.* Le domaine de connaissance du génie logiciel est donc très vaste, et varie selon les approches utilisées pour développer un logiciel, et selon les phases du cycle de vie de l'application qui sont concernées.

Parmi cette variété d'approches, nous nous intéressons particulièrement aux *middlewares* qui nous semblent primordiaux pour aborder la phase de maintenance. Le but des *middlewares* est de raccourcir le cycle de vie des applications, en rendant leur conception, leur exécution et/ou leur maintenance plus aisée, plus rapide et/ou moins coûteuse. Les *middlewares* sont fréquemment complétés par (ou intégrés dans) des outils de développement plus généraux. Ces outils se concentrent sur la phase de conception du logiciel, en offrant une assistance au développeur, les *middlewares* viennent se placer à l'exécution entre le logiciel et le système ciblé, et offrent une couche d'abstraction permettant une exécution simplifiée. Ces deux approches ne sont pas mutuellement exclusives, si bien que, lors de la création d'une application, ils s'entremêlent de façon naturelle.

Les applications ubiquitaire, en raison des nombreuses contraintes qu'elles doivent satisfaire, et des nombreuses propriétés qu'elles doivent exposer, sont particulièrement complexes à concevoir, à exécuter et à maintenir. C'est pourquoi l'utilisation d'outils de développement et de *middlewares* appropriés est indispensable. Sans ces facilités, la gestion de la complexité des applications ubiquitaires démultiplie l'effort nécessaire à leur réalisation et leur mise en place, et leur cycle de développement est considérablement allongé. En effet, les besoins d'une application ubiquitaire sont extrêmement

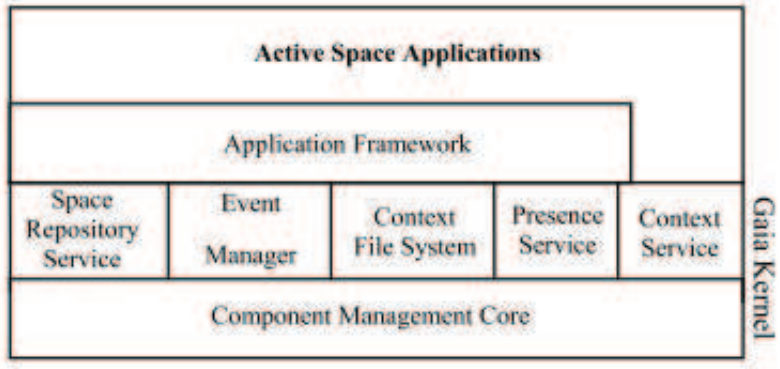

FIGURE 2.5 – Architecture de Gaia [Román 2002]

difficiles à cerner de façon exhaustive, et ils changent bien souvent pendant l'exécution de celle-ci. Les étapes suivantes du cycle de développement doivent donc prendre en compte cette variabilité des besoins, et fournir des moyens d'y répondre en adaptant une application à l'exécution.

Les travaux qui sont présentés ci-après présentent diverses approches visant à réduire la difficulté de conception, d'exécution et/ou de maintenance d'applications ubiquitaires.

2.6 *Middlewares* spécifiques pour l'ubiquitaire

Nous examinons dans cette section un ensemble de *middleware* et d'environnements de développement et d'exécution associés. Ces outils ont été conçus pour la plupart par des équipes de recherche se focalisant sur l'informatique ubiquitaire.

2.6.1 Gaia/Olympus [8]

Gaia est un *middleware* développé par un groupe de recherche de l'University of Illinois qui s'intéresse particulièrement aux environnements physiques de type *active spaces*. Ces derniers sont définis comme étant des espaces aux frontières bien délimitées, contenant des objets tangibles, des appareils utilisateurs. Gaia est présenté comme un méta-système d'exploitation pour ces *active spaces*, dans le sens où il apporte une couche d'abstraction à l'environnement et aux ressources qui le compose. La gestion de la mobilité de l'utilisateur est rendue possible par la sauvegarde des applications actuellement utilisées et de leurs données dans des sessions. Lors du déplacement de l'utilisateur, ces sessions sont dynamiquement projetées sur les ressources qui l'entourent.

La figure 2.5 montre l'architecture du *framework* Gaia : le noyau de Gaia repose sur une couche logicielle appelée *'Component Management Core'*. Celle-ci se charge de gérer dynamiquement le chargement, le déchargement, le transfert, la création et la destruction des composants de Gaia. Cette couche est basée sur l'architecture *CORBA* (*Common Object Request Broker Architecture*), qui permet aussi la distribution des composants entre les différentes nœuds formant l'*active space*. Au-dessus de cette brique fondamentale, le noyau Gaia fournit cinq services, directement utilisables par les applications pervasives :

8. `http://gaia.cs.uiuc.edu/`

1. Un **registre de ressources** qui est l'inventaire des ressources présentes dans l'*active space*. Les capacités de chaque ressource y sont décrites, et un langage de requête permet aux applications d'interroger le registre afin d'y trouver une ressource adaptée.
2. Un **gestionnaire d'événement** qui se charge de notifier les applications lors d'événements relatifs à l'*active space*, ou des applications participantes : ajout ou retrait composant, apparition ou disparition d'une ressource, entrée ou départ d'un utilisateur, *etc.* Les événements transitent par des canaux, auxquels les applications doivent s'abonner pour recevoir les événements liés. Les applications peuvent aussi participer en envoyant des événements sur ces canaux, et même définir leurs propres canaux de communication.
3. Un **système de fichiers contextuel** qui structure les données et ressources hiérarchiquement selon le contexte d'utilisation. Les applications peuvent donc accéder aux données, voire enregistrer leurs données, et ceci de façon transparente et sensible à leur contexte d'exécution. Par exemple, la requête suivante va rechercher la liste des parasols du jardin Nord par temps ensoleillé :

   /type :/parasol/**location :**/North-Garden/**weather :**/sunny

   Les applications et les utilisateurs peuvent lire et écrire dans ce système de fichier, ainsi que définir leurs propres opérateurs de sélection.
4. Un **service de contexte** qui contient des informations sur le contexte de l'*active space*. Les applications peuvent interroger ce service et ainsi connaître leur contexte d'exécution, mais aussi déduire des informations de contexte de plus haut niveau, par exemple quelle est l'activité en cours. Un modèle de règle et un langage associé permettent d'interroger le contexte, et d'y écrire des faits, par exemple :

   Context(number_of_person, RoomF018, >, 0) ⇒ **Context**(is_free, RoomF018, =, false)
5. Un service de présence fournit des informations sur la présence dans l'*active space* de personnes, de dispositifs et d'applications. Les entités logicielles doivent régulièrement attester de leur présence en envoyant un signal (*heartbeat*) à ce service. La non-réception de ce signal avant échéance signifie la disparition de cette entité, qui sera alors retirée du registre. La présence de personnes dans l'*active space* est déterminée activement par un ensemble de capteurs spécifiques.

L'équipe à l'origine du système Gaia a plus tard décrit dans [Ranganathan 2005] le langage Olympus. C'est un langage de haut niveau spécifique à la programmation des *active spaces* qui s'appuie sur Gaia pour le support des applications à l'exécution. Ce modèle abstrait offre deux caractéristiques principales qui sont particulièrement intéressantes :

1. la **découverte sémantique** des entités : un développeur spécifie les entités qu'il manipule (services, applications, périphériques, objets tangibles, lieux, utilisateurs) de manière abstraite, en utilisant des ontologies. À l'exécution, ces entités décrites sont résolues et liées à des entités existantes en prenant en considération le contexte courant : politique de l'*active space*, préférence de l'utilisateur, *etc.*
2. des **opérations de haut niveau** pour l'*active space* : les opérations les plus courantes (démarrage, arrêt, mouvement d'un utilisateur, ...) sont intégrées dans le modèle de programmation. Les développeurs peuvent donc directement utiliser ces opérations de haut niveau dans leurs programmes sans se soucier de la manière dont elles vont être exécutées en pratique.

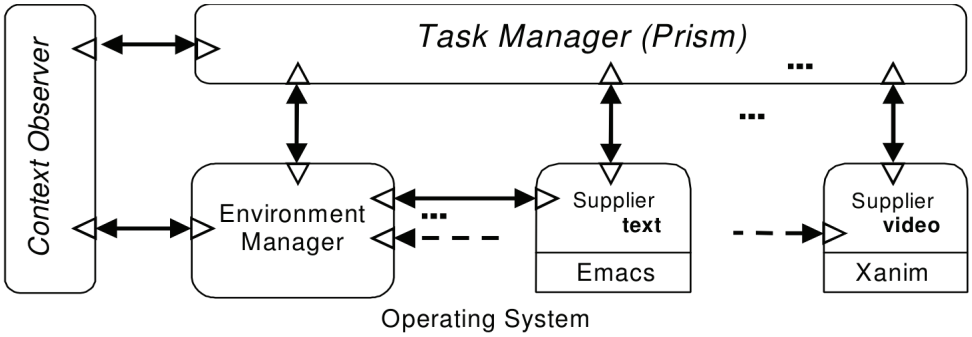

FIGURE 2.6 – Architecture du projet Aura [Sousa 2002]

Olympus définit des opérations élémentaires relatives aux *active spaces* ; il est cependant impossible d'ajouter simplement des opérations supplémentaires. De plus, la conception d'applications ubiquitaires est une tâche complexe, et ni Gaia ni Olympus ne fournissent d'assistance pour guider cette conception, et faciliter le travail de développement.

2.6.2 Aura [9]

Tandis que Gaia cible les environnements pervasifs de type *active space*, le projet Aura se focalise sur une approche centrée sur l'utilisateur. Ce projet est dirigé par David Garlan à l'université Carnegie Mellon. Le principe général est que la ressource la plus précieuse d'un environnement pervasif est l'attention de l'utilisateur, et non plus la puissance de calcul ou la capacité mémoire des appareils. Tout est donc fait dans Aura pour assister l'utilisateur de manière transparente, sans détourner inutilement son attention. Ainsi, chaque utilisateur dispose de son instance personnelle et dédiée d'Aura, qui le suit et l'aide à accomplir ses tâches dans un environnement pervasif.

Aura détermine quelles sont les intentions de l'utilisateur et les projette sur les services disponibles dans l'environnement d'exécution courant, comme montré par la figure 2.6. L'attention de l'utilisateur est donc économisée dans le sens où il n'a plus à explicitement « ouvrir » un service en particulier. Par exemple, l'utilisateur peut saisir un stylo, ce qui sera interprété par Aura comme une intention d'écrire du texte, ce qui déclenchera l'ouverture de l'application correspondante, par exemple *LibreOffice Writer*. Le choix de l'application est dépendant des préférences de l'utilisateur et du contexte courant ; par exemple si l'utilisateur était en train de tenir son téléphone au moment où il saisit son stylo, Aura pourrait plutôt ouvrir l'application servant à rédiger un *SMS*, ou un courriel.

Le fait de considérer les tâches des utilisateurs comme entité principale facilite le travail de développement des applications centrées sur l'utilisateur, car le développeur n'a plus à se soucier de quels services en particulier seront utilisés lors de l'exécution. Cependant, cette liaison automatique des tâches aux services disponibles est rendu possible par l'introduction de certaines contraintes de résolution : les fournisseurs de services doivent se conformer à l'interface de programmation uniforme d'Aura, qui est propriétaire. Ainsi, pour réutiliser des services existants (tel l'éditeur de texte *Writer*), il faut, en plus de fournir la description abstraite du service, écrire un adaptateur spécifique. Une autre limitation d'Aura est le manque d'assistance fournie lors de

9. http://www.cs.cmu.edu/~aura/

FIGURE 2.7 – Architecture d'un environnement Oxygen [Rudolph 2001]

l'implémentation d'un service ou d'un composant d'une application. Ce manque rend la conception et la maintenance d'applications pervasives particulièrement délicat et coûteux.

2.6.3 Oxygen [10]

Le projet Oxygen est une initiative du *MIT* (*Massachusetts Institute of Technology*) ciblant un environnement où l'informatique est partout accessible librement, tel l'air que nous respirons. Cet environnement fournit aux utilisateurs une puissance de calcul et des capacités de communication, disponibles de façon ambiante et transparente.

Les plateformes matérielles ciblées par le projet Oxygen sont regroupées en trois catégories. Les *E21s* sont des environnements physiques augmentés par l'ajout d'une multitude de capteurs. En plus de capacités sensorielles, ils fournissent à l'environnement une infrastructure de calcul et d'affichage. Les *H21* sont des dispositifs d'interaction légers et mobiles utilisés pour interagir avec le système. Le *N21* est le réseau ambiant qui permet la communication et la coordination des différents éléments. La figure 2.7 montre la composition et les interactions entre les différents éléments d'un environnement Oxygen.

Les applications Oxygen sont fournies par les *E21* de l'environnement local et peuvent interagir avec les *H21* de leurs utilisateurs, par le biais du réseau sans fil *N21*. Les interactions prennent en compte le contexte des utilisateurs, comme par exemple leur position autour de la table, les objets qu'ils montrent, l'activité courante, *etc.*

Une des limitations de cette approche est la trop grande cohérence entre les dispositifs, qui ne favorisent pas l'hétérogénéité. Dans la réalité, si les *E21s* peuvent effectivement être relativement homogènes au sein d'un même *smart space*, ou *smart building*, les appareils mobiles qui entrent en scène peuvent être très hétérogènes, chaque utilisateur pouvant utiliser son *smartphone* ou son ordinateur portable personnel par exemple. [11]

10. `http://oxygen.lcs.mit.edu`
11. « philosophie *BYOD* » : *Bring your own device*

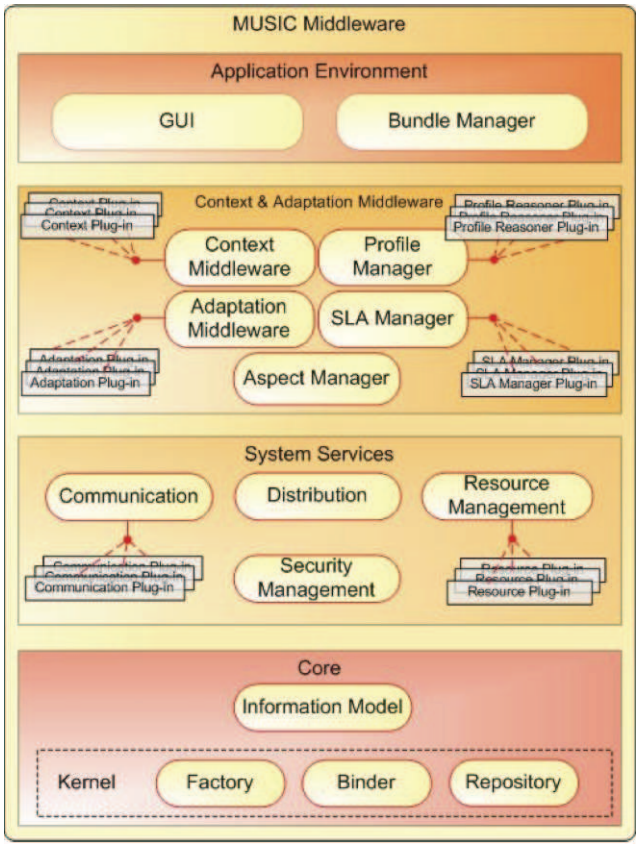

FIGURE 2.8 – Architecture du middleware du projet Music [Hallsteinsen 2010]

2.6.4 Music [12]

Music est un projet européen qui propose un environnement de développement d'applications afin de faciliter la création et l'exécution d'applications mobiles auto-reconfigurables qui s'adaptent au contexte de l'utilisateur. Music fournit un *framework* qui comprend un *middleware* dédié à l'exécution des applications mobiles reconfigurables et adaptatives, un ensemble d'outils visant à simplifier la conception et le développement de ces mêmes applications, ainsi qu'une méthodologie spécifiant comment cette plateforme et ces outils viennent s'intégrer dans une architecture dirigée par les modèles (*MDA*).

Music architecture son *middleware* en quatre couches, comme le montre la figure 2.8 :

1. Le **cœur** a pour but de fournir une abstraction des technologies sous-jacentes, découplant les couches supérieures des plateformes d'exécution, autant matérielles que logicielles. Il fournit aussi un modèle commun d'information et des structures de données qui seront utilisées pour représenter les applications.
2. Les **services systèmes** fournissent un support pour la communication distribuée entre les hôtes connectés en réseau, l'enregistrement et la liaison entre services distants, la communication entre ces services indépendamment du protocole de communication, une architecture orientée service pour l'exécution de ces services, la gestion de l'accès aux ressources locales et distantes ainsi que la gestion de la

12. `http://ist-music.eu/`

sécurité des services.

3. Le ***middleware* de contexte et d'adaptation** fournit un support pour la sensibilité au contexte aux composants du *middleware* et des applications. Cette couche est en charge de collecter, gérer et stocker des informations de contexte et de les rendre disponibles aux applications. En plus d'une modélisation du contexte, elle fournit un entrepôt de données de contexte et un processeur de requêtes qui permettent d'interroger l'historique des informations de contexte. Les autres composants, dont le *middleware* d'adaptation, reçoivent des notifications de changement du contexte et réagissent en utilisant leur connaissance de ces événements et du contexte.
4. L'**environnement d'application** présente des interfaces utilisateurs pour interagir avec le *middleware* et ajouter et mettre à jour les applications.

Bien que Music définisse son *middleware* comme ouvert et indépendant des technologies sous-jacentes, beaucoup de ses concepts sont hérités de et basés sur *OSGi*. De ce fait, la technologie *OSGi* est utilisée par l'implémentation de référence de ce *middleware*. Cette technologie fournit les bases d'une architecture orientée service modulaire, et la possibilité de s'exécuter sur différentes plateformes mobiles. La technologie à services *OSGi* sera étudiée plus précisément dans la suite de ce chapitre.

Le *framework* Music offre un studio de développement, pour les concepteurs d'applications sensibles au contexte. En utilisant Music Studio et la méthodologie précédemment mentionnée, un développeur peut construire un modèle d'application indépendant de la plateforme, définir les composants atomiques nécessaires et leurs dépendances contextuelles, exprimer comment ces composants doivent s'agencer ensemble, et décrire quels sont les différents scénarios de déploiement possibles. Music Studio permet la création d'un modèle *UML* indépendant de la plateforme (*PIM : platform independent model*) qui peut être automatiquement transformé en un code source dépendant de la plateforme. Ce code source est ensuite complété par les développeurs, qui spécifient quelles sont les fonctions utilitaires (*utility functions*) nécessaires pour l'adaptation, et implémentent la logique métier des composants de l'application. Une fois que l'application a été packagée pour la plateforme cible, des outils de test et de simulation permettent de feindre des changements de contexte et de surveiller l'adaptation des applications.

2.6.5 DiaSuite [13]

DiaSuite est une suite d'outils couvrant principalement la phase de développement des applications ubiquitaires. DiaSuite fournit un langage, DiaSpec, dédié à la description d'un système ubiquitaire, de son environnement et de ses applications. Comme dépeint par le figure 2.9, deux acteurs sont impliqués dans l'utilisation de ce langage :

1. l'**expert du domaine** qui décrit quelles sont les entités dont la présence est requise dans l'environnement, tels que les dispositifs, les informations qu'ils produisent et les opérations qu'ils sont capables d'effectuer.
2. l'**architecte logiciel** qui spécifie quels composants vont participer à l'application, tels que la taxonomie de contexte et les composants de contrôle qui contiennent la logique applicative.

13. `https://diasuite.inria.fr/`
14. Source : `https://diasuite.inria.fr`

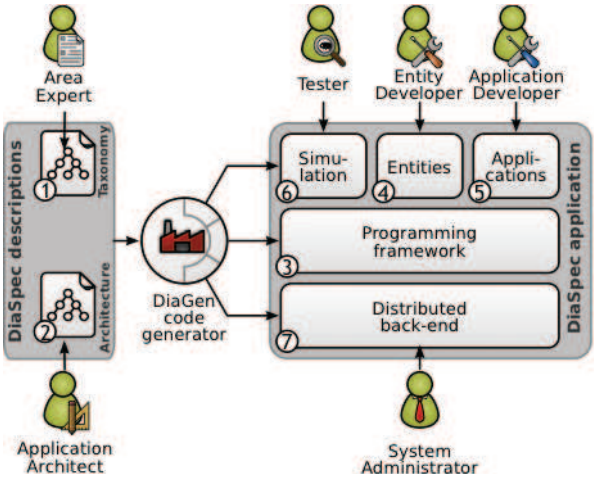

FIGURE 2.9 – Cycle de développement d'une application avec DiaSuite [14]

Tout comme Music, DiaSuite génère un code source à partir d'une spécification afin de guider les développeurs durant l'implémentation des entités de contexte et des composants de contrôle. Le code implémenté et la spécification DiaSpec servent ensuite à approvisionner le *framework* qui va permettre l'exécution les entités de contexte réifiées et liées aux dispositifs ; c'est ainsi que la sensibilité du système au contexte est supportée.

DiaSuite comprend aussi un outil de simulation pour faciliter la phase de validation de l'application : DiaSim. Cet outil permet de définir des environnements de simulation, de modéliser un environnement pervasif et de le parsemer de dispositifs et d'entités définies dans la spécification DiaSpec. L'éditeur graphique de scénario permet de définir le scénario de simulation, qui va envoyer des stimuli aux dispositifs et aux entités simulées. Ces scenarii constituent une véritable suite de test pour l'application développée et la plate-forme d'exécution qui en découle. Il est de plus possible de mettre en scène ces scenarii dans un environnement hybride, combinant entités réelles et simulées [Bruneau 2009]. La figure 2.10 montre un exemple d'environnement tel qu'il est simulé par DiaSim.

2.6.6 WComp [15]

WComp est un environnement qui permet de faciliter le développement et supporte l'exécution d'applications ubiquitaires. Ce projet a été conçu en 2003 conjointement par une équipe de l'université de Nice-Sophia Antipolis et le CNRS [Cheung 2003]. Il vise à réduire la complexité posée par les propriétés des environnements ubiquitaires, plus précisément la mobilité des dispositifs et leur hétérogénéité dans un environnement d'exécution dynamique.

Le modèle de développement d'applications WComp s'appuie un modèle à composants légers au-dessus de la technologie *Microsoft .NET*. Les composants utilisent et fournissent des *WebServices*. Une application est donc une composition structurelle de *WebServices*, implantés sous la forme de composants. Les interactions entre les différents services et les dispositifs physiques de l'environnement sont assurées via les protocoles *UPnP* et *DPWS*.

15. `http://www.wcomp.fr/`

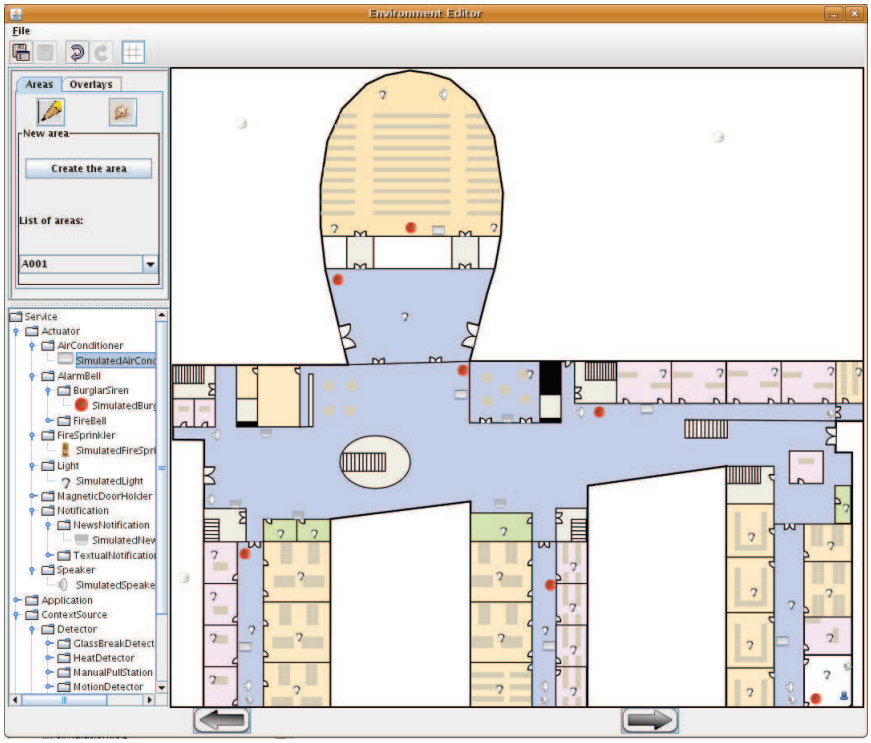

FIGURE 2.10 – Éditeur de simulations DiaSim [Bruneau 2009]

Les compositions structurelles sont décrites par une approche *SCLA* [16] [Tigli 2010] qui assemble les services fournis et requis par les composants, et gère l'acheminement des événements entre ces services. Les composants sont utilisés comme des boîtes noires, mais leur assemblage peut être adapté en décrivant les changements structurels à effectuer avec des aspects. La résolution des dépendances de services entre les composants est effectuée à l'exécution, selon la disponibilité des différents services et/ou dispositifs. Cette assemblage s'effectue toujours conformément à la définition *SCLA* de l'application.

2.6.7 Kalimucho [17]

Kalimucho est un projet développé à l'université de Pau. C'est une plateforme dédiée au développement et à l'exécution d'applications multimédia sur des appareils mobiles, tels que des *smartphones*. L'environnement constitué de dispositifs nomades est par nature instable, et sujet à de nombreuses fluctuations : hétérogénéité des réseaux, pertes de signal, panne de batterie, mobilité des usages, *etc.* La plateforme Kalimucho prend en compte les changements de l'environnement d'exécution afin d'adapter une application multimédia (par exemple la diffusion d'un flux vidéo). Bien que conçu spécifiquement dans le cadre d'applications mobiles, sa sensibilité au contexte et sa capacité d'adaptation en font un projet pouvant s'intégrer dans des environnements ubiquitaires.

L'objectif de la plateforme Kalimucho est de maintenir un qualité de service suffisante, malgré les événements pouvant survenir sur le réseau mobile. Pour cela il

16. *Service Lightweight Component Architecture*
17. `http://www.kalimucho.com/`

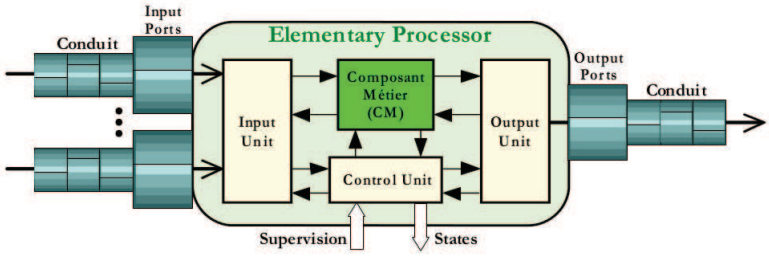

FIGURE 2.11 – Structure d'un composant Kalimucho [Bouix 2008]

se base sur un modèle à composant orienté flot de données : Osagaia. Ce dernier fournit un support à l'exécution pour des composants métiers (cf. figure 2.11), et synchronise les flux multimédia entre eux. Ces flux circulent à travers des conduits, géré par Korrontea, connectant les composants métiers entre eux, qui prennent en charge la synchronisation des données, et leur distribution sur les mailles du réseau. Lorsqu'une modification de la topologie du réseau survient, ou que son débit devient trop faible, Osagaia peut modifier les conduits afin qu'ils empruntent d'autres routes, et ainsi prévenir des répercussions sur la qualité de service [Bouix 2005, Da 2014].

La plateforme Kalimucho va pouvoir réaliser des adaptations de plus haut-niveau à partir des informations remontées par les composants, des conduits, et des environnements d'exécution. Il va par exemple pouvoir connaître de niveau de la batterie d'un dispositif, et réduire la charge des composants de ce dispositifs si ce niveau est trop faible. Kalimucho va aussi pouvoir traiter les données circulant dans les conduits, afin de réduire leur coût en bande passante et s'adapter ainsi aux aléas des réseaux mobiles. Par exemple, un utilisateur qu'il s'éloigne trop de son point d'accès au réseau pourra continuer de regarder un flux vidéo en mode dégradé (noir et blanc, ou baisse de la résolution de l'image) [Louberry 2011].

2.6.8 Synthèse

Si chacun des travaux présentés ci-dessus présente des aspects qui facilitent la mise en œuvre d'applications pervasives, on remarque cependant une grande diversité tant dans les approches adoptées que dans les domaines d'application envisagés. Malgré ces différences, tous répondent à un ou plusieurs des défis à relever pour intégrer des applications dans un environnement ubiquitaire.

Une des premières différences qui apparaissent après l'étude de ces travaux sont les différentes visions de l'environnement ubiquitaire. On peut en effet distinguer deux catégories d'environnements ubiquitaires ciblés :

1. celle dite des *active spaces*, où le système est géographiquement délimité : par exemple une pièce, un étage ou un bâtiment.
2. celle dite *user-centric* où le système est focalisé sur un utilisateur en particulier, le suivant lors de ces déplacements.

Certains des travaux cités se positionnent clairement en faveur d'une des deux approches, par exemple Gaia qui est conçu spécifiquement pour les *actives spaces*, ou encore Aura, dont chaque instance « suit » son utilisateur. D'autres travaux ont un positionnement moins évident, ou plus prudent, et ne distinguent pas explicitement ces deux configurations d'environnement. Si un positionnement clair, d'un côté ou

de l'autre, permet aux applications d'avoir un contexte d'exécution mieux défini, et de mieux répondre aux spécificités de ces types d'environnement, il restreint en contrepartie les domaine de ces applications à un sous-ensemble de l'informatique pervasive.

Certaines caractéristiques structurelles des environnements et applications ubiquitaires, comme l'hétérogénéité ou la distribution, sont très bien couverts par la majorité des travaux présentés. Les approches spécifiques comme les génériques ont en effet une modélisation assez complète de ce qu'est un environnement ubiquitaire, et de comment une application s'exécute en son sein. Il y a cependant d'autres aspects, liés à l'évolution des environnements et des applications, qui ne sont pas traités de manière suffisante, voire totalement occultés. Ainsi, le dynamisme des environnements et applications ubiquitaires est très mal supporté dans la plupart des travaux. Cette limitation rend difficile l'évolution des applications dans des environnements pourtant constamment en mouvement.

Un autre aspect essentiel, l'autonomie des systèmes, est laissé à l'écart. Si, comme nous allons le voir dans le chapitre suivant, la réalisation de systèmes autonomes est une tâche complexe, elle nécessite cependant un support minimal de la plateforme d'exécution. Aucun des travaux présentés ci-dessus ne présente de caractéristique autorisant leur gestion de l'autonomie.

Des approches dites génériques peuvent elles aussi offrir les qualités nécessaires à l'élaboration et à l'exécution d'applications ubiquitaires. Ces travaux n'ont pas été spécifiquement conçus pour le domaine de l'informatique ubiquitaire, mais tentent de réduire la complexité de développement d'application en gérant les aspects non-fonctionnels, et offrent leur support. Ces approches sont généralement extensibles et permettent donc d'ajouter la gestion des caractéristiques propres aux applications ubiquitaires.

2.7 *Middlewares* génériques

2.7.1 Composants logiciels

Il existe un ensemble de *middlewares* génériques, la plupart issus de travaux en génie logiciel, qui sont utilisés pour le développement d'applications ubiquitaires, même s'ils n'offrent pas forcément toutes les propriétés requises. Nous pouvons citer par exemple :

- **Fractal** [Bruneton 2006] est un modèle a composant générique à la base de plusieurs travaux [Bouchenak 2006, David 2005, Leclercq 2004, Romero 2010]. Différentes implémentations de Fractal existent dans plusieurs langages de programmation : *C* (Think [Fassino 2002], Cecilia), *C++* (Plasma), *Java* (Julia [Bruneton 2006], AOKell [Seinturier 2006], ProActive), *Smalltalk* (FracTalk) et *.NET* (FractNet). Chacun de ces langages apporte des propriétés supplémentaires à ce modèle et chacun est destiné à un domaine particulier. Julia, l'implémentation de référence, permet la spécification de composants en *Java* et leur manipulation à l'exécution. Fractal ne définit pas de mécanisme standard pour le *packaging* ou le déploiement de composants car celui-ci dépend de l'implémentation et du langage choisis. Il offre néanmoins des mécanismes pour faire évoluer dynamiquement les applications : créer et détruire des instances de composants et les liaisons entre eux.
- **K-Component** est un modèle à composants pour applications auto-adaptatives

[Dowling 2001b, Dowling 2001a]. L'originalité de ce modèle vient du fait que chaque composant dispose d'une vue partielle du système. Il permet à chaque composant de réaliser des adaptations individuelles à travers une coordination complexe et décentralisée. L'évolution des applications réalisées à travers ce modèle à composants est dynamique. La reconfiguration est réalisée par l'ajout, la suppression ou la mise à jour des composants et connecteurs.

– **Kevoree** est un modèle à composants *open source* récent gérant des applications adaptatives et distribuées par l'utilisation des modèles à l'exécution (*Model@Runtime*) [Morin 2009]. Ce modèle se distingue des modèles à composants traditionnels par la définition du concept de canaux de communication (*channels*). Ces canaux permettent de relier des composants distants déployés dans des plates-formes hétérogènes, selon une sémantique propre à chaque canal. Kevoree est capable à la fois de modéliser l'architecture d'un système distribué hétérogène et de gérer des évolutions au cours de son fonctionnement. Il vise à faciliter le développement d'applications dans le contexte de l'informatique ubiquitaire, naturellement distribuée [Fouquet 2012].

Ces middlewares sont remarquables pour leur propriété de gestion du dynamisme : ils peuvent être adaptés ou s'adapter à l'exécution en fonction du contexte, ce qui est une propriété fondamentale pour les applications ubiquitaires. Ils sont, pour la plupart, fondés sur la notion de composant logiciel [Szyperski 2011]. Les composants logiciels sont des briques fonctionnelles de base que l'on peut composer et déployer indépendamment. D'après Heineman [Heineman 2001], un composant doit être conforme à un modèle à composants. Un tel modèle définit l'infrastructure et le cadre nécessaire permettant la conception, le développement, le déploiement et l'exécution des composants. En particulier, le modèle définit les types de composant qui sont ensuite utilisés pour créer des unités d'exécution appelées instances de composant, ainsi que les rapports qu'ils peuvent avoir les uns aux autres.

Récemment, les modèles à composants ont intégré la notion de service logiciel. Cette notion est aujourd'hui très utilisée dans le domaine des *middlewares* pour applications ubiquitaires (y compris ceux cités précédemment). Nous présentons cette notion dans la section suivante.

2.7.2 Services logiciels

Un service est une entité logicielle qui fournit un ensemble de fonctionnalités définies dans une description de service. Cette description comporte des informations sur la partie fonctionnelle du service mais aussi sur ses aspects non-fonctionnels. À partir de cette spécification, un consommateur de service peut rechercher un service qui correspond à ses besoins, le sélectionner et l'invoquer en respectant le contrat qui a été accepté par les deux parties [Chollet 2008].

L'objectif de l'approche à services est la construction d'applications à partir d'entités logicielles indépendantes, tout en assurant un faible couplage entre ces entités. La définition, publication, utilisation des services se fait dans une infrastructure bien définie qui explicite les interactions possibles entre fournisseurs et utilisateurs. Cette infrastructure s'appelle une architecture à services (*Service Oriented Architecture*) [Papazoglou 2003]. *CBDI* [18] [Sprott 2004] propose la définition suivante :

18. *Component Based Development and Integration*

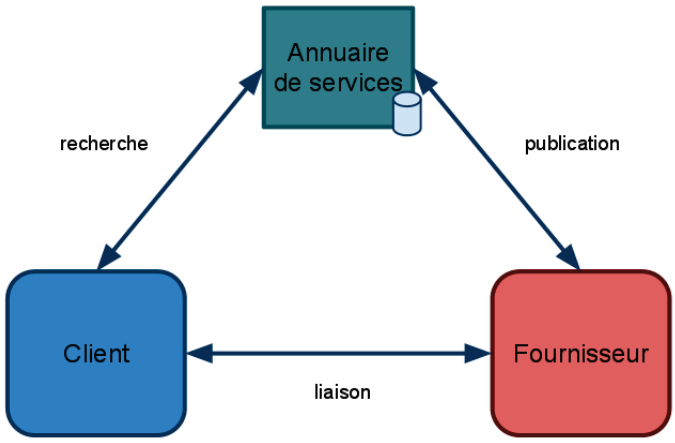

FIGURE 2.12 – Acteurs fondamentaux de l'approche à service

> « *The policies, practices, frameworks that enable application functionality to be provided and consumed as sets of services published at a granularity relevant to the service consumer. Services can be invoked, published and discovered, and are abstracted away from the implementation using a single, standards-based form of interface.* »

Une architecture à services propose une définition précise de la notion de services et fournit un environnement permettant le développement et l'exécution des services. Cet environnement définit notamment les politiques et les patrons d'interactions entre services et fournit également un ensemble d'outils et de fonctions techniques de base. Les architectures à services s'attachent toutes à abstraire les services de leur technologie d'implantation.

Toutes les architectures à services reposent sur trois acteurs de base et doivent mettre en œuvre leurs interactions (voir figure 2.12). Ces acteurs sont :

- les **fournisseurs de service** qui décrivent des services selon une spécification bien établie et les communiquent à une entité centralisatrice, souvent appelée registre,
- l'**annuaire de services** qui contient l'ensemble des descriptions des services proposés et valides à un instant donné,
- les **consommateurs de service**, encore appelés clients, qui utilisent des services après les avoir sélectionnés au sein du registre (ou toute entité regroupant des services).

Le découplage architectural entre le client et le fournisseur de service permet de construire des applications dynamiques où l'utilisation de services peut être faite sur demande. La localisation, la liaison et l'utilisation du service peuvent donc être faites à tout moment : statique, pendant la conception, dynamique, lors de l'exécution, semi-statique, pendant le déploiement.

De façon plus précise, le fournisseur de service représente une personne/une organisation capable de fournir des fonctionnalités sous forme de service. Après le développement d'un service, un fournisseur doit mettre à disposition des éventuels utilisateurs les informations nécessaires pour pouvoir utiliser le service, c'est-à-dire la description du service. La description du service rassemble en premier lieu toutes les informations concernant les fonctionnalités fournies par le service, ainsi que, le cas

échéant, ses propriétés non-fonctionnelles et les types de communication acceptés.

Comme indiqué précédemment, la description du service est ensuite publiée dans un registre de services. Elle contient toute l'information nécessaire à son utilisation. Cette description permet aussi d'identifier les caractéristiques non-fonctionnelles du service, telles que les politiques d'utilisation, les contraintes et, dans certains cas, la qualité du service fournie.

Un consommateur de service, interroge le registre de services pour s'enquérir des services disponibles qui correspondent à ses besoins. La découverte d'un service est réalisée grâce à la description du service disponible dans l'annuaire. Après avoir sélectionné le service qu'il veut utiliser, le consommateur du service peut, dans certains cas, négocier auprès du fournisseur les termes suivant lesquels il peut utiliser ce service. À la fin de la négociation, un accord de service [Aiello 2005] est réalisé entre le consommateur et le fournisseur. La plupart du temps, cet accord de service contractualise les termes de l'utilisation du service par le consommateur sans garantie totale du résultat. Grâce aux informations disponibles dans la description du service, le consommateur de service peut, dès lors, réaliser la liaison et appeler les fonctionnalités du service.

L'interaction entre ces acteurs peut être locale ou distribuée. Cette approche présente les trois primitives d'interaction suivantes :

- la **publication de service** : le fournisseur de service met à disposition l'information nécessaire pour l'utilisation du service, c'est-à-dire enregistre la description du service dans un annuaire de services,
- la **recherche de service** : le client, ou consommateur de service, interroge l'annuaire de services pour trouver un service qui corresponde à ses besoins,
- la **liaison**, si l'annuaire contient une description de service correspondant aux besoins du consommateur, le client effectue une liaison en utilisant l'information du service obtenue à partir de l'annuaire. Cette liaison est une communication entre le client et le fournisseur de service et permet l'utilisation des capacités fournies par le service.

2.7.3 iPOJO [19]

Apache Felix iPOJO est un projet *open source*, qui prône l'utilisation d'un modèle à composants orientés services. En effet, si l'architecture orientée services offre des qualités intéressantes pour la conception d'applications (couplage faible entre composants, substituabilité, dynamisme, *etc.*), la conception de tels applications reste très complexe. La gestion correcte de la disponibilité et du dynamisme des services est très technique, fastidieuse, et source de nombreuses erreurs. C'est donc dans le but de faciliter la conception de composants orientés services qu'iPOJO propose un modèle de développement simple, à base de composants, qui masque ces aspects non-fonctionnels. Ces aspects sont décrits autour du code métier du composant, et pris en charge à l'exécution par le conteneur du composant, comme illustré par la figure 2.13. Le *framework* iPOJO est un sous-projet de la plateforme à services Apache Felix [20], implémentation de la spécification OSGi [21]. Il est un modèle à composant orienté service très utilisé dans l'industrie, notamment par les serveurs d'applications OW2 JOnAS [22],

19. http://www.ipojo.org
20. http://felix.apache.org
21. http://www.osgi.org
22. http://jonas.ow2.org

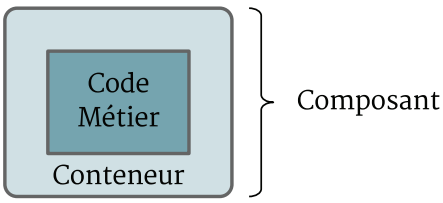

FIGURE 2.13 – Composant : conteneur et code métier

```
@Component // <- Définition de type de composant
@Provides // <- Fourniture de service (implicitement de type LightFollowMe)
public class LightFollowMeImpl implements LightFollowMe {

    // Dépendance de service
    @Requires
    private PresenceSensor sensor;

    // Autre dépendance de service
    @Requires
    private Light[] lights;

    // Propriété de configuration
    @Property
    private int autoPowerOffPeriod;

    // Méthode métier
    public void powerOff() {
        for (Light l : lights) {
            l.setPower(OFF);
        }
    }

    // ... suite du code métier ...

}
```

FIGURE 2.14 – Composant : conteneur et code métier

Peergreen [23], ainsi que la plateforme pour applications mobiles UbiDreams [24].

Le *framework* iPOJO repose donc sur un modèle de développement à base de *POJO* [25] : le développeur écrit un composant iPOJO comme étant une classe *Java*, contenant des champs ainsi que des méthodes, sans code spécifique au conteneur. Cette classe décrit le fonctionnement du cœur métier du composant iPOJO, définit un squelette, appelé **type de composant**, qui permettra la création d'objets, conformes à ce squelette, appelés **instance de composants**, ou simplement composants (sous-entendu, à l'exécution).

En plus de code métier du composant, le développeur **annote** la classe *Java*. Ces annotations donnent à iPOJO les instructions nécessaires pour savoir à quel endroit il doit intervenir : injection de dépendance, de propriété de configuration, fourniture de service, *etc.* En bref, le comportement métier du composant est **implanté** par le déve-

23. `http://www.peergreen.com`
24. `http://www.ubidreams.com`
25. Acronyme de *Plain Old Java Object* : objet *Java* simple incarnant un composant. Cet objet est simple dans le sens où il n'interagit pas avec le conteneur de composant, dont il est d'ailleurs, idéalement, totalement indépendant.

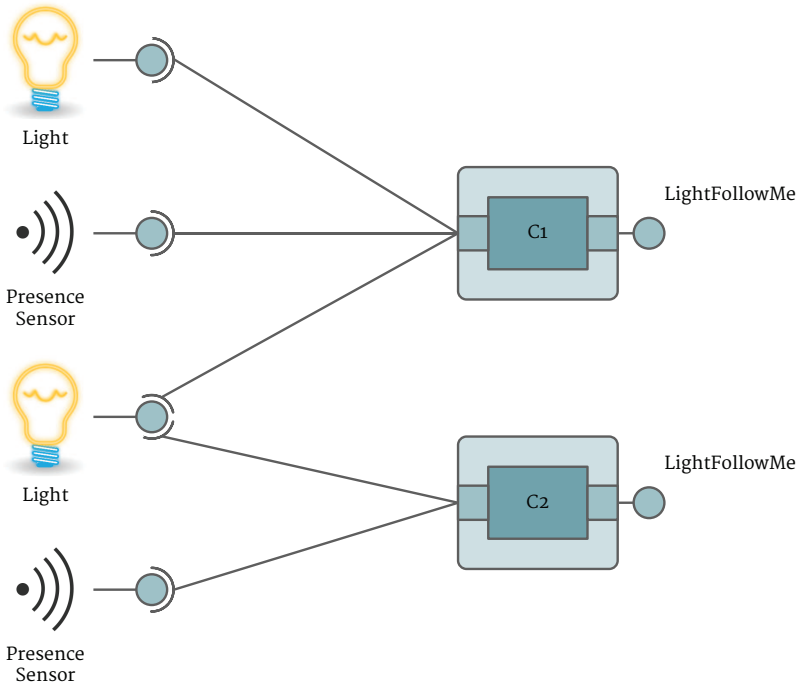

FIGURE 2.15 – Exemple de composants iPOJO à l'exécution

loppeur, tandis que les aspects non-fonctionnels sont **décrits** par des annotations. La figure 2.14 montre un exemple de définition d'un type de composant iPOJO, qui fournit un service (interface `LightFollowMe`), dépend de deux types de service (`PresenceSensor` et `Light`), et dont le comportement peut être configuré par une propriété (champ `autoPowerOffPeriod`).

À l'exécution, ce type de composant va permettre la création d'une ou plusieurs instances de composants iPOJO. Ces instances sont indépendants les unes des autres : chacune ayant son propre état, fournissant son propre service, et pouvant dépendre d'un ensemble de services différent. Les objets métier (*POJO*) de ces instances vont être encapsulés à l'intérieur de **conteneurs de composant**. Le rôle du conteneur est principalement de gérer le cycle de vie du composant et du ou des *POJOs* contenus. Ce conteneur est à géométrie variable, et permet l'inclusion de différentes extensions appelées ***handlers* de composant**. Ces *handlers* vont permettre la gestion d'aspects non fonctionnels et/ou techniques du composant, telles les dépendances de services, la (re)configuration, la fourniture de services, *etc.* Ces *handlers* sont en fait l'expression, à l'exécution, des annotations présentes dans la classe définissant le type de composant.

La figure 2.15 montre l'exemple de deux instances c_1 et c_2 de composants (dont le type a été défini dans la figure précédente) à l'exécution. Chacune de ces instances est composée d'un conteneur, d'un objet métier, et de plusieurs *handlers*. Les objets métiers sont des instances de la classe définie dans la figure précédente. Les différents *handlers* sont les « incarnations » à l'exécution des aspects non-fonctionnels décrits dans cette même classe. Par exemple, les *handlers* gérants les dépendances de service correspondent aux annotations `@Requires` décorant les champs `sensor` et `lights`.

Une des fonctionnalités primordiales du conteneur est la **gestion du cycle de vie** du composant. En effet, l'environnement d'exécution est dynamique et peut, dans certains cas, ne pas satisfaire les conditions nécessaires à l'exécution de composants. Dans

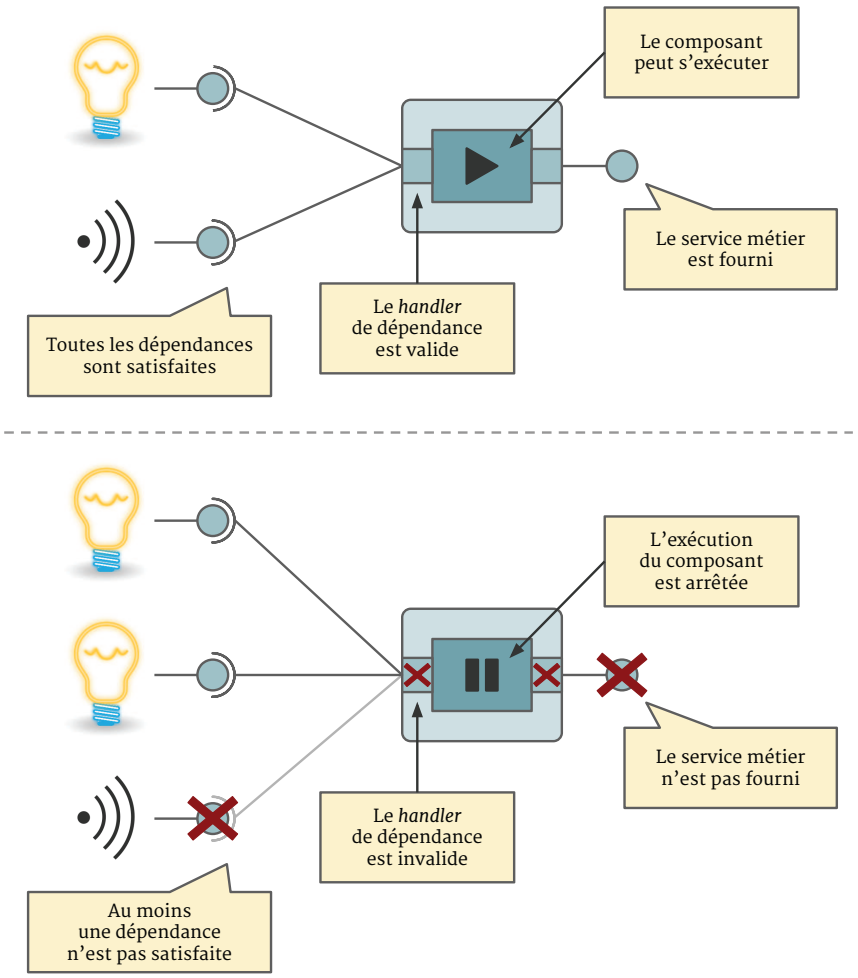

FIGURE 2.16 – Exemple de composants valides et invalides

ce cas, le composant est dit **invalide** : son exécution est suspendue jusqu'à ce que les conditions deviennent favorables à la reprise de son exécution. La figure 2.16 montre l'exemple de deux composants, dont un qui est invalide à cause d'une dépendance de service non-satisfaite. Le conteneur déduit la validité du composant en fonction de la validité des *handlers*.

Dans cet exemple, la non-validité de la dépendance entraîne l'invalidation du *handler* gérant les dépendances de services, qui à son tour entraîne l'invalidation de l'instance de composant toute entière. L'instance étant invalidée, elle ne peut plus fournir de service. Si, par la suite, un service satisfaisant cette dépendance apparaît, le *handler* gérant les dépendances de services redeviendra valide. Ce changement d'état du handler entraînera la validation de l'instance du composant, qui pourra reprendre son exécution et fournir à nouveau son service.

Il existe de nombreux types de *handlers*, permettant de gérer une multitude d'aspect non-fonctionnels différents : services, configuration, administration, journalisation, transactions, persistance, distribution, *etc.* Le *framework* iPOJO fournit, de base, les *handlers* permettant la configuration et la gestion des dépendances et fournitures de services. Il est toutefois possible d'étendre les composants iPOJO en ajoutant d'autres *handlers*, en fonction des aspects à gérer.

Toutefois, la gestion des dépendances et des fournitures de services constituent la pierre angulaire d'une architecture à base de composants à service. Comme nous l'avons vu précédemment, ce type d'architecture promeut la flexibilité et l'adaptabilité nécessaires à la gestion autonomique de la plateforme. C'est dont sur ces deux aspects fondamentaux que la suite de cette section va se pencher plus particulièrement.

Dépendances de services

Une dépendance de service dans un composant iPOJO est principalement décrite par six propriétés, qui définissent à la fois quels sont les services potentiellement utilisables par le composant, comment les sélectionner, et comment réagir lors d'événements liés au dynamisme de ces services. Ces propriétés sont :

- le **type de service** attendu : c'est l'interface qui définit les opérations fonctionnelles supportées par le service.
- la **cardinalité**, qui définit combien de services peuvent être utilisés simultanément par le composant : un seul ou plusieurs.
- le caractère **obligatoire ou optionnel** de la dépendance, qui recouvre, en partie, la cardinalité de la dépendance. Une dépendance optionnelle non satisfaite n'empêche pas l'exécution du composant.
- le **filtre de sélection** permet de spécifier quels services peuvent être utilisés, en fonction de leurs types et des propriétés de service qu'ils exposent.
- un **comparateur de service** qui ordonne les services en fonction de leurs propriétés mais aussi des besoins du composant.
- la **politique de liaison** qui détermine dans quels cas un service peut être remplacé par un meilleur (par rapport aux résultats du comparateur) service, ou bien quand le composant doit être invalidé ou réinitialisé lorsqu'un service qu'il utilise disparaît ou est modifié.

À l'exécution, le *handler* de dépendance de service combine ces propriétés afin de déterminer comment il doit chercher les services, lesquels il peut sélectionner, dans quel ordre, et aussi de quelle manière il peut se lier à ces services. Le *handler* va aussi gérer l'aspect dynamique des services en surveillant le registre de service. Les

apparitions de nouveaux services, les modifications de propriétés de service et les disparations sont toutes susceptibles de modifier l'état de la dépendance de service. Comme nous l'avons vu, une dépendance non satisfaite peut entraîner l'invalidation du composant entier et l'arrêt de son exécution.

Dans la version actuelle d'iPOJO, ces étapes sont entièrement automatisées par le conteneur de composant. Le code fonctionnel peut donc paramétrer le fonctionnement du **handler** de dépendance, mais aucune facilité n'est offerte pour sa gestion externe.

Les fonctionnalités précises d'une dépendance de service iPOJO, et sont comportement en environnement dynamique ont été détaillées dans [Escoffier 2013].

Fournitures de services

Un composant iPOJO peut fournir un ou plusieurs services. Le code fonctionnel contient des instructions permettant de spécifier les caractéristiques su ou des services fournis :

- les **interfaces fonctionnelles** du service, qui définissent les actions supportées par les services fournis.
- les **propriétés de services** qui décrivent l'état du service. Ces propriétés peuvent à la fois contenir des informations fonctionnelles sur le service, mais aussi des aspects non fonctionnels (qualité, localisation) qui vont guider le sélection des consommateurs de service. Ces propriétés peuvent être modifiées en cours d'exécution ; c'est même un des piliers de l'approche à service dynamique.
- la **politique de fourniture** du service qui détermine comment le composant réagit aux liaisons et utilisations des services qu'il fournit. Dans le cas général, le même objet peut servir à tous les consommateurs, mais il est possible de fournir un objet différent pour chaque session d'utilisation du service.

De la même manière que pour les dépendances de service, le *handler* de fourniture va combiner les caractéristiques décrites dans le code fonctionnel du composant et automatiser la publication du ou des services à l'exécution. La gestion du dynamisme est elle aussi automatisée : si le composant change la valeur d'une propriété de service, le *handler* va notifier le registre de service de cette modification.

Les mécanismes de fourniture et de dépendance de services permettent à iPOJO de proposer un modèle de développement simplifié. La préoccupation majeure est de gérer au maximum les aspects non-fonctionnels et/ou techniques des composants. Ceci permet de réduire drastiquement le nombre d'erreurs dans le code du composant, et d'améliorer significativement la qualité du logiciel ainsi fourni. La disposition des composants n'est pas explicite : la résolution des dépendances et les liaisons sont effectuées de façon tardive, à l'exécution. L'architecture est donc souple puisqu'elle peut s'adapter dynamiquement selon la disponibilité et les propriétés des services.

2.8 Conclusion

L'objectif de ce chapitre était de souligner quelques points cruciaux :

1. L'évolution de l'informatique a permis l'émergence d'une nouvelle vision qui fusionne monde informatique avec l'environnement du monde réel.
2. L'informatique ubiquitaire vise à fournir aux utilisateurs des informations et des services de manière transparente, au travers de l'environnement physique.

3. De nombreux domaines de recherche, tels que l'électronique, les télécommunications, les logiciels, sont plus ou moins directement impliqués dans la réalisation de cette vision.
4. La nature des environnements ubiquitaires impose aux applications un certain nombre de caractéristiques et de contraintes qui les rendent difficiles à développer et à maintenir.
5. Les solutions existantes ne répondent que partiellement aux problèmes posés pour le développement, l'exécution et la maintenance de telles applications. C'est ce qui motive ce travail de thèse.

Parmi les principaux griefs que l'on peut retenir concernant les approches existantes, le manque de la gestion du dynamisme est sans doute l'un des plus préoccupants. Un environnement ubiquitaire fait partie intégrante du monde réel, et est donc soumis à son évolution perpétuelle. Une application de ce monde doit pouvoir bouger, changer selon les circonstances et s'adapter afin de correspondre à l'environnement qui l'entoure et aux besoins des utilisateurs. Sans dynamisme au niveau applicatif, les environnements ubiquitaires ne peuvent offrirent la flexibilité nécessaire pour se fondre dans ce monde en mouvance.

Ce dynamisme est d'ailleurs un requis essentiel pour un autre aspect mal géré par les travaux présentés : la gestion de l'autonomie. Les environnements et applications ubiquitaires doivent être en mesure d'évoluer de manière autonome, sans la gouvernance continue d'un administrateur. Le chapitre suivant va présenter l'approche de l'informatique autonomique, qui tente de répondre aux défis de l'administration massive des systèmes.

De plus, les approches permettant de répondre aux défis lancés par la vision de l'informatique ubiquitaire se distinguent en deux grandes catégories :

- les approches spécifiques se basent sur la notion d'environnement ubiquitaire et fournissent les primitives permettant aux applications d'y interagir. Les applications qu'elles permettent de construire sont très fortement orientées. Ces types de travaux sont essentiellement exploratoires : les approches proposées sont expérimentales et ne sont généralement pas adaptées à une utilisation industrielle et/ou commerciale.
- les approches génériques permettent d'ajouter à des méthodes de développement existantes et éprouvées la gestion de tous les aspects techniques et/ou non-fonctionnels, dont ceux relatifs à l'informatique ubiquitaire. Les architectures à base de composants et à base de services sont déjà très largement employées en industrie ; leur souplesse et extensibilité permet généralement de les adapter afin de satisfaire les contraintes des applications ubiquitaires.

Le *middleware* Apache Felix iPOJO utilise l'approche des composants à services, permettant de soulager le développeur de la gestion des nombreux aspects techniques, tel le dynamisme. Il constitue à ce titre une base très pertinente pour la conception et le développement d'applications ubiquitaires.

Chapitre 3

Informatique Autonomique

Les environnements dans lesquels les systèmes informatiques modernes s'exécutent sont très exigeants. Leurs besoins spécifiques imposent des contraintes significatives sur la façon de concevoir et d'y exécuter les logiciels. Comme nous l'avons vu précédemment, les environnements pervasifs, qui illustrent parfaitement cette sophistication, constituent un véritable défi pour le domaine du génie logiciel. En effet, cette vision tend à mettre en symbiose l'informatique avec le monde réel, et oblige à radicalement changer la manière avec laquelle les applications sont conçues, développées et exécutées. Les impacts sur la phase d'exécution des logiciels, et leur maintien dans en état de fonctionnement, est aussi particulièrement affectée.

Ce chapitre se focalise sur le concept d'**informatique autonomique**[1]. Apparue dans un contexte centré sur la problématique de l'administration de larges systèmes, cette approche définit néanmoins des méthodes permettant de répondre aux besoins des applications s'exécutant en environnement complexe et dynamique.

1. traduction du terme originel *'Autonomic Computing'*, que l'on retrouve parfois aussi sous le terme d'informatique autonome. Dans ce manuscrit, nous utiliserons le terme d'« informatique autonomique ».

Sommaire

3.1 Complexité d'administration croissante

Les systèmes informatiques ont vu, au fil des années, leur complexité s'accroître considérablement. Le génie logiciel est apparu comme une réponse à la crise du logiciel, pour lutter contre cette inexorable croissance anarchique en structurant et formalisant la conception des logiciels. Comme mentionné précédemment (cf. chapitre 1), cette discipline découpe la mise en œuvre d'applications en plusieurs phases logiques. Une fois qu'une application est déployée, la phase de maintenance va permettre le suivi et l'évolution de l'application.

Comme nous allons le voir, les problématiques d'adaptation des applications aux environnements dynamiques concernent principalement la phase de maintenance. En effet, un logiciel doit pouvoir interagir avec toutes sortes d'entités la machine hôte et ses périphériques matériels, les ressources et services distants, les équipements mobiles ainsi que les utilisateurs locaux ou distants. La diversité et la dynamique de cet environnement d'exécution fait qu'il est impossible d'en connaître la portée lors de la conception initiale du logiciel. Telle que caractérisée par [Lehman 1980], cette classe de logiciels, les *E-programs*, est vouée à évoluer en permanence, sans quoi elle dégénérera inévitablement.

La maintenance est souvent définie comme la phase d'évolution d'un logiciel après sa livraison initiale :

> « *Modification of a software product after delivery to correct faults, to improve performance or other attributes, or to adapt the product to a modified environment.* »
>
> [IEE 1998]

> « *The maintenance process [...] is activated when a system undergoes modifications to code and associated documentation due to an error, a deficiency, a problem, or the need for an improvement or adaptation. The objective is to modify an existing system while preserving its integrity.* »
>
> [Singh 1996]

Parmi ces définitions se distinguent notamment différentes catégories de maintenances [Lientz 1980] :

- **corrective** : destinée à corriger une défaillance découverte (c'est-à-dire un *bug*).
- **perfective** : qui vise à améliorer le logiciel, pour augmenter ses performances par exemple, ou son esthétique visuelle.
- **adaptative** : pour garder le logiciel fonctionnel dans un environnement changé.
- **préventive** : pour prévenir d'éventuels problèmes ultérieurs.

L'ajout de fonctionnalités à un logiciel n'est pas directement cité dans cette liste des évolutions des logiciels. Selon les cas, elle pourra être vue comme une évolution adaptative, si la fonctionnalité résulte d'un changement dans l'environnement (utilisateur par exemple) ou cette fonction devient nécessaire, ou perfective si cette fonctionnalité améliore l'utilisabilité du logiciel.

Ces définitions, directement issues des travaux préliminaires sur le génie logiciel, sont focalisées sur la notion de **produit logiciel** (*'software product'*). Il est intéressant de constater que l'évolution des instances de ce produit logiciel, déployées sur des systèmes informatisés, ne sont que très peu considérées par ces travaux. Ceci est donc en complet déphasage avec les méthodes utilisées aujourd'hui pour développer et

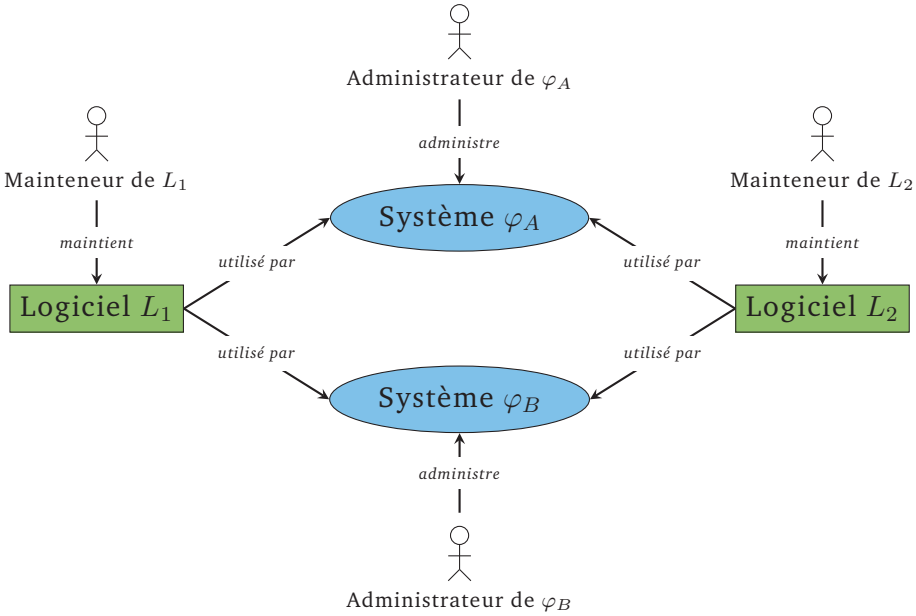

FIGURE 3.1 – Maintenance logicielle et administration système

mettre en exécution des systèmes dans des environnements complexes, distribués à large échelle, ou encore virtualisés dans le *Cloud*. C'est donc deux domaines de la maintenance qui divergent par leurs approches, bien que leur but soit finalement le même. Dans la suite de cette thèse, nous utiliserons les définitions suivantes afin de bien différencier les deux domaines :

Définition 1 — La maintenance logicielle est la modification du produit logiciel après sa livraison initiale, afin de corriger les défaillances de ce logiciels ou d'y ajouter de nouvelles fonctionnalités.

Définition 2 — L'administration d'un système est la tâche consistant à maintenir ce système en état de fonctionnement ou à le restaurer en cas de défaillance. fonctionnalités.

Bien que ces deux domaines soient continus en apparence, il apparaît certaines divergences qui ont peu à peu creusé le fossé entre la maintenance logicielle et l'administration. Comme le montre la figure 3.1, l'évolution d'un logiciel a un impact sur tous les systèmes où celui-ci est utilisé, alors que l'administration d'un système prend en compte la configuration et les interactions entre plusieurs logiciels, installés sur ce système. La principale contradiction est que le mainteneur d'un produit logiciel sera jugé principalement en fonction du nombre de fonctionnalités ajoutées à ce logiciel, alors que l'administrateur d'un système est lui primé sur la stabilité de ce système. Ce conflit entre évolution et stabilité est exacerbé par le fait que les équipes en charge de ces deux tâches appartiennent à des corps de métier généralement très différents. Certaines récentes approches, comme *DevOps* [2] tentent de réconcilier les deux approches de la maintenance en favorisant la constante communication entre ces deux aspects : maintenance et administration.

2. mot-valise, contraction de *'Development'* et *'Operations'*.
`http://en.wikipedia.org/wiki/DevOps`

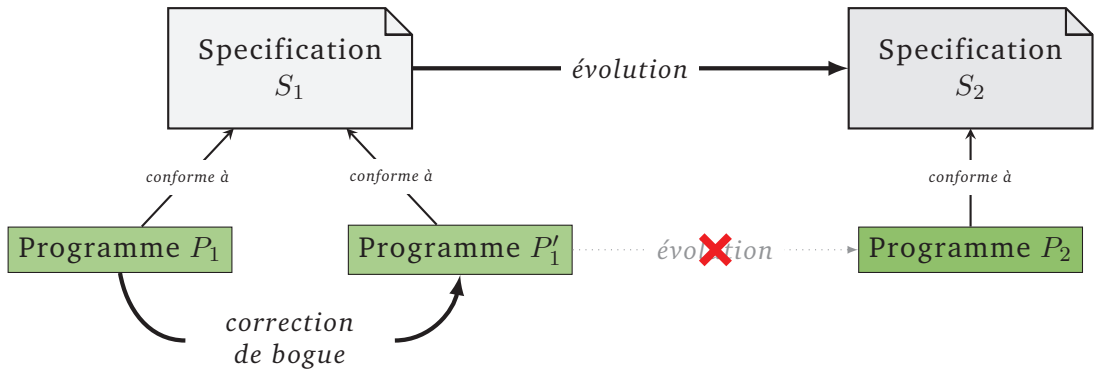

FIGURE 3.2 – Évolution des *S-Programs*

L'administration est donc par essence une tâche complexe, car elle doit à la fois faire face à l'évolution des logiciels qui composent un système, mais aussi gérer les changements dans l'environnement d'exécution qui affectent ce système. Parmi les diverses tâches d'administration, on retrouve par exemple :

- la supervision de l'état du système (souvent appelée monitoring)
- la mise à jour des applications ou du système d'exploitation, pour corriger des bogues ou des failles de sécurité,
- l'optimisation du système, pour par exemple diminuer les temps de réponse, ou desservir plus d'utilisateurs simultanément,
- la sauvegarde périodique de l'état du système, et la restauration après une panne,
- l'installation de nouveaux logiciels ou composants, pour fournir de nouvelles fonctionnalités,
- l'ajout, la suppression ou la modification des machines qui exécutent le système,
- la configuration de l'assemblage de logiciels composant le système.

Dans la suite de cette section, nous allons voir à quels enjeux et à quelles difficultés l'administration des systèmes doit faire face, et pourquoi cette situation posent de sérieux problèmes de stabilité des systèmes et de coûts de leur entretien. L'administrateur doit en effet gérer, parmi d'autres préoccupations, l'évolution du système, son intégration avec d'autres systèmes, le tout en maximisant sa disponibilité et ses performances.

3.1.1 Évolution des systèmes

Un des défis majeurs auxquels les systèmes informatiques et leurs administrateurs doivent faire face est l'évolution. Tout comme les êtres vivants, les systèmes et leurs logiciels évoluent dans un environnement dynamique et doivent pouvoir réagir pour s'adapter à ses changements. Ils ne disposent toutefois pas des mécanismes de réplication, de mutation et de sélection naturelle, nécessaires à cette évolution au sens biologique.

Cette considération de l'évolution des logiciels n'a pas toujours été prise en compte, ou a eu différentes significations au fil des âges. Par exemple, à la fin des années 1970, aux prémices de la programmation structurée et du génie logiciel, les systèmes étaient définis et conçus en fonction d'une spécification formelle, statique et immuable. Dans sa classification, Meir Lehman a nommé ce type de logiciel les *S-programs* [Lehman 1980]. Ces programmes évoluent dans un environnement strictement prévisible et défini par une spécification de programme. La marge d'évolution de

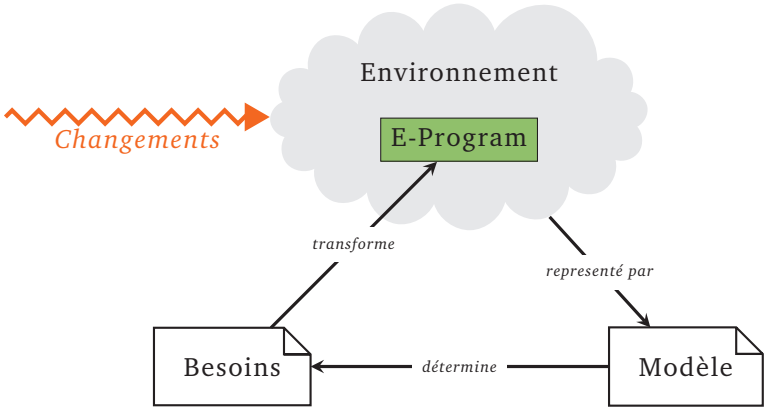

FIGURE 3.3 – Boucle d'évolution des *E-Programs*

tels programmes, et des systèmes qui les utilisent, est très réduite : principalement la correction des bogues ou l'amélioration des performances. L'ajout de fonctionnalités n'est pas possible sans modifier la spécification, ce qui revient à créer un programme différent, tel que dépeint par la figure 3.2. La vaste majorité des *S-Programs* que l'on peut rencontrer aujourd'hui est cantonnée à des fonctionnalités très réduites, car très simples à spécifier : calculs mathématiques, routines d'affichage sur un écran, tri de listes, *etc.*

Une autre catégorie de logiciels décrite par Lehman est celle des *P-Programs*, appelés ainsi parce qu'ils servent à résoudre un problème bien défini. Le problème est généralement assez abstrait, ou un modèle de représentation d'un problème du monde réel. Ce type de logiciels, bien que non fortement contraints par une spécification formelle, doivent considérer un ensemble de règles qui définissent le problème, les solutions fournies ne pouvant être évaluées sans connaître précisément le contexte du problème. Les *P-Programs* ne se focalisent donc pas sur la spécification du problème, mais sur la validité et la pertinence des solutions fournies par rapport au contexte actuel du problème Par exemple, les programmes de jeu d'échec, exemples typiques de *P-Programs*, sont jugés sur la qualité des mouvements proposés, qui sont évalués en fonction de la disposition actuelle de l'échiquier. Ces programmes étant contraints par des règles procédurales décrivant le problème, qu'ils ne doivent en aucun cas enfreindre, leurs perspectives d'évolution se portent surtout sur l'amélioration des solutions fournies, ou sur les performances pour y parvenir.

La catégorie d'applications qui caractérise le mieux les applications modernes est la classe des *E-Programs*. Celle-ci est définie par Lehman comme l'ensemble des applications qui sont en interaction directe avec le monde réel, celui où nous vivons. Le comportement de ces applications doit être totalement conditionné par l'état de cet environnement concret. C'est d'ailleurs principalement sur cet aspect que les *E-Programs* sont évalués : la capacité d'adaptation, réactive et proactive. Il en découle que l'évolution de ces applications est requis, pour s'adapter aux modifications d'un environnement d'exécution inconnu. Les applications pervasives sont une excellente illustration de cette catégorie de systèmes évoluant en contexte dynamique.

La figure 3.3 montre quelles sont les interactions entre un *E-Program* et son environnement d'exécution. Chaque changement dans cet environnement amène à une évolution du programme. Ce même programme peut influer sur l'état de l'environnement : directement, par le biais d'actionneurs, ou indirectement, en affichant des données susceptibles de modifier le comportement des utilisateurs par exemple. Le

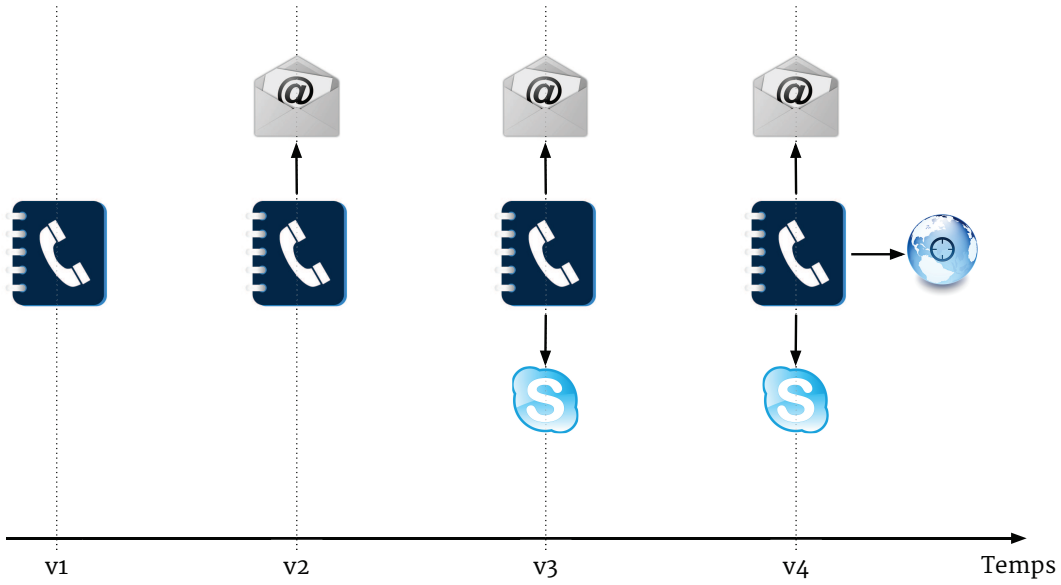

FIGURE 3.4 – Exemple d'évolution d'application

système et son environnement forment donc une boucle de rétroaction.

Parallèlement à sa classification de l'évolution des programmes, Lehman a formulé huit lois qui régissent cette évolution, parmi lesquelles on retrouve notamment :

Loi 1. **Continuing Growth** (croissance continue) : la taille du programme augmente sans cesse, au fur et à mesure que des fonctionnalités sont ajoutées pour maintenir un niveau d'utilisabilité acceptable.

Loi 2. **Increasing Complexity** (augmentation de la complexité) : l'évolution d'un système implique une croissance naturelle de sa complexité, si rien n'est fait pour la contenir.

Loi 3. **Declining Quality** (dégradation de la qualité) : la qualité des systèmes déclinent inexorablement sans une maintenance continue.

Ces trois points font que l'évolution des systèmes est une tâche d'administration indispensable, mais néanmoins complexe. Il est de plus nécessaire que cette prise en charge de l'évolution se fasse en permanence, sans quoi la qualité générale du système se dégrade rapidement.

Dans un contexte actuel où le nombre de dispositifs informatisés croît sans cesse, les logiciels prennent une place de plus en plus importante. Comme nous le verrons ci-après, la bonne évolution de ces logiciels est un besoin important, sinon primordial. Les trois lois de Lehman nous montrent cependant que cette évolution demande un effort permanent. Afin de réduire les coûts liés à la maintenance et à l'évolution des logiciels, il est donc nécessaire de :

- prévoir la capacité d'évolution du logiciel, dès le début de sa conception,
- soutenir l'évolution du logiciel durant toute sa phase d'exploitation,
- assurer une veille constante pour éviter ou atténuer les dérives citées dans les lois de Lehman.

Afin d'illustrer ce besoin d'évolution, considérons par exemple une application mobile, apparemment très simple, de répertoire de contacts (cf. figure 3.4). Une

première version de ce logiciel, très rudimentaire, pourrait se contenter, pour chaque entrée, de stocker un nom et un numéro de téléphone. En navigant dans le répertoire, il est dont possible de sélectionner un contact et d'engager un appel, ou d'envoyer un SMS [3]. Des années plus tard, les téléphones se sophistiquent et sont capables de se connecter à l'Internet ; notre application doit donc maintenant être capable de stocker une adresse e-mail, et de proposer un moyen d'envoyer un e-mail à un contact. L'année suivante, les progrès technologiques récents permettent d'engager des appels visiophoniques ; notre application pourrait proposer d'enregistrer, par exemple, les comptes Skype des contacts. L'intégration de récepteurs GPS [4] dans les téléphones actuels permet de nous avertir de la proximité d'un contact catalogué comme ami ; notre application doit pouvoir évoluer en ce sens.

Cette application, bien que volontairement simplifiée, illustre la constante nécessité d'évolution d'un système. Cette évolution permet de lutter contre l'obsolescence programmée du système, par exemple en intégrant les nouvelles possibilités offertes par l'environnement d'exécution du système.

3.1.2 Complexité d'intégration

Un autre enjeu majeur auquel doivent faire face les systèmes informatiques et leurs administrateurs est l'intégration. Si les premiers systèmes informatisés étaient isolés, les vagues de l'informatique distribuée et mobile ont rendu possible la communication entre systèmes, et l'ont largement favorisée. Il est de nos jours peu concevable de construire un environnement informatique fermé, sans possibilité d'interconnexion ou de communication avec l'extérieur. Chaque système devient un acteur à part entière de l'environnement, observant et participant à son évolution, coopérant avec les systèmes voisins.

Cet environnement d'exécution est donc complexe par nature, tissé de systèmes hétérogènes et dynamiques devant pourtant interagir de façon transparente. La prise en compte de tous les éléments qui forment cet environnement ambiant amène à concevoir des logiciels et des systèmes qui peuvent découvrir ces éléments, puis les intégrer. Seule cette symbiose peut conférer à l'environnement une apparence uniforme, naturelle et transparente.

Deux grandes difficultés se posent lors de l'interaction d'un système avec l'environnement qui le contient : la gestion de l'**hétérogénéité** et du **dynamisme**. L'hétérogénéité est due à la multitude et à la variété des acteurs à prendre en compte : utilisateurs, dispositifs communicants, appareils mobiles, services distants, *etc.* Chacune de ces entités apportent diverses formes d'interaction, sous différents formats, et il est nécessaire de s'adapter à chacun d'eux si on veut les comprendre. Le système devant communiquer ces objets doit donc fournir suffisamment de flexibilité pour permettre ces interactions. Le dynamisme est quant à lui un caractère intrinsèque de l'environnement réel dans lequel doit évoluer l'application. De nouvelles entités peuvent apparaître dans le périmètre d'influence de l'application, ou disparaître. Cela est d'autant plus vrai que l'informatique distribuée et mobile s'est popularisée : les perturbations du réseau ou la mobilité des dispositifs et de leurs utilisateurs affectent la disponibilité à certaines fonctionnalités. L'intégration de ces entités doit donc pouvoir se faire dynamiquement, dès qu'une opportunité d'utiliser ces entités apparaît possible et/ou nécessaire.

3. *Short Message Service*, populairement désigné sous le nom de « texto »
4. *Global Positioning System*, système de positionnement par satellite de portée mondiale

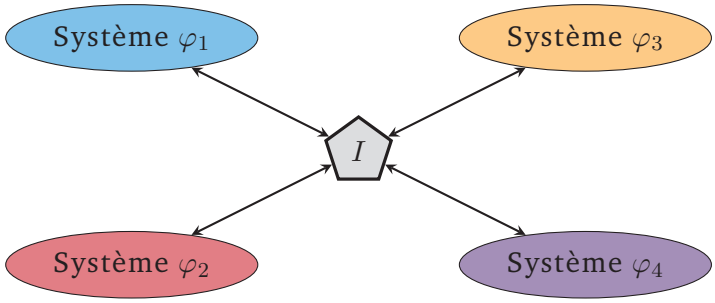

FIGURE 3.5 – Architecture centralisée pour l'intégration de systèmes

Une des architectures proposées pour l'intégration de systèmes est celle décrite dans la figure 3.5 : l'intégration centralisée. Des techniques d'intégration très couramment utilisées actuellement, telles que les ESB [5] ou JBI [6] établissent un bus central où sont routées et/ou traitées les données échangées entre les systèmes qui y sont reliés. Ainsi, chaque système est connecté au médiateur central, et évite de devoir gérer lui-même l'interconnexion avec tous les autres systèmes présents [Garcia Garza 2012]. Du point de la maintenance et de l'administration, cette architecture offre comme principal avantage un unique point de contrôle et de configuration. Cela évite à l'administrateur de devoir surveiller chaque système séparément : il a une vue d'ensemble de l'assemblage des systèmes [Wiederhold 1992]. En cas de problème, il peut en identifier les systèmes incriminés et intervenir séparément sur ceux-ci. Cette approche est très utilisée, notamment dans le domaine des applications d'entreprise, qui agrègent des systèmes patrimoniaux [7] afin de fournir de nouvelles fonctionnalités, sans le surcoût associé au changement de leurs infrastructures.

Cette approche apporte aussi un lot d'inconvénients, et il n'est pas toujours possible de l'appliquer littéralement. Par exemple, dans les environnements pervasifs, il est difficile de garantir la présence et la disponibilité permanente de ce médiateur central. Son absence oblige donc les systèmes présents à devoir gérer eux-mêmes cette intégration, amenant des problèmes de passage à l'échelle. De plus, la centralisation des interactions fragilisent l'ensemble : le médiateur central devient un point individuel de défaillance [8]. Le coût de mise en place de ce système central de médiation doit aussi être considéré, en plus de sa maintenance qui est capitale pour la communication inter-systèmes. L'approche d'intégration centralisée, si elle permet de découpler les différents systèmes entre eux, ne fait que déplacer le problème d'administration, en ajoutant le surcoût de mise en place et d'entretien du système d'intégration central.

Des approches d'intégration décentralisées, telles l'intégration réactive, ou basée sur les agents, répartisse la tâche d'intégration entre les différents systèmes, qui communiquent entre eux en utilisant des événements. Si les problèmes de la présence et de la vulnérabilité du médiateur central sont ainsi évités, le problème du passage à l'échelle rend difficile l'application de ces approches à des systèmes aux ressources limitées, tels que les réseaux de capteurs ou les dispositifs mobiles. De plus, comme le montre la figure 3.6, la complexité d'administration de cette fédération de systèmes croît très rapidement quand le nombre de systèmes à intégrer augmente. Il apparaît

5. *Enterprise Service Bus*
6. *Java Business Integration*
7. *'legacy systems'* en anglais
8. *'"Single Point of Failure"'* en anglais
9. Source : `http://www.mulesoft.org/documentation/display/current/Meet+Mule`

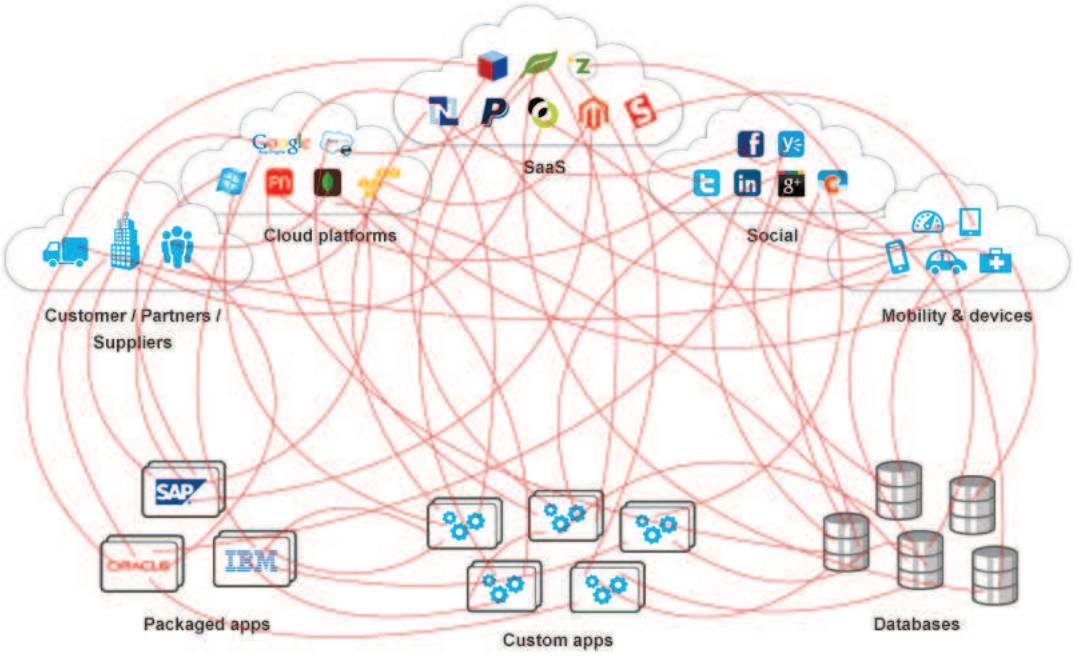

FIGURE 3.6 – Complexité d'une intégration décentralisée [9]

donc que l'intégration continuelle des systèmes est une tâche complexe, qui requiert des interventions de la part de l'administrateur.

3.1.3 Abolition des interruptions de service

L'utilisation de systèmes informatisés dans certains domaines amène divers risques, parmi lesquels les pannes des systèmes impliquant qu'ils ne puissent plus jouer leur rôle. Selon le secteur d'activité concerné, les conséquences de ces pannes peuvent occasionner des inconvénients pratiques, des pertes financières plus ou moins lourdes, voire même des catastrophes mettant en jeu des vies humaines. Parmi les systèmes dits **critiques** dont la défaillance n'est particulièrement pas souhaitable, on retrouve les systèmes bancaires gérant les transactions financières, les logiciels de contrôle du trafic aérien ou ferroviaire, les systèmes embarqués des satellites artificiels, *etc.*

Mais plus généralement, il existe des systèmes qui peuvent être qualifiés de critiques par les entreprises qui les maintiennent : système de gestion de la clientèle, des facturations, sites Internet, *etc.* Une défaillance dans ce type de système peut entraîner pour l'entreprise une cessation temporaire d'activité, des pertes financières importantes et ternir son image de marque. Ces systèmes sont très largement diffusés, et se trouvent donc être les premiers exposés à ce type de défaillance.

Beaucoup d'études se sont penchées sur les conséquences des pannes dans les systèmes informatiques de grande ampleur, utilisés à des fins commerciales. Le coût d'une interruption de service est en moyenne de 440 000 euros par heure pour une entreprise, et peut dépasser le milliard d'euros à l'échelle d'un pays [10]. Parallèlement, la difficulté ou l'impossibilité pour une entreprise d'accéder à ses données aurait

10. `http://www.silicon.fr/les-pannes-informatiques-coutent-tres-cher-aux-entreprises-22933.html`

```
$ uptime
23:00:06 up 3 days, 40 min,  2 users,  load average: 0,52, 0,42, 0,42
```

FIGURE 3.7 – Exemple d'*uptime* sur un système d'exploitation Linux

Taux de disponibilité		**Temps d'indisponibilité**		
en « neuf »	en %	par année	par mois	par semaine
« un neuf »	90 %	36,5 j	72 h	16,8 h
« deux neuf »	99 %	3,65 j	7,2 h	1,68 h
« trois neuf »	99,9 %	8,76 h	43,2 min	10,1 min
« quatre neuf »	99,99 %	52,56 min	4,32 min	1,01 min
« cinq neuf »	99,999 %	5,26 min	25,9 s	6,05 s
« six neuf »	99,9999 %	31,5 s	2,59 s	0,61 s
« sept neuf »	99,99999 %	3,15 s	0,26 s	0,06 s

TABLE 3.1 – Taux de disponibilité et temps d'indisponibilité

déjà coûté plus de 17 milliards d'euros à l'Europe [11]. Pour des entreprises reposant massivement sur la distribution de services sur Internet, les conséquences de pannes peuvent donc être catastrophiques. Ainsi, le site Internet *Amazon.com* a enduré des pertes allant jusqu'à 31 000 dollars par minute, pendant plusieurs heures [12].

Il existe plusieurs indicateurs qui permettent de mettre évidence la disponibilité des systèmes. Une des plus simples est l'*uptime*, qui caractérise la durée pendant laquelle une machine ou un service est disponible sans interruption. En cas de redémarrage, cette durée est remise à zéro. Cette durée peut être un indice précieux de la robustesse et de la stabilité d'un système. Par exemple, un système de fichier et d'impression est resté connecté pendant plus de 16 ans, avant que son disque dur ne succombe au poids des années [13]. Si l'*uptime* peut être un indicateur intéressant, il est toutefois insuffisant pour considérer sa disponibilité : une coupure du réseau pourra rendre un service indisponible sans pour autant affecter son *uptime*. La figure 3.7 illustre le temps d'uptime d'un système suivi d'autres indicateurs quantifiant la charge moyenne de ce système (*load average*).

Le domaine de la haute-disponibilité a défini plusieurs indicateurs précis permettant de caractériser la disponibilité d'un système. Par exemple, le **taux de disponibilité** durant une période donnée est défini comme le rapport entre la durée de disponibilité et la durée considérée. Le *MTTR* [14] est la durée moyenne de réparation des dysfonctionnements. Ces données statistiques sont indispensables pour avoir un aperçu global de la disponibilité d'un système, et pouvoir anticiper les frais causés par les défaillances plus ou moins fréquentes. Le tableau 3.1 montre quelles sont les taux de disponibilité les plus courants (mesurés en « neuf »), et quelles sont les périodes d'indisponibilité maximales autorisées pour maintenir ces taux de disponibilité. Le terme de haute-disponibilité est généralement employé pour des taux égalant ou

11. http://www.journaldunet.com/solutions/dsi/couts-indisponibilite-des-donnees-en-france.shtml
12. http://news.cnet.com/8301-10784_3-9962010-7.html
13. http://arstechnica.com/information-technology/2013/03/epic-uptime-achievement-can-you-beat-16-years/
14. *mean time to recover*, temps moyen jusqu'à la remise en service

dépassant les « quatre neuf » (99,99 %) [15].

Les techniques utilisées pour parvenir à fournir et maintenir un taux de disponibilité élevé sont souvent très lourdes. Il est en effet nécessaire de pouvoir anticiper des risques de natures très diverses, comme une coupure de courant, des attaques malveillantes, une catastrophe naturelle, ou même des dégâts militaires [16]. Bien que tous les systèmes ne soient pas aussi critiques, il convient, avant sa mise en production, de prévoir quels seront les conséquences des pannes, et d'accorder la robustesse, en répliquant les composants les plus fragiles par exemples. L'administrateur de systèmes hautement disponibles doit pouvoir intervenir à n'importe quel moment afin de remplacer un composant matériel ou logiciel, et donc de minimiser le risque ou la durée des dysfonctionnements.

3.1.4 Recherche d'optimalité

La recherche d'une meilleure disponibilité est un objectif majeur pour les administrateurs d'un système. Il est aussi très souvent souhaitable que ce système fonctionne de manière optimale. En effet, l'envergure de la diffusion de larges systèmes d'information, de traitement ou de stockage, fait qu'il peut être très coûteux de les entretenir. Ce coût fixe d'exploitation est autant causé par son administration que par sa consommation : électricité, réfrigération, bande passante, remplacement de pièces défectueuse, *etc.* Si l'évolution et la modernisation de l'informatique permet de produire des systèmes de plus en plus performants, la demande est telle que la consommation de ces systèmes croît sans cesse. Jonathan Koomey a ainsi mesuré que la consommation électrique des *data centers* avait doublé entre 2000 et 2005 [Koomey 2007].

La raison première pour laquelle un système devrait être optimisé est donc la rentabilité de son exploitation. Par exemple, un service optimisé permettre de réduire l'impact énergétique lors de son utilisation par un client. De plus, le système pourra accueillir plus de clients simultanément, et donc potentiellement générer plus de bénéfices. Cette optimisation est particulièrement intéressante pour les systèmes avec un trafic très important On retrouve ainsi en première ligne les sites Internet de réseaux sociaux, les moteurs de recherche et les sites d'actualité. *Facebook*, par exemple, compte plus d'un milliard d'utilisateurs, et reçoit plusieurs centaines de millions de visites par mois. Face à un tel volume d'utilisateurs et de visites, il est très intéressant d'optimiser le service fourni ; même une très légère amélioration causera des économies substantielles, tant le coefficient multiplicateur est élevé.

La réduction du gaspillage de ressources, telle que la puissance de calcul non-utilisée, est aussi un moyen efficace d'optimiser un système informatique. L'utilisation moyenne d'un serveur informatique de dépasserait pas les 6 % [17]. Il peut donc être intéressant pour une entreprise de mutualiser ses ressources informatiques avec des partenaires, afin de réduire et de partager les coûts de maintenance et d'administration. L'approche du *cloud computing* [18] prône la virtualisation des systèmes informatiques, ceux-ci étant en fait localisés chez des hébergeurs spécialisés qui prennent charge la maintenance matérielle, et souvent une partie de la maintenance logicielle. Cette approche permet au propriétaire du système de s'affranchir de l'acquisition et de

15. `http://www.glohse.com/?tag=uptime-institute`

16. Selon le mythe très répandu, *ARPAnet*, l'ancêtre de l'Internet, aurait été conçu dans le but d'assurer les communications même en cas de frappes nucléaires, en contexte de pleine guerre froide.

17. `http://www.lemondeinformatique.fr/actualites/lire-les-datacenters-veritables-gachis-energetiques-25993.html`

18. informatique en nuage, ou informatique dématérialisée

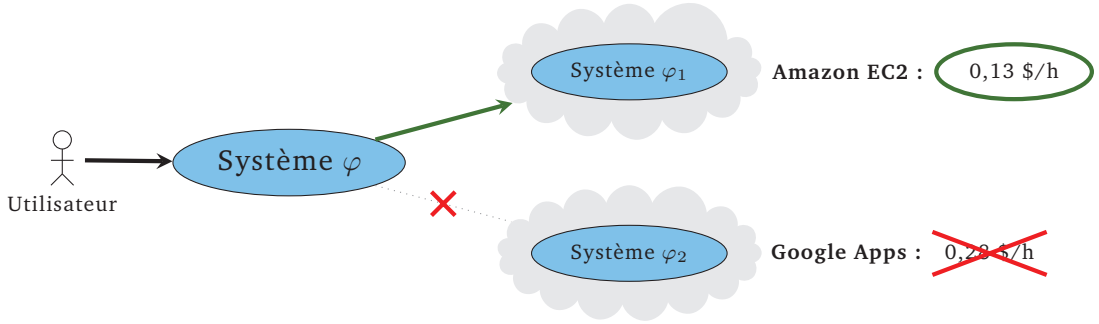

FIGURE 3.8 – Optimisation du coût de fonctionnement d'un système sur le cloud

l'entretien d'un parc informatique souvent très coûteux, et finalement utilisé bien en dessous de ses possibilités. Il doit alors payer à l'hébergeur ce que son système consomme réellement, en terme de puissance de calcul, d'espace mémoire et de trafic réseau : c'est le principe du *pay-per-use*. Si cette approche permet de faire des économies, les systèmes doivent être optimisés afin que la facture de l'hébergeur soit la moins coûteuse possible. La figure 3.8 montre une manière d'adapter dynamiquement le choix de l'hébergeur en fonction du prix courant de fonctionnement.

D'autres considérations peuvent aussi amener à optimiser son système, comme la réduction de l'empreinte écologique des systèmes. Le mouvement du *Green IT* [19] se penche sur cette problématique. Tout le cycle de vie du système d'information est quantifié par rapport à son impact sur l'écosystème : toxicité des composants matériels, consommation énergétique, gaspillage de ressources, gestion des déchets, *etc.* [Melville 2010], en cette période où l'éco-responsabilité devient un besoin préoccupant [20]. Si le choix de matériel économique et écologique se fait lors de la conception du système, la surveillance de la consommation énergétique et sa régulation est la tâche de l'administrateur. Celui-ci doit aussi veiller à ce que les ressources soient exploitées au maximum, pour ne pas gaspiller inutilement de la puissance de calcul, par exemple.

La notion de *smart grid* se place du côté des fournisseurs d'énergie. L'électricité étant une énergie difficile à stocker, il est nécessaire de connaître au mieux les besoins énergétiques des consommateurs afin d'adapter la production et ainsi éviter au maximum les gaspillages et prévenir les éventuels *blackouts*. Si les énergies renouvelables, celles ayant la plus faible empreinte écologique, commencent à se répandre en Europe, la production intensive d'électricité, notamment en période de pointe, demande parfois la mise en route de centrales thermiques brûlant des combustibles fossiles. Une des méthodes couramment utilisée pour « lisser » la consommation est la pratique de tarifs différenciés [21] selon la période de consommation. Pour l'administrateur d'un système, il devient donc nécessaire d'adapter la charge du système en tenant compte du coût instantané de l'électricité, par exemple en décalant dans la nuit les tâches non-urgentes et très consommatrices (sauvegardes, statistiques journalières, *etc.*).

Enfin, il existe des domaines d'application ou l'optimisation n'est plus une option, mais une véritable nécessité. Certains domaines reposent sur l'optimisation de leurs

19. aussi appelé *Green Computing*, informatique verte, ou informatique écoresponsable
20. La présentation du premier volet du V^{e}rapport du GIEC (Groupe d'experts intergouvernemental sur l'évolution du climat) confirme définitivement le rôle des activités humaines sur le réchauffement climatique.
21. tarification heures pleines pour les périodes de grande consommation, et tarification heures creuses.

systèmes et la mette en application dans un contexte de concurrence agressive. Par exemple, les systèmes de *high frequency trading* [22] fournissent le moyen de passer des ordres de commandes dont la durée d'exécution ne dépasse pas les 150 microsecondes. Les clients de tels systèmes peuvent alors suivre en temps réel, à très faible latence, les cous du marché. La rapidité avec laquelle les programmes clients peuvent détecter une situation avantageuse, planifier des ordres et les mettre à exécution est cruciale : c'est l'algorithme le plus rapide qui est susceptible de générer le plus de profit. Il faut néanmoins pondérer ce potentiel profit avec le coût des ressources nécessaires pour l'atteindre. Au final, c'est dont le programme avec le rapport rapidité d'exécution sur ressources nécessaires le plus élevé qui se révélera stratégiquement le plus intéressant.

Le fait de pouvoir améliorer le rendement des systèmes permet de mieux rentabiliser leur coût d'exploitation, voire de dégager un profit. Cette optimisation est un choix qui doit être prévu dès la conception des systèmes, mais qui a un impact sur toute sa durée de vie, et donc sur son entretien et sa maintenance. L'administrateur a donc le rôle d'adapter le système afin de réduire son coût ; ces opérations se manifestant principalement par le remplacement du matériel obsolète, la temporisation d'activités consommatrices d'énergie ou la virtualisation d'une partie ou de la totalité du système dans le nuage.

3.1.5 Des erreurs humaines inévitables

Pour toutes les raisons développées précédemment, l'administration d'un système est une tâche complexe. Les administrateurs sont très sollicités par les changements dans l'environnement d'exécution, et doivent être prêts à adapter ou réparer le système à tout moment. Le manque d'adaptation fait dériver naturellement le système vers un déphasage avec le contexte réel, ce qui peut entraîner des défaillances ou un état sous-optimal, et dans tous les cas un effort ultérieur pour rattraper ces dérives :

> « *As an evolving program is continually changed, its complexity, reflecting deteriorating structure, increases unless work is done to maintain or reduce it.* »
>
> [Lehman 1980]

Les conséquences d'éventuelles défaillances peuvent être désastreuses, autant économiquement qu'humainement, selon la nature des systèmes concernés. Une métaphore souvent évoquée quant à la dérive des systèmes évolutifs est celle de la *dette technique* [23] [Cunningham 1993]. La dette technique est la mesure de la dérive d'un système au cours de son évolution, et elle caractérise l'effort nécessaire pour ramener ce système dans un état « idéal », qui facilite son évolution. Ainsi, chaque choix qui est fait lors de l'évolution du système est susceptible d'augmenter cette dette technique, selon les priorités qui sont fixées [Brown 2010].

Parmi les causes les plus communes de dette technique, on retrouve le manque de documentation, la non conformation aux standards de codage, le manque de vérification de certains paramètres pouvant affecter la sécurité du système, *etc.* Certains outils, tels que *SonarQube* [25] permettent de visualiser et de quantifier cette dette technique, ainsi que de scruter sa progression tout au long de l'évolution du logiciel. Les

22. transactions à haute fréquence
23. *technical debt* en anglais
24. Source : `http://nemo.sonarqube.org/dashboard/index/382019`, au 14 décembre 2013
25. `http://www.sonarqube.org`

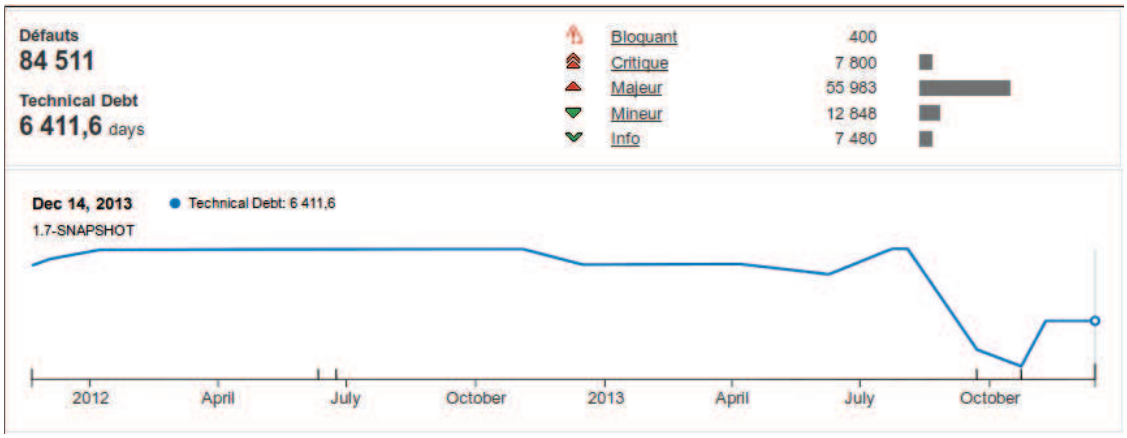

FIGURE 3.9 – Dette technique du projet *OpenJDK 7* [24]

principales sources d'information permettant de calculer cette dette sont l'analyse statique du code source du logiciel ainsi que les résultats de ces tests (de couverture principalement). Les critères permettant de mesurer cette dette technique sont toutefois assez nombreux, et doivent être affinés selon le logiciel considéré. Par exemple, la figure 3.9 montre en exemple d'estimation de la dette technique du projet *OpenJDK 7* [26]. Cette dette, exprimée en nombre de jours d'effort à fournir, est d'environ 6 416 jours, pour pouvoir résoudre les 84 511 défauts détectés, ce qui représente un travail (et un investissement) colossal. Ces outils se focalisent généralement sur le projet logiciel, et il est difficile de transposer ces méthodes et ces critères de mesure à des systèmes en production, basés sur ces mêmes logiciels. Dans l'exemple ci-dessus, rien ne permet d'estimer la dette technique des nombreux systèmes en production utilisant *OpenJDK 7*.

Les administrateurs aussi avoir une expertise complète des systèmes qu'ils doivent entretenir. Pour certains systèmes anciens, cela peut poser certains problèmes : les administrateurs maîtrisant les technologies oubliées sont rares et coûteux. On peut par exemple citer la pénurie d'experts *COBOL* [27] dans le milieu bancaire. Une grande majorité des systèmes bancaires utilisent encore des programmes anciens développés dans ce langage, qui n'est quasiment plus enseigné aujourd'hui. Les experts *COBOL* se font donc de plus en plus rares, mais sont cependant très recherchés pour maintenir et administrer les systèmes bancaires vieillissants, pourtant vitaux. L'expertise des administrateurs a donc un prix, qui est difficile à anticiper.

Les administrateurs ont donc beaucoup d'aspects à gérer : évolution, intégration, disponibilité et optimisation des systèmes. Il y a cependant encore bien des facettes du métier de l'administration. Parmi celles-ci, l'aspect juridique doit être considéré : l'administrateur est responsable du système informatique, et de l'utilisation qui en est faite. Il peut être mis en cause si le système a commis, ou a servi à commettre des infractions, comme le téléchargement de fichiers soumis aux droits d'auteur [28]. Il peut aussi devoir garantir la confidentialité des données personnelles des utilisateurs, et donc être poursuivi en cas de fuite de ces informations [29]. Pour les systèmes financiers,

26. http ://openjdk.java.net/projects/jdk7/
27. *COmmon Business Oriented Language*, langage de programmation structuré très employé les années 1960 à 1980
28. `http://2003.jres.org/actes/paper.130.pdf`
29. `http://www.foruminternet.org/telechargement/documents/ca-par20011217.pdf`

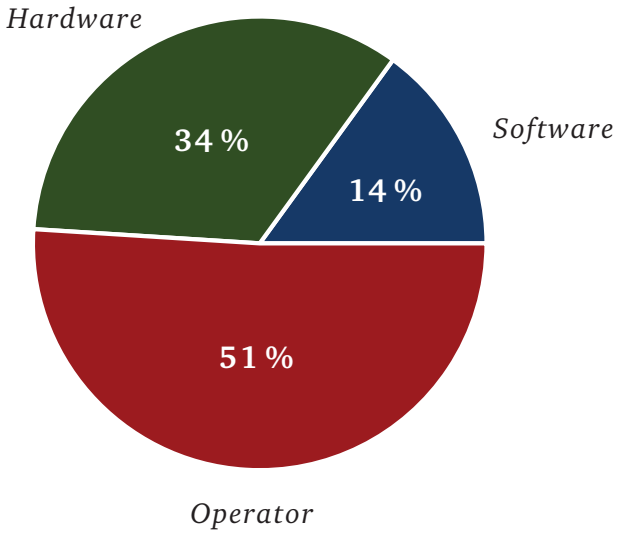

FIGURE 3.10 – Causes d'erreur pour trois sites Internet en 2000 [Patterson 2002]

l'administrateur a l'obligation légale de produire un audit journalier des transactions effectuées. Cet audit va permettre à l'autorité de régulation des marchés de pouvoir détecter d'éventuelles fraudes.

La diversité des activités de l'administrateur et son rôle indispensable font qu'il est la clé de voûte de la survie et de l'évolution du système. Sans lui, le système serait voué à stagner et à défaillir à la moindre panne. Paradoxalement, l'administrateur est aussi l'une des principales causes de ces pannes : l'erreur est humaine, et la sollicitation parfois extrême de l'administrateur peut le pousser à commettre des inadvertances. La figure 3.10 résulte d'une étude comptabilisant le nombre d'erreurs sur trois sites Internet pendant six mois, relevant à chaque fois la cause de ces erreurs [Patterson 2002]. Elle nous montre que plus de 50 % des erreurs sont causées par l'administrateur lui-même. Les administrateurs sont donc la cause la plus fréquente des pannes de systèmes informatiques, et multiplier la charge de travail ne ferait qu'augmenter ce taux d'erreur. S'il est impossible d'empêcher ces erreurs humaines [30], l'allègement des tâches d'administration permettrait au moins de les réduire. De nombreux outils d'administration existent, visant à assister l'administrateur dans ses tâches quotidiennes, et à automatiser les actions répétitives (les plus sujettes aux erreurs). Malgré tout, l'administrateur emploie très facilement des méthodes dites manuelles : des programmes ou scripts faits par lui-même, spécifiques au système. Ces « recettes maison » d'administrateur, bien que souvent très efficaces et ciblées, rendent opaque la compréhension globale du système. L'administrateur devient donc irremplaçable parce qu'il devient le seul gardien du savoir concernant le système. Ces méthodes manuelles sont aussi une des principales sources d'erreur causées par les administrateurs.

D'une manière plus humoristique, un des corollaires de la « célèbre » loi de Murphy [31], appelée seconde loi de Gilb de la non-fiabilité, stipule que « tout système dont la fiabilité dépend d'un être humain n'est pas fiable ». L'informatique autonomique, comme nous allons le voir, va s'attacher à réduire cette dépendance du système sur

30. `http://wwwv1.agora21.org/ari/nicolet1.html`
31. *'Anything that can go wrong, will go wrong'*, Edward A. Murphy Jr.

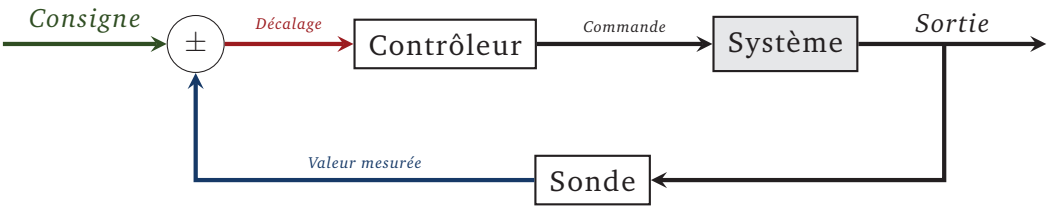

FIGURE 3.11 – Boucle de rétrocontrôle en automatique

l'administrateur, en le rendant plus autonome.

3.2 Informatique autonomique

L'informatique autonomique est une vision apparue alors que les défis liés à l'administration des systèmes semblaient incontournable. Dans un manifeste publié en 2001, Paul Horn a expliqué pourquoi la façon d'administrer les systèmes informatiques devait changer à terme [Horn 2001]. La complexité des systèmes devient telle qu'il est très difficile pour l'administrateur de maîtriser l'ensemble. Cette difficulté freine l'évolution et altère la cohésion globale des systèmes. C'est donc pour répondre à cette problématique préoccupante que l'approche autonomique a été énoncée. En quelques mots, elle vise à donner plus d'**autonomie** à un système afin qu'il puisse prendre des décisions de lui-même, et donc délester l'administrateur d'une importante charge de travail.

3.2.1 Inspirations et origines

Parmi les inspirations de l'informatique autonomique, on retrouve, comme dans beaucoup d'autres inventions humaines, des concepts tirés de l'étude du vivant. Ainsi, les mammifères disposent d'un système nerveux spécifique, appelé système nerveux autonome [32] qui prend en charge les activités inconscientes de l'organisme. Digérer, respirer, faire battre notre cœur, contrôler notre pression sanguine sont autant d'activités vitales dont nous ne nous préoccupons guère. Cette partie du cerveau va donc assurer le bon fonctionnement de nos fonctions les plus vitales, tandis que notre cerveau conscient va pouvoir se concentrer sur des activités plus complexes, comme chercher de la nourriture par exemple. L'arc réflexe est un exemple de réaction inconsciente encore plus basique, l'influx nerveux ne passant même plus par l'encéphale. Dans la vision d'*IBM* de l'informatique autonomique, l'administrateur joue le rôle du cerveau conscient du système, tandis que le système doit pouvoir gérer de façon autonome les adaptations vitales les plus basiques.

D'autres domaines ont inspiré la vision de l'informatique autonomique. La théorie du contrôle, très utilisée en automatique, définit des règles afin d'adapter un système en fonction de sa dynamique et de celle de son environnement. Dans le cas d'un système instable ou d'un environnement dynamique, difficile à modéliser, le contrôle en boucle fermée permet de prendre en compte la réponse du système par rapport à la consigne donnée. La rétroaction négative permet de prendre en considération le décalage entre l'objectif du système et l'état actuel de l'environnement. Selon

32. en anglais : *'Autonomic nervous system'*

l'écart mesuré, le système, piloté par un contrôleur, agit sur l'environnement jusqu'à s'approcher le plus possible de son objectif, comme le montre la figure 3.11.

L'informatique autonomique n'est pas la première approche visant à donner plus d'autonomie aux systèmes informatiques. Certains composants matériels et certains programmes exposent depuis longtemps un certain caractère autonome. Les mémoires *ECC* [33] sont par exemple capables de corriger automatiquement un certain nombre d'erreurs de corruption de données. Edsger Dijkstra a mené des travaux précurseurs dans le domaine des réseaux auto-stabilisés : ces derniers peuvent s'ajuster selon les perturbations, la charge, les nouvelles routes disponibles[Dijkstra 1986]. Certains logiciels montrent aussi des caractéristiques autonomiques : les anti-virus, clairement inspirés du domaine de l'immunologie, sont capable de détecter des fichiers potentiellement dangereux et de les neutraliser, sans aucune action de l'utilisateur.

Si des comportements autonomes ont émergé bien avant 2001, ils n'ont concerné que des systèmes informatiques très réduits, spécifiques, ou proposé des adaptations très simples. Une exception cependant : la catégorie des systèmes auto-adaptatifs semblent correspondre à cette volonté d'insuffler une certaine autonomie aux systèmes informatiques. De nombreux travaux précurseurs ont étudié et formalisé cette classe de système, et décrit comment un système peut s'adapter, par exemple, en changeant son architecture ou en reconfigurant certains de ses composants [Bradbury 2004, Oreizy 1999]. Certaines de ces opérations sont périlleuses lorsque le système est en cours de fonctionnement ; il faut alors isoler les zones impactées pour les manipuler [Kramer 1998]. D'autres approches, telles la **tranquillité**, profitent de l'inactivité temporaire d'un composant pour effectuer des adaptations sur celui-ci, sans devoir paralyser une partie du système [Vandewoude 2006]. Les systèmes auto-adaptatifs constituent le fondement pratique de l'informatique autonomique, jetant les bases techniques permettant aux logiciels et aux systèmes de participer à leur propre évolution.

C'est sur ce socle d'influences diverses que l'informatique autonomique a proposé une nouvelle vision pour résoudre le problème de l'administration de systèmes complexes et dynamiques

3.2.2 Définitions

Comme nous allons le voir, l'informatique autonomique couvre un spectre très large de préoccupations, et définit plusieurs caractéristiques complémentaires. Il existe donc une grande variété de définitions, reposant sur ses caractéristiques élémentaires, et influencées par les préoccupations considérées.

> « *The essence of autonomic computing is self-management, the intent of which is to free system administrators from the details of system operation and maintenance and to provide users with a machine that runs at peak performance 24/7.* »
>
> [Kephart 2003]

Dans cette définition, on retrouve bien cette préoccupation première de soulager l'administration des systèmes des détails techniques. La disponibilité et l'optimisation du système est aussi abordée, un utilisateur devant être en mesure de l'utiliser en tout temps, au maximum de ces capacités.

33. *Error-correcting Code Memory*, mémoire à correction automatique

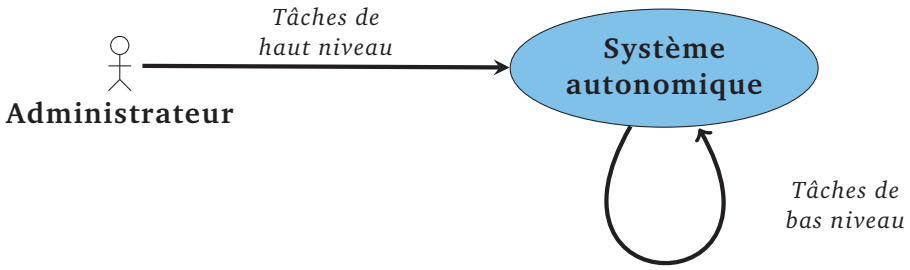

FIGURE 3.12 – Système autonomique

> « *The aim [is to] create the self-management of a substantial amount of computing function to relieve users of low level management activities, allowing them to place emphases on the higher level concerns of running their business, their experiments or their entertainments.* »
>
> [Sterritt 2005]

Cette autre définition concentre l'attention de l'utilisateur sur des préoccupations de haut niveau, le système devant gérer lui-même les « basses besognes ». On peut y voir un parallèle intéressant avec l'informatique ubiquitaire (cf. chapitre 2), où l'attention de l'utilisateur de doit pas être détournée par des préoccupations non-fonctionnelles. Ici, le rôle de l'administrateur et de l'utilisateur sont confondus. Selon les domaines, il peut être important de bien différencier les deux rôles, tant ils peuvent être cloisonnés.

> « *The Autonomic Computing Paradigm has been inspired by the human autonomic nervous system. Its overarching goal is to realize computer and software systems and applications that can manage themselves in accordance with high-level guidance from humans.* »
>
> [Parashar 2005]

Ici, le système doit, tel le système nerveux autonome, s'autogérer en accord avec les buts de haut niveau des humains qui l'utilisent. En fait, toutes ces définitions s'accordent à dire que l'humain ne doit pas gérer lui-même tous les aspects techniques liés à l'administration d'un système. Un système autonomique offre une vue simplifiée à ses utilisateurs, où les fonctions basiques, voire vitales, de bas niveau sont satisfaites de manière autonome. Seuls des objectifs de haut-niveau sont présentés à l'humain, en accord avec ses préoccupations, et tel que dépeint par la figure 3.12. Ceci lui permet donc de se concentrer sur ce qu'il veut faire avec le système, sans devoir se soucier de comment il va pouvoir le faire.

Ces définitions évoquent certains mots-clés qu'il convient, dans un souci de clarté, de préciser :

– l'**utilisateur** du système est la personne qui doit interagir avec le système. L'informatique autonomique vise à rendre cette interaction plus naturelle. Bien que les utilisateurs finaux [34] du système soient des entités qui méritent d'être considérées, l'informatique autonomique se focalise sur l'administration des systèmes.

34. traduction littérale du terme commercial anglais *'end user'*.
`http://en.wikipedia.org/wiki/End_user`

Dans ce contexte, le principal utilisateur des systèmes est donc l'**administrateur**. Celui-ci n'a donc pas à se soucier de comment le système fonctionne, ni comment il doit faire pour le maintenir en état de fonctionnement. C'est au système lui-même de saisir les intentions de l'administrateur et s'adapter en conséquence, tout en prenant en compte son contexte d'exécution actuel. Nous verrons ci-après que les utilisateurs, et plus particulièrement leurs objectifs, sont, du point de vue du système, un constituant à part entière du contexte de l'application autonomique.

– les **interactions** avec le système autonomique sont définies en fonction de ce que l'utilisateur souhaite, du but qu'il cherche à accomplir ou de l'état qu'il veut appliquer au système. Selon les différentes définitions, ces interactions prennent le nom de buts, de politiques, ou encore d'intentions. Si les approches ou les interprétations divergent, toutes s'accordent à dire que c'est la nature de ces interactions qui est le fondement même de l'informatique autonomique. Les mécanismes internes du système doivent être exclus de ces interactions, seule doit apparaître les concepts importants pour l'utilisateur. La nature de ces interactions permet donc un **découplage conceptuel** entre l'utilisateur et le système.

Pour synthétiser cet idéal de l'informatique autonomique, nous proposons la définition suivante :

Définition 3 — L'informatique autonomique est la discipline visant à élaborer des systèmes informatiques autonomiques, c'est-à-dire étant capable de s'adapter à leur contexte d'exécution en fonction de préoccupations de haut-niveau.

L'informatique autonomique a pour but de rendre des systèmes autonomiques. Si cette assertion semble assez évidente dans les définitions proposées, la caractérisation de système autonomique n'est pas aussi unanime. Il y'a néanmoins trois axes récurrents définissant les systèmes autonomiques :

1. ils se doivent de s'adapter au contexte d'exécution. Cependant, l'interprétation du contexte d'exécution est ironiquement assez dépendante du contexte du système, de son domaine d'application.
2. ils sont autogérés : les mécanismes d'adaptations sont réalisés par le système lui-même. Il existe bien entendu divers degrés d'interprétation de ce que signifie cette autogestion, que nous verrons par la suite.
3. ils sont guidés par des actions, buts ou politiques de haut niveau. Ces termes étant assez abstraits, ils mènent à diverses interprétations.

Les points 1 et 2 correspondent aux critères caractérisant les systèmes auto-adaptatifs, évoqués dans la section 3.2.1. Le point 3 marque cependant une différence importante entre les deux domaines. Là où les systèmes auto adaptables s'adaptent de leur propre chef, les systèmes autonomiques suivent un ordre établi, inspiré par une vision de haut niveau de l'administrateur.

Les définitions de ce qu'est un système autonomique varient grandement selon les interprétations. On retrouve plusieurs visions de ce qu'est un système autogéré :

– « *Each autonomic element will be responsible for managing its own internal state and behavior and for managing its interactions with an environment that consists largely of signals and messages from other*

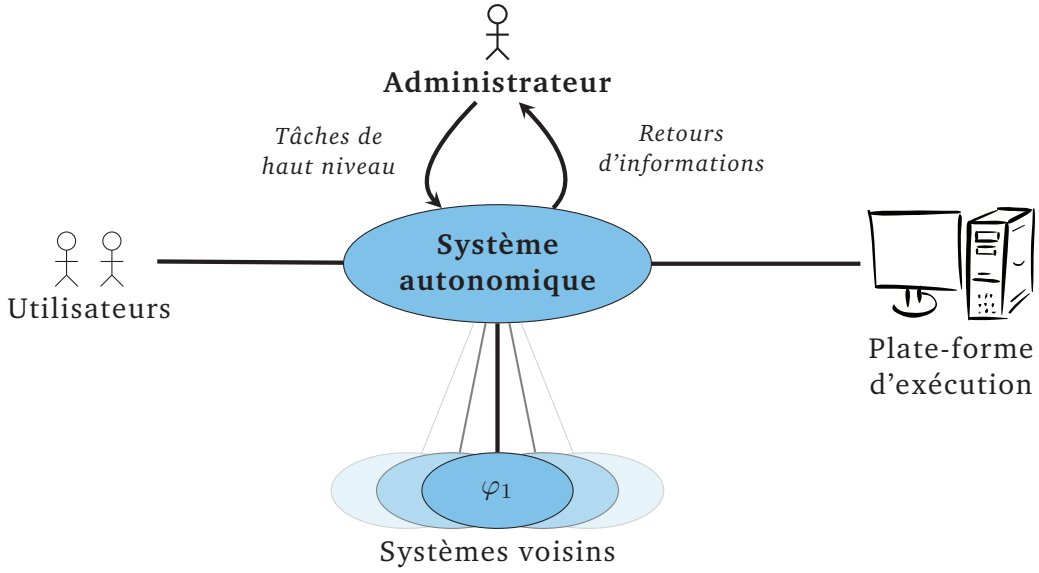

FIGURE 3.13 – Contexte d'un système autonomique

> *elements and the external world.* »
>
> [Kephart 2003]

– « *[The] term autonomic computing is [...] being used to describe [...] systems [...] in which the query plan, resource management or routing decision changes to reflect the current environmental context ; reflecting dynamism in the system.* »

  [Huebscher 2008]

D'après ces définitions, un système autonomique doit se préoccuper de son état interne et de l'environnement dans lequel il évolue. Ces sources d'information, indispensable pour qu'un système autonomique puisse s'autogérer, forment le **contexte** du système. L'état interne est très important à considérer pour le système autonomique, et va participer à dégager cette « notion de **soi** », qui le différencie de l'« autre », c'est-à-dire l'environnement extérieur, pouvant inclure d'autres systèmes (autonomiques ou non). Si l'on considère les buts de haut-niveau de l'utilisateur, la nature du contexte d'un système autonomique est très proche de celle de contexte utilisée en informatique ubiquitaire (cf. chapitre 2). On retrouve ainsi, entremêlées, les trois composantes majeures du **contexte ubiquitaire**, tel que défini par Bill Schilit [Schilit 1994] : l'environnement physique, l'environnement de l'utilisateur et l'environnement d'exécution.

Cette décomposition du contexte peut cependant être affinée, en considérant les entités en rapport avec un système autonomique, illustrée par la figure 3.13 :

- l'**administrateur** est l'interlocuteur principal du système autonomique. Ses intentions doivent être clairement connues du système, exprimées sous la forme de buts de haut niveau.
- les **utilisateurs finaux** du système ne perçoivent pas directement l'aspect autonomique du système qu'ils utilisent. Le système doit lui, par contre, prendre en compte le contexte d'utilisation pour, par exemple, se protéger d'actions malveillantes.
- la **plate-forme d'exécution** est le support permettant le fonctionnement du

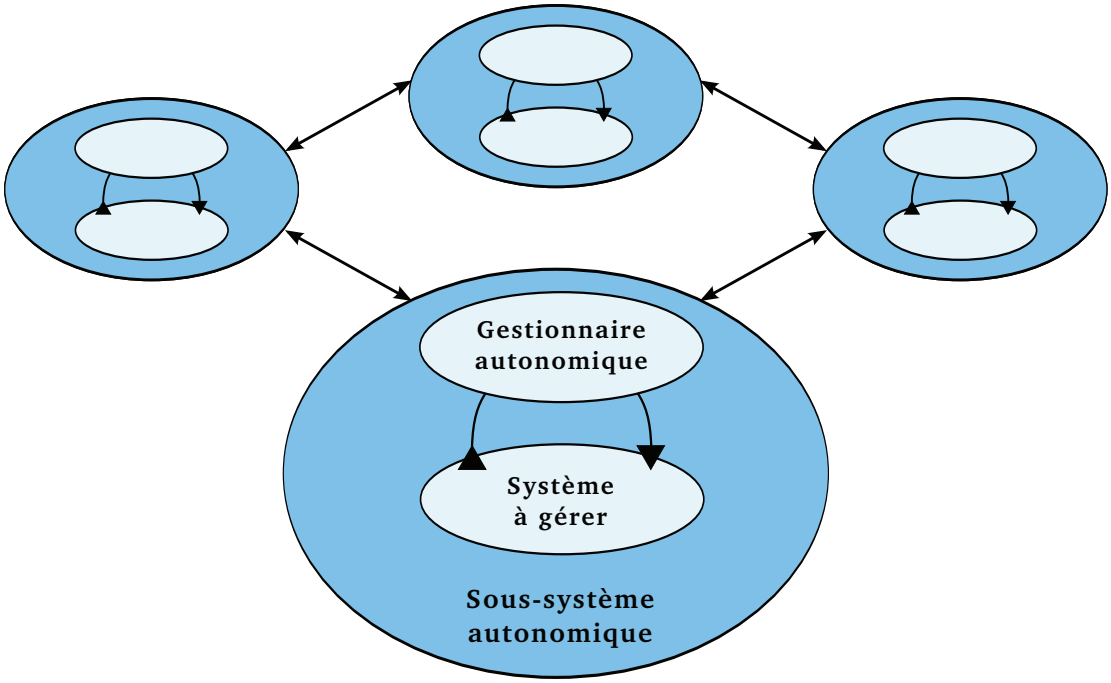

FIGURE 3.14 – Systèmes et éléments autonomiques

système autonomique. La base matérielle et logicielle doit être surveillée par le système, afin de devoir réagir à une pénurie de mémoire, ou l'apparition d'un nouveau réseau sans fil, par exemple.
- les **systèmes voisins** peuvent influer sur le comportement du système : certaines ressources doivent par exemple être partagées, ou des actions coordonnées.
- le **système autonomique** lui-même qui doit être surveillé et adapté en fonction des sources de contexte précédentes.

Ce contexte forme donc la base de connaissance nécessaire à la compréhension et à l'application des politiques d'autogestion du système. À partir de cette énumération du contexte, nous proposons la définition suivante d'un système autonomique :

Définition 4 — Un système autonomique est un système informatique tirant du contexte dans lequel il s'exécute – plate-forme, systèmes voisins et lui-même – les informations nécessaires pour s'adapter afin de satisfaire les buts de haut niveau de son administrateur.

La granularité des systèmes autonomiques n'est pas évoquée dans cette définition. Il existe pourtant différentes appellations : *IBM* par exemple distingue un système autonomique dans son ensemble des éléments autonomiques qui le composent, comme le montre la figure 3.14 [Kephart 2003]. Nous considérons, dans la suite de cette thèse, que les termes sont équivalents : les éléments autonomiques d'*IBM* pouvant aisément être assimilés à des sous-systèmes autonomiques, de granularité plus fine. Dans la section suivante, nous verrons quelles sont les caractéristiques de cette autogestion des systèmes autonomiques, et sous quelles formes elles peuvent s'exprimer.

3.2.3 Propriétés autonomiques

Dans son manifeste de 2001, *IBM* a énoncé plusieurs points essentiels qui, ensemble, caractérisent les systèmes autonomiques (Horn, 2001). Ces critères ont, plus tard, été exprimées sous la forme de **quatre** propriétés essentielles :

Prop.1. **Auto-configuration** : tout système autonomique doit se reconfigurer continuellement afin de s'adapter aux variations de son contexte. L'environnement d'exécution du système peut subir des modifications aussi diverses qu'imprévisibles. Il est donc nécessaire, en vue de continuer à satisfaire les buts de haut-niveau fixés par l'administrateur, que le système puisse changer sa propre configuration. En pratique, cela peut se manifester par l'ajout ou la suppression de composants, le déploiement de composants sur de nouvelles plateformes matérielles, *etc.* L'adaptation est la clé de voûte de l'évolution du système, et va donc lui permettre de s'organiser, de se réparer et de se protéger face aux changements de son milieu de vie.

Prop.2. **Autoréparation** : lorsqu'un dysfonctionnement survient au sein d'un système autonomique, ce dernier doit être en mesure de déceler le problème, de trouver une solution pour le résoudre, et de l'appliquer. La robustesse du système autonomique doit lui permettre de faire face aux défaillances, en les détectant et en les corrigeant. Le dynamisme et l'hétérogénéité de l'environnement font qu'il est difficile de prendre en compte toutes les situations possibles. Face à un état non-prévu ou incohérent occasionnant une défaillance, un système autonomique doit être en mesure d'empêcher cette dernière de mettre hors service la partie incriminée du système, ou de se propager à l'ensemble du système. Cette propriété caractérise donc la robustesse et la tolérance aux fautes du système autonomique. Comme nous le verrons par la suite, cette capacité de « survie » du système à un environnement hostile est un des bases primitives de la conscience au contexte.

Prop.3. **Auto-optimisation** : le système autonomique doit pouvoir utiliser **au mieux** les ressources dont il dispose. Il doit donc en permanence considérer son propre état afin de déterminer comment il peut s'améliorer. Le système autonomique doit s'autoévaluer en fonction de plusieurs critères de qualité, tels que la consommation mémoire, le temps de réponse, la consommation en énergie, et en fonction de son contexte d'exécution actuel : disponibilité des ressources, taux d'utilisation des ressources, point de congestion, *etc.* La décision d'optimiser tout ou partie du système revient donc à analyser ces différents critères, et à faire des compromis, principalement en fonction des buts de haut-niveau exprimés par l'administrateur. Par exemple, un système autonomique sur une plateforme mobile favorisera l'amélioration des performances lorsqu'il il sera branché à une source de courant, mais privilégiera l'économie d'énergie lorsqu'il fonctionnera sur batterie.

Prop.4. **Autoprotection** : le système autonomique doit être en mesure de se protéger contre les éventuelles menaces de l'environnement extérieur aussi bien que de son contexte interne. De telles menaces peuvent être de différentes natures : causées par un environnement devenu hostile, par des utilisateurs maladroits ou malveillants, voire une partie du système devenant incontrôlable (*bug*, virus, porte dérobée). L'autoréparation a pour but de résoudre ces problèmes lorsqu'ils sont rencontrés, mais, comme le dit l'adage, « mieux vaut prévenir que guérir ». Un système autonomique doit donc anticiper les causes possibles de dysfonctionnement et doit mettre en place des mécanismes de défense appropriés.

Lors de la découverte d'une faille de sécurité, le système pourra par exemple automatiquement télécharger et remplacer le logiciel impliqué.

Ces quatre propriétés fondamentales des systèmes autonomiques sont couramment appelées les propriétés *auto-** ou encore *self-CHOP* [35]. Parmi les publications traitant de l'informatique autonomique, on retrouve une multitude d'autres propriétés *auto-** qui étendent les quatre propriétés fondamentales :

- L'**auto-anticipation** caractérise l'aptitude du système à anticiper les événements à venir, notamment les problèmes.
- L'**autocritique** revient à confronter aux buts de haut niveau l'état du système dans son contexte, et à remettre cet état en cause si nécessaire.
- L'**autodestruction** impose au système autonomique de se désactiver s'il ne remplit plus sa fonction, ou si ses actions s'opposent aux buts de haut-niveau de l'administrateur, ou aux utilisateurs.
- L'**auto-stabilisation** vise à mettre le système dans un état stable après une perturbation de son environnement d'exécution.
- L'**auto-apprentissage** définit la capacité du système à inférer des connaissances (savoir) et des comportements (savoir-faire) à partir des situations précédemment rencontrées.

Cette liste de propriétés, appelées propriétés *auto-** étendues [36] n'est pas exhaustive : un nombre toujours croissant de caractéristiques *auto-** sont mises en évidence. Les critères caractérisant les systèmes autonomiques font souvent débat, chacun souhaitant ajouter ou mettre en avant des propriétés parmi d'autres. Il faut cependant noter qu'il est possible d'exprimer ces propriétés en fonction d'une ou plusieurs des quatre propriétés *auto-** fondamentales, ou inversement. Il est important de souligner ici que l'informatique autonomique ne peut pas se résumer à insuffler ces propriétés aux systèmes. L'informatique autonomique est une vision plus générale qui permet d'agencer les systèmes, aussi variés soient-ils, et de leur permettre d'exposer ces propriétés, fondamentales ou étendues. On peut aussi reprocher à cette caractérisation des systèmes autonomiques de manquer de pragmatisme, les concepts évoqués étant bien souvent très abstraits, ou étant dérivés de domaines autres que l'informatique. Dans la section suivante, nous allons décrire quel est la mise en œuvre pratique de l'application des principes de l'informatique, notamment leurs impacts sur l'architecture des systèmes.

3.2.4 Structures des systèmes autonomiques

Dans cette section nous allons voir quelles sont les diverses façons de concevoir les systèmes afin de leur donner ce caractère autonomique recherché. Plus précisément, les systèmes autonomiques peuvent être élaborés selon différentes approches architecturales. La notion d'architecture logicielle a été brièvement introduite précédemment (cf. section 1.1). Les différentes études et implémentations des systèmes autonomiques et de leur élaboration ont mis en avant principalement trois grands types d'architecture : les systèmes monolithiques, les boucles système–gestionnaire autonomique et les boucles *MAPE-K*.

35. en anglais : *'self-**C**onfiguration, self-**H**ealing, self-**O**ptimization, self-**P**rotection'*
36. *'extended self-* properties'* en anglais

Système monolithique

La première manière qui peut venir à l'esprit lorsqu'un système autonomique doit être construit est l'approche dite **monolithique**. Historiquement, elle est aussi la première à avoir été utilisée, bien avant l'émergence du terme d'« informatique autonomique », dans les précurseurs des systèmes auto-adaptables. Dans cette approche, l'architecture globale du système ressemble à une boîte noire, dans laquelle sont mêlés la logique applicative ainsi que la logique autonomique, incarnant la sensibilité au contexte et la logique d'adaptation.

Vus de l'extérieur, ces systèmes interagissent avec les entités qui constituent leur contexte (cf. figure 3.13). Lorsque l'on se penche sur la structure interne de tels systèmes, les composants réalisant les tâches de surveillance des différentes sources de contexte, les mécanismes permettant d'adapter le système sont difficilement discernables de la logique métier du système. Les différentes préoccupations sont mélangées, ce qui rend le système difficile à analyser, à comprendre et donc à faire évoluer. Cette architecture ne favorisant pas l'administration des systèmes, elle contrarie l'esprit même de l'informatique autonomique. La conception de tels systèmes n'est pas favorisée par les méthodes récentes d'élaboration de logiciels, prônant la séparation des préoccupations.

Cette architecture n'est clairement pas un choix pertinent pour la conception de systèmes autonomiques. Cependant, en plus de sa valeur historique, on peut encore la rencontrer pour des systèmes autonomiques de taille très réduite, tels certains dispositifs embarqués.

Boucle système – gestionnaire autonomique

Le couplage entre les parties fonctionnelles d'un système et la gestion des propriétés autonomiques est donc la principale faiblesse de l'architecture monolithique. Un raffinement de cette architecture, proposé par *IBM*, permet de clairement compartimenter les préoccupations : une partie gérant les aspects métiers du système, une autre fournissant les aspects autonomiques à ce système [IBM 2006]. Si, vu de l'extérieur, ce type d'architecture offre les mêmes propriétés et interactions d'un système autonomique monolithique, sa structure interne la rend plus facile à implémenter, à comprendre et à administrer.

La figure 3.15 montre comment le gestionnaire autonomique, en inspectant et adaptant le système à gérer, permet de rendre l'ensemble du système autonomique. Ce gestionnaire expose à l'administrateur des indicateurs de haut niveau décrivant l'état du système, et lui permet d'exprimer ses intentions d'administration, sous la forme de buts de haut niveau. À l'intérieur de ce système, il existe un découpage entre la logique métier du système et la partie réalisant l'adaptation autonomique. Ces deux parties communiquent par l'intermédiaire de **points de contrôle**[37], permettant au gestionnaire autonomique de monitorer et d'intervenir sur le système à gérer. Il existe deux catégories de points de contrôle : les **senseurs** et les **effecteurs**.

Les différentes entités composant le système autonomique dans son ensemble sont donc les suivantes :

– Le **système à gérer** contient l'ensemble de la **logique métier** mettant en œuvre les objectifs du système. Il est la partie centrale du système, l'entité minimale qui doit être présente pour que le système soit fonctionnel. La nature du système

37. *touchpoints* en anglais

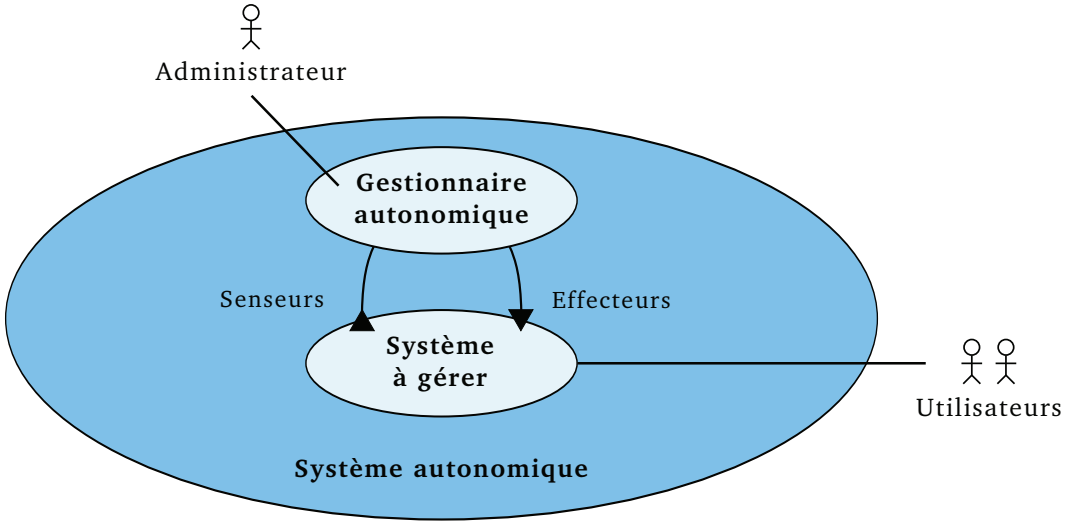

FIGURE 3.15 – Architecture système – gestionnaire autonomique

à gérer peut être très variable : dispositif physique (capteur, interface réseau, cellule de batterie, *etc.*), composant logiciel (mémoire cache, système de fichier, *etc.*) ou un ensemble complexe regroupant les deux (machine + système d'exploitation, pilote de satellite artificiel, de centrale nucléaire, *etc.*). Le système à gérer, aussi appelé **élément administré**, doit être en mesure de fonctionner de manière indépendante, sans son gestionnaire autonomique.

- Le gestionnaire autonomique est chargé d'insuffler les propriétés autonomiques au système. Il communique avec le système à gérer au travers des points de contrôle, senseurs et effecteurs, et l'administre en fonction des buts de haut niveau fixés par l'administrateur. Ce gestionnaire, en fonction des données relevées par les senseurs, prend des décisions d'adaptation du système à gérer, qu'il transmet via les effecteurs. Nous verrons dans la section suivante que le gestionnaire autonomique peut être lui-même décomposé en plusieurs composants réalisant une partie de ces tâches de surveillance et d'adaptation.
- Les points de contrôle constituent les interfaces de communication entre le système à gérer et le gestionnaire autonomique. Il existe deux grandes catégories de points de contrôle :
  - Les **senseurs** récoltent des données sur le système à gérer et les transmettent au gestionnaire autonomique (*push*), ou lui permettent de les récupérer (*pull*). Le terme senseur désigne à l'origine un dispositif physique relevant des mesures environnementales (température, pression, *etc.*). La notion de senseur, telle qu'utilisée dans le domaine de l'informatique autonomique, est plus large, regroupant à la fois les senseurs physiques (thermomètre, capteur de présence, *etc.*) et des composants logiciels capturant certaines données du système (consommation mémoire, activité du microprocesseur, débit réseau, *etc.*) La nature des données collectées par les senseurs, ainsi que la façon de les collecter, sont des paramètres qui dépendent totalement du type de senseur utilisé.
  - Les **effecteurs** ont une fonction symétrique à celle des senseurs. Ils permettent d'agir sur l'élément administré, de modifier son état ou son comportement. Leur présence dans le système à administrer est un prérequis pour permettre l'adaptation de celle-ci par le gestionnaire autonomique. On peut distinguer deux

grandes catégories d'effecteurs au sein d'un système. Les premiers permettent de reconfigurer (c'est-à-dire modifier) une partie du système (typiquement un composant logiciel). Les deuxièmes vont modifier l'architecture du système, en ajoutant ou retirant des composants. La possibilité d'effectuer ces modifications « à chaud », sans arrêter entièrement le système, est une propriété qui, comme nous le verrons, favorise l'évolution continue du système.

Cette séparation des points de contrôle en deux catégories distinctes est théorique. Il est courant, en pratique, de rencontrer des points de contrôle qui agissent tantôt comme des senseurs, tantôt comme des effecteurs. Par exemple, un senseur peut être doté d'une bascule indiquant son état courant, activé ou désactivé, et peut donc, du point de vue du gestionnaire autonomique, être aussi considéré comme un effecteur.

Dans cette architecture, il faut noter que la logique métier du système n'a pas « conscience » de l'existence d'un gestionnaire autonomique. C'est ce découplage entre la logique fonctionnelle et la partie autonomique qui est justement recherchée. D'un autre côté, il est illusoire de penser que le gestionnaire autonomique est lui aussi totalement découplé de la partie fonctionnelle. Certains gestionnaires autonomiques, dits génériques, offrent la possibilité de gérer une très grande variété de différents systèmes. Cependant, les aspects qu'ils gèrent sont souvent très réduits, et insuffisants pour qualifier le couple système – gestionnaire de complètement autonomique. Il en résulte que tout gestionnaire autonomique est couplé au système qu'il doit gérer, dédié à l'administration de ce système en particulier.

Ce découplage, unidirectionnel, permet de ne pas bouleverser les méthodes de conception des systèmes à gérer : ils doivent, en plus de leurs interfaces fonctionnelles, fournir les interfaces d'administration (*touchpoints*) suffisantes pour être observés et adaptés par un gestionnaire autonomique. La vision de l'informatique autonomique, telle que proposée par *IBM*, va cependant plus loin, en proposant un découpage par activité du gestionnaire autonomique : la boucle *MAPE-K*.

Découpage du gestionnaire autonomique : la boucle MAPE-K

La boucle système–gestionnaire autonomique permet de faciliter la construction de système fournissant des caractéristiques autonomiques. La mise en œuvre d'un gestionnaire autonomique reste néanmoins une tâche difficile : ce gestionnaire doit accomplir différentes tâches telles que la surveillance et l'adaptation du système, tout en tenant compte du contexte d'administration de ce système. Le mélange de de ces diverse composantes au sein d'un gestionnaire autonomique monolithique, bien que possible, augmente la complexité d'appréhension de ce gestionnaire, diminue sa qualité et rend difficile sa maintenance.

IBM a proposé un standard s'architecture de gestionnaire autonomique découpant ce dernier en cinq entités aux rôles bien délimités [Kephart 2003]. La figure 3.16 illustre l'agencement des cinq composantes de cette architecture de référence, appelée l'architecture *MAPE-K* [38]

– La partie ***Monitoring*** est chargée de gérer la remontée d'informations provenant du système à gérer. Les différents capteurs fournissent des données brutes sur l'état courant du système. Le couche de monitoring va avoir pour but de sélectionner les données remontées et de les agréger afin d'en extraire des informations significatives, celles qui vont guider les choix du reste du gestionnaire

38. abréviation de *'**M**onitoring, **A**nalysis, **P**lanning, **E**xecution - **K**nowledge'*

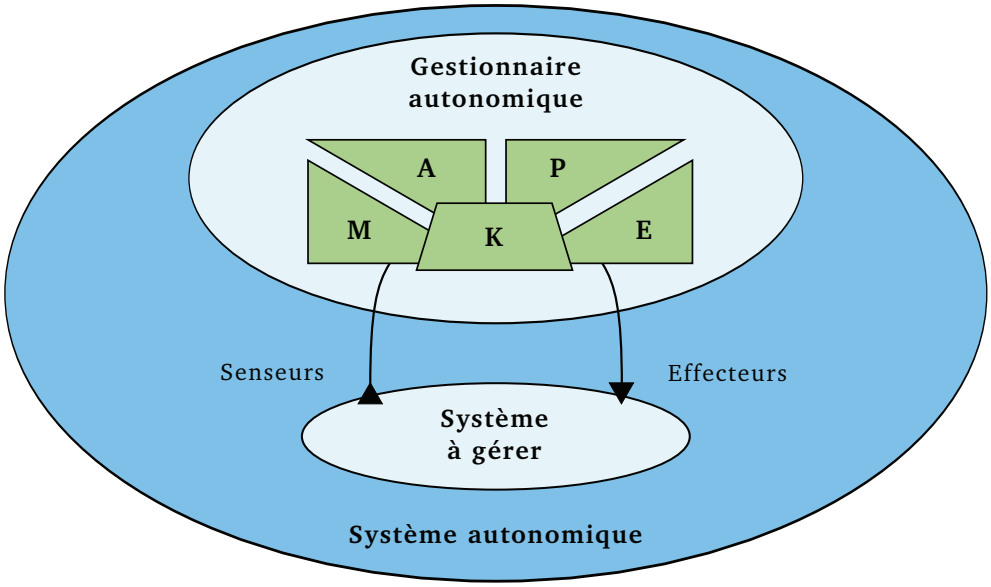

FIGURE 3.16 – Architecture *MAPE-K*

autonomique.

- La partie ***Analysis*** récupère les données résultats. Elle identifie les dysfonctionnements et les optimisations puis détermine la nécessité d'effectuer des modifications sur l'élément administré autonomique pour régler les problèmes identifiés. L'analyse utilise la base de connaissances pour effectuer des corrélations, des courbes de tendances, des statistiques, des probabilités, des séries de numériques horodatées, *etc.*
- La partie ***Planning*** prend en charge la définition des actions à effectuer sur l'élément administré pour résoudre les problèmes identifiés par l'analyse. Elle génère un plan d'actions.
- La partie ***Execution*** est en charge du calendrier des actions définies par la planification. Le calendrier consiste, par exemple, à définir une date ou un horaire pour l'exécution des actions qui doivent être exécutées. Elle est aussi le point de contact avec les effecteurs.
- La partie ***Knowledge*** est une composante essentielle de la boucle autonomique. Sans elle, la boucle autonomique serait une simple boucle de rétro-action. La base de connaissances contient l'ensemble des informations nécessaires pour la gestion de l'élément autonomique. Elle est commune entre la Surveillance, l'Analyse, la Planification et l'Exécution (*MAPE*). Elle peut contenir des mesures, des politiques, des états, des résultats des exécutions, des topologies, *etc.* Elle peut aussi être mise à jour par la récupération d'une base de connaissances d'un ou plusieurs autres gestionnaires autonomiques.

3.3 Niveaux d'autonomie

3.3.1 Maturité des systèmes autonomiques

IBM a défini cinq niveaux de maturité par rapport à l'autonomie. Ils commencent par le niveau 1 appelé basique et se terminent par le niveau 5, degré exprimant un système dit autonomique :

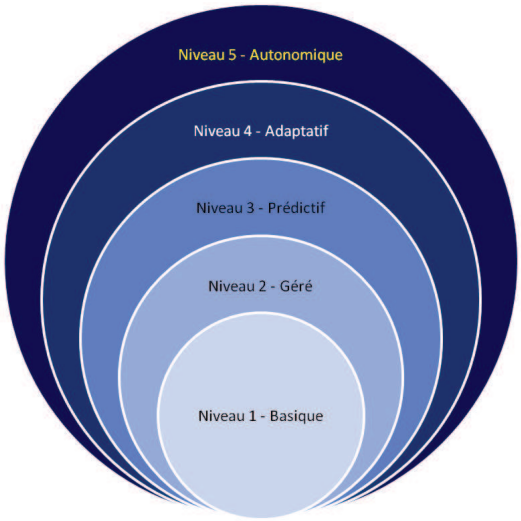

FIGURE 3.17 – Augmentation des fonctionnalités autonomiques

Niv.1. le **niveau basique** : le système agrège plusieurs sources de données. L'analyse et la correction de problèmes sont manuelles. L'intervention des experts et des administrateurs est importante.

Niv.2. le **niveau géré** : des outils permettent la consolidation des mesures effectuées. Les informations collectées sont étudiées par des administrateurs et des experts ; les actions sont manuelles.

Niv.3. le **niveau prédictif** : le système collecte des données et les analyse. Le système propose des actions. Les experts et les administrateurs décident de leur mise en place ou non.

Niv.4. le **niveau adaptatif** : le système collecte des données et effectue des corrélations. Il est capable de s'adapter pour des décisions simples. Le système est plus résilient et agile. L'intervention des experts et des administrateurs est réduite.

Niv.5. le **niveau autonome** : les experts et les administrateurs définissent des objectifs de haut niveau, ce sont des objectifs à long terme. Le système s'autogère en conformité avec ses objectifs. Les experts et les administrateurs n'interviennent que très rarement.

En fait, ces cinq niveaux illustrent le champ d'application de la boucle autonomique. Plus le périmètre d'application de la boucle autonomique s'agrandit, plus la boucle autonomique augmente son périmètre d'application. À tout moment, les bonnes pratiques des experts, des utilisateurs et des administrateurs sont prises en compte pour aider à définir le niveau supérieur.

La figure 3.18 extraite de [Murch 2004] illustre le pourcentage des solutions SI à chaque niveau de l'informatique autonomique. Ces mesures ont été effectuées par *IBM*. Pour l'année 2002, 40 % des entreprises proposaient des solutions de niveau 1. D'années en années, la figure nous montre une décroissance des solutions de niveau 1 et 2, alors qu'en même temps les niveaux 3 à 5 ne font que croître. Il faut par ailleurs savoir que le temps de mise sur le marché de solutions est largement supérieur à quatre ans. Nous avons ainsi, avec cette illustration, une mise en évidence que l'informatique autonomique est aussi adaptée aux systèmes patrimoniaux.

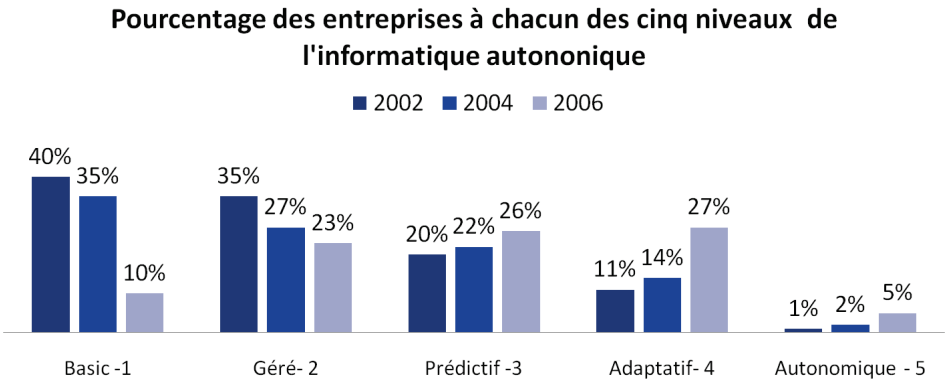

FIGURE 3.18 – Niveau de maturité de l'informatique autonomique dans les entreprises [Murch 2004]

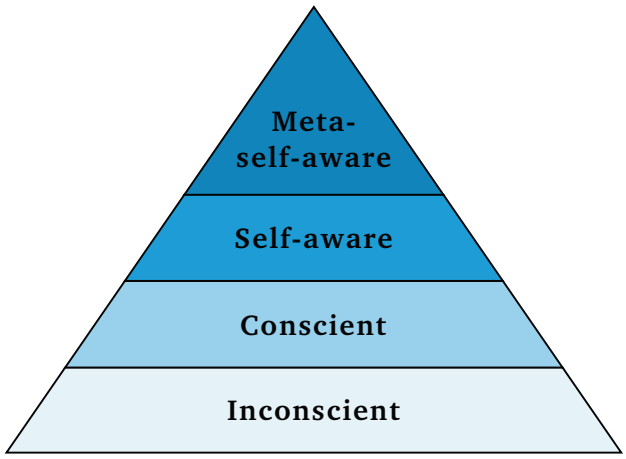

FIGURE 3.19 – Pyramide de la conscience

3.3.2 Vers les portes de la conscience ?

L'informatique autonomique tire ses inspirations de nombreuses disciplines. Il est intéressant de constater par exemple que les qualificatifs servant à désigner les systèmes autonomiques, plus ou moins matures, servent aussi à distinguer les différents niveaux de conscience chez l'être humain. Ainsi, sans trop sombrer dans l'anthropomorphisme, un système autonomique peut être grossièrement comparé à une entité dotée d'une conscience restreinte.

L'étude de la psychologie découpe la conscience humaine en différents niveaux [Morin 2006], tel qu'illustré par la figure 3.19 :

Niv.1. Le niveau le plus bas symbolise **l'inconscience**. Dans cet état, un individu ne réagit pas aux stimuli extérieurs, mais conserve néanmoins, sans dans les cas de traumatismes graves, ses fonctions vitales (respiration, contractions cardiaques, *etc.*). L'inconscience représente un état de vie minimaliste, où l'individu subit sans agir. Les besoins vitaux nécessitant un état plus conscient (alimentation, protection du froid, *etc.*) ne peuvent être satisfaits.

Niv.2. Dans le niveau suivant, appelé **conscience**, l'individu est éveillé : il perçoit le monde qui l'entoure et peut réagir à des stimuli. L'individu est capable de

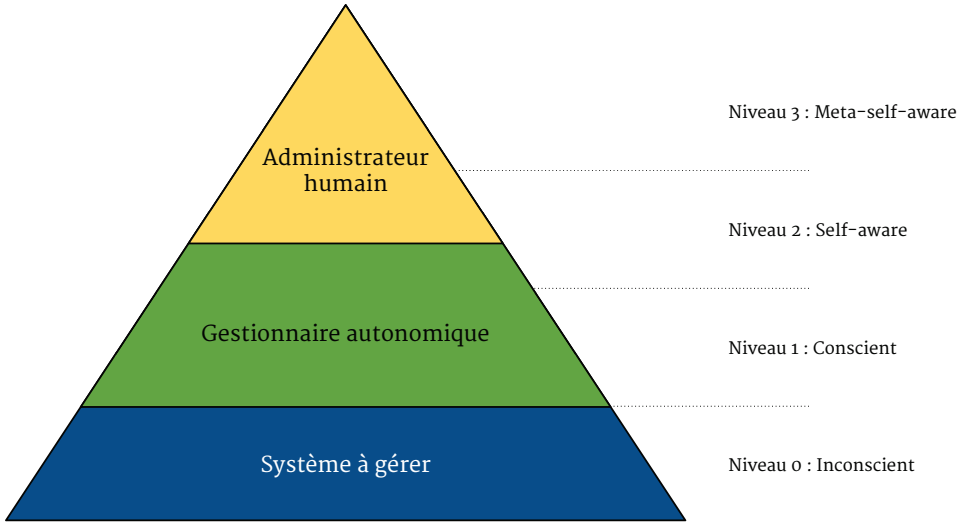

FIGURE 3.20 – Niveaux de conscience d'un système autonomique

répondre à ses besoins physiologiques, et de se mouvoir dans son environnement.

Niv.3. Le niveau suivant est la **conscience de soi** (*self-awareness*). Dans cet état, un individu est conscient de son existence, et de sa place dans l'environnement qui l'entoure. Les raisonnements abstraits permettent l'établissement de concepts tels que le bonheur, les projets futurs, la situation financière, *etc.* Les réflexions émanant de cet état de conscience, bien qu'abstraites, résultent encore grandement sur des perceptions.

Niv.4. Enfin, le niveau final est la « **méta-conscience de soi** » (*meta-self-awareness*). Cet état symbolise en fait la conscience de la conscience : la sienne, mais aussi cette des autres êtres conscients. Ce niveau permet le raisonnement symbolique de très haut-niveau.

Vouloir rendre les systèmes informatiques plus autonomes, c'est leur faire grimper cette pyramide de la conscience. Il est bien sûr très difficile de transposer littéralement ces états qui s'appliquent à l'humain. Cependant, le niveau d'abstraction des adaptations de ces systèmes permettent de leur attribuer un niveau de conscience équivalent, tel que l'illustre la figure 3.20 :

- Les systèmes informatiques non-autonomes peuvent être rangés dans la catégorie « systèmes inconscients ». Les changements de l'environnement d'exécution ne sont pas prévus, et perturbent ou sont mêmes fatals à ce type de système.
- Les systèmes autonomiques permettent de « sentir » l'environnement d'exécution, et de réagir aux changements lorsqu'ils surviennent. Ces systèmes sont sensibles au contexte, et rentrent donc dans la catégorie des « systèmes conscients ». Selon le niveau des actions d'adaptation et des connaissances du système induites, les systèmes autonomiques peuvent parfois même dépasser ce niveau de « conscience » pour effleurer celui de la « conscience de soi », avec des concepts parfois très abstraits.
- L'administrateur d'un système est seul capable d'atteindre pleinement les niveaux de conscience supérieurs. Le gestionnaire autonomique décharge l'administrateur

des tâches exigeant un niveau de conscience moindre. Ce délestage permet à ce dernier de se focaliser sur les tâches les plus complexes et abstraites.

3.4 Besoins

Selon [Ganek 2004], un système autonomique doit avoir les propriétés suivantes :

1. avoir une **base de connaissances**, qui reflète les états et les contextes des activités des constituants du système ;
2. être capable de **comprendre et d'analyser les conditions environnementales** : le système doit pouvoir s'apercevoir et comprendre ses implications lors de chaque changement environnemental ;
3. être capable de comprendre et d'analyser les conditions environnementales. Celles-ci peuvent être globales ou spécifiques à un ou plusieurs sous-ensembles de constituants du système.

La mise en place de ces trois caractéristiques repose sur les notions d'observation et d'adaptation que nous examinons ci-après. Elle repose également sur la définition d'un formalisme approprié pour la représentation de la connaissance. Ce formalisme doit être d'un niveau d'abstraction suffisamment élevé pour permettre une écriture facile et directe des tâches d'analyse et de planification.

Nous fournissons maintenant quelques éléments significatifs concernant les notions d'observation et d'adaptation.

3.4.1 Observation

La surveillance collecte les données des capteurs et met à jour la base de connaissances. Le processus qui effectue des mesures, ne doit pas perturber l'exécution du système. La surveillance peut, par exemple, déclencher un événement par l'intermédiaire du capteur, sans perturber le traitement d'un événement système d'un point de vue fonctionnel et temporel. Les mesures fournies par les capteurs doivent avoir une précision suffisante pour caractériser le système observé.

Sans observation pertinente, le gestionnaire autonomique ne pourra pas atteindre ou même maintenir ses objectifs. Prenons l'exemple de pannes non-franches (dégradations lentes des performances du système). En pratique, elles demandent beaucoup de temps et d'énergie aux administrateurs et aux experts pour identifier les sources du dysfonctionnement. Sans information proche de la source d'erreur, la détermination du problème est extrêmement longue et difficile. C'est pour cela que pour gagner du temps de disponibilité du système, généralement le correctif consiste en une série de modifications de type contournement (le problème n'est pas corrigé, seule son apparition est supprimée). Seule une connaissance approfondie et une expérience de la dynamique d'exécution de l'application permettent la définition des contournements. En fonction de l'implantation du système, il ne faut donc pas hésiter à déterminer les données pertinentes qui peuvent être enfouies dans les couches les plus basses (matériel, système d'exploitation) jusqu'aux plus hautes.

Les capteurs délivrent généralement des données non corrélées. Généralement, on retrouve des données qui permettent d'obtenir :

– des mesures de charge du système,
– des temps de réponse du système,

- des mesures de latence,
- le nombre d'événements traités et/ou générés par unité de temps,
- le nombre de fichiers ouverts,
- le nombre de connexions *TCP/IP* ouvertes,
- des mesures distribuées sur plusieurs systèmes nécessitant des mécanismes de synchronisation d'heure (une mesure de charge ou de consommation électrique exécutée au même moment par tous les systèmes déclenchée par un *top* horaire),
- la topologie du système (par exemple, les chemins de communications disponibles).

Le code pour récupérer ces informations peut être interne ou externe au système administré. S'il est externe, il pourra prendre le rôle de médiateur entre le système et la surveillance. En tant que médiateur, il peut agréger et/ou filtrer des données. Dès que des données sont regroupées en plusieurs systèmes, il est nécessaire de mettre en place un mécanisme d'identification. Celui-ci permet d'associer de manière unique la mesure et le système mesuré. Ces médiateurs de données ont été décrits par [Garlan 2001] et sont couramment appelés *gauges*. Les *gauges* peuvent être configurées.

Le processus de génération des mesures consomme de la ressource système, ce qui peut dégrader les performances métier de l'application. La seconde conséquence est au niveau de la boucle autonomique. Un flux important et permanent des mesures rend difficile le fonctionnement nominal de la surveillance (sa capacité maximale de traitement pourrait être dépassée). En offrant aux capteurs des fonctions d'activation et de désactivation, la surveillance peut piloter le flux et la charge d'exécution des processus de génération des mesures.

Pour rendre plus facile le fonctionnement de la surveillance, nous avons vu que les capteurs peuvent être externes, ils pourront prendre la fonction de médiateurs de données (agrégation, filtrage). Ils peuvent également être internes, il faudra alors implanter du code non-fonctionnel qui pourra effectuer une mesure directe ou indirecte.

Il nous reste à nommer quelques techniques et principes largement répandus permettant de communiquer les mesures des capteurs à la surveillance ; citons les mécanismes suivants :

- de **notification**, le capteur notifie la fraîcheur d'une information, la donnée peut être ou non incluse dans l'événement,
- de **surveillance périodique** (ou non). La tâche de surveillance est à l'initiative de la mise à jour des données issues des capteurs,
- de **messageries** avec la définition des patrons d'échanges de messages entre la tâche de surveillance et les capteurs,
- des **appels de méthodes** (*callback*).

Une surveillance dynamique permet d'arrêter, de démarrer, de configurer des capteurs. Elle surveillance peut même injecter du code dynamiquement et le supprime lorsqu'il n'est plus nécessaire. Plus simplement, elle peut configurer de façon dynamique les capteurs (augmenter/diminuer la fréquence d'échantillonnage, injecter un message spécifique), surveiller des capteurs à la demande (en fonction du contexte d'exécution). L'exécution de l'analyse est étroitement liée à celle de la surveillance. En effet, l'analyse peut nécessiter la mise à jour d'informations supplémentaires (augmenter la période d'échantillonnage d'une mesure, mettre dans la base de connaissances des informations non encore mises à jour). La surveillance vient chercher un ensemble d'informations dans les capteurs sur demande de l'analyse.

3.4.2 Adaptation

L'adaptation est un processus qui modifie un système. Elle peut être soit statique, soit dynamique. Une adaptation statique nécessite un arrêt du système. L'adaptation dynamique autorise des modifications pendant l'exécution du système. Pour les utilisateurs et/ou les autres applications interconnectées, la disponibilité du système sera partielle (elle ne peut pas être totale pendant la modification).

Les effecteurs peuvent agir sur le système par le biais de la configuration. Usuellement, on distingue les paramètres de réglages qui sont modifiables à la volée (adaptations dynamiques) et les paramètres de configuration que l'on ne peut pas modifier dynamiquement (adaptation statique). Dès la conception du système, il est important de privilégier le développement des paramètres de réglages. Ils sont plus difficiles à implanter car ils requièrent la prise en compte du dynamisme.

Un deuxième type d'adaptation plus profonde est nécessaire pour répondre à des besoins de flexibilité. Elle agit au niveau structurel. Kramer et Magee [Kramer 1990] ont présenté les besoins pour des modifications de type dynamiques. L'approche utilise essentiellement la séparation des préoccupations à savoir l'implantation du métier, de l'isolation des processus de modification. Les propriétés sur les modifications ainsi définies sont :

- les modifications doivent minimiser les perturbations du système,
- les modifications doivent laisser le système dans un état consistent,
- les modifications doivent être indépendantes des algorithmes et des protocoles utilisés par le système,
- les modifications doivent être déclaratives,
- les modifications doivent être décrites en termes de structure du système (il ne doit pas y avoir de modifications à grains fins, par exemple, pas de modifications de la structure interne d'un composant).

Il existe plusieurs patrons dont deux sont largement utilisés :

- le patron d'interposition dynamique implémenté par *CORBA* [Baldoni 2003]. Un intercepteur est inséré au niveau du serveur ou du client. L'intercepteur redirige les requêtes vers un *wrapper* spécifique.
- le remplacement dynamique de composant [Soules 2003] qui est une évolution du patron précédent.

Ces deux patrons nécessitent de mettre la partie du système qui doit être modifiée dans un état **quiescent** [Kramer 1990]. Ce besoin provient de la nécessité de démarrer une modification lorsque l'état du système est consistant, de le modifier et de rendre de nouveau consistant après la modification. L'état de quiescence étant difficile à atteindre, certains auteurs [Vandewoude 2006] proposent plutôt de viser un état de tranquillité pour effectuer des adaptations dynamiques.

3.5 Synthèse

L'informatique autonomique est une initiative ambitieuse s'attelant à la gestion de systèmes informatiques complexes. Son principal objet est de réduire considérablement l'effort requis pour l'administration des systèmes, et par conséquent de diminuer leur coût de maintenance et d'évolution. Ceci est accompli en introduisant des gestionnaires autonomiques coordonnés capables de remplacer les administrateurs humains en intervenant de manière rapide et automatisée pour réparer les erreurs, optimiser l'utilisation des ressources, configurer de nouvelles ressources ou protéger le système

contre des défaillances ou des attaques.

La complexité intrinsèque des systèmes informatiques modernes est donc directement reflétée dans les systèmes autonomiques de gestion qui les contrôlent. *IBM* a proposé une architecture de référence, *MAPE-K*, pour organiser de tels systèmes autogérés. Cette architecture conceptuelle a été implémentée de diverses manières par plusieurs équipes de recherche qui visent différents critères de qualité et domaines d'application. Parmi les projets les plus aboutis aujourd'hui, citons :

- **Jade** [de Palma 2008, Sicard 2008] est un projet développé par l'équipe SARDES de l'INRIA. Son principal objectif est de permettre la gestion autonomique d'applications patrimoniales. Jade propose de les encapsuler dans des composants standardisés disposant d'interfaces d'administrations unifiées. Des plans de reconfigurations peuvent ensuite y être appliqués.
- **Auto-Home** [Bourcier 2011, Bourcier 2008] est un projet initié par l'université de Grenoble. Il fournit des solutions permettant d'assister les concepteurs pour le développement d'applications autonomiques basées sur des architectures hiérarchiques orientées services. Il se focalise sur le domaine des applications pervasives domotiques.
- **AutoMate** [Parashar 2006] est un projet mis en place par l'université de Rutgers. Il vise à explorer les technologies clés afin de permettre le développement d'applications autonomiques qui gèrent la complexité, le dynamisme et l'incertitude associés aux environnements à base de grilles informatiques. Il inclut des sous-projets, tels le modèle de programmation autonomique Accord [Liu 2006], l'infrastructure multi-agents basé sur des règles Rudder [Li 2004] ainsi que le middleware d'interaction sur les contenus Meteor [Jiang 2008].
- **Unity** [Chess 2004, Tesauro 2004] est un projet développé par *IBM* au centre de recherche Thomas J. Watson. L'objectif de ce projet est de permettre la gestion autonomique de systèmes distribués. Unity se concentre principalement sur la gestion autonomique d'applications s'exécutant dans des grappes et des grilles informatiques. Plus spécifiquement, Unity fournit une architecture pour permettre l'allocation autonomique de ressources (comme les serveurs par exemple) aux différentes applications, en suivant les politiques de gestions fixées par les administrateurs.
- **Rainbow** [Garlan 2004] est un projet développé à l'université Carnegie Mellon par l'équipe de David Garlan. Rainbow propose l'utilisation de modèles architecturaux à l'exécution (*Model@Runtime*) afin de faciliter l'adaptation dynamique des systèmes. C'est une approche plutôt globale pour l'ingénierie des systèmes auto-adaptatifs, comprenant un *framework*, un langage décrivant les adaptations et des mécanismes d'exécution de ces adaptations. Le *framework* inclut une boucle de contrôle surveillant le système et en déduit les adaptations appropriées sur le modèle de l'architecture.
- **Ceylon** [Diaconescu 2008, Maurel 2010b, Maurel 2010a] est un projet développé à l'université de Grenoble. Il propose de construire des gestionnaires autonomiques en intégrant des tâches administratives spécialisées, où chaque tâche implémente un aspect de gestion spécifique (surveillance d'une certaine caractéristique de système ciblé, détection de problèmes spécifiques, planning de l'application d'une solution particulière, ou modification de certains ressources gérées). L'intégration de tâches est réalisée durant l'exécution, afin de produire des chaînes de gestion qui sont adaptées au dynamisme des conditions d'exécution et aux requis. Ces tâches peuvent être connues au moment du dé-

veloppement et/ou découvertes et déployées pendant l'exécution du système. Suivant le contexte d'exécution et l'état des applications gérées, un gestionnaire de tâches de haut niveau active ou désactive des tâches spécifiques et forme une chaîne de tâche répondant aux situations observées. Ce gestionnaire de tâches est guidé par les stratégies de gestion de haut niveau dictées par l'administrateur.

Ces différents *frameworks* implantent tous l'architecture de référence *MAPE-K* définie par *IBM*. Ils répondent aux besoins fondamentaux explicités dans ce chapitre, à savoir :

- Ils définissent une base de connaissance reflétant les contextes d'exécution internes et externes. Cette base repose sur du code de surveillance, souvent très avancé.
- Ils sont capables de comprendre et d'analyser la situation courante à partir des informations fournies par la base de connaissance.
- Ils sont capables de planifier et d'effectuer des modifications à différents grains. Ces modifications reposent sur des mécanismes d'adaptation fournis par le *framework*.

Il est cependant frappant de constater que ces *frameworks* reposent sur des solutions techniques et des formalismes très différents. Cette grande hétérogénéité est un des caractéristiques essentielles de l'informatique autonomique.

Proposition

Chapitre 4

Vers une machine d'exécution pour applications autonomiques

La première partie de ce manuscrit s'est intéressée à l'étude des approches permettant d'augmenter le degré d'autonomie des applications pervasives. Précisément, nous avons défini les applications pervasives, présenté leurs caractéristiques essentielles et pointé le manque des solutions actuelles en matières d'autonomie.

La deuxième partie de cette thèse introduit une extension au modèle à composant à service iPOJO [Escoffier 2008] afin de permettre d'augmenter l'autonomie d'applications reposant sur iPOJO ainsi que de gérer et de contrôler cette autonomie. En effet, bien que le modèle iPOJO permette le développement d'applications dynamiques, le degré d'autonomie de ces applications est contraint par l'implémentation et à l'architecture de l'application. Il nous a donc paru important d'étendre ces capacités afin de pouvoir administrer, et contrôler le comportement dynamique de ces applications. En complément, il est nécessaire de faciliter la mise en place de tels contrôleurs tout en ne négligeant pas leur extensibilité. En d'autres termes, cette thèse propose un canevas logiciel afin de faciliter la mise en place de boucle autonomique autour d'applications dynamiques reposant sur le modèle iPOJO.

Ce chapitre rappelle les enjeux des adaptations dynamiques et souligne les difficultés de mise en place de boucles autonomiques. Ensuite, la problématique adressée par cette thèse est décrite ainsi que l'approche globale proposée.

Sommaire

4.1 Problématique

Dans la première partie de ce manuscrit, nous avons présenté l'informatique pervasive. Ce nouveau domaine repose sur un environnement saturé d'équipements électroniques capables de calculer et de communiquer entre eux de façon à assister les êtres humains dans leur vie quotidienne de façon transparente et naturelle. Les applications pervasives sont rapidement devenues très populaires et sont désormais de plus en plus répandues. On en rencontre dans les bâtiments, les maisons, les espaces publics et même au sein des usines dites connectées. Elles soulèvent également un grand intérêt dans le domaine de la santé, un enjeu sociétal majeur aujourd'hui [1].

Nous avons vu que l'informatique ubiquitaire est caractérisée par les propriétés suivantes :

1. Elle a pour but de rester quasiment invisible et repose sur une interaction homme machine discrète, naturelle et transparente.
2. Elle est intrinsèquement répartie entre différents dispositifs de calcul, mobiles ou fixes, connectés par des services réseaux. Ces dispositifs, qui restent généralement discrets, sont en constante interaction entre eux et avec leur environnement physique.
3. Elle est sensible au contexte afin de pouvoir optimiser les services rendus dans un environnement changeant.

Nous avons vu également que le développement et la maintenance d'applications pervasives demeurent des activités complexes. Ces applications doivent s'exécuter dans des environnements distribués, hétérogènes, ouverts et très dynamiques. Elles doivent donc être capables de s'adapter dynamiquement et ce, de façon autonome puisque l'intervention d'un administrateur à chaque évolution de contexte n'est pas envisageable. Le besoin en adaptation dynamique est essentiellement dû à des changements fréquents de l'environnement d'exécution des applications.

L'adaptation dynamique est très difficile à implanter au sein des applications pervasives. En effet, pour y parvenir, une application doit être consciente des possibilités d'évolution ainsi que de son état au sein d'un écosystème plus vaste. Or l'accès à cette connaissance et son utilisation posent deux problèmes majeurs :

1. Elle rend les applications très complexes à développer car la logique d'adaptations se rajoute à la logique métier des applications.
2. Elle ne permet que des adaptations anticipées lors du développement. Or, il est aujourd'hui très difficile de prédire avec précision les différentes évolutions de l'environnement d'exécution d'une application.

Afin d'illustrer ces deux points, prenons l'exemple d'une application ubiquitaire « simple » : un *Light Follow Me*. Cette application allume les lumières dès que l'utilisateur entre dans une pièce et les éteint lorsqu'il sort. Il est malheureusement impossible de prédire lors du développement de l'application l'agencement de la maison où elle sera exécutée ainsi que les lampes présentes. De plus, l'ensemble des lampes et leurs localisations peuvent évoluer lors de l'exécution de l'application. De nouvelles lampes peuvent être ajoutées dans la maison, d'autres peuvent être retirées, et certaines peuvent ne plus être fonctionnelles.

1. Par exemple le projet *Medical* (`http://medical.imag.fr`) vise à maintenir les personnes âgées à leur domicile, en proposant une plateforme logicielle gérant l'hétérogénéité des différents équipements installés (détecteurs de chute, alarmes incendie, *etc.*).

Pour répondre à de telles exigences, de synchronisation notamment, le code applicatif devient rapidement complexe et source d'erreurs. Typiquement, l'implantation d'adaptations demande de manipuler des concepts complexes tels que des **sémaphores** et des **threads** afin, par exemple, de garantir l'intégrité de l'état de l'application. Il faut également introspecter les applications pour décider des besoins en adaptation et du bon timing. Ce niveau de technicité n'est pas approprié pour les développeurs métier qui, dans notre exemple, sont experts de l'éclairage.

Pour rendre de telles applications plus autonomes, robustes et agrandir leur espace de viabilité, il est souhaitable de disposer d'une machine d'exécution prenant en charge une partie, voire la totalité, des adaptations au contexte. Cette externalisation, souvent effectuée à l'aide d'une boucle de contrôle ou autonomique, offre plusieurs avantages, notamment :

1. elle sépare le code d'adaptation du code métier ;
2. elle permet une analyse plus globale de la situation afin d'optimiser, non pas une application, mais l'ensemble du système.

Le problème d'anticipation, cité précédemment, intervient également au niveau des politiques d'adaptations. Il n'est pas rare de vouloir ajouter ou de contrôler l'autonomie d'une application existante en cours d'exécution. En effet, suite a des changements d'objectifs ou de contraintes métiers, il peut être nécessaire de mettre en place un nouveau gestionnaire autonomique autour d'une application bien après son déploiement initial afin d'optimiser son exécution pour atteindre les nouveaux objectifs.

Ces dernières années, de nombreux modèles à composants sont apparus afin de faciliter le développement d'applications reconfigurables dynamiquement [Oreizy 1999] ou dynamiques [Papazoglou 2008]. Bien que ces modèles permettent le développement d'applications plus autonomes, ces aspects restent très difficiles à développer. En effet, l'état de l'application et les reconfigurations appliquées dépendent fortement du modèle utilisé. Il est difficile d'étendre ce modèle avec des données métiers ou provenant d'autres briques techniques. Les actions de reconfigurations peuvent donc toucher les parties métiers d'une application mais aussi les parties du système sur lequel elle repose.

Nous nous sommes intéressés en particulier au *framework* iPOJO, développé par l'équipe Adèle [2]. Ce modèle proposé par Clément Escoffier en 2008 [Escoffier 2008] permet le développement d'applications dynamiques à base de services. iPOJO implémente les concepts des modèles à composant à service au-dessus de la plate-forme Java et plus précisément *OSGi*. iPOJO propose à la fois un modèle de développement simple et une modélisation du dynamisme flexible et puissante. À ce sujet, iPOJO propose aujourd'hui le modèle de dépendance de service le plus complet, et couvre une grande partie des besoins en dynamisme [Escoffier 2013]. iPOJO a atteint une maturité permettant son utilisation industrielle, et est utilisé à l'heure actuelle par plus de 400 entreprises.

Cependant, iPOJO ne couvre pas l'intégralité des fonctionnalités requises pour une gestion autonome de la dynamicité. De plus certaines fonctionnalités sont fournies mais leur utilisation est délicate et réservé aux développeurs experts [3]. Ainsi,

2. `http://www-adele.imag.fr`

3. Par exemple, l'ajout de la gestion d'aspects non-fonctionnels (*'handlers'*) au conteneur est une caractéristique de l'extensibilité d'iPOJO. Il nécessite cependant une connaissance très approfondie des mécanismes internes d'iPOJO, si bien qu'en pratique très peu de développeurs peuvent correctement étendre ces conteneurs de composants.

bien qu'iPOJO propose de base des mécanismes d'introspection, ceux-ci utilisent un formalisme complexe sans notifications.

Dans le chapitre précédent a été présentée la notion d'informatique autonomique. Nous pensons, en effet, que des propriétés autonomiques peuvent permettre de faire face à la complexité croissante des besoins en dynamisme. Ces propriétés peuvent permettre de diminuer significativement les coûts d'exploitation et de maintenance, d'une part, et parer à l'absence d'administrateurs, d'autre part. Les systèmes autonomiques reposent sur des capacités importantes en matière d'introspection et d'adaptation et demandent une formalisation avancée de l'état d'exécution des systèmes autogérés.

Nous faisons le constat suivant : malgré le besoin croissant en autonomie et en dynamisme, la complexité de mise en place de boucle de contrôle freine l'essor de l'informatique autonomique. En particulier, l'intégration avec l'application mais également le domaine métier et les autres composants du système limitent ces développements à de rares experts.

4.2 Objectifs

Le but de ce travail est de permettre la réalisation de boucles de contrôle au sein d'applications développées avec le modèle iPOJO. Précisément, notre objectif est d'étendre iPOJO et son écosystème d'outils afin de faciliter le développement d'applications autonomiques ; c'est-à-dire capables de s'autogérer. Il s'agit donc essentiellement de fournir aux développeurs d'applications et de gestionnaires autonomiques les moyens d'implanter ou de spécifier aisément les propriétés d'autogestion. Cette solution ne doit pas contraindre le développeur et lui donner une grande flexibilité. De plus l'application administrée ne doit pas nécessairement avoir été développée avec l'idée d'être administrée. Cette contrainte demande de fournir au développeur toutes les informations et leviers pour la gestion autonomique.

Faciliter la mise en place de boucles de contrôle est un objectif ambitieux. En effet, l'implantation de ces boucles de contrôle demande de nombreuses fonctionnalités, telles que :

1. l'introspection de l'état de l'application – il est nécessaire de pouvoir comprendre l'état courant de l'application ainsi que son architecture ;
2. la connaissance de l'écosystème environnant – l'application s'exécute dans un environnement qu'il est nécessaire de connaître afin de prendre des décisions optimales ;
3. l'accès à des données métier – l'administration d'une application requiert la manipulation de données métier provenant de l'application mais également d'autres sources ;
4. la manipulation de l'application – l'application doit pouvoir être reconfigurée et son architecture modifiée ou contrainte ;
5. la manipulation de l'environnement d'exécution de l'application – non seulement l'application peut être manipulée, mais certains environnements d'exécution possèdent des primitives de reconfiguration ;
6. la protection de la cohérence de l'application lors des reconfigurations – l'état de l'application ne doit pas être corrompu lors d'une reconfiguration ;
7. la minimalisation de l'interruption de service – les reconfigurations doivent être appliquées en limitant au maximum les interruptions de service.

Il est nécessaire d'étendre iPOJO pour permettre au programmeur de réaliser ces nouvelles fonctionnalités de façon relativement aisée. Précisément, nous pensons qu'il est nécessaire de fournir aux développeurs de boucles autonomiques :

- une représentation **unifiée** de l'application et de son environnement,
- des moyens de reconfigurations fonctionnelles et architecturales,
- une représentation unifiée des données métier,
- des mécanismes de notifications afin de réagir aux différents changements.

La gestion autonomique d'applications iPOJO demande non seulement d'augmenter iPOJO, mais également de fournir des outils annexes de représentation et d'accès. Le rôle du canevas proposé est de fournir toutes les informations et leviers d'action nécessaires à la gestion autonomique. Cette représentation de la connaissance mélange des données techniques et métier qu'il est nécessaire de lier et de corréler.

Bien que notre thèse se focalise sur iPOJO, les concepts abordés sont transposables à d'autres modèles. Il faut également bien comprendre qu'iPOJO, est un outil programmable. Il est bien sur possible d'ajouter dans iPOJO un certain nombre de propriétés autonomiques génériques, comme l'est d'ailleurs la gestion des dépendances de service. Cependant, la majeure partie des propriétés autonomiques attendues sont métier et donc ne peuvent pas être implantées directement dans le modèle à composant. Au contraire, elles donnent lieu à du code spécifique écrit par les experts du domaine. Aujourd'hui ce travail est extrêmement complexe, faute de disposer des facilités et abstractions nécessaires.

Nous proposons d'utiliser le terme *Autonomic Ready* afin de définir une machine d'exécution supportant l'intégration facilitée et dynamique de gestionnaires autonomiques, plus précisément :

Définition 5 — Une machine d'exécution **Autonomic Ready** permet d'exécuter des applications fournissant les primitives requises à leur supervision par un gestionnaire autonomique, sans interruption.

Nous allons donc nous attacher à trouver des moyens pour rendre la machine d'exécution iPOJO *Autonomic Ready*.

4.3 Proposition

4.3.1 Présentation générale

Afin d'atteindre les objectifs précédents, nous avons modifié le *framework* iPOJO de façon à permettre aux développeurs d'insérer de multiples boucles de contrôle au sein d'applications dynamiques.

Chaque boucle de contrôle correspond à un gestionnaire autonomique tel que défini au chapitre 3 et est responsable de la gestion d'un aspect particulier. Elle comprend une partie introspection (*monitoring*), une partie décision et une partie action (*reconfiguration*), comme il se doit en informatique autonomique. Ces boucles de contrôle sont dirigées par des buts de haut niveau, souvent métier, définis par des administrateurs du système ou de l'application.

Pour chaque application, le programmeur peut ainsi créer une boucle de contrôle globale et des boucles de contrôle plus locales au niveau de chaque composant et liaison. Au niveau global, le rôle de la boucle de contrôle est de maintenir la topologie et les configurations les plus appropriées à l'application en fonction des buts d'admi-

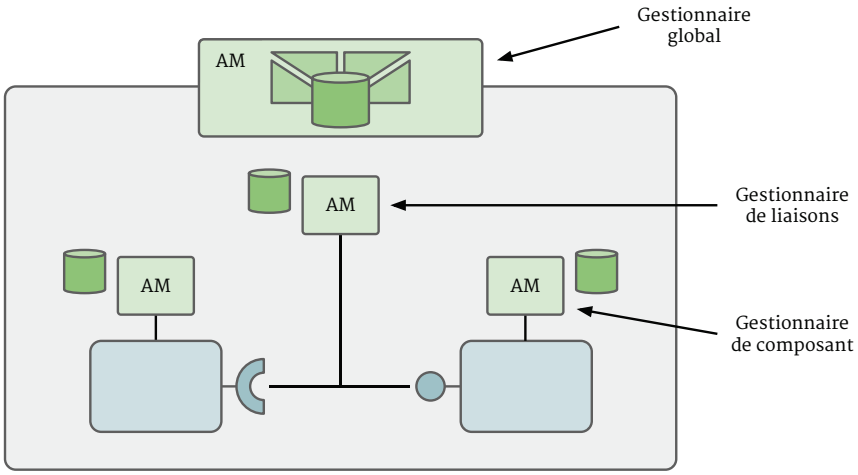

FIGURE 4.1 – Organisation des gestionnaires autonomiques

nistration, du contexte d'exécution et de la connaissance courante. Elle a également en charge la gestion des conflits entre les boucles locales. Au niveau des composants, le rôle des boucles de contrôle est de garantir l'utilisation des meilleures ressources en fonction des buts d'administration et de l'environnement. Ces boucles de plus bas niveau ont également un niveau de technicité plus important et peuvent prendre en charge des politiques de sécurité et de chargement de classes. Ces boucles sont locales et ne peuvent pas affecter d'autres éléments que les composants eux-mêmes. Au niveau des liaisons, des boucles de contrôle permettent de guider et de raffiner le choix des composants utilisés. Elles ne peuvent affecter directement les composants, mais interviennent dans l'établissement d'une ou plusieurs liaisons entre composants. Ce niveau est nécessaire pour permettre un établissement optimal de la topologie commandée par la boucle globale. De nombreux paramètres rentrent en ligne de compte lors de l'établissement de liaisons entre composants. La boucle globale ne peut pas gérer l'intégralité de ces aspects, cependant elle influence et contraint les liaisons grâce au gestionnaire de liaison.

Cette approche est illustrée par la figure 4.1 ci-dessus. Dans ce schéma, les boucles de contrôle sont symbolisées par l'acronyme *'AM'* (pour *Autonomic Manager*) classiquement utilisé en informatique autonomique. Remarquons que les trois boucles ont des bases de connaissances différentes. La boucle de plus haut niveau possède une connaissance plus vaste (mais moins précise) du système qui contient souvent des données métier. Pareillement, les langages et les techniques utilisées pour les gestionnaires globaux et locaux sont différents. Cela est dû à la nature des informations manipulées (au niveau de l'introspection et des adaptations), à la complexité des stratégies d'autogestion à exprimer, mais aussi aux différentes échelles de temps de réaction attendue.

Au final, un code d'application autonomique fondé sur cet iPOJO étendu comprend donc une partie spécifique à la logique métier de l'application (les composants), et une partie dédiée à l'autogestion de ces composants et de leurs liaisons. Cette seconde partie est dirigée par les buts de l'administrateur qui peuvent évoluer au cours du temps et ainsi modifier les politiques de gestion. Comme nous le verrons, ces deux parties peuvent interagir. Un enjeu important ici est de trouver le bon équilibre dans ces interactions de façon à ne pas impacter les performances ni complexifier le modèle

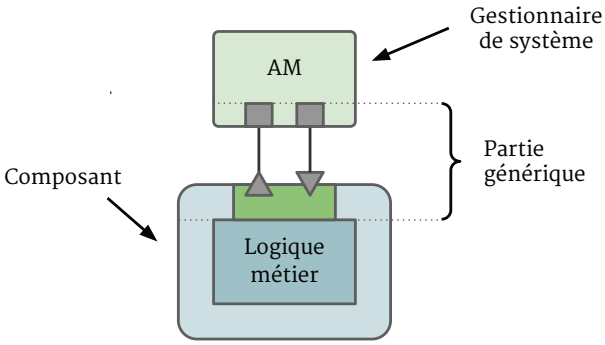

FIGURE 4.2 – Gestionnaire autonomique de composant

de développement tout en apportant les services d'autogestion attendus.

Chaque boucle de contrôle, ou gestionnaire autonomique, comprend une partie générique et une partie spécifique. La partie générique est directement implantée dans le code du framework iPOJO et fournit entre autre les mécanismes d'introspection et d'adaptation. La partie spécifique est à la charge des programmeurs métier et comprend le code purement applicatif.

Nous avons choisi de définir et de générer une couche d'abstraction entre les niveaux spécifiques et génériques de façon, justement, à faciliter l'écriture de la partie générique. La couche d'abstraction représente le système à l'exécution et maintien un modèle de celui-ci. Ce modèle peut être causal, mais cette propriété est extrêmement complexe à maintenir [4]. Si l'on replace ce modèle dans la boucle *MAPE-K*, il couvre les entités de *monitoring*, d'exécuteur, et fait également partie de la base de connaissance, comme le montre la figure 4.2. Les entités d'analyse et de planification sont à la charge du développeur car nécessite une connaissance pointue du domaine métier. Il en est de même pour le pilotage du *monitoring* et de l'adaptation.

Les gestionnaires locaux sont plus proches du système et le manipulent directement. Le code générique couvrant l'introspection et les reconfigurations est directement implanté dans le cœur du framework iPOJO. Ce code apparaît sous la forme d'interfaces d'introspection et de reconfiguration des composants et des liaisons. Ainsi il est possible de vérifier et de modifier la configuration des composants et des liaisons. Ces interfaces sont contrôlables de façon fine et dynamique.

La partie spécifique est fondamentalement différente entre les gestionnaires locaux et globaux. Bien que les deux utilisent le langage Java, qui apporte toute latitude pour créer du code avancé, les interfaces et abstractions sont différentes. Les gestionnaires locaux utilisent des accès de bas niveau et traitent des données techniques, alors que les gestionnaires globaux ont plus de connaissances métier et manipulent une représentation globale du système, tel que montré par la figure 4.3.

Nous insistons sur le fait que l'implantation des parties génériques est complexe. Il est important, par exemple, que l'introspection soit configurable pour correspondre au mieux à la situation courante et qu'elle n'impacte pas les performances au-delà des limites permises. De même, les adaptations doivent garantir la conservation des données et des états propres à l'exécution. Les différents types de boucles de contrôle

4. Les systèmes s'appuyant sur la machine virtuelle Java d'*OpenJDK* ne peuvent pas maintenir une représentation du système totalement causale à cause de limitations et de choix de conception de la machine virtuelle.

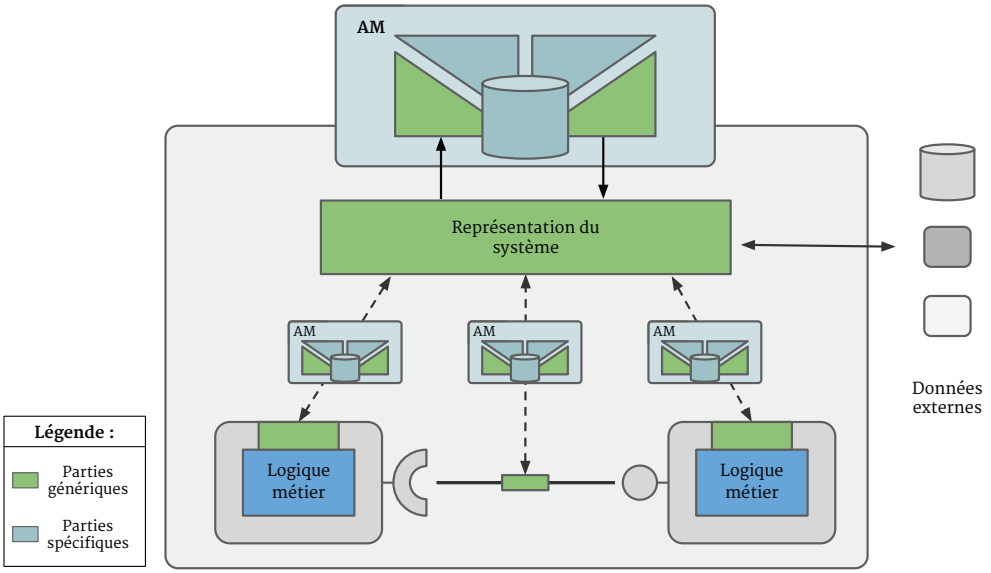

FIGURE 4.3 – Parties spécifiques et parties génériques de la gestion autonomique

demandent des techniques d'introspection, d'adaptation et de raisonnements différentes. Cela est dû à la nature des informations manipulées mais aussi aux différentes échelles de temps de réaction attendue.

Dans les sections suivantes, nous détaillons les éléments essentiels de notre proposition, à savoir :

- la notion de buts dirigeant les activités autonomiques,
- la notion de base de connaissance construite par iPOJO pour faciliter le code spécifique d'auto-gestion,
- les capacités d'auto-gestion au niveau des composants,
- les capacités d'auto-gestion au niveau des liaisons entre composants,
- les capacités d'auto-gestion au niveau applicatif global.

4.3.2 Buts d'administration

Le principe de l'informatique autonomique est de réduire le niveau d'intervention des administrateurs. Le rôle d'un administrateur est ainsi d'orienter le fonctionnement d'un système par l'intermédiaire de buts de gestion de haut niveau. Son stress et sa charge de travail sont ainsi réduits puisqu'il n'a plus à gérer des détails techniques de bas niveau qu'il maîtrise mal.

Les buts prennent généralement la forme d'objectifs métier devant être réalisés par le système. La plupart du temps, ils sont exprimés comme des critères caractérisant l'état d'un système. La détermination des actions à mettre en œuvre pour satisfaire ces critères et leur réalisation sont laissées à la charge des gestionnaires autonomiques. Dans notre cas, les buts correspondent à des critères permettant de qualifier et d'orienter l'exécution des applications supervisées. Les buts peuvent être décomposés en sous-buts permettant d'exprimer des directives plus précises.

On peut considérer deux types de buts. Tout d'abord, les buts et sous-buts génériques valables pour toutes les applications dynamiques développées avec iPOJO. Ces buts sont essentiellement liés aux actions génériques sur les composants et liaisons

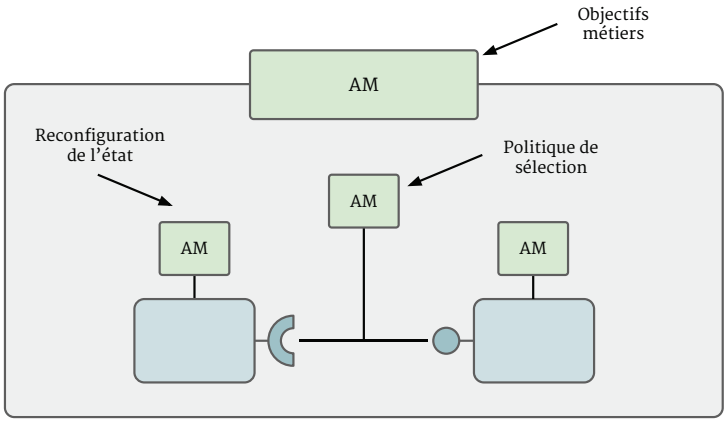

FIGURE 4.4 – Différences d'expression des buts en fonction du niveau de gestion

entre composants. Ils peuvent donner lieu à des réactions génériques, communes à toutes les applications. Ensuite, il y a les buts et sous-buts spécifiques aux applications qui sont directement définis par les programmeurs métiers. Ces buts spécifiques sont définis en même temps que les gestionnaires autonomiques (les boucles de contrôle). En effet, ce sont des critères qui sont explicitement utilisés par les gestionnaires pour orienter leurs réactions. On peut les considérer comme des paramètres d'entrée des gestionnaires autonomiques.

Il faut noter que les syntaxes utilisées pour exprimer les objectifs sont différentes suivant la cible. La boucle globale reçoit des taches exprimées dans ces concepts de haut-niveau. Les gestionnaires de liaisons reçoivent des stratégies exprimées sur des concepts métiers déjà traduits en concepts techniques. Enfin les gestionnaires de composants reçoivent des reconfigurations techniques.

Nous détaillerons dans la suite de ce document les différents formalismes retenus pour les buts en fonction des gestionnaires visés.

4.3.3 Représentation des connaissances et modèle du système

Un point crucial dans la gestion autonomique d'une application est la base de connaissance contenant l'ensemble des données requises afin de mener à bien la politique d'autogestion. Dans notre approche, comme nous l'avons vu, plusieurs boucles de contrôle coexistent. Elles reposent sur des connaissances différentes, issues de sources différentes et formalisées de façons différentes. Pour autant, on retrouve quelques critères essentiels dans la représentation de cette connaissance qui simplifient la conception et l'exécution de gestionnaires autonomiques :

- **Accès universel** : La grande variété des sources de contexte fait qu'il est nécessaire de prendre en compte des types de données très différentes, autant sur le plus sémantique que syntaxique. S'occuper de l'alignement de ces données est une tâche lourde et habituellement inefficace. L'accès aux informations de contexte doit donc reposer sur des concepts à la fois simples et universels.
- **Utilisabilité** : La représentation du contexte est le point névralgique d'un système autonomique, accédée à la fois par les applications sensibles au contexte, l'environnement d'exécution et l'administrateur du système. Cette représentation doit donc à la fois être facilement manipulable par les différents composants du système autonomiques, et de plus être compréhensible et utilisable par l'admi-

nistrateur. L'accès aux informations du contexte doit se faire au travers d'une interface applicative facile à prendre en main.
- **Transformable** : L'informatique ubiquitaire comprend une vaste diversité d'applications et de scénarii, qui nécessitent des points de vue souvent très variés, pour les mêmes informations de contexte. Les applications et gestionnaires doivent dont être en mesure de transformer la représentation universelle du contexte qui leur est proposée en une perspective plus adaptée, centrée sur ses besoins.
- **Relationnel** : Les entités formant le contexte sont étroitement liées, et en constante interaction. Ces relations entre les constituants du contexte représentent une connaissance importante, qui doit être représentée.
- **Dynamisme** : La sensibilité au contexte nécessite de capturer les changements intervenant sur le contexte. Sa représentation doit donc refléter ce dynamisme de l'environnement d'exécution et permettre son observation.
- **Extensibilité** : Les applications et systèmes évoluent. Il est illusoire de prétendre fournir une représentation satisfaisant toutes les applications existantes et celles à venir. La représentation du contexte doit donc permettre d'introduire de nouveaux concepts et de nouvelles entités.
- **Auto-descriptif** : L'accès à la représentation du contexte doit permettre de manipuler les entités sous-jacentes. La représentation doit donc décrire quels sont les moyens permettant de manipuler ces entités, et quelles actions ces entités supportent.

La plateforme d'exécution iPOJO ne fournit aucun support pour la représentation de la connaissance à l'exécution. Aujourd'hui, cette notion de connaissance, à l'intérieur des composants, est implicite, et sa représentation est à la charge du développeur. De plus, aucune représentation du contexte de la plateforme d'exécution n'est fournie, obligeant les gestionnaires autonomiques ainsi que les administrateurs des systèmes à devoir le récupérer par leurs propres moyens.

Nous proposons dans cette thèse la construction automatisée de connaissance, au niveau des composants, des liaisons entre composants et au niveau de l'architecture globale.

4.3.4 Adaptation autonomique des composants

Les capacités autonomiques que nous avons ajoutées à iPOJO concernent en premier lieu les composants iPOJO eux-mêmes. Rappelons que chaque composant iPOJO définit un ensemble de propriétés internes, un état de fonctionnement, des dépendances de services et des services fournis [5]. Dans cette section, nous présentons la façon dont nous avons conçu les gestionnaires autonomiques associés aux composants et les techniques d'introspection et d'adaptation utilisées.

Gestionnaire interne ou externe

Tout d'abord le gestionnaire autonomique d'un composant peut être inclus dans le conteneur du composant ou séparé. Lorsque le gestionnaire est inclus dans le conteneur du composant, il est connu lors du développement du composant, alors que dans le deuxième cas, le gestionnaire peut être déployé séparément (soit avant, voire en cours de fonctionnement). Il faut noter que bien que le gestionnaire interne,

5. Le modèle d'iPOJO est en fait plus complet, mais dans cette thèse nous ne nous sommes intéressés qu'à ces aspects qui sont les plus importants pour la gestion autonomique.

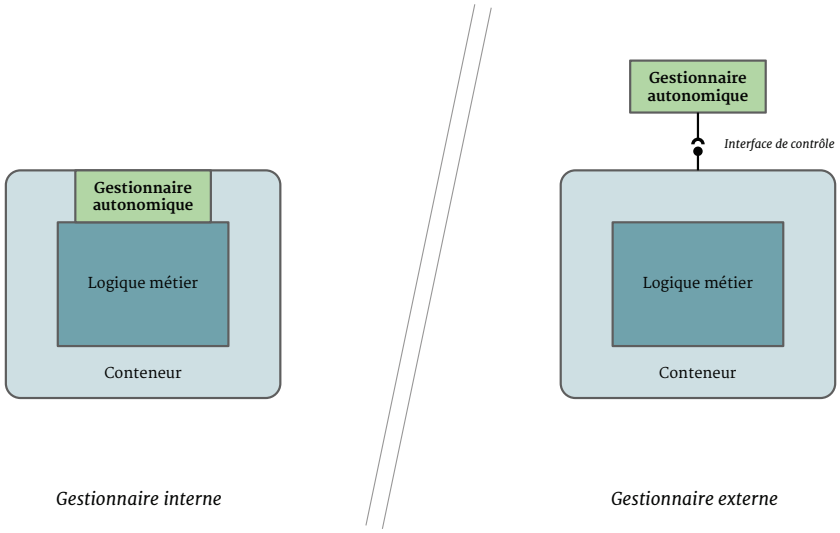

FIGURE 4.5 – Gestionnaires autonomiques internes et externes

c'est-à-dire faisant partie du conteneur, doit suivre le modèle de développement du conteneur iPOJO ce qui peut rendre le développement plus complexe mais lui donne accès à des primitives de synchronisation que le gestionnaire externe ne peut pas utiliser. En dehors de cette différence, les accès et actions possibles sont équivalents.

Lorsque le gestionnaire autonomique est externe, il s'appuie sur des primitives d'introspection et d'action fournies par iPOJO. Ces opérations, regroupées dans une interface de contrôle, fournissent la partie générique du gestionnaire.

Introspection

Le gestionnaire a besoin d'obtenir des informations sur le composant afin de remplir son rôle. Cependant, dans notre contexte, l'application est nativement dynamique ce qui implique des changements hors du contrôle du gestionnaire. C'est pourquoi les mécanismes d'introspection doivent non seulement permettre l'obtention de l'état courant mais également un mécanisme de notifications afin de prévenir le gestionnaire de tout changement.

La capture de l'état courant contient des informations techniques sur le composant administré, c'est-à-dire son état, la valeur des propriétés, les dépendances de services et leurs états et enfin les services fournis. Chacun de ces éléments peut changer dynamiquement. Par exemple l'état d'une dépendance de service dépend de l'ensemble des fournisseurs disponibles. Cet ensemble évolue avec le temps se qui influence directement l'état de la dépendance de service. Bien entendu, le gestionnaire choisi les notifications importantes pour sa logique d'adaptation.

Actions

Du fait du positionnement des gestionnaires autonomiques de composants, le gestionnaire a accès à des primitives d'administration de bas niveau afin de mettre une politique plus large orchestrée par le gestionnaire autonomique globale. Les objectifs reçus par le gestionnaire de composants peuvent influencer :

– les propriétés internes : la valeur des propriétés peut être changée,

– son cycle de vie : l'état du composant peut être influencé,
– les services requis : les dépendances de services peuvent être reconfigurées pour devenir plus ou moins spécifique, ou changer leur politique de résilience au dynamisme,
– les services fournis : la publication des services peut être contrôlée ainsi que les propriétés publiées.

À noter que l'état du composant est protégé durant ces reconfigurations, par un mécanisme proche de *tranquility* [Escoffier 2013, Vandewoude 2006]. L'ensemble des actions possibles est détaillé dans le prochain chapitre.

4.3.5 Adaptation autonomique des liaisons

La liaison entre composants iPOJO est une liaison à service. Il s'agit d'une liaison faiblement couplée pouvant évoluer dynamiquement en fonction des fournisseurs du service recherché et de critères variés (filtre, cardinalité, résilience au dynamisme, *etc.*).

Bien qu'il soit possible de manipuler les liaisons entre composants à partir des composants eux-mêmes (en manipulant les dépendances de services et les services fournis), le code requis est très technique et ne permet pas d'intégrer aisément des données et stratégies métier dans la résolution de la liaison. En effet, la sélection de service est exprimée en utilisant une syntaxe *LDAP* [6] non-appropriée aux sélections sophistiquées et métier.

Il serait aussi possible de laisser au gestionnaire autonomique gérer et adapter les liaisons, lui qui dispose d'une vue globale de l'application et de son architecture. La gestion des liaisons nécessite cependant la connaissance et la prise en compte de nombreuses données de contexte, dont certaines de bas niveau, par exemple :

– des propriétés/politiques du système, et de la plateforme d'exécution ;
– la configuration de l'application, des composants impliqués dans les liaisons ;
– les fils d'exécution (*threads*), notamment l'identité et/ou les préférences de l'utilisateur initiant un accès,
– le contexte ubiquitaire dans un sens plus large, comme par exemple la localisation de dispositifs, leur état de fonctionnement, *etc.*

Un gestionnaire autonomique dédié à une ou plusieurs liaisons va donc rassembler les informations disponibles au niveau local, tandis que le gestionnaire global va raisonner sur des informations de plus haut-niveau, en fonction des buts d'administration. L'adaptation des liaisons est guidée par le gestionnaire global, mais effectuée par le gestionnaire de liaisons, qui est directement en relation avec les entités à reconfigurer.

Positionnement

Les gestionnaires autonomiques de liaisons se branchent sur les dépendances de services d'un composant (ou d'un ensemble de composant) ainsi que sur les fournitures de service d'un (ou plusieurs) composants.

Ce positionnement particulier donne une grande latitude au gestionnaire pouvant ainsi suivre le dynamisme des services mais également impacter la sélection des fournisseurs utilisés par la dépendance. Un gestionnaire de liaison peut se contenter de ne gérer qu'une « demi-liaison », c'est-à-dire le côté fourniture de service, ou le

6. *Lightweight Directory Access Protocol*, dont la syntaxe de filtrage permet d'exprimer des contraintes de sélection sur les entrées de l'annuaire.

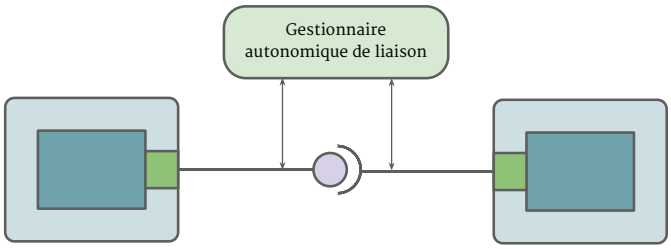

FIGURE 4.6 – Positionnement des gestionnaires autonomiques de liaison

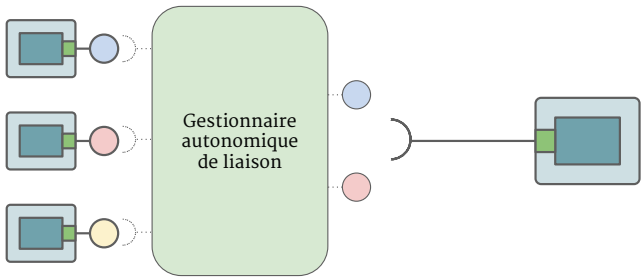

FIGURE 4.7 – Changement de la visibilité de services

côté dépendance de service. En pratique, rien n'empêche le même gestionnaire de gérer les deux « côtés » d'une ou plusieurs liaisons. Dans la suite de ce manuscrit, nous parlerons de gestionnaires de liaisons pour désigner indifféremment les gestionnaires de fournitures de service ou les gestionnaires de dépendances de services.

Conformément à la vision de l'informatique autonomique, les gestionnaires de liaisons sont déployés indépendamment de l'application, et peuvent être ajoutés ou enlevés dynamiquement. Les composants métier peuvent ainsi continuer de fonctionner sans l'aide des gestionnaires.

Logique d'adaptations

Les gestionnaires de liaisons contiennent la logique d'adaptation qui peut soit influencer la visibilité des fournisseurs de service, soit raffiner la sélection du ou des fournisseurs à utiliser. Cette logique peut être paramétrée par un ensemble de critères, potentiellement métier, fournit par le gestionnaire global.

Les gestionnaires de liaisons sont intégrés dans le processus de résolution de la dépendance de services et donc sont appelés à chaque changement (arrivée, départ ou modification d'un fournisseur de service). Ils peuvent également demander une réévaluation complète de la dépendance lorsqu'ils sont reconfigurés avec de nouveaux objectifs.

Ces politiques de liaisons vont permettre d'agir sur l'ensemble des services fournis par un composant, ou l'ensemble des services perçus par un composant. Par les actions concrètes de ces politiques, on retrouve notamment le changement de visibilité d'un service (figure 4.7) ou la modification du classement de services (figure 4.8). L'ensemble des opérations applicables sur les liaisons de services sera détaillé dans le chapitre suivant.

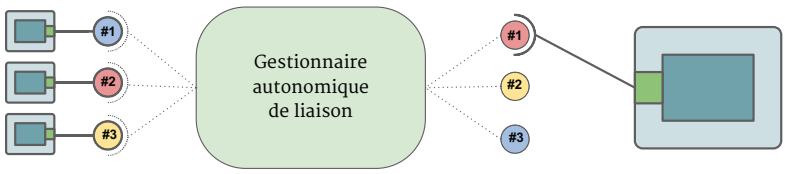

FIGURE 4.8 – Changement du classement de services

4.3.6 Adaptation autonomique globale

Comme indiqué précédemment, nous avons également décidé d'ajouter une boucle de contrôle globale pour gérer une application dans sa globalité. Il s'agit à ce niveau de gérer les paramètres de configuration et les différents éléments de l'application ainsi que sa topologie. Ce dernier point peut se traduire par exemple par l'ajout, le retrait ou le remplacement d'un composant mais également la reconfiguration des liaisons entre les composants. Une telle boucle de contrôle doit être capable de surveiller l'état global de l'application en cours d'exécution, de prendre des décisions d'adaptation si nécessaire, et de mettre en œuvre ces adaptations.

Introspection

Le gestionnaire autonomique global doit posséder une vue du système de haut-niveau lui permettant de décider des reconfigurations à effectuer. Cette vue contient non seulement l'ensemble des composants utilisés par le système et les liaisons entre eux mais également des données métier collectées soit par des composants spécifiques, soit récupérées à partir d'une source de données. Par exemple, dans le contexte de l'informatique ubiquitaire, le gestionnaire global a accès au contexte physique (température, luminosité, *etc.*).

Pour l'ensemble des données remontées, il est important pour le gestionnaire de pouvoir avoir une capture de l'état mais également un ensemble de notifications permettant de suivre l'évolution. Le gestionnaire compose sa propre vue et les notifications importantes en fonction de son but.

Adaptation

Le gestionnaire global a accès à un vaste panel d'adaptations possibles. Pour cela, il interagit avec un modèle de représentation de la machine d'exécution. Ce modèle fournit également un ensemble d'opérations d'adaptations de haut niveau.

Le modèle présenté est synchronisé, c'est-à-dire que les changements d'état de la plateforme d'exécution sont répercutés sur l'état du modèle, et vice et versa. Les actions de reconfiguration du gestionnaire global reposent sur les gestionnaires de composants et de liaisons présents.

4.3.7 Architecture de référence proposée

Dans le cadre de cette thèse, nous proposons de fournir les parties génériques permettant le développement et l'exécution d'applications autonomiques spécifiques.

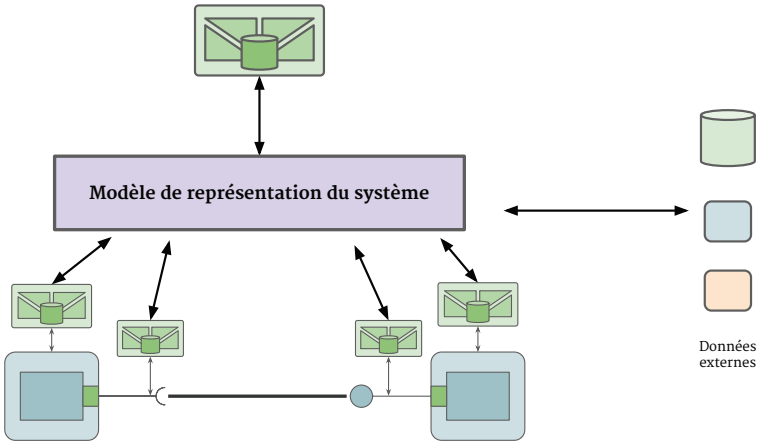

FIGURE 4.9 – Positionnement du modèle de représentation

Notre canevas propose donc un modèle de développement pour les applications autonomiques. L'architecture de référence des applications ubiquitaires autonomiques développées au-dessus de ce canevas repose sur les axiomes suivants :

1. **Séparation des préoccupations** : les aspects fonctionnels (métiers) des applications et leurs aspects autonomiques sont clairement séparés.
2. **Architecture à composants orientés services** : la partie métier des applications est construite sous forme d'un assemblage de composants fonctionnels (C). Chaque composant peut utiliser les services (S) disponibles sur la plateforme d'exécution et fournir de nouveaux services.
3. **Dynamisme de l'architecture** : les services sont dynamiques et peuvent apparaître, disparaître, changer à tout moment. L'assemblage de composants métiers, liés entre eux par des services, forment donc une architecture dynamique.
4. **Dynamisme des composants** : chaque composant métier pris indépendamment peut nécessiter une inspection de son état et une reconfiguration de cet état. Un composant doit donc être conçu en envisageant sa manipulation par un gestionnaire externe.
5. **Représentation du système** : le système doit fournir une représentation de lui-même (R). Cette représentation est un fondement de sa gestion par un gestionnaire autonomique. Un administrateur humain peut aussi avoir à inspecter cette représentation du système afin d'identifier des problèmes et adapter les politiques de haut-niveau.
6. **Gestionnaire autonomique d'application** : l'application est gérée par un gestionnaire autonomique *MAPE-K* global (AM_{global}). Ce gestionnaire offre une interface permettant à un administrateur de configurer les objectifs de haut niveau de l'application. Le gestionnaire global repose sa base de connaissance sur le modèle représentatif (R) de l'application.
7. **Gestionnaires autonomiques de composants, de liaisons** : Afin de décomposer les buts de haut-niveau en tâches plus spécifiques et techniques, le gestionnaire global (AM_{global}) s'appuie sur un ensemble de gestionnaires autonomiques *MAPE-K* de composants ($AMs_{composant}$) et de liaisons ($AMs_{liaison}$). Ceux-ci sont en charge,

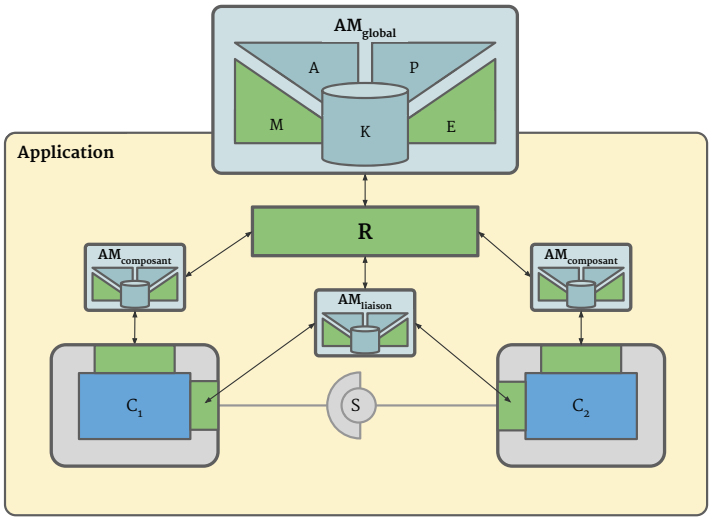

FIGURE 4.10 – Architecture d'une application autonomique selon le canevas proposé

respectivement, d'adapter les composants et les liaisons auxquels ils sont dédiés. Les gestionnaires de liaisons peuvent éventuellement à leur tour être décomposés en gestionnaires de fournisseur de services et en gestionnaires de dépendance de services.

La figure 4.10 illustre l'architecture d'une application conçue selon les principes énoncés ci-dessus. Les parties en gris sont les mécanismes existants, fournis par la couche d'exécution (points 1 à 4). Les parties en vert sont les extensions de cette couche d'exécution, qui permettent la mise en œuvre d'applications autonomiques. Ces mécanismes fournissent les parties génériques qui vont permettre le branchement des différents gestionnaires autonomiques sur l'application (points 5 à 7). Enfin, les parties en bleu représentent les parties applicatives spécifiques : elles comprennent le code métier des composants applicatifs, ainsi que l'implantation des différents gestionnaires autonomiques. Bien que la représentation du système soit considérée comme générique, elle contient de nombreuses données spécifiques issues des différents gestionnaires autonomiques. La couche d'exécution fournit un modèle de base représentant l'exécution de la plateforme ; ce modèle est ensuite étendu par les données provenant des bases de connaissance spécifiques des gestionnaires autonomiques.

4.4 Synthèse

La proposition de cette thèse vise à fournir les mécanismes permettant de simplifier grandement la mise en œuvre d'une application autonomique. Au développement, le code métier est découpé en composants liés par des interfaces de services. La gestion autonomique est séparée de la logique métier. La conception de la partie de gestion autonomique de l'application est centrée autour d'une représentation uniforme du système. Au sommet, le gestionnaire global traduit les buts de haut niveau de l'administrateur en tâches spécifiques, exécutées par les gestionnaires autonomiques de composants et de liaisons. À l'exécution, le système est en permanence représenté par le modèle central. Ce dernier est scruté et augmenté par tous les gestionnaires autonomiques de l'application.

Le chapitre suivant va s'attacher à préciser les parties spécifiques de la gestion autonomique en montrant quelles sont les données qu'elles remontent aux gestionnaires et les actions qu'elles permettent d'exécuter. Les caractéristiques du modèle central de représentation du système et ses interactions avec les différents gestionnaires autonomiques seront aussi expliquées.

Chapitre 5

Extension de la machine d'exécution iPOJO

Le chapitre précédent a décrit les principes permettant une conception facilitée d'applications autonomiques. La machine d'exécution doit pouvoir offrir des mécanismes élémentaires d'adaptation des composants pendant leur fonctionnement. En offrant un modèle d'exécution adaptable, les gestionnaires autonomiques gagnent en simplicité d'expression, et peuvent ainsi remonter des informations plus pertinentes et exécuter des actions de plus haut-niveau, sans devoir se soucier du fonctionnement de la couche d'exécution. Le modèle à composants à services iPOJO fournit un certain nombre de points d'adaptation pour les composants, et constitue donc un candidat idéal pour l'ajout des principes Autonomic-Ready susmentionnés.

Ce chapitre va maintenant détailler la mise en œuvre de ces mécanismes au sein du *framework* iPOJO. Les modifications proposées vont donc permettre de rendre la couche d'exécution suffisamment adaptable pour l'intégration naturelle de boucles de contrôle autonomiques.

Sommaire

5.1 Rappels

Dans le cadre de cette thèse, nous avons choisi comme fondation l'approche à composants orientés service en nous appuyant sur le *framework* iPOJO. Ce dernier favorise la modularité des applications et le couplage faible entre composants ; autant de qualités qui facilitent l'adaptation des applications et leur gestion par un gestionnaire autonomique. La version d'iPOJO de base n'offre cependant pas toutes les caractéristiques permettant la conception et le support à l'exécution d'applications autonomiques.

Comme nous l'avons vu précédemment, la clé de l'adaptation d'un système repose sur la capacité de sa machine d'exécution à exposer un modèle satisfaisant aux différents gestionnaires autonomiques. Ce modèle se doit de remplir certain critères fondamentaux, afin de permettre la surveillance et la modification du système par le gestionnaire, ainsi que de faciliter sa conception et son évolution :

- le modèle doit être **réflexif**. Il doit permettre l'introspection de l'état de la plateforme : non seulement il doit être possible de récupérer, à tout moment, un instantané de l'état actuel, mais il est nécessaire de pouvoir être notifié lorsque certains changements d'états, significatifs pour le gestionnaire autonomique, surviennent. Le modèle doit aussi permettre l'intercession, c'est-à-dire la modification du comportement de la plateforme, pour mettre en œuvre les adaptations commandées par le gestionnaire autonomique.
- le modèle doit fournir un **niveau d'abstraction** satisfaisant : les données remontées au gestionnaire autonomique ainsi que les structures qu'il manipule doivent écarter les considérations techniques de bas niveau, spécifiques à la plateforme. La simplicité du modèle est un gage de facilité de son utilisation. Il doit cependant offrir un accès complet à tous les éléments s'exécutant sur la plateforme.

La plateforme d'exécution Apache Felix iPOJO a été conçue dans le but de faciliter la conception des applications dynamiques à base de composants à service. Elle favorise la gestion de la modularité et du dynamisme des applications tout en masquant sa complexité derrière un modèle de développement simple, focalisé sur le code métier. La gestion des aspects techniques, le dynamisme notamment, est une tâche complexe et source de nombreuses erreurs (*deadlocks*, *race conditions*, *etc.*). iPOJO encapsule le code fonctionnel à l'intérieur d'un conteneur, et permet aux applications de s'exécuter naturellement dans des environnements hétérogènes et dynamiques.

Comme indiqué dans le chapitre précédent, nous avons décidé d'étendre iPOJO pour permettre le développement aisé de boucles de contrôle multiples. Ces boucles de contrôle doivent être en mesure de récupérer de manière simple les informations sur les éléments qu'elles gèrent : composant, liaison ou application. De plus, les composants et les liaisons doivent pouvoir être adaptés par ces mêmes gestionnaires. Les gestionnaires récupèrent donc des informations de contexte depuis les éléments gérés (entre autres) et adaptent ces éléments en fonction de ce contexte. La principale manifestation de cette difficulté d'adaptation concerne l'adaptation de l'architecture des applications. Les dépendances de service sont exprimées sous forme d'interface et de filtres de sélection. iPOJO permet au développeur d'un composant de spécifier ce filtre de sélection, et au gestionnaire autonomique de le modifier à la volée. Cependant, le niveau d'expression de ce filtre, et sa syntaxe rébarbative (basée sur *LDAP*) font qu'il est, en pratique, peu utilisé par les concepteurs de composants, qui préfèrent effectuer la sélection de manière programmatique. De plus, l'écriture de gestionnaire

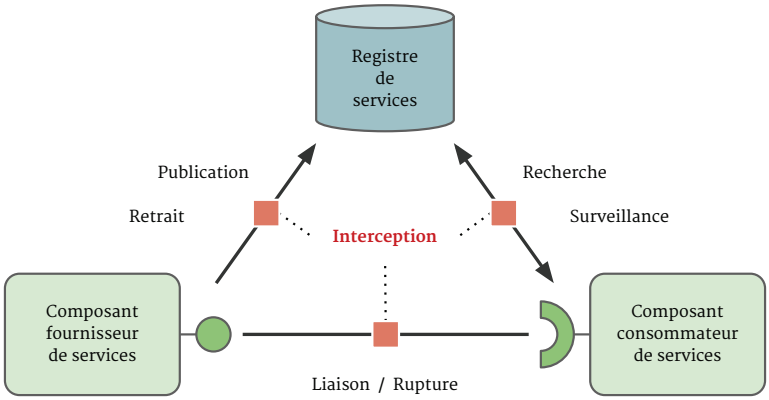

FIGURE 5.1 – Interception des appels au registre de services

autonomique s'en trouve grandement complexifiée : ces filtres ne permettent pas d'inclure des variables de contexte, et le gestionnaire doit donc le rafraîchir à chaque fois qu'un changement contexte est susceptible de modifier la sélection de services couramment utilisée. Ceci fait que les filtres de sélection calculés par le gestionnaire autonomique sont souvent de très grande taille, ce qui amène des problèmes de performances pour la plateforme, et de lisibilité pour l'administrateur.

5.2 Extensions requises et contraintes

La mise en œuvre d'un modèle réflexif nécessite de changer la manière dont les composants utilisent et fournissent des services au sein de la plateforme. Dans le schéma classique de l'informatique orientée service (*SOC*), le principal interlocuteur est le registre de service, contenant les descriptions et les références vers tous les services. C'est ce registre qui est en charge d'enregistrer les publications de services, de répondre aux requêtes de services et d'assurer la liaison entre fournisseurs et consommateurs de service. Le *SOC* dynamique impose de plus de pouvoir surveiller l'évolution du contenu de ce registre : arrivées, départs et modifications de services.

Un gestionnaire autonomique devant gérer des composants à services dynamiques va donc devoir, afin de permettre de scruter et d'altérer le comportement de la plateforme à service, se placer entre les composants à gérer et le registre, comme illustré dans la figure 5.1. Cette interception va comprendre deux axes principaux, qui vont conditionner les actions que le gestionnaire autonomique va pouvoir effectuer :

1. L'interception des fournisseurs de service va permettre au gestionnaire de s'informer des capacités offertes par le composant. Le gestionnaire peut alors intervenir en modifiant le service fourni avant sa publication dans le registre. Par exemple, le gestionnaire peut ajouter des informations non-fonctionnelles, comme la qualité de service, qui pourront aider les consommateurs à mieux choisir les services qu'ils utilisent.
2. L'interception des consommateurs de service va permettre au gestionnaire d'obtenir et de modifier la liste de services perçus par la dépendance de service du consommateur. Un service de mauvaise qualité pourra par exemple être masqué, ou bien dévalué. Le gestionnaire autonomique, conscient du contexte, va pouvoir

adapter ainsi l'architecture entre les composants, en changeant leur façon de voir les services environnants.

La mise en œuvre de ce mécanisme d'interception va améliorer la réflexivité de la plateforme, la rendant plus facile à adapter en fonction du contexte. Certaines contraintes doivent cependant être respectées, afin, principalement, de conserver tous les avantages offerts par l'approche à composant orientés service, et par iPOJO plus particulièrement :

- Il ne doit pas y avoir d'impact sur le modèle de développement de composants. La façon de concevoir les composants à service iPOJO doit rester absolument identique. Le code fonctionnel ne doit pas se préoccuper des interceptions du gestionnaire autonomique. Celles-ci doivent donc être transparentes et permettre la gestion de composants existants, développés avant cette proposition (rétro-compatibilité descendante).
- Les intercepteurs doivent pouvoir être mis en place ou retirés dynamiquement. C'est au gestionnaire autonomique de déterminer quel ensemble de composant et de services il doit surveiller. Suivant l'évolution du contexte, cet ensemble peut changer. Le gestionnaire doit pouvoir par exemple, intercepter de nouveaux composants.
- L'impact de l'ajout du mécanisme d'interception doit avoir un impact mineur sur les performances d'exécution des composants. La plateforme iPOJO étant utilisée dans de nombreux dispositifs aux capacités restreintes (smartphones, tablettes, set-top-box, *etc.*), ou dans des contextes d'utilisation intensive (serveurs d'applications), l'interception doit être la plus imperceptible possible.

Le modèle de représentation des ressources doit, quant à lui, exposer toutes les propriétés définies précédemment (cf. section 5.4). L'introduction de cette nouvelle fonctionnalité ne doit pas non plus changer le modèle de programmation d'iPOJO : les composants développés avant l'introduction de ce modèle de représentation doivent continuer à fonctionner normalement, sans aucune adaptation du code métier.

Ce modèle de représentation doit par contre être assez souple pour servir de socle à la constitution de gestionnaire autonomique, en lui permettant de récolter, de surveiller et de modifier sa base de connaissance. Il doit de plus supporter la collaboration entre gestionnaires hiérarchiques et/ou distribués, en évitant d'exposer des problèmes de cohérence et de d'inter-blocage lors d'accès concurrents.

Organisation des contributions

L'architecture de référence d'une application autonomique, telle que proposée par notre modèle de développement, a été détaillée dans le chapitre précédent (section 4.3.7). Les parties spécifiques d'une application autonomique sont, d'après cette architecture, en relation avec quatre types d'entité générique, à savoir :

1. Les **conteneurs de code métier** des composants. Ces parties sont chargées de permettre aux gestionnaires de composants d'interagir avec le code métier du composant, et de surveiller son état.
2. Les **dépendances de services** permettent aux composants d'utiliser les services de la plateforme. Les gestionnaires de liaisons peuvent surveiller quels sont les services utilisés, et changer la façon de sélectionner et d'utiliser ces services.
3. Les **fournitures de services** permettent aux composants de fournir un service. Les gestionnaires de liaison peuvent surveiller les services fournis et modifier la façon dont le service est fourni.

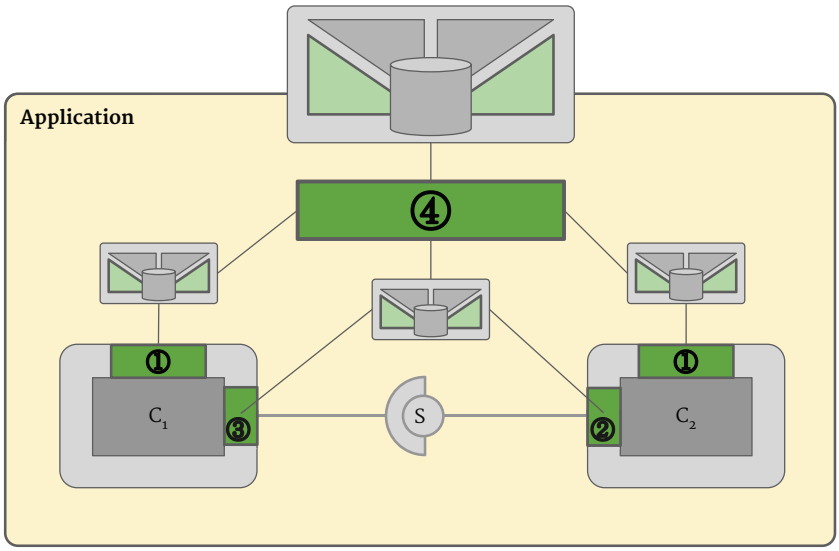

FIGURE 5.2 – Parties génériques d'une application autonomique

4. La **représentation de la plateforme d'exécution** fournit aux différents gestionnaires autonomiques une vue de l'état courant de l'application et du système. Les gestionnaires basent leur management de l'application sur les informations disponibles dans ce modèle de représentation. Ils peuvent aussi enrichir ce modèle avec leur propre connaissance.

Ces parties génériques sont le pivot d'une gestion autonomique simplifiée. Les parties *Monitoring* et *Execution* des différentes gestionnaires autonomiques deviennent très réduites, voire complètement génériques : les mécanismes d'instrumentation du système à gérer sont entièrement contenus dans les entités génériques énoncées ci-dessus. La figure 5.2 rappelle l'emplacement de chacune des différentes parties spécifiques de l'application autonomique.

Dans la suite de ce chapitre, chacune des quatre entités génériques sera présentée. Les opérations élémentaires qu'elles fournissent ainsi que les mécanismes qui permettent ces opérations seront détaillés. La section 5.3.1 montre comment la plateforme d'exécution est rendue réflexive, et comment cela permet aux gestionnaires d'interagir avec les conteneurs de composant (point 1). L'interception des dépendances de services (point 2) est détaillée dans la section 5.3.2, celle des fournitures de services (points 3) dans la section 5.3.5. L'impact de ces propositions sur la notion d'architecture applicative est analysé dans la section 5.3.6. La section 5.4 plonge dans le modèle de représentation (point 4) et montre quelles sont les possibilités permettant la consultation et l'enrichissement de ce modèle.

5.3 Introspection par l'API

La machine d'exécution iPOJO offre un certain niveau d'autonomie aux composants métiers, en fournissant une gestion du dynamisme des services. Chaque composant est géré par un gestionnaire dédié qui va gérer cet aspect non-fonctionnel essentiel pour l'évolution du composant, de la plateforme, et donc du système. Ce gestionnaire est cependant limité, dans la mesure où il n'offre pas les capacités réflexives nécessaires

au branchement d'un gestionnaire autonomique. Il faut donc instrumenter la machine d'exécution, la doter de nouveaux types de capteurs et d'effecteurs réflexifs afin d'autoriser une gestion autonomique simplifiée et efficace.

Cette instrumentation se développe en trois axes majeurs, à savoir :

1. L'**interception du code métier du composant**, de ses changements d'état et de ses accès. Ceci va permettre à un gestionnaire autonomique spécifique, connaissant l'implémentation du composant, de le surveiller et d'agir sur le code métier du composant.
2. L'**interception des dépendances de service** du composant. Elle va permettre au gestionnaire de liaison de gérer quels sont les services qui peuvent être utilisés par la dépendance de service. Elle permet aussi de modifier la vision qu'a la dépendance des services qui lui sont proposés.
3. L'**interception des fournitures de service** du composant. Les services fournis par un composant peuvent être modifiés par le gestionnaire de liaison. Ces modifications sont transparentes pour le composant qui expose le service, mais sont visibles par le registre de service et tous les autres composants de la plateforme.

5.3.1 Interception au niveau du conteneur

Un gestionnaire autonomique peut avoir besoin d'accéder au cœur métier d'un composant. Si cette assertion peut paraître étrange, tant les gestionnaires autonomiques courants sont présentés comme étant génériques et indépendants de l'application, il y a de nombreux cas où cela est nécessaire. Cela nécessite donc que la machine d'exécution fournisse un moyen efficace de surveiller le code métier, et de pouvoir modifier son comportement.

Dans le cadre d'iPOJO, le modèle de développement est basé sur le concept de POJO. Le gestionnaire doit donc pouvoir intercepter tous les accès à ces objets, c'est-à-dire :

- La construction de nouvelles instances de ces objets.
- Les accès à l'état de ces objets : lectures et modifications des valeurs de leurs champs.
- Les interactions entre objets, matérialisés par des appels de méthodes.

De plus, iPOJO permet l'ajout de *'handlers'* dans le conteneur des composants. Ces handlers sont des gestionnaires qui soutiennent l'exécution du composant en gérant un ou des aspects non-fonctionnels. C'est par exemple ainsi que sont gérées les fournitures et les dépendances de service. Ces handlers doivent, afin d'agir de manière transparente avec le code métier, avoir accès au même type d'interception que le gestionnaire autonomique. Ils peuvent d'ailleurs être considérés comme des proto-gestionnaires autonomiques miniatures. Ils peuvent cependant être nécessaires à l'exécution du composant et de l'application.

iPOJO fournit donc déjà, afin de permettre aux *handlers* d'interagir avec le code métier, des mécanismes d'interception basiques, permettant de réaliser les trois types d'interception mentionnés ci-dessus. La façon dont ces mécanismes sont fournis ne convient cependant pas au branchement d'un gestionnaire autonomique pour plusieurs raisons :

- Le gestionnaire ne peut pas empêcher l'appel d'une méthode. Il peut seulement être notifié lorsque l'appel est initié et lorsqu'il se termine (succès ou erreur). De plus, les paramètres d'appels de la méthode ne peuvent être modifiés.
- L'ordre d'interception entre le gestionnaire autonomique et les *handlers* est complexe à maintenir. Les *handlers* sont annotés avec une priorité, qui détermine

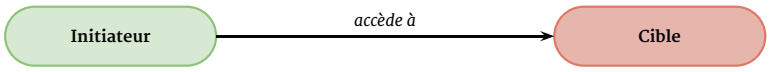

FIGURE 5.3 – Accès classique

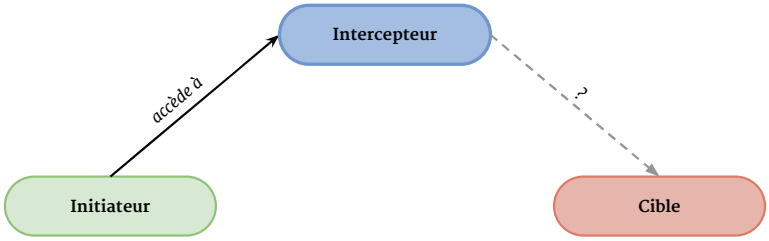

FIGURE 5.4 – Accès intercepté

dans quel ordre ils seront appelés. Ces priorités sont statiques, et, dans le cas où des *handlers* supplémentaires sont nécessaires, peuvent générer des conflits lors des interceptions. Selon le moment où le gestionnaire commence à intercepter un composant, il peut être notifié soit avant, soit entre, soit après les *handlers*.

– L'interception des accès aux attributs du POJO se fait, entre les *handlers*, de manière collaborative. Dans le cas d'une écriture par exemple, si la valeur à écrire est modifiée par deux *handlers* différents, une erreur cd conflit peut survenir.

Ces limitations du modèle d'interception actuel d'iPOJO sont la manifestation de la vision originelle de cette plateforme. Le gestionnaire autonomique était vu comme un *handler* comme un autre, gérant l'aspect autonomique du composant, au même rang que les autres aspects non-fonctionnels (dynamisme des services, persistances, administration, *etc.*). Or le gestionnaire, conscient du composant, de l'application et de la plateforme dans sa globalité, joue le rôle d'arbitre entre le code métier, les *handlers* et les adaptations au contexte. Il doit donc avoir la possibilité d'intercepter à la fois le POJO métier et les *handlers*, afin de pouvoir résoudre les éventuels conflits, selon le contexte.

Interceptor pattern

Afin de pouvoir agir sur le comportement des composants en toute transparence, et sans modification du code métier, le gestionnaire autonomique doit pouvoir intercepter les accès à ce dernier. Le patron architectural *interceptor pattern* permet, en se positionnant entre un élément cible et les entités qui y accèdent, d'interception toute communication.

La figure 5.3 illustre, de manière très abstraite, l'accès d'une entité à une autre, qui doit être interceptée. La figure 5.4, quant à elle, montre l'architecture de ces deux mêmes entités une fois qu'un intercepteur a été ajouté entre elles. Cette interception est transparente dans le sens où :

– L'élément cible **ne sait pas** que les accès le concernant sont interceptés, et ne change donc rien à sa manière de réagir aux sollicitations extérieures.

– L'élément initiateur accède **exactement de la même manière** à l'entité cible,

seule la destination de l'accès change. Dans de nombreuses mises en œuvre, l'intercepteur est d'ailleurs masqué aux éléments initiateurs, qui pensent donc interagir avec l'élément cible réel.

La figure 5.4 montre également que l'intercepteur est libre de transmettre les accès interceptés à l'entité ciblée. Il peut aussi ignorer l'accès, ou accéder différemment à la cible, du moment que ce comportement reste transparent du point de vue de l'initiateur. Ce patron architectural est donc un excellent choix pour l'interception par le gestionnaire autonomique des accès aux éléments à gérer.

Une extension de ce patron, non décrite dans sa formulation d'origine, peut d'ailleurs donner un rôle plus actif à l'intercepteur, en lui permettant d'initier lui-même des accès à la ressource cible. Il pourra ainsi effectuer les opérations autonomiques de base, à savoir :

- La remontée d'informations concernant l'élément à gérer, et sa surveillance.
- L'exécution d'actions sur cet élément, initiée par le gestionnaire autonomique via l'intercepteur actif.

L'application quasi-littérale de ce patron architectural devrait, en théorie, pouvoir donner aux gestionnaires autonomiques toute les libertés d'action nécessaires pour administrer les éléments à gérer. Toutefois les *handlers* d'iPOJO ont besoin eux aussi de pouvoir intercepter les accès du composant métier. Ces interceptions peuvent entrer en conflit avec les politiques de gestion du gestionnaire autonomique. Il faut donc un moyen de permettre des interceptions par plusieurs intermédiaires, gestionnaire autonomique du composant et *handlers*, tout en donnant au gestionnaire autonomique la possibilité de réagencer les différentes interceptions.

Chaînes d'interception

Le modèle à composant doit donc permettre à la fois aux *handlers* existants et au gestionnaire autonomique du composant d'intercepter tous les accès au code métier. Si, dans de rares cas, les *handlers* peuvent avoir conscience les uns des autres, ils ignorent totalement la présence du gestionnaire autonomique, et encore moins ses intentions. Comment donc faire cohabiter ces différents acteurs au sein du conteneur en permettant à chacun d'intercepter les accès au code métier ?

Le concept de **chaîne d'interception** permet à plusieurs intercepteurs de cohabiter de manière relativement transparente. De plus, la manipulation de la chaîne d'interception va permettre de réorganiser l'orchestration des différents intercepteurs. Seul le gestionnaire autonomique a la connaissance nécessaire afin d'effectuer ces réagencements de manière fiable, sans gêner les *handlers* ni compromettre leur cohérence ou celle du code métier.

Une chaîne d'interception, comme son nom l'indique, va enchaîner les différents intercepteurs les uns aux autres. Lors d'un accès, les différents intercepteurs seront appelés successivement, suivant l'ordre préétabli de la chaîne. Cependant, chaque intercepteur peut avoir le choix d'altérer l'accès :

- Soit en modifiant les paramètres de l'accès (type ou propriétés d'un message par exemple), ce qui affectera tous les intercepteurs suivants.
- Soit en terminant prématurément la chaîne d'interception, évitant ainsi les intercepteurs suivant mais aussi l'élément cible final.

Cette chaîne devient donc, au niveau architectural, le seul intercepteur entre les éléments initiant les accès et les éléments cibles, comme le montre la figure 5.5. C'est le contenu de la chaîne d'interception, géré par le gestionnaire autonomique, qui va

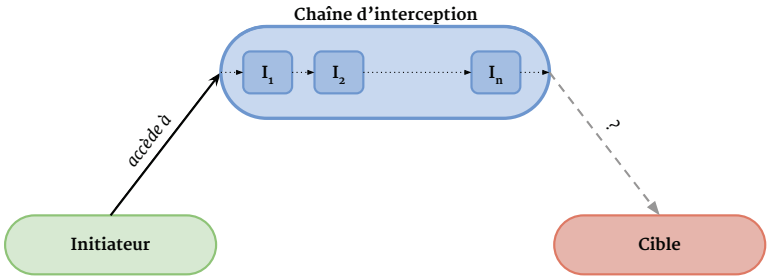

FIGURE 5.5 – Accès intercepté par une chaîne d'interception

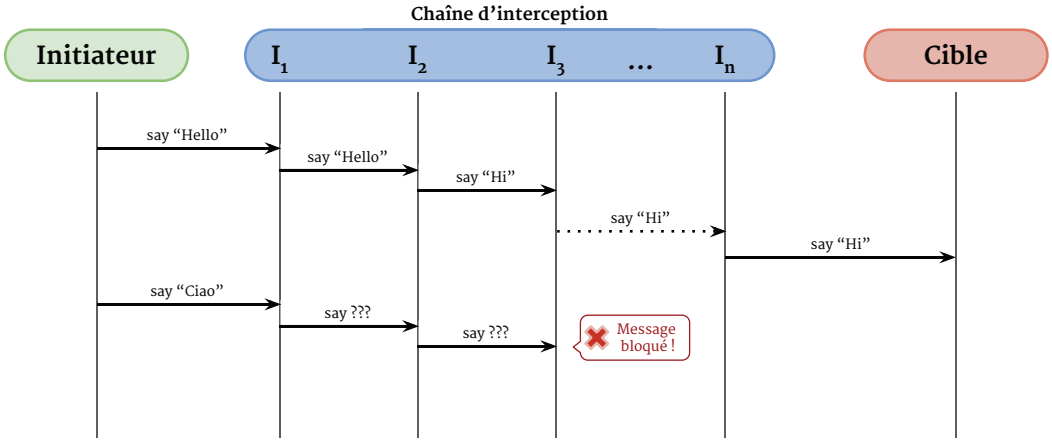

FIGURE 5.6 – Exemples d'interception de messages par une chaîne d'interception

permettre l'orchestration entre les différents intercepteurs : ceux émanant des *handlers* ainsi que ceux introduits par le gestionnaire autonomique dédié au composant. Ce gestionnaire a donc le contrôle total de la chaîne, et peut :

- Changer l'ordre des intercepteurs, en ajouter dynamiquement, en enlever également.
- Modifier, entre chaque intercepteur, les paramètres d'accès.
- Terminer prématurément la chaîne d'interception, ou bien, contre les directives d'un intercepteur, la poursuivre malgré tout.

À ce grand pouvoir du gestionnaire incombe aussi une grande responsabilité : celle de maintenir, pour chaque interception, la cohésion de l'ensemble. Encore une fois, seul le gestionnaire autonomique peut avoir la connaissance suffisante pour effectuer cette orchestration sans risque ; cela ne garantit pas, en pratique, qu'il en soit toujours capable.

La figure 5.6 illustre deux interceptions de messages par une chaîne d'interception. Le premier message intercepté traverse la chaîne et atteint l'élément ciblé, avec toutefois quelques modifications. Le deuxième message est quant à lui « avalé » par la chaîne et n'atteindra donc jamais sa cible.

La combinaison de l'application de l'*interceptor pattern* et de l'utilisation de chaînes d'interception permet donc au gestionnaire d'intervenir à tous les niveaux entre les

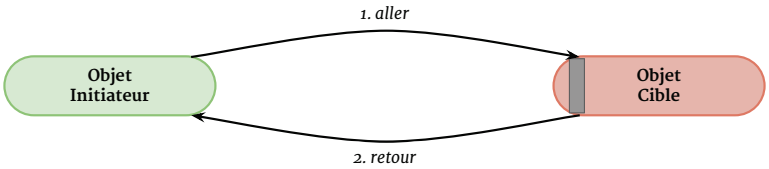

FIGURE 5.7 – Accès synchrones aux objets

initiateurs, les *handlers* et les cibles. Si l'interception de messages se prête particulièrement bien à ce mécanisme, son application au paradigme promu par iPOJO, le concept de POJO, demande cependant encore un effort d'adaptation.

L'interception en programmation orientée objet est une approche adoptée par de nombreux *middlewares*, principalement pour l'injection de dépendances et/ou de contexte. On peut par exemple citer la spécification Java *EE CDI* [1], le framework Unity pour les technologies Microsoft *.NET* [2], et plus généralement l'application des principes de la programmation orientée aspects (*AOP*) [3].

Application à la « philosophie POJO »

iPOJO repose sur une approche de développement et d'exécution de composants développés et exécutés sur des plateformes orientées objet (en l'occurrence Java et *OSGi*). La programmation orientée objet définit des types d'accès particuliers sur les objets : les accès aux **attributs** de l'objet, et les **appels de méthodes** sur ces objets. La plupart des langages orientés objet (Java, Smalltalk, C#, *etc.*) autorisent nativement les **accès de type synchrone** aux objets, ce qui est plus naturel et facile à mettre en œuvre. Les accès de type asynchrone restent possibles en utilisant des bibliothèques spécialisées, basées sur les accès synchrones natifs et les possibilités de créer de nouveaux fils d'exécution [4].

Les accès synchrones aux objets imposent à l'initiateur de l'accès d'attendre une réponse de la cible. Cette réponse peut contenir des données relatives au résultat du traitement de l'appel d'une méthode, ou de la lecture de la valeur d'un attribut. Certains accès ne renvoient cependant aucune donnée ; ils doivent cependant attendre que la cible ait terminé le traitement de l'accès. Les accès synchrones peuvent donc être décomposés en deux parties, comme illustré par la figure 5.7 :

- L'**accès aller** est le message allant de l'initiateur de l'accès jusqu'à la cible de ce même accès. Ce message contient la cible de l'accès, le type d'accès ainsi que des paramètres liés au type d'accès (nom de l'attribut, lecture/écriture, nom de la méthode, paramètres de l'appel de méthode, *etc.*).
- Le **retour d'accès** est le message correspondant à la réponse de la cible à l'accès débuté par l'initiateur. Ce message peut contenir le résultat de l'exécution de la méthode cible, ou de la lecture de l'attribut ciblé, mais pas nécessairement. Ce message a aussi pour but de signaler la fin de l'accès à l'initiateur. Ce dernier étant bloqué durant toute l'exécution de l'accès, il ne peut recommencer son exécution qu'après réception du message retour.

1. `http://docs.oracle.com/javaee/6/tutorial/doc/giwhb.html`
2. `http://msdn.microsoft.com/en-us/library/ff649614.aspx`
3. `http://aopalliance.sourceforge.net/doc/org/aopalliance/intercept/package-summary.html`
4. Appelés aussi processus légers, ou plus couramment *'threads'*.

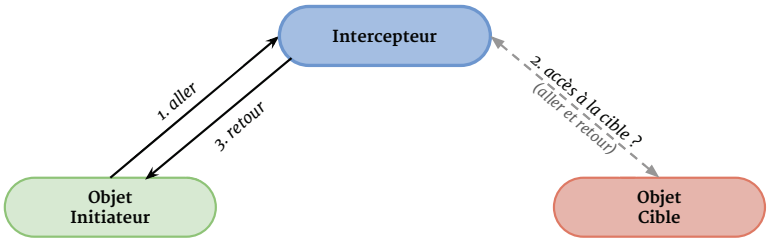

FIGURE 5.8 – Mise en œuvre de l'interception des POJOs

Du point de vue de l'interception de ces accès synchrones, les intercepteurs doivent être en mesure de surveiller, modifier ou capturer ces deux parties des échanges. Il en résulte que chacun des éléments de la chaîne d'interception doit pouvoir effectuer des opérations avant la propagation du message aller, et avant la propagation du message retour. La terminaison prématurée de l'accès, opération quelquefois nécessaire, consiste à émettre un message retour au lieu de propager le message aller à la suite de la chaîne d'interception.

En terme de *POJO*, les différents types d'accès que le gestionnaire autonomique doit être en mesure d'intercepter sont les suivants :

- Les **appels de méthodes** : l'initiateur souhaite exécuter une méthode particulière du *POJO* ciblé. L'initiateur peut paramétrer cet appel de méthode, et peut nécessiter une valeur de retour, garantissant la bonne exécution de la méthode, ou une exception, lancée, par exemple, en cas d'erreur. Le type des paramètres, le type de retour ainsi que le type des exceptions pouvant être lancées par la méthode du POJO définissent la signature cette méthode. Les intercepteurs, afin de préserver la transparence, doivent impérativement respecter cette signature afin de ne pas perturber l'exécution de l'initiateur.
- Les **accès aux attributs** : ils peuvent être des accès en lecture ou bien en écriture. Dans le cas d'une lecture, seul le nom de l'attribut demandé est passé en paramètre de l'accès. Le type de l'attribut est fixé par la définition du *POJO*, et les intercepteurs ne peuvent donc pas renvoyer n'importe quoi à l'initiateur de l'accès, ni accéder n'importe comment à la cible, spécialement dans le cas d'une écriture d'attribut.
- La **construction d'objets** : chaque *POJO* est créé par un constructeur, méthode spéciale permettant de configurer l'état initial de l'objet. Le framework iPOJO permet d'injecter des paramètres de configuration ou des dépendances de services dans les paramètres du constructeur. Selon les paramètres et services spécifiés à la construction, iPOJO permet même de choisir quel est le constructeur le plus approprié pour l'initialisation du *POJO*. Le gestionnaire autonomique doit pouvoir être en mesure d'intercepter les appels aux constructeurs et de modifier les paramètres d'appels afin d'influencer sur le choix du constructeur effectivement appelé. Le conteneur de composant a pour rôle de sélectionner le constructeur le plus approprié aux paramètres d'initialisation modifiés par la chaîne d'interception. Toutefois, il peut arriver qu'il n'y ait aucun constructeur compatible, ou, à l'inverse, qu'il y ait ambigüité sur le constructeur à sélectionner. C'est au gestionnaire autonomique de faire en sorte d'éviter ces cas limites.

L'ajout du mécanisme de chaîne d'interception doit donc, afin de réduire les erreurs

et les corruptions de l'état des composants, vérifier que les intercepteurs se conforment bien aux différents schémas d'accès, en respectant les types des paramètres et des attributs des composants. Le *framework* doit forcer la conformité des interceptions, et toute violation doit être correctement signalée, car elle est le signe d'un dysfonctionnement grave dans le composant, dans le gestionnaire autonomique, voire les deux.

Un cas particulier d'interception d'accès est le cas où l'initiateur et la cible sont confondus. En effet, les accès du POJO par lui-même peuvent être interceptés ; ils sont même la fondation du mécanisme d'injection de propriétés et de dépendances d'iPOJO. Si ce mécanisme est potentiellement très puissant, puisqu'il permet de modifier le comportement interne de l'objet de façon transparente, il demande la plus grande précaution de la part des intercepteurs y participant. Il est en effet très facile d'introduire des comportements non-gérés par l'objet originel, voire des effets de bord pouvant corrompre l'état de l'objet de façon irrécupérable.

Handler hot-plug

Comme expliqué précédemment, les mécanismes permettant de gérer les aspects non-fonctionnels des composants iPOJO sont présentés sous la forme de *handlers*, composants particuliers qui étendent le conteneur du composant métier. Le gestionnaire autonomique, a un contrôle total sur le conteneur de chaque composant, et se charge donc de surveiller et d'agir sur chacun de ses constituants : *POJO*, *handlers* et chaînes d'interceptions.

Les *handlers* doivent afin de réaliser leurs fonctions, être en mesure d'intercepter les accès au code métier du composant. Les chaînes d'interception, présentées ci-dessus, permettent à la fois au gestionnaire autonomique et aux *handlers* d'intercepter ces accès, tout en donnant au gestionnaire autonomique la possibilité d'arbitrer ces accès. Ce mécanisme de chaîne d'interception permet l'ajout ou le retrait d'intercepteur à la volée, tout en protégeant cette même chaîne des incohérences résultant d'accès concurrents inévitables dans un environnement d'exécution dynamique. Par exemple, si un accès est en cours d'interception et qu'un intercepteur souhaite s'inclure dans la chaîne, il ne sera pas sollicité pour l'accès en cours, mais sera bien appelé pour tous les accès ultérieurs.

Auparavant, les *handlers* d'iPOJO ne pouvaient pas être ajoutés à la volée : la définition d'un type de composant dressait une liste exhaustive des *handlers* participants. En effet, certains de ces *handlers* modifient profondément la nature du type de composant, en y ajoutant des propriétés, des facettes, des requis et des aptitudes (telles les dépendances et fournitures de services). Ces *handlers* structurels participent à la définition du type de composant, et ne peuvent pas être ajoutés ou retirés dynamiquement sans avoir un impact conséquent sur toutes les instances de composant en cours d'exécution.

Il y a cependant certains *handlers*, facultatifs, qui ne nécessitent aucune modification du type de composant, et méritent donc de pouvoir être attachés ou détachés pendant l'exécution du composant. iPOJO fournit déjà un moyen d'attacher ce type de *handler* à toutes les instances de composant créées par la suite, sans toutefois fournir de mécanisme pour les détacher. Ces *handlers* peuvent donc être attachés avant la création de composants, et reste attachés durant tout le cycle de vie des composants créés ultérieurement.

L'informatique autonomique nécessite des mécanismes plus fins, afin de pouvoir

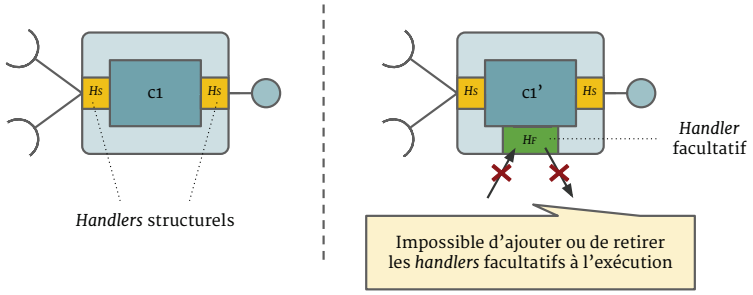

FIGURE 5.9 – Handlers structurels et handlers facultatifs

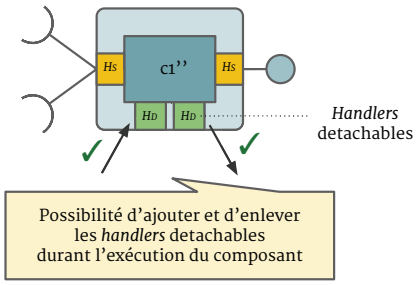

FIGURE 5.10 – Handlers détachables

adapter à la volée certains aspects des composants qui ne rentrent pas dans leur définition. La distribution, la gestion des droits d'accès ou encore la journalisation sont des illustrations de tels aspects, qui ne nécessitent pas de modifier la définition de type de composants. Le gestionnaire autonomique doit pouvoir être en mesure d'activer ou de désactiver ces aspects, afin d'optimiser l'effort d'adaptation nécessaire. Cette capacité d'optimisation est un des critères essentiels de l'évaluation d'une plateforme *Autonomic Ready*. Les *handlers* tels que proposés dans la version initiale d'iPOJO ne fournissent hélas pas la flexibilité nécessaire pour satisfaire ce critère exigeant.

Imaginons par exemple un *handler* réalisant des mesures de performance sur le composant auquel il est attaché. Ces mesures peuvent servir au gestionnaire autonomique à guider des optimisations de l'architecture afin, par exemple, de réduire les délais de latence. Dans la version de base d'iPOJO, une fois ce *handler* attaché à l'instance, il est impossible de l'en retirer. Il en résulte un gaspillage de performances une fois que le *handler* est devenu inutile, alors que son fonction est justement de mesurer les performances afin d'aider à les améliorer.

Dans la version étendue d'iPOJO, nous proposons d'introduire un nouveau type de *handler*, appelé ***handler* détachable**, qui peut être attaché et détaché à la volée, pendant l'exécution du composant. Ce type de handler ne peut pas participer à la description du type de composant, qui reste figée. Cependant, il peut ajouter des fonctionnalités ou gérer certains aspects non-fonctionnels du composant (distribution ou journalisation parmi d'autres).

La souplesse offerte par le mécanisme de chaîne d'interception, décrit précédemment, permet d'ajouter ou de retirer des intercepteurs pendant l'exécution du composant, tout en garantissant la cohérence de l'état du composant et celle des accès interceptés. Le branchement de *handlers* détachables au cours de l'exécution des composants s'en trouve donc grandement facilité.

En définitive, l'interception des accès à la logique métier des composants iPOJO permet à la fois aux *handlers* de gérer les aspects non-fonctionnels de ce composant, mais aussi au gestionnaire autonomique de surveiller et d'adapter son état. Le gestionnaire est en effet, par définition, omniscient : il connaît l'état du composant, de la plateforme, de l'environnement, le contexte d'utilisation, *etc.* Lui seul possède donc les connaissances nécessaires à l'adaptation du composant.

Les types d'adaptation proposés ci-dessus ne dépassent pas la portée du composant : les impacts sont locaux, et ne sont *a priori* pas perceptibles par les autres composants de la plateforme. Dans la section suivante, nous allons voir que l'interception des « connecteurs » des composants, ses dépendances et des fournitures de services, nous allons pouvoir changer les liaisons **entre** les composants, et adapter l'architecture de la plateforme.

Les gestionnaires de liaisons, pivots des adaptations architecturales, s'articulent autour de deux concepts clé d'iPOJO, et plus généralement des architectures orientées service : la gestion des dépendances de services, et la gestion des fournitures de services.

5.3.2 Interception au niveau des dépendances de service

Les dépendances de services permettent aux composants de déléguer certaines de leurs tâches à des services tiers disponibles sur la plateforme. Ces services peuvent être dynamiques, c'est-à-dire apparaître, changer et disparaître pendant l'exécution de la plateforme et des applications qui y reposent. iPOJO permet aux composants de décrire leur dépendances vers des services tiers, mais aussi de décrire leurs réactions face aux dynamisme de ces services [Escoffier 2013]. Ces caractéristiques permettent :

- la séparation des préoccupations, puisque les aspects techniques liés au dynamisme ne sont pas à la charge du développeur.
- le couplage faible entre composants, puisqu'ils ne dépendent pas directement les uns des autres : ils spécifient les interfaces des services dont ils ont besoin.
- la flexibilité de la plateforme et des applications, puisque l'architecture n'est plus figée, et peut être adaptée en fonction de la disponibilité et de la qualité des services.

Ce dernier point est particulièrement important dans le cadre de l'informatique autonomique, dont l'adaptabilité est une des expressions les plus apparentes. Cependant, comme nous allons le voir, bien qu'iPOJO permette cette flexibilité au niveau des dépendances de service, leur prise en charge par un gestionnaire autonomique de liaison est très complexe. Nous proposons donc une révision des mécanismes de gestion des dépendances des services : une partie générique étendue, qui permet la délégation à un gestionnaire autonomique, ainsi qu'une interface d'interception autorisant la mise en œuvre d'adaptations spécifiques.

Dans la suite de cette section, nous allons tout d'abord présenter les principales caractéristiques des dépendances de service, puis nous montrerons comment un gestionnaire autonomique de liaison doit pouvoir prendre en charge le mécanisme de résolution de services et de liaison, tout en conservant ces caractéristiques essentielles au code métier.

Caractéristiques d'une dépendance de service

Les caractéristiques générales d'une dépendance de service ont déjà été exposées dans la section 2.7.3. La combinaison de ces caractéristiques permet à la logique

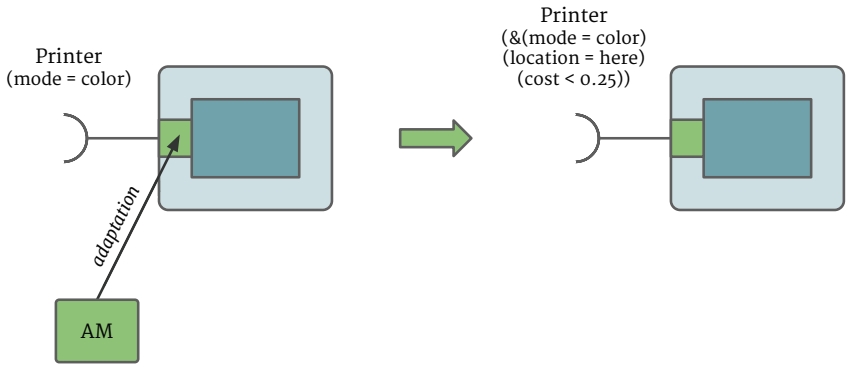

FIGURE 5.11 – Entrelacement des politiques de filtrage d'une dépendance de service

métier d'un composant de répondre à un très grand nombre de situations. L'adaptation de la dépendance par un gestionnaire de liaison est par contre très limitée, car le gestionnaire doit jongler entre les aspects non-fonctionnels du composant (comme la qualité des services par exemple) et les aspects métier. À l'exécution, les deux aspects se retrouvent complètement mélangés, et il est difficile de distinguer quelles sont les caractéristiques inhérentes au composant de celles redéfinies par le gestionnaire de liaison.

L'exemple le plus élémentaire est la caractéristique de filtrage de la sélection de service. Un composant peut dépendre d'un service de type imprimante, avec la restriction que celle-ci supporte l'impression en couleur. Le gestionnaire de liaison peut, quant à lui, n'autoriser l'utilisation que des imprimantes présentes dans un périmètre restreint, grâce à sa connaissance du contexte d'utilisation. La figure 5.11 montre comment ces deux contraintes peuvent se retrouver fusionnées dans un filtre de sélection. La manipulation du filtre de sélection par le gestionnaire de liaison est donc lourde et inadaptée.

Certaines opérations, telles que le masquage de certains services, l'ajout de propriétés de services non-fonctionnelles ou dépendantes du contexte sont très difficiles, voire impossibles à réaliser en utilisant les cinq caractéristiques (type, cardinalité, obligatoire/optionnelle, filtre de sélection, comparateur et politique de liaison), car elles sont très orientées vers leurs utilisations par le code métier. Ces opérations sont pourtant nécessaires pour permettre aux gestionnaires de liaisons de réaliser les buts de haut niveau dictés par les gestionnaires de niveau supérieur.

Le comparateur de service permet, en théorie, au code métier du composant de sélectionner le meilleur service possible. Toutefois, les paramètres permettant de juger et de classer un service sont nombreux, et bien souvent sans aucun rapport avec la fonctionnalité première du composant : temps de réponse, sécurité, disponibilité, *etc.* Cette comparaison est donc une tâche qui incombe au gestionnaire de liaison, qui, conscient du contexte, peut décider quels sont les meilleurs services présents dans la plateforme.

Nous proposons donc un nouveau modèle de dépendance, compatible avec le modèle standard, qui permet une gestion véritable des dépendances de service par un gestionnaire. Cette gestion se fait de manière transparente pour le code métier du composant. Le gestionnaire doit être en mesure d'influencer les mécanismes de recherche et de sélection des services, mais aussi l'ordonnancement des services selon des critères dépendants du contexte. Ce nouveau modèle s'articule donc autour de

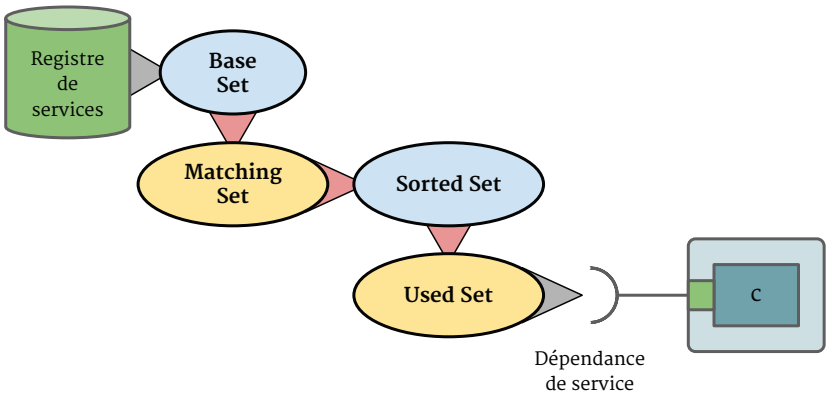

FIGURE 5.12 – Ensembles de services dans une dépendance

deux concepts clé : l'interception du registre de service, et le tri de l'ensemble des services utilisables.

Dans ce nouveau modèle, l'ensemble des services vus par une dépendance sont décomposés en différents sous-ensembles, et chacun des mécanismes d'adaptation des gestionnaires autonomiques va pouvoir modifier un de ses sous-ensembles, et répercuter ses changements sur les sous-ensembles suivants. La figure 5.12 montre la succession des ensembles de services considérés par une dépendance de service, tel que le propose notre nouveau modèle de dépendance. Les ensembles déjà existants dans iPOJO sont colorés en bleu, les ensembles ajoutés par le modèle proposé sont colorés en orange :

- Le registre de service contient tous les services fournis sur la plateforme d'exécution : c'est l'ensemble de départ.
- Le ***Base Set*** est l'ensemble des services du registre qui satisfont le filtre métier de la dépendance de service.
- Le ***Matching Set*** est un nouvel ensemble, manipulable par un gestionnaire autonomique de liaison, qui permet d'ajouter, de retirer ou de modifier certains services. Cet ensemble permet donc de modifier la perception par un composant des services disponibles.
- Le ***Sorted Set*** est l'ensemble des services du *Matching Set*. Comme son nom l'indique, cet ensemble est trié, ce qui permet de favoriser l'utilisation de certains services par rapport à d'autres. L'ordre des services est déterminé par le comparateur métier du composant (s'il est présent), mais peut être modifié par un gestionnaire de liaison.
- Le ***Used Set*** contient les objets qui sont utilisés par le composant. Chaque objet correspond à un service du *Sorted Set*. Un gestionnaire de liaison peut modifier ces objets afin de changer le comportement du service utilisé par le composant.

Dans la figure précédente, les nouvelles étapes permettent de déterminer les services d'un ensemble à partir de l'ensemble précédent ont été colorés en rouge. Le modèle existant d'iPOJO permet de déterminer le *Base Set* à l'aide d'un filtre métier, et le *Sorted Set* avec un comparateur de services métiers. Notre modèle ajoute donc deux ensembles intermédiaires de services, et ajoute trois étapes qui permettent à un gestionnaire de liaison de modifier ces ensembles.

La suite de cette section va détailler les différentes étapes permettant de modifier ces ensembles, et donc d'agir sur une dépendance de service. La première de ces étapes

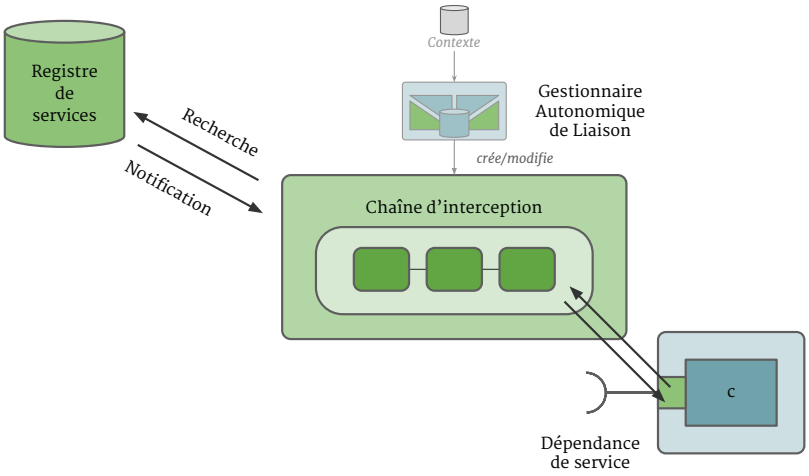

FIGURE 5.13 – Interception du registre de service

est l'**interception du registre de services**, qui permet de déterminer le *Matching Set*. La deuxième étape est l'**ordonnancement des services**, qui est un nouveau moyen de déterminer le *Sorted Set*. Enfin, l'**interception des liaisons aux services** permet de modifier le *Used Set* : l'ensemble des objets réellement utilisés par le composant.

Interception du registre de services

Dans une architecture orientée service, le registre de services est l'élément central, car il détient la liste de tous les services présents sur la plateforme, leurs propriétés, et le moyen de les utiliser. C'est donc naturellement cette entité qu'utilisent les dépendances de service d'iPOJO (sous la forme de *handlers* de dépendance) afin de permettre l'injection de dépendance à l'intérieur des objets métier. Cependant, comme nous l'avons vu, les *handlers* et le code métier ne disposent pas des informations ni des mécanismes nécessaires afin d'adapter les liaisons vers ces services en toute connaissance du contexte.

Un gestionnaire autonomique, plus particulièrement un gestionnaire autonomique de liaison, doit pouvoir gérer cette dépendance, de manière totalement transparente et invisible pour le code métier du composant. Afin de permettre ces opérations de gestion externe des dépendances de service, nous proposons de modifier la façon avec laquelle le *handler* de dépendances cherche, filtre et sélectionne les services au travers du registre de services. Le *handler* de dépendance se voit muni d'un mécanisme de chaîne d'interception, qui permet sa perception du registre de service. Ce mécanisme donne toute la latitude nécessaire au gestionnaire de liaison pour effectuer les opérations de gestion mentionnées ci-dessus.

La figure 5.13 montre la mise en place d'une interception entre le registre de services et la dépendance du composant. Cette interception agit en écoutant et modifiant les accès de la dépendance vers le registre de service, mais aussi en modifiant les évènements émanant du registre à l'attention de la dépendance, qui scrute ces notifications.

La structure en chaîne, décrite précédemment (section 5.3.1), permet au gestionnaire de liaison de décomposer la prise en charge de la liaison en plusieurs blocs distincts et indépendants, chacun pouvant se concentrer sur un aspect non-fonctionnel

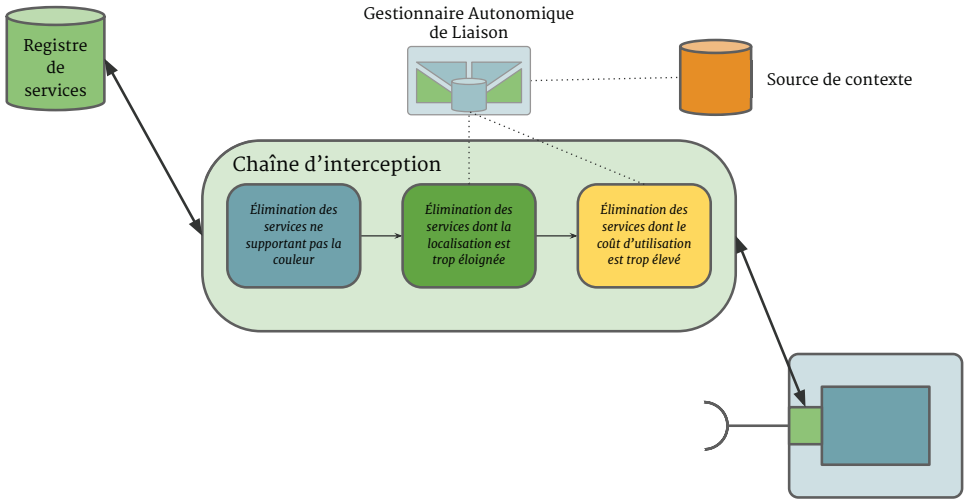

FIGURE 5.14 – Exemple de chaîne d'interception de dépendance de service

particulier. Les intercepteurs situés en première ligne sont généralement chargés d'éliminer le plus de services candidats possible, afin d'alléger la charge des intercepteurs suivants. Ces derniers peuvent quant à eux se concentrer sur la modification des services, tout du moins la perception que le composant va en avoir, selon le contexte d'exécution et d'utilisation de la dépendance.

En reprenant l'exemple précédent, celui du composant dépendant d'un service d'impression, le gestionnaire de liaison pourrait par exemple être décomposé en trois intercepteurs distincts, tels qu'illustrés dans la figure 5.14 :

- un intercepteur défini par un filtre métier, éliminant les services d'impression ne supportant pas la couleur,
- un deuxième recherchant la localisation des imprimantes correspondantes, et excluant celles qui ne se situe pas à proximité de l'utilisateur du service.
- un troisième intercepteur qui détermine le coût par page d'une impression et élimine les services jugés trop coûteux. Cet intercepteur pourrait noter les services en fonction de leur coût d'utilisation, mais cette tâche peut être traitée de façon beaucoup plus aisée et intuitive avec les ordonnanceurs de services, décrits dans la section suivante.

Seuls les services ayant traversé les trois intercepteurs de la chaîne seront présentés au code métier du composant. Les autres ne seront même pas visibles. Les intercepteurs doivent aussi transmettre au composant les évènements provenant du registre de service, et doivent dans certains cas les modifier :

- Les arrivées de nouveaux services, potentiellement intéressants doivent traverser la chaîne d'interception. Ces notifications ne seront transmises au composant que si le nouveau service est accepté par tous les intercepteurs formant la chaîne.
- Si un service sélectionné est modifié, il faut réévaluer s'il satisfait toujours les critères de sélection (c'est-à-dire s'il traverse toujours la chaîne). Si le service modifié traverse la chaîne, la notification provenant du registre est directement transmise au composant. Sinon, un évènement de départ de service est transmis à la place. Le service n'est pas vraiment parti, mais dorénavant caché au composant par la chaîne d'interception.
- Si un service non-sélectionné est modifié, cette modification peut faire en sorte

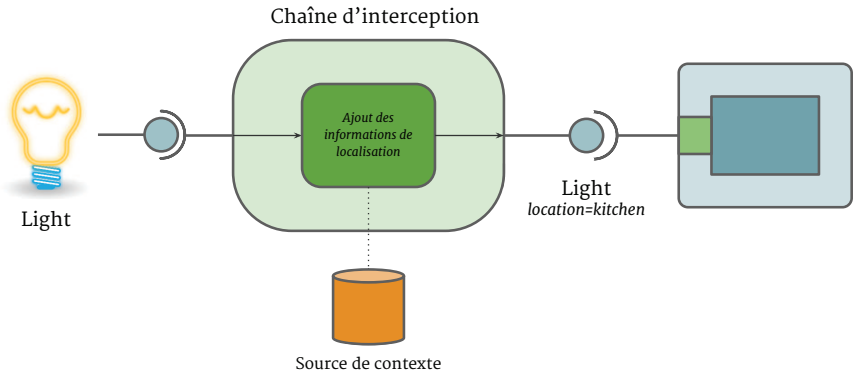

FIGURE 5.15 – Injection de propriétés de contexte sur un service

que le service soit maintenant sélectionnable. Le service traverse les différents intercepteurs, et s'il atteint la fin de la chaîne, une notification d'arrivée de nouveau service est transmise au composant. Le service ne vient pas d'arriver, mais était auparavant masqué par le gestionnaire.

– Les notifications de départ des services sélectionnés sont toujours transmises directement au composant, afin qu'il prenne les mesures nécessaires pour ne plus l'utiliser.

Il faut noter que les deux derniers intercepteurs dépendent du contexte des services de la plateforme, en l'occurrence leur localisation et leur coût. Ces données peuvent changer à tout moment. Il est donc nécessaire que les intercepteurs de la chaîne puissent, après avoir détecté de tels changements, invalider une partie ou l'ensemble de services sélectionnés et recalculer un ensemble correspondant au contexte mis à jour. Ces intercepteurs sont donc **actifs**, car ils peuvent initier des actions sans qu'elles ne soient nécessairement provoquées par des notifications du registre de service, ou par des requêtes de la dépendance de service.

Le filtrage de services, en fonction de leurs propriétés non-fonctionnelles ou dépendantes du contexte, est l'une des possibilités offertes par les chaînes d'interception placées sur les dépendances de service. Il y a cependant des cas plus complexes, ou le simple filtrage ne suffit pas à gérer de manière efficace la dépendance. Il est parfois nécessaire de modifier les services, ou plus exactement de **modifier** la perception du composant.

Si nous reprenons l'exemple de l'application *LightFollowMe*, le composant réalisant l'allumage et l'extinction des lumières doit pouvoir connaître précisément les dispositions de celles-ci, ainsi que la localisation des capteurs de présence, et la disposition des pièces de l'appartement. Ces informations sont, par essence, des données du contexte de l'environnement physique. Le gestionnaire autonomique maintient une connaissance de ces paramètres. Ce n'est par contre pas au code métier du composant de récupérer ces informations, quels que soient les moyens nécessaires pour les récupérer.

Le gestionnaire autonomique doit donc être en mesure d'insuffler sa connaissance du contexte au code métier du composant. Dans ce cas précis, la connaissance portant sur des services, c'est donc au gestionnaire de liaison de communiquer ces informations, concernant les services sélectionnés, au code métier du composant. En pratique,

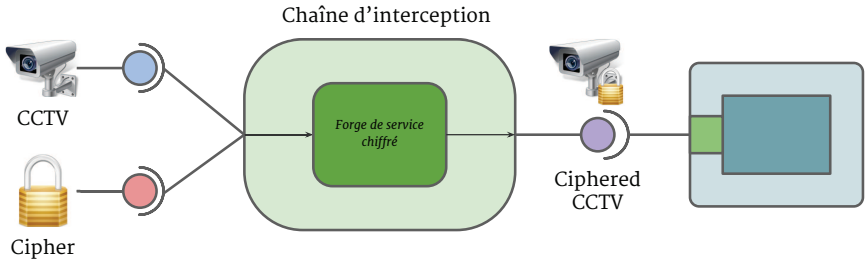

FIGURE 5.16 – Injection de service forgé par le gestionnaire de liaison

il va annoter les services en ajoutant des propriétés de localisation sur chacun d'eux. Ces propriétés ne sont pas ajoutées aux services réels, mais le gestionnaire présente au composant un pseudo-service modifié, reflétant l'état du vrai, auquel s'ajoutent les propriétés de localisation. La figure 5.15 illustre par exemple le fait que le code métier dispose des informations de localisation sur un service utilisé, alors que ces informations ne figurent pas sur le service original : elles sont récupérées, par le gestionnaire, depuis une source de contexte.

Enfin, il peut arriver que gestionnaire ait besoin de « forger » des services spécifiquement pour un composant en particulier. Il s'appuie le plus souvent sur des services disponibles sur la plateforme, mais en modifie la nature, ou ajoute du code de gestion d'un aspect non fonctionnel. Le service original, s'il existe, est totalement masqué au code métier qui perçoit seulement le service forgé par le gestionnaire. Dans ce cas-là, le composant utilise un service qui n'existe pas sur la plateforme d'exécution, puisqu'il est généré par le gestionnaire. Ce dernier est alors en charge de faire correspondre ce service virtuel avec le ou les services réels associés. Il peut intercepter les appels à ce service et effectuer, selon le contexte, les opérations nécessaires. Tout ceci est totalement transparent du point du vue du code métier qui, comme dans le cas des « services augmentés », pense utiliser un service provenant directement du registre.

La figure 5.16 montre un exemple de gestionnaire de liaison qui compose deux services, un service de vidéosurveillance et un service de chiffrement, afin de fournir au code métier du composant un service de vidéosurveillance sécurisé qu'il attend. Ce service virtuel est créé spécifiquement pour cette instance du composant, et peut par exemple contenir des données spécifiques à cette utilisation (clé de chiffrement de session par exemple, mémoire tampon des images enregistrées, *etc.*).

L'interception des accès au registre de service permet de supporter plusieurs scénarii d'adaptation par le gestionnaire autonomique, comme le filtrage de service, l'ajout de propriétés contextuelles ou encore l'injection de services forgés sur mesure pour le composant. Ce mécanisme n'offre cependant pas toute la souplesse nécessaire pour comparer les services les uns aux autres, et permettre de sélectionner le ou les meilleurs candidats en fonction des besoins du composant et du contexte de la plateforme d'exécution.

Ordonnancement des services par le gestionnaire de liaison

Une des autres techniques pour la mise en œuvre de l'adaptation architecturale par le gestionnaire autonomique, par le biais des gestionnaires autonomiques de liaison est l'ordonnancement des services. Cette technique permet de modifier l'ordre apparent des services tels qu'ils sont vus par le code métier du composant. Par défaut, la

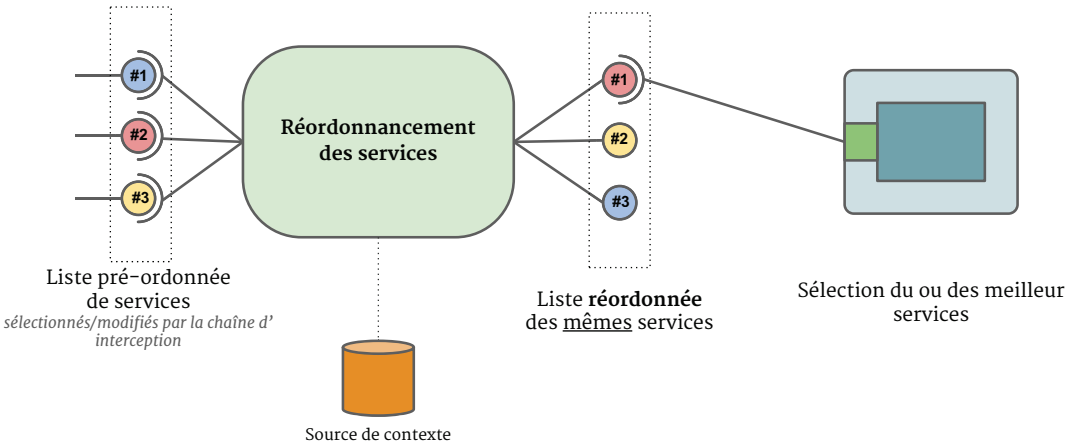

FIGURE 5.17 – Ré-ordonnancement des services

plateforme *OSGi*, sur laquelle repose iPOJO, ordonne implicitement les services selon leur date d'enregistrement. Les services disponibles depuis la plus longue période sont donc favorisés par rapport aux services plus récents. Comme nous l'avons vu, iPOJO permet au code métier de spécifier un comparateur afin de modifier cet ordre tacite. Nous avons également vu que les données de contexte accessibles au code métier sont très limitées, et qu'il est donc extrêmement difficile d'écrire un comparateur conscient du contexte tout en conservant la séparation des préoccupations au sein du code métier.

La modification de l'ordre des services permet de changer le ou les services choisis par le composant lorsque plusieurs candidats sont disponibles. Une des motivations les plus fortes qui nous poussent à fournir ce mécanisme d'ordonnancement externe est l'optimisation de la plateforme. Le gestionnaire doit en effet être en mesure de proposer le ou les meilleurs services au composant, selon la situation actuelle de la plateforme (charge, capacités de calcul, divers critère de qualité) mais aussi selon le contexte d'utilisation (préférences, compatibilité, langue maternelle de l'utilisateur, *etc.*).

Le comparateur de service se situe après la chaîne d'interception mentionnée dans la section précédente, c'est-à-dire qu'il va agir sur l'ensemble de services préalablement filtrés, augmentés et/ou forgés par la chaîne d'interception. Ceci permet de trier l'ensemble des services en prenant en compte les informations de contexte ajoutées par la chaîne d'interception.

Le comparateur doit se tenir informé des évènements liés aux services sélectionnés : les notifications qui lui parviennent peuvent déclencher une réévaluation complète de l'ordonnancement des services. D'autres évènements peuvent aussi déclencher cette évaluation, telle un changement dans une source de contexte extérieure. Le comparateur a donc aussi un rôle actif, puisqu'il peut décider, à tout moment, d'invalider l'ensemble trié courant des services afin d'en proposer une nouvelle version, mise à jour, au code métier du composant.

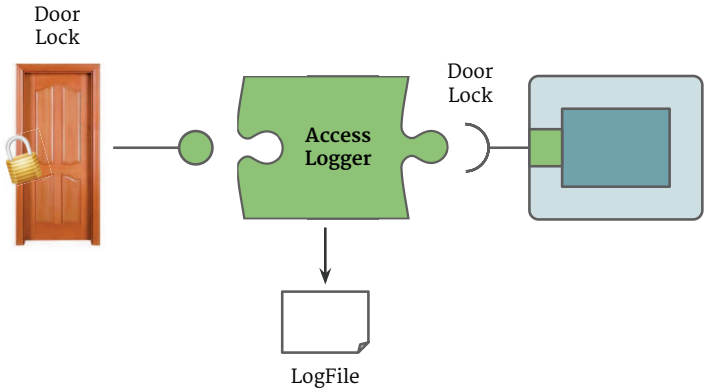

FIGURE 5.18 – Exemple d'interception de la liaison à un service

5.3.3 Interception des liaisons de service

L'interception du registre de service permet de proposer au composant des services modifiés, voire virtuels, qui embarquent des informations de contexte et/ou gère des aspects non-fonctionnels. Cette approche est adaptée dans le cas de fusion de services, où le service proposé est de nature différente des services fusionnés. Cependant, dans la plupart des cas, le service à injecter est un mandataire [5] d'un service existant. Un objet enrobant le service réel est alors proposé au composant, qui l'utilise comme s'il s'agissant du service réel. Cet objet transmet les accès au service réel tout en effectuant des actions spécifiques avant et après ces accès. Il peut dans certains cas interdire ces accès (gestion des droits d'accès par exemple), ou les court-circuiter (mémoire tampon enregistrant les résultats des appels précédents).

Ce type d'interception n'intervient généralement pas au niveau de la sélection et de l'ordonnancement des services, et il n'est donc pas adapté d'utiliser les mécanismes décrits ci-dessus afin de le mettre en œuvre. En fait, ce type d'interception se fait au moment de la **liaison** du composant au service : au moment où il commence à l'utiliser. Le but est ici de fournir au composant un objet mandataire au lieu du service auquel il veut se lier. Cette opération est totalement transparente pour le composant, car le type de l'objet mandataire doit être compatible avec le type de service demandé.

Ce type d'interception n'intervient pas au niveau de l'architecture de l'application : les liens entre composants et services restent identiques. Ce mécanisme permet pourtant d'intégrer la gestion d'aspects non-fonctionnels directement dans le service proposé au composant. Ces aspects peuvent être traités séparément par les différents intercepteurs de la chaîne, comme le montre la figure 5.18.

5.3.4 Bilan des intercepteurs de dépendances de services

Les différentes approches énoncées ci-dessus permettent au gestionnaire autonomique de liaison de modifier la résolution des dépendances, d'affecter la visibilité de certains services, d'ajouter des propriétés contextuelles, voire même d'injecter des pseudo-services dans le composant. Ces techniques peuvent être composées afin d'apporter au code métier les services les plus pertinents en fonction du contexte. La figure 5.19 détaille l'ensemble d'une dépendance de service prise en charge par un

5. Ou *proxy* en anglais

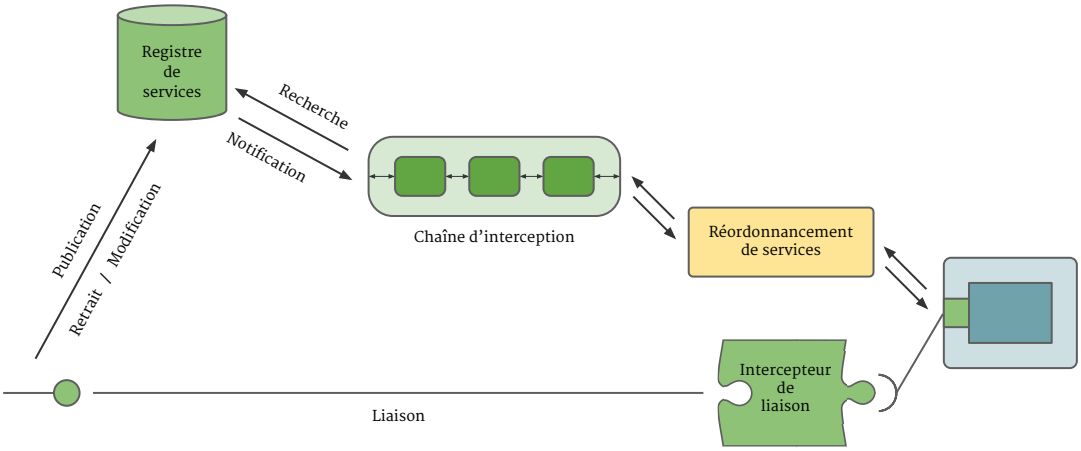

FIGURE 5.19 – Dépendance de service *Autonomic Ready*

gestionnaire autonomique de liaison.

L'interception des dépendances de service permet au composant de percevoir un environnement d'exécution augmenté par les informations de contexte qui sont insufflées de manière transparente par son gestionnaire autonomique de liaison. Cependant, ces données contextuelles peuvent parfois être directement fournies par le service lui-même. C'est justement sur ce point que se penche l'interception des fournisseurs de service.

5.3.5 Interception au niveau des fournitures de service

Un composant fournissant un service expose généralement aux autres composants les propriétés fonctionnelles relatives à ce service, et seulement celles-ci. Or nous avons vu que le choix d'utiliser ou non un service est aussi largement déterminé par les informations de contexte. Un composant dépendant d'un ou plusieurs services peut, grâce à l'interception des dépendances, choisir et utiliser ces services avec les informations de contexte communiquées par le gestionnaire de liaison. Ce mécanisme est particulièrement bien adapté lorsque la sélection et l'utilisation des services doivent être « personnalisées » pour un composant consommateur particulier.

Certains informations de contexte sont universelles, et doivent pouvoir être consultables par n'importe quel composant de la plateforme. Ces informations pourraient être ajoutées par les chaînes d'interception de chaque dépendance de service. Cette méthode est cependant relativement lourde et redondante. Il est en effet plus simple d'ajouter ces informations à la source : directement sur le service fourni.

De la même manière que l'interception des dépendances de service change la façon de percevoir des services de la plateforme, l'interception des fournitures de service modifie la manière d'exposer un service aux autres composants. Il va ainsi être possible de changer le service fourni en fonction du contexte sans toucher au code fonctionnel.

Comme nous l'avons vu précédemment (section 2.7.3), la fourniture d'un service est caractérisée par le type de service fourni, les propriétés du service fourni, et la politique de fourniture du service. La politique de fourniture détermine la relation entre le ou les objets métiers du composant et les utilisateurs du service. Bien souvent, le même objet sert tous les clients, mais il peut arriver que le code métier doive fournir un objet spécifique pour chaque client, afin par exemple de maintenir un état dédié

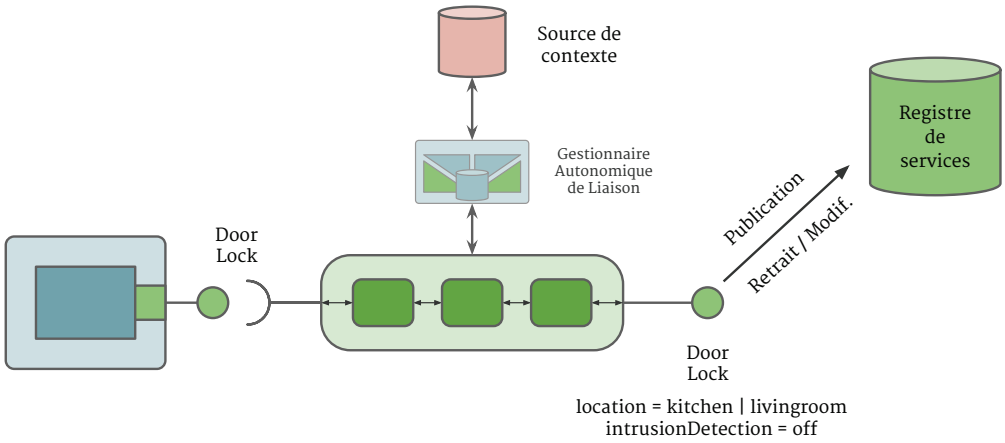

FIGURE 5.20 – Affinage des services fournis

pour ce client (session d'utilisateur par exemple). Cette politique est très liée à la manière dont le code métier est conçu, et n'est donc pas facilement modifiable par un gestionnaire autonomique. Le type de service fourni par le composant n'est pas non plus quelque chose que le gestionnaire peut modifier, car il fait partie intégrante de la spécification du composant.

Le gestionnaire autonomique de liaison va, par contre, pouvoir affiner le service fourni en y ajoutant des propriétés contextuelles avant sa publication dans le registre. L'interception de la publication de service va permettre à chacun des intercepteurs de la chaîne de rajouter et/ou modifier ces propriétés, tel qu'illustré dans la figure 5.20. Les notifications concernant le service exposé sont aussi interceptées, afin de pouvoir réagir de façon appropriée. Les intercepteurs ont aussi un rôle actif, puisqu'ils peuvent déclencher une modification des propriétés du service publié lors d'un changement de contexte. La chaîne d'interception permet aussi l'ajout et le retrait d'intercepteurs pendant l'exécution du composant, ce qui permet d'effectuer un raffinement à la demande du service.

L'autre fonctionnalité essentielle des intercepteurs de fourniture de service est l'enrobage du service réel dans un objet mandataire. Ce mécanisme est équivalent à l'interception des liaisons de services mentionnées dans la section précédente, sauf que le service est enrobé à la source, et non plus à l'utilisation. Ceci va permettre de gérer un ou plusieurs aspects non fonctionnels directement dans l'objet mandataire. Cet objet étant publié en tant que service dans le registre, tous les composants qui l'utilisent supporteront ces aspects gérés de manière transparente. C'est en quelque sorte une généralisation de l'interception de liaisons à ce service, pour tous les consommateurs.

La figure 5.21 montre un exemple d'utilisation de l'enrobage de service fourni : un intercepteur va ajouter au service d'impression la gestion de la facturation. Cet aspect est transparent pour le composant fournissant le service, qui n'a pas besoin d'être modifié. L'intercepteur ajoute aussi une propriété de service pour indiquer le prix d'utilisation du service.

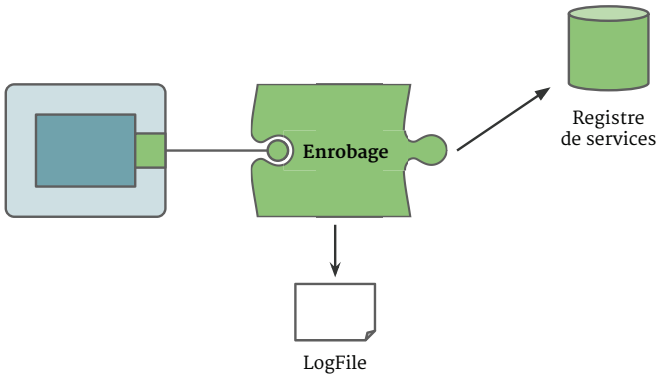

FIGURE 5.21 – Exemple d'enrobage de service fourni

5.3.6 Impacts sur l'architecture

L'interception des dépendances et des fournitures de service permettent d'adapter les composants, le comportement des services qu'ils fournissent et utilisent ainsi que les liaisons entre composants. En modifiant d'un côté ou d'un autre les caractéristiques des liaisons entre composants, le gestionnaire autonomique va pouvoir influencer l'architecture des applications.

Il est cependant intéressant de s'attarder sur la manière avec laquelle sont effectuées ces adaptations. Dans la vaste majorité des modèles à composants existants, des adaptations architecturales sont réalisées de façon explicite, en **ordonnant** par exemple au composant A de se lier au composant B. L'approche retenue dans cette contribution est toute autre : le gestionnaire, en manipulant les connecteurs des composants (dépendances et fournitures de services) va exprimer des **intentions** qui vont affecter la manière de fournir, de rechercher et de sélectionner un service.

La résolution de ces intentions garde donc un caractère flexible et adaptable, puisqu'un simple changement dans le contexte peut occasionner un changement de l'architecture, conforme aux intentions, sans que les gestionnaires autonomiques n'aient à réexprimer cette adaptation. Les intentions sont ici l'expression des buts de haut niveau du gestionnaire autonomique global, ainsi que la décomposition en sous-buts de (moins) haut niveau pilotant les gestionnaires autonomiques de niveau inférieur (de composant et de liaison).

La figure 5.22 montre comment le changement d'une propriété de contexte peut avoir un impact sur l'architecture de l'application. Le gestionnaire autonomique de liaison ne réalise aucune adaptation en réaction à ce changement de contexte, il ne fait que propager l'évènement vers la dépendance de service qui se charge de l'interpréter en réévaluant le service qu'elle peut utiliser.

La gestion des dépendances et des fournitures de services par des gestionnaires de liaisons permet de remonter ces informations aux gestionnaires autonomiques de niveau supérieur. Le recoupement de ces informations donne un aperçu de l'architecture globale courante des composants, services et applications qui s'exécutent sur la plateforme, comme illustré par la figure 5.23.

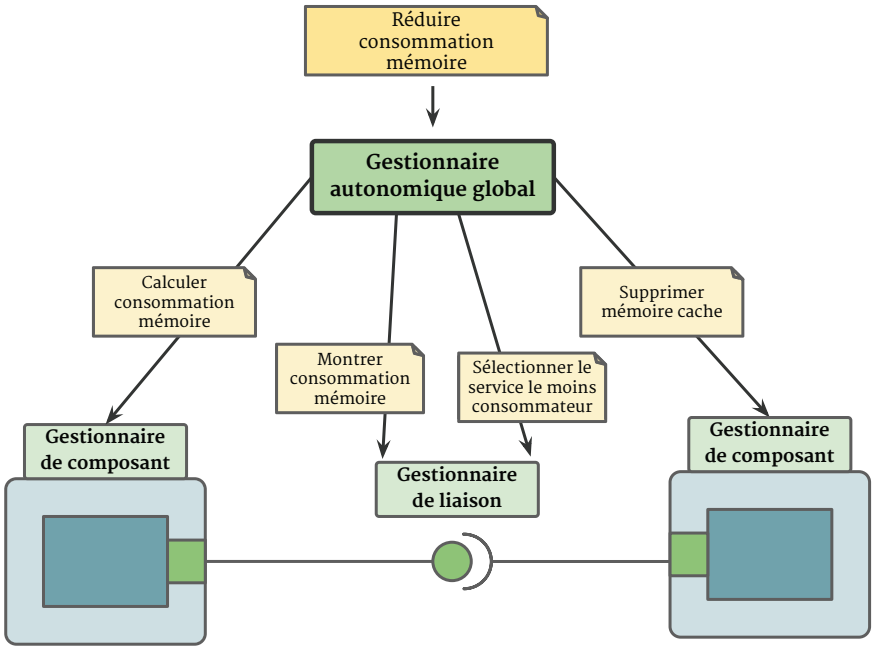

FIGURE 5.22 – Adaptation autonomique de l'architecture conformément aux intentions

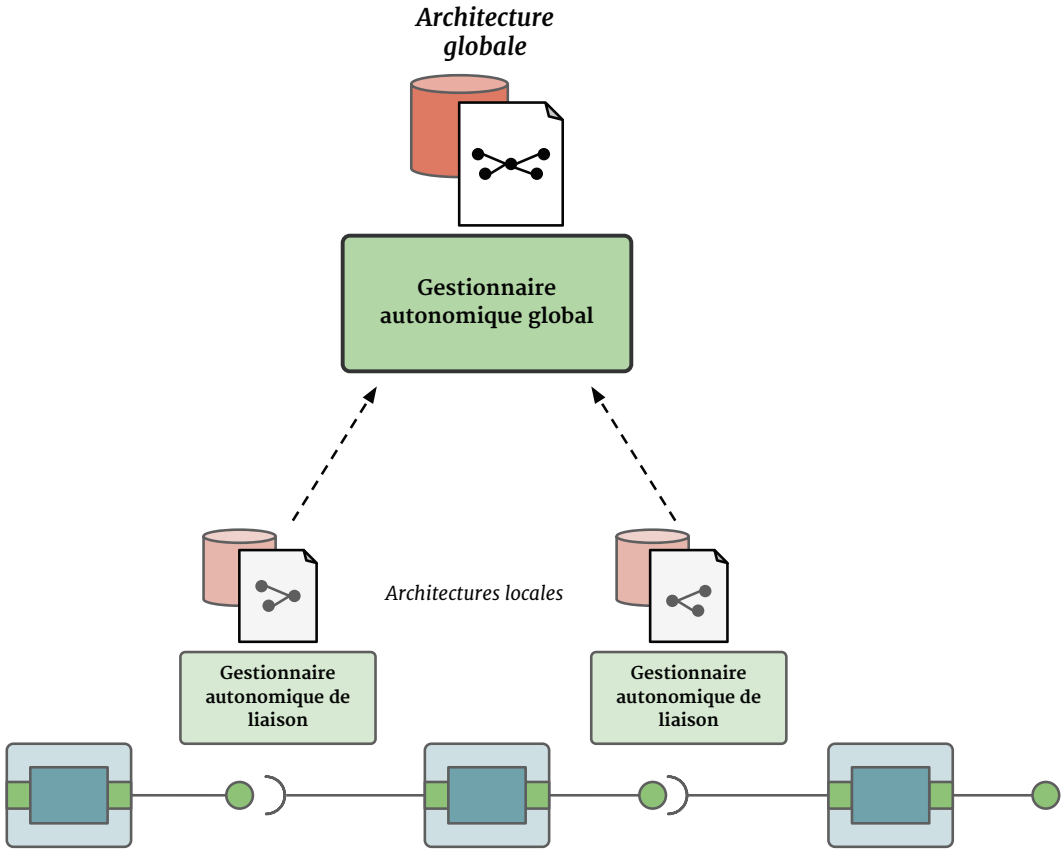

FIGURE 5.23 – Remontée de l'architecture globale

5.4 Représentation de la connaissance

Dans tous les mécanismes d'adaptation par les gestionnaires autonomiques que nous avons vus précédemment, le contexte tient une place très importante. Il est en effet la base de connaissance qui guide les gestionnaires pour chaque adaptation, des composants ou des liaisons entre composant. Cette thèse a présenté les similitudes entre les notions de contexte dans le domaine de l'informatique pervasive et de connaissance en informatique autonomique. Ces deux notions peuvent même, dans le cas d'applications pervasives gérées par un gestionnaire autonomique être confondues.

L'architecture à composants orientés service est un socle logiciel permettant de faciliter les adaptations à la volée des applications et de la plateforme d'exécution par un gestionnaire autonomique. Cependant, les composants et services présents sur la plateforme peuvent être très hétérogènes, chacun étant développé de façon isolée, et ayant une vision très subjective de ce qu'est le contexte. Les différents gestionnaires autonomiques doivent donc collecter des informations de contexte provenant d'une grande variété de sources différentes.

La mise en œuvre de cette tâche est hélas longue et délicate : la prise en compte de l'hétérogénéité des sources de contexte et la gestion du dynamisme de ces informations peuvent rapidement rendre la conception de gestionnaire autonomique difficile. Afin de simplifier cette conception, nous proposons un modèle pivot de représentation qui prend en charge ces aspects non-fonctionnels et offre une représentation unifiée de l'application et de son environnement.

Ce modèle, s'il masque le caractère hétéroclite des différentes entités composant le contexte, ne doit par contre pas altérer la nature profondément dynamique et adaptable de la plateforme d'exécution et des composants et services. Il doit permettre à la fois de récolter de façon simple et uniforme les informations provenant de diverses sources de contexte, autoriser le stockage d'informations déduites par le gestionnaire autonomique et fournir au gestionnaire le moyen d'adapter les diverses entités de la plateforme.

Ce modèle représentationnel forme donc une couche intermédiaire d'interaction entre les gestionnaires autonomiques et les entités qui constituent la plateforme à l'exécution. Ce concept se rapproche de la notion de *meta-object protocol*[6] [Kiczales 1991] : le niveau de base correspondant aux composants, services et autres entités de la plateforme, et le niveau *méta* correspondant au modèle abstrait présenté au gestionnaire autonomique. La synchronisation entre le niveau de base et le niveau *méta* est assurée par :

- La machine d'exécution, plus particulièrement les conteneurs de composants, pour la partie générique.
- Les différents gestionnaires autonomiques pour les parties spécifiques.

La synchronisation entre les deux niveaux du *meta-object protocol* constitue la partie la plus délicate dans sa mise en œuvre. Comme nous l'avons mentionné dans le chapitre précédent, fournir un modèle causal est très complexe, voire impossible, en fonction des capacités de la machine d'exécution sous-jacente. C'est pourquoi le niveau *méta* que nous proposons n'est pas une représentation directement causale du niveau de base, mais plutôt un instantané[7]. Il est néanmoins possible de scruter les changements intervenant sur ce modèle, afin d'en saisir tout le dynamisme.

Selon le niveau de gestionnaire autonomique pris en considération, (global, de

6. Protocole à méta-objets, souvent abrégé en *MOP*
7. En anglais : *snapshot*

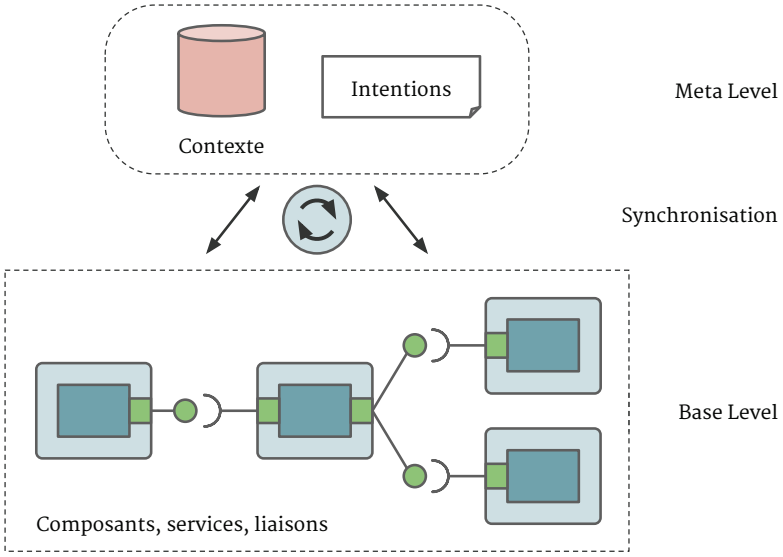

FIGURE 5.24 – *Meta-object protocol* de la plate-forme d'exécution

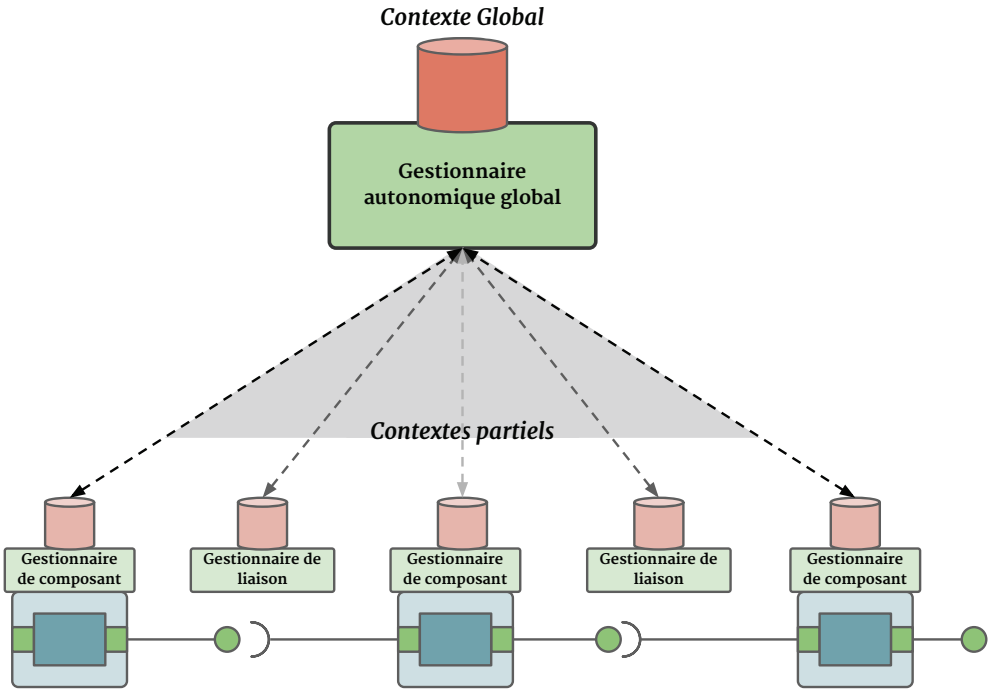

FIGURE 5.25 – Composition des différentes visions du contexte

l'application, du composant, de liaison), les contextes considérés diffèrent. En effet, si le gestionnaire autonomique central doit avoir une vue d'ensemble, les gestionnaires locaux ne considèrent que les informations de contexte concernant les domaines qu'ils gèrent (composant, service, *etc.*). Le contexte global est donc composé des informations remontées par les gestionnaires locaux depuis les contextes locaux. Les différents gestionnaires autonomiques doivent donc synchroniser les contextes entre eux, comme le montre la figure 5.25.

Pour la représentation abstraite du contexte, nous nous appuyons sur la notion de **ressource** telle que définie dans le style d'architecture *REST*[8] [Fielding 2000]. Après avoir brièvement présenté les grandes lignes de ce style d'architecture, nous proposerons notre modèle de ressource, inspiré de *REST*, qui servira de base uniforme à la représentation du contexte.

5.4.1 Principes du style d'architecture *REST*

Le style d'architecture *REST* a été proposé par Roy Fielding, afin de répondre aux enjeux de la distribution des applications sur l'Internet [Fielding 2000]. Ce style d'architecture client-serveur repose sur la notion universelle de **ressource**, dont l'état, maintenu par le serveur, est transmis au client et/ou modifié par celui-ci. Les interactions entre clients et serveurs sont sans état[9], c'est-à-dire qu'il n'y a aucune information de session ou de contexte dans la connexion entre clients et serveurs. Ceci permet entre autre la mise en mémoire cache des états des ressources par des clients ou serveurs mandataires. Les ressources sont accessibles par le biais d'une interface uniforme qui permet :

- L'**identification** : chaque ressource est munie d'un identifiant unique qui permet de la retrouver, et de la distinguer des autres ressources.
- La **manipulation** : les ressources sont des représentations manipulables d'entités variées. La façon d'obtenir et de changer l'état de ces ressources est uniforme.
- L'**auto-description** : les ressources doivent contenir les informations permettant leur interprétation par les clients. Ces informations doivent à la fois être interprétables par un programme et lisibles par un humain.
- **HATEOAS**[10] : les ressources sont liées les unes aux autres par des liens dits hypermédia. Ces liaisons sont elles aussi auto-descriptives, et permettent la navigation entre ressources.

Le style d'architecture *REST* expose des qualités qui permettent des interactions simples et unifiées avec les ressources constituant une application. Bien qu'il ait été conçu spécifiquement pour les applications distribuées, en particulier les applications Web, ses caractéristiques d'accès uniforme permettent de l'utiliser pour représenter et de manipuler une grande variété d'entités logicielles et physiques. Dans la suite de cette section, nous proposons un modèle de représentation du contexte, dans le sens ubiquitaire et autonomique du terme, basé sur l'interface d'accès uniforme de *REST*.

5.4.2 Représentation uniforme du contexte proposée

Une représentation uniforme du contexte est, comme nous l'avons vu, un prérequis pour la simplification du développement des gestionnaires autonomiques, et donc des

8. *REpresentational State Transfer*
9. *stateless* en anglais
10. *'Hypermedia As The Engine Of The Application State'* : hyper media en tant que moteur de l'état de l'application

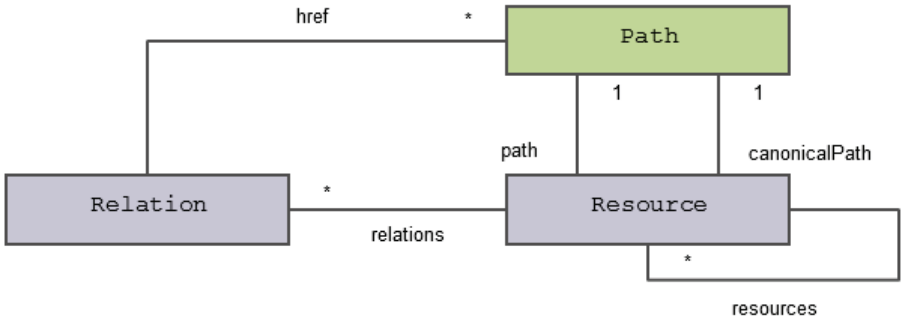

FIGURE 5.26 – Modèle de ressource proposé

applications autonomiques. Le style d'architecture *REST* propose une représentation uniforme de toutes les entités sous la forme de ressources. Ces ressources peuvent être liées entre elles, ce qui permet d'en déduire leur corrélation, et sont auto-descriptives, ce qui simplifie leur manipulation. Cette uniformisation couvre à la fois la simplification d'accès aux entités du système et le masquage de leur hétérogénéité.

Le dynamisme de l'environnement d'exécution est par contre plus difficile à appréhender en utilisant uniquement les principes du style d'architecture *REST*. Il est nécessaire, comme nous l'avons vu, de pouvoir surveiller les changements de cet environnement, afin de pouvoir réagir à ces changements (réactif) ou en anticiper de nouveaux (proactif). Il est aussi nécessaire pour un gestionnaire autonomique d'accéder, à partir du modèle de représentation, aux véritables entités qui sont représentées (*base level*), et *REST*, en masquant totalement ces entités réelles, ne permet pas d'y accéder.

Ces deux points font que le modèle de ressource du style d'architecture *REST* doit être légèrement adapté afin de pouvoir représenter le contexte d'un environnement dynamique, et exprimer toutes ses caractéristiques.

Modèle de ressource

Tout d'abord, le modèle de ressource inspiré de *REST* est utilisé afin de représenter la large variété des entités présentes sur la plateforme d'exécution. Ce modèle, illustré par la figure 5.26, identifie chaque ressource par un identifiant unique : son chemin canonique (*canonical path*). Plusieurs chemins peuvent mener à une seule et même ressource, mais, parmi ces chemins, un seul identifie cette ressource de façon unique. Les ressources sont liées entre elles par des **relations**. Ces relations sont auto-descriptives, et contiennent les informations pour qu'un client puisse les interpréter sémantiquement et les parcourir si nécessaire.

Chaque ressource a un état, qui est représenté sous la forme d'un ensemble clé-valeur. Les propriétés d'état peuvent être individuellement lues, modifiées et effacées,

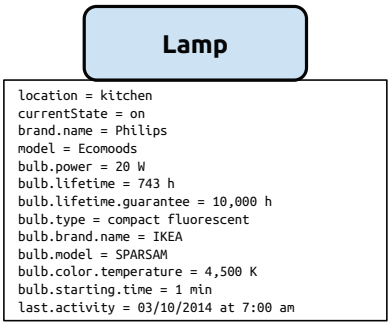

FIGURE 5.27 – Exemple de ressource et de son état

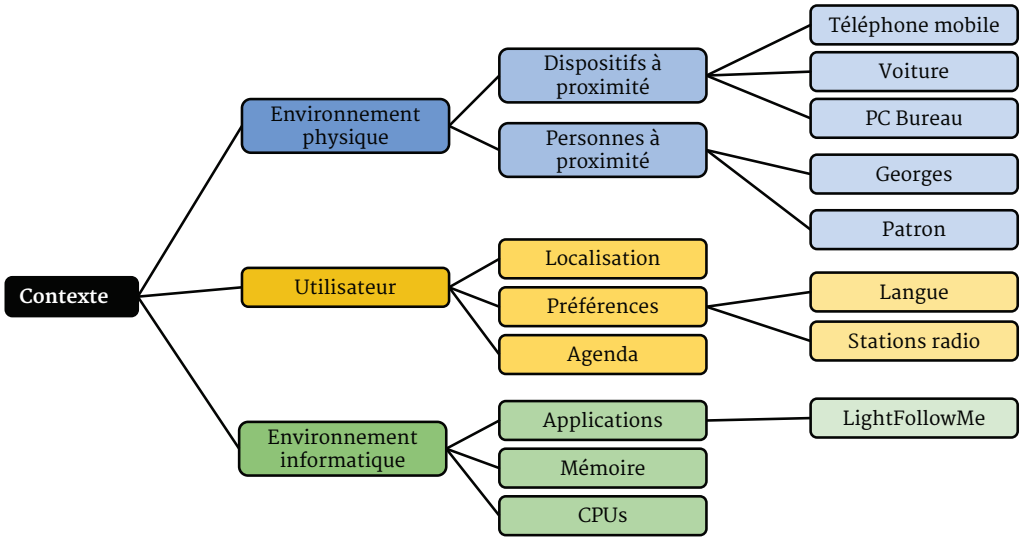

FIGURE 5.28 – Exemple d'arborescence de ressources

comme décrit ci-après. La figure 5.27 montre un exemple de ressource contenant de telles propriétés d'état.

Organisation logique des ressources

Comme mentionné ci-dessus, chaque ressource est accessible par le biais d'un ou plusieurs chemins, dont un chemin canonique qui identifie la ressource. Ces chemins sont structurés, à la manière d'un arbre, et permettent d'organiser les ressources de manière hiérarchique, selon une arborescence.

La figure 5.28 montre un exemple de ressources organisées selon une arborescence. Cette structure logique en arbre est similaire à l'organisation des fichiers dans un système de fichiers : les ressources intermédiaires pouvant être comparées à des répertoires, et les ressources terminales (ou feuilles) à des fichiers ordinaires.

La présentation des ressources sous la forme d'une arborescence n'est qu'une manière de les organiser logiquement. Cette présentation permet d'organiser les représentations des entités de la plateforme selon un schéma très classique et facile à

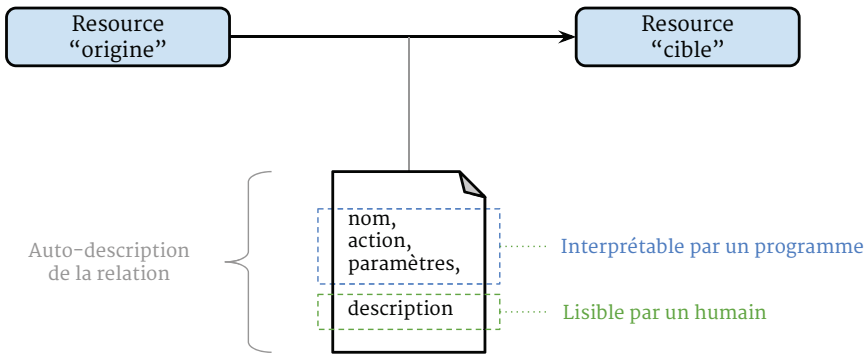

FIGURE 5.29 – Schéma d’une relation entre deux ressources

utiliser. Cette arborescence ne permet par contre pas d’exprimer toutes les relations entre les ressources : pour cela il faut définir une structure en graphe.

Relations inter-ressources

Parmi les différentes entités de la plateforme d’exécution, il en existe qui sont liées, plus ou moins explicitement. Il est donc normal que les représentations de ces entités puissent exprimer ces liens qui unissent les ressources entre elles. Ces relations sont un constituant essentiel de la description de l’état des ressources, puisqu’il permet de décrire les divers rapports qui peuvent exister entre ces ressources. Il y a aussi diverse manières d’interpréter sémantiquement ces relations : dépendance, inclusion, utilisation, couplage, *etc.* Ces relations doivent pouvoir être dynamiques : par exemple, une ressource représentant un composant va être liée à une ressource représentant le service qu’elle utilise actuellement. Ce service pouvant changer, cette relation doit pouvoir être modifiée ou défaite.

Le modèle de ressource proposé permet de définir, de modifier et de casser les relations entre les ressources. Ces relations sont annotées de manière à pouvoir être interprétées à la fois par un humain (description) et par un programme consultant les ressources (instructions d’interprétation). La figure 5.29 montre le schéma général d’une relation, tel que défini dans notre modèle.

Certaines relations sont implicitement déduites de l’organisation logique entre les ressources. Ainsi, une ressource parente aura une relation vers chacune de ses ressources filles, et inversement, les ressources filles auront une chacune relation pointant vers leur ressource parente.

On remarquera aussi que les relations sont unidirectionnelles : elles ne sont navigables que dans un seul sens. Les relations bidirectionnelles sont en général plus difficiles à définir et à auto-décrire, car la sémantique dépend beaucoup du sens de parcours de la relation. Avec notre modèle, il faut donc définir deux relations de sens inverse afin de permettre ce parcours dans les deux sens. De plus, les relations ne permettent de relier que deux ressources entre elles. Si une ressource doit être liée avec n ressources, il faudra donc définir n relations distinctes.

La figure 5.30 montre un exemple d’utilisation des relations entre ressources. Ces relations permettent d’exprimer ici des liens sémantiques entre les entités repré-

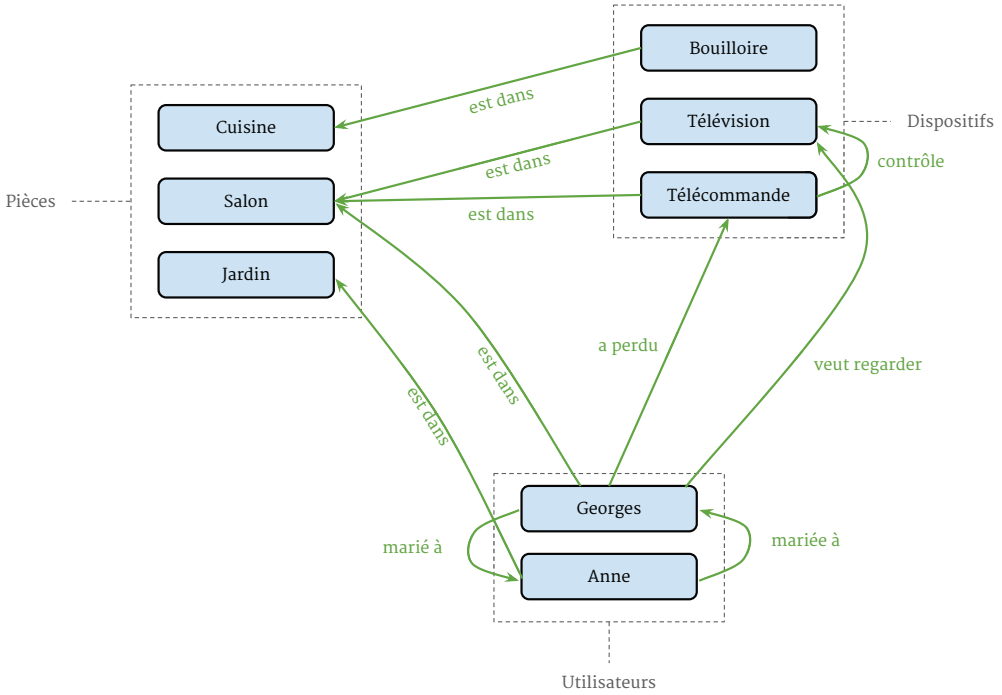

FIGURE 5.30 – Exemple de graphe de ressources

sentées : utilisateurs, dispositifs, pièces de la maison. Cet exemple illustre bien la diversité des informations de contexte que les relations peuvent représenter. Au même titre que l'état des ressources, les relations entre les ressources permettent le tissage de nouvelles informations de contexte (par exemple : `Georges.currentLocation = Salon`).

Le modèle de ressource et de relations fournit donc un niveau d'expression assez complet pour traduire les différentes natures des informations de contexte. La suite de cette section va maintenant détailler la manière d'accéder, de modifier, d'adapter, d'étendre et de transformer ces ressources. Ces opérations sont nécessaires afin de proposer un modèle suffisamment simple et flexible pour l'implémentation de gestionnaire autonomiques d'application.

Observe, Create, Read, Update, Delete

Afin de pouvoir interagir avec les ressources de façon uniforme, il est nécessaire de définir des opérations de manipulations élémentaires. Bien que l'approche *REST* ne mentionne pas explicitement quelles sont les opérations permettant d'agir sur les ressources, le domaine des bases de données définit un ensemble d'opérations élémentaires permettant la manipulation des données, et donc des propriétés de contexte. Ces opérations, au nombre de quatre, sont souvent désignées sous l'acronyme *CRUD* [11], et sont les suivantes :

– ***CREATE*** : permet la création de nouvelles ressources. Cette opération permet de créer et de définir en même temps, de manière atomique, l'état initial de la ressource.

11. *Create, Read, Update, Delete*

- ***READ*** : opération de lecture de l'état d'une ressource, ce qui inclut les relations sortantes vers d'autres ressources. Cette opération permet d'obtenir un instantané de l'état courant de la ressource.
- ***UPDATE*** : opération de modification de la ressource. Les interactions étant sans état (*stateless*), l'intégralité du nouvel état de la ressource doit être communiquée. Il n'est pas possible de modifier partiellement la ressource (*patch*).
- ***DELETE*** : suppression de la ressource.

Ces quatre opérations permettent de manipuler à la fois l'état des ressources mais aussi les relations entre ces ressources. Elles sont donc suffisantes pour remodeler le graphe des ressources, et exprimer les changements pouvant survenir dans le contexte.

Cependant, il est nécessaire de rajouter une pseudo-opération qui permet de surveiller l'état d'une ressource, et de recevoir des notifications lors des modifications de son état ou de sa suppression. Comme mentionné précédemment, cette surveillance est nécessaire afin de permettre la réactivité et la proactivité du gestionnaire autonomique :

- ***OBSERVE*** : surveille une ressource. Les événements de modification de l'état de la ressource, de modification de ses relations sortantes, ou de suppression de la ressource sont communiqués au client, qui peut réagir en conséquence.

Dans la suite de ce manuscrit, nous mentionnerons ces cinq opérations sont l'acronyme ***Ó CRUD*** [12], ou bien opérations **CRUD étendues** (sous-entendu pour la prise en compte du dynamisme).

Adaptations

Le modèle de représentation proposé au gestionnaire autonomique permet en théorie d'abstraire toutes les entités présentes sur la plateforme. Le niveau *méta* est donc l'entité avec laquelle les gestionnaires autonomiques construisent leur connaissance du contexte, modifient ce contexte et adaptent les applications.

Il y a cependant parfois des opérations de bas niveau qui doivent être réalisées par les gestionnaires autonomiques, qui doivent donc accéder directement aux entités du niveau de base. Le *meta-object protocol*, par définition, masque totalement ces entités de bas niveau, et le gestionnaire autonomique a alors la charge de retrouver, à partir des ressources non-typées du niveau *méta*, les entités du niveau de base qui y sont rattachées.

Cette opération de résolution des entités à partir de leur représentation peut être une tâche fastidieuse, voire impossible, selon le détail des informations proposées par le niveau *méta*. Nous proposons donc, dans notre modèle, une opération d'**adaptation des ressources**. L'adaptation d'une ressource permet au gestionnaire autonomique de retrouver l'entité qui est représentée, si elle existe. Une ressource pouvant représenter plusieurs facettes de la même entité, ou même représenter simultanément plusieurs entités différentes, le gestionnaire doit connaître le type de l'entité qu'il souhaite retrouver.

La figure 5.31 montre comment l'adaptation permet par exemple au gestionnaire autonomique d'accéder aux instances de composants iPOJO et aux services en utilisant l'opération d'adaptation sur les ressources qui les représentent. Cette adaptation permet de redescendre la montée en abstraction fournie par le modèle de ressources.

Il faut noter que certaines ressources peuvent être totalement virtuelles, et ne correspondent donc à aucune entité réelle. Logiquement, l'adaptation de telles ressources

12. À l'irlandaise : *Observe, Create, Read, Update, Delete*

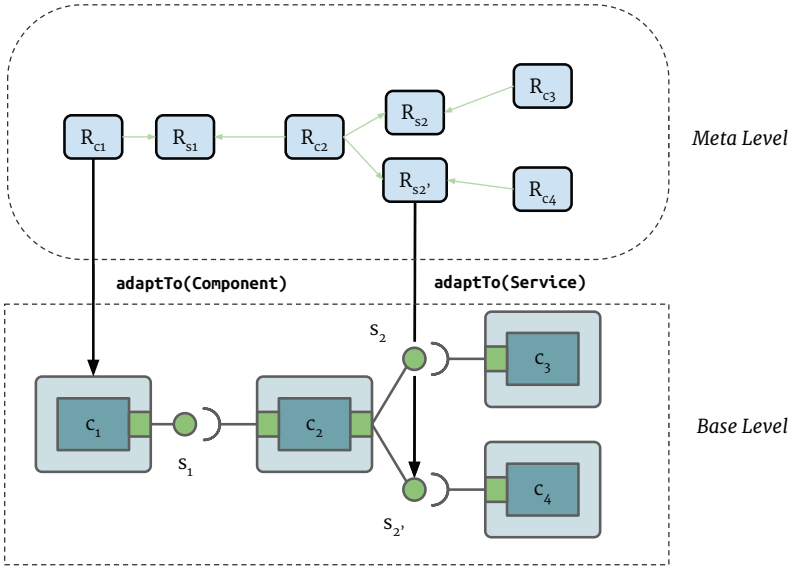

FIGURE 5.31 – Adaptation de ressources

ne fournira rien. Il en est de même si le gestionnaire essaie d'adapter une ressource vers un type d'entité incompatible. Le gestionnaire doit donc connaître à l'avance quelle ressource peut être adaptée en quel type d'entité.

L'opération d'adaptation va à l'encontre du principe d'abstraction proposé par le modèle de représentation des ressources. Son utilisation est donc déconseillée, car elle viole le règle d'accès uniforme. Dans les rares cas où elle est nécessaire, l'adaptation des ressources apporte une simplification de l'écriture du gestionnaire autonomique en lui évitant la recherche des entités de bas niveau à partir de leurs représentations.

Domaines d'extension

Les principes du modèle proposé jusqu'ici permettent au gestionnaire autonomique un accès uniforme aux ressources, aux relations entre elles et leur surveillance. L'adaptation permet d'inverser l'abstraction promue modèle. Toutefois, aucune explication n'a été fournie sur la manière d'abstraire les entités de la plateforme sous forme de ressources : ceci est le rôle des domaines d'extension.

La représentation des ressources est à la fois générique et théorique, et permet la création, la modification et la suppression de ressources totalement virtuelles et déconnectées des réalités de la plateforme d'exécution. La communication entre ce modèle et la plateforme d'exécution nécessite d'établir des « ponts » entre ces deux « mondes ». On peut comparer ces ponts aux pilotes de périphériques dans un système d'exploitation, qui sont des passages entre le monde logique du système et le monde physique des dispositifs.

Le modèle proposé est, de base, virtuel, mais peut être étendu par l'adjonction de domaines d'extension. Chaque domaine d'extension est un pont vers des types d'entité particuliers, et permet de les représenter dans notre modèle sous forme de ressource. Par exemple, la figure 5.32 montre comment un domaine d'extension permet de représenter les instances de composants iPOJO, un autre les services de la plateforme, et un dernier l'état de la machine physique. Chaque domaine gère donc un aspect bien spécifique du niveau de base.

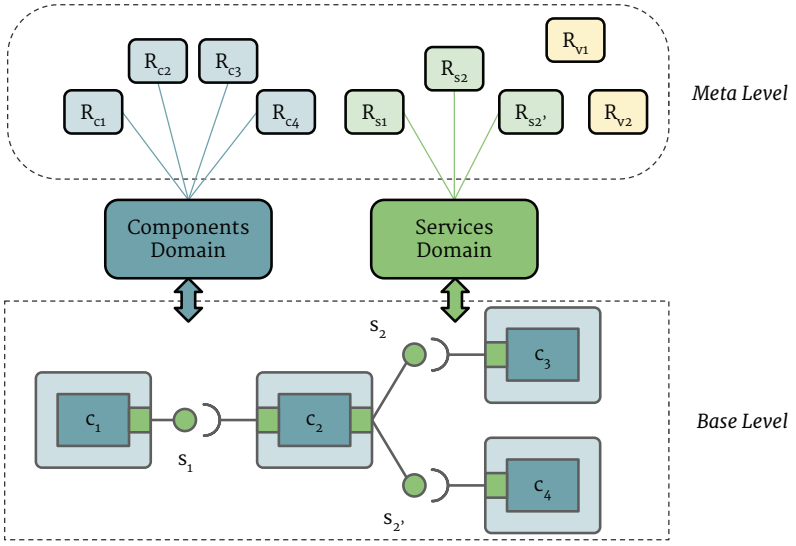

FIGURE 5.32 – Exemples de domaines d'extension du modèle

Certains domaines d'extension sont indispensables aux gestionnaires autonomiques pour représenter l'architecture de la plateforme d'exécution :

- Le **domaine des services** représente l'ensemble des services présents sur la plateforme, et reflète leur dynamique.
- Le **domaine des composants** représente les composants iPOJO s'exécutant sur la plateforme, ainsi que leur état. Le dynamisme étant une de leurs caractéristiques essentielles, il est aussi reflété par les ressources maintenues par ce domaine. Ce domaine doit aussi présenter les concepts de dépendance de service et de fourniture de service.
- Le **domaine système** représente l'état courant de la machine virtuelle (*JVM*) et physique.

D'autres domaines peuvent cependant être ajoutés afin de fournir des informations de contexte complémentaires, par exemple :

- Le **domaine des systèmes** fichiers qui représente les fichiers de la machine sous la forme de ressources.
- Le **domaine utilisateur** qui stocke et maintient les préférences de ou des utilisateurs du système.

En plus de fournir de nouvelles ressources, les domaines d'extension peuvent aussi nécessiter l'altération de ressources existantes. De plus, un gestionnaire autonomique peut avoir à transformer sa représentation locale des ressources, afin qu'elle soit plus adaptée aux concepts manipulés par ce gestionnaire. Ces deux opérations sont rendues possibles par la transformation de ressources.

Transformation de ressources

Il est parfois nécessaire pour un domaine ou pour un client de modifier la vue d'une ressource existante. La ressource peut être modifiée directement, mais cela peut générer des conflits avec le domaine d'extension ou le gestionnaire autonomique qui a créé et qui maintient l'état de cette ressource. Il faut plutôt transformer cette ressource sans la toucher : les transformations ne seront visibles que pour les utilisateurs de la

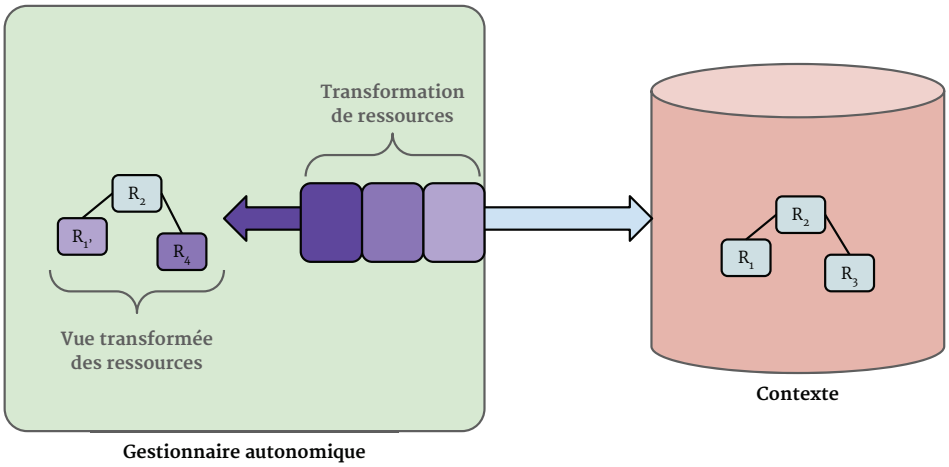

FIGURE 5.33 – Transformation locale de ressources

ressource, mais son état n'aura pas véritablement changé.

Ce mécanisme est nécessaire pour deux raisons principalement :

- Un gestionnaire autonomique peut avoir besoin d'une vue locale du graphe des ressources. Cette vue est adapté spécifiquement aux besoins de ce gestionnaire, et présente des informations de contexte qui concernent ce gestionnaire, et lui seul : c'est une **transformation locale**.
- Un domaine d'extension peut, en plus de présenter de nouveaux types de ressources, avoir à étendre des ressources existantes afin de fournir des informations supplémentaires relatives à ce domaine. Ces informations sont exposées à l'ensemble des utilisateurs des ressources : c'est une **transformation globale**.

Dans le premier cas, le client qui utilise la ressource peut manuellement appliquer les transformations afin d'avoir sa vue personnalisée des ressources. Ce mécanisme ne modifie en rien le modèle de ressources proposé, mais nous fournissons des outils facilitant cette transformation locale. Le client peut appliquer une succession de transformation sur une ressource jusqu'à obtenir une ressource qui correspond à son niveau d'expression, tel qu'illustré par la figure 5.33. À la manière des chaînes d'interception, la chaîne de transformation est en charge de refléter les changements d'état de la ressource réelle sur la ressource transformée, et inversement.

Dans le cas de la transformation globale, le domaine d'extension doit pouvoir transformer la ressource sans impact sur la ressource elle-même : chaque accès à la ressource sera en fait réalisé sur une version transformée de celle-ci. Par exemple, la figure 5.34 illustre le cas du domaine d'extensions des composants qui transforme les ressources-services afin d'ajouter des relations vers les composants qui les fournissent et les utilisent. Le gestionnaire accède de façon transparente aux ressources, sans se douter qu'elles ont été transformées à la volée.

La transformation de ressources permet donc, à la fois au niveau local et aussi au niveau global d'étendre et de transformer la représentation des ressources, afin de l'adapter à des points de vue particuliers ou de présenter des informations globales relatives à un domaine particulier.

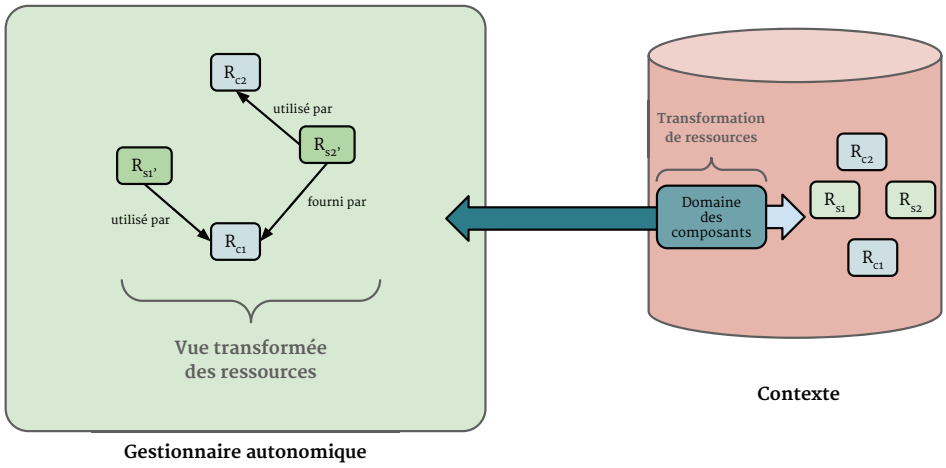

FIGURE 5.34 – Exemple de transformation globale de ressources

5.5 Conclusion

L'ensemble de la proposition énoncée ci-dessus peut se résumer en trois points fondamentaux :

- L'extension du conteneur de composant, afin de faciliter sa gestion par un gestionnaire de composant.
- La modification des interactions entre les composants et les services, ce qui permet de modifier, au niveau local, la résolution des services, leur comportement ou leurs liaisons.
- La présentation des informations de contexte sous la forme de ressources, et la possibilité d'appliquer des opérations sur ces ressources, afin de favoriser leur utilisabilité par des gestionnaires autonomiques locaux et leur montée en abstraction pour le gestionnaire autonomique globale.

Une attention toute particulière a été apportée sur la conservation du modèle de développement des composants iPOJO. Comme nous le verrons dans le chapitre suivant, la grande majorité des modifications apportées ne nécessitent aucun changement ni recompilation des composants existants, qui peuvent donc s'exécuter normalement sur la plateforme d'exécution étendue.

Si le comportement à l'exécution des composants est individuellement identique, la manière de les surveiller, de les adapter et de changer leur topologie a été profondément bouleversée. Ses changements permettent de combler les limitations d'iPOJO citées en début de chapitre, qui empêchaient la gestion des composants par des gestionnaires autonomiques. Ces composants sont maintenant réflexifs, et permettent d'extraire et d'injecter des informations de contexte essentielles pour les gestionnaires autonomiques. Les différents contextes sont présentés de manière uniforme par un modèle à ressource. Enfin, la connaissance du ou des contextes permet aux gestionnaires de liaisons de manipuler les mécanismes de résolution des services et de liaison, dans le but d'appliquer des intentions d'adaptation directement issues des buts de haut niveau du gestionnaire autonomique.

Le chapitre suivant montre la mise en œuvre de l'ensemble des concepts proposés dans cette contribution. L'implantation de ces mécanismes au sein du framework Apache Felix iPOJO y est détaillée. Les concepts proposés et de leurs implantations y

sont ensuite évalués et validés.

Validation

Chapitre 6

Implantation et Validation

Dans les chapitres précédents, nous avons proposé un modèle pour la construction d'applications autonomiques. Il répond aux besoins de simplification de conception et d'exécution d'applications autonomiques, dans des environnements hétérogènes et dynamiques. Les environnements ubiquitaires représentent les archétypes de tels contextes d'exécution : fondamentalement dynamiques et imprévisibles. Les approches de développement classiques ne permettent que très rarement de prendre en compte ces contraintes lors de la conception d'applications. L'informatique autonomique apporte des concepts permettant d'insuffler une capacité de changement et d'adaptation dans les applications.

Le modèle proposé dans les chapitres 4 et 5, en s'appuyant sur une machine d'exécution supportant le dynamisme et l'adaptation, permet théoriquement de simplifier les applications s'exécutant dans des environnements ubiquitaires, ou plus généralement des environnements dynamiques. Cette proposition nécessite la construction d'une représentation du système, ainsi que la modification de la couche d'exécution de bas niveau.

Ce chapitre s'articule autour de deux axes majeurs :

- L'implantation des modifications de la machine d'exécution, ainsi que du modèle de représentation du système seront présentées.
- La validation des propositions et de leurs implantations, en s'appuyant sur des cas d'utilisation dans le domaine de l'informatique ubiquitaire.

Sommaire

6.1 Implantation de la proposition

Le chapitre précédent propose deux axes permettant la simplification des applications autonomiques depuis la conception jusqu'à leur exécution. Les deux améliorations proposées sont :

- L'extension de la machine d'exécution iPOJO, afin de la rendre *'Autonomic-Ready'*, et permettre ainsi les applications adaptables par des gestionnaires autonomiques.
- Un modèle de représentation du système, qui permet de regrouper dans un modèle uniforme les données de contexte issues de la machine d'exécution, de l'environnement ubiquitaire et des utilisateurs du système.

L'implantation des améliorations proposées est donc logiquement découpée en deux éléments, complémentaires mais dissociables. Les modifications de la machine d'exécution ont vocation à fusionner avec le projet iPOJO lui-même, afin que ces améliorations puissent profiter à toutes les applications futures développées à partir de ce *framework*. Le modèle de représentation du système peut plutôt être qualifié de « compagnon » à l'exécution. Il n'est pas nécessaire pour l'exécution d'applications, mais son utilisation permet, comme nous allons le montrer, de grandement simplifier la gestion autonomique de ces applications.

La suite de cette section va donc présenter l'implantation de ces deux propositions : la première étant l'ajout des mécanismes d'interception au sein du *middleware* Apache Felix iPOJO, et la deuxième un modèle de représentation uniforme du système.

6.1.1 Modifications de la couche d'exécution iPOJO

La première partie de l'implantation de la proposition est donc la modification du *framework* Apache Felix iPOJO. Comme nous l'avons vu dans les deux chapitres précédents, les principes prônés par iPOJO permettent de concevoir des applications modulaires, qui peuvent prendre en compte le dynamisme de l'environnement d'exécution. Les points adaptables de l'application se situent à deux niveaux :

- Les composants peuvent être reconfigurés lors de l'exécution. Ces adaptations permettent de changer le comportement du composant, et donc de réagir à un changement de l'environnement d'exécution (par exemple la localisation de l'utilisateur a changé).
- Les liaisons entre les composants sont dynamiques. Un composant exprime des dépendances sur les types de services dont il a besoin, et fournit d'autres de services aux autres composants.

Ces points d'adaptation des applications à base de composants à services ne sont cependant pas suffisants pour permettre la gestion autonomique de ces applications, comme nous l'avons vu dans le chapitre 4. C'est pourquoi il a fallu ajouter les fonctions nécessaires au *framework* iPOJO.

Les modifications apportées correspondent aux mécanismes d'adaptation décrits dans le chapitre 5, c'est-à-dire l'interception du code métier des composants, l'interception des dépendances de services, et l'interception des fournitures de services. Ces adaptations sont génériques, car utilisables par n'importe quel gestionnaire autonomique, quel que soit le domaine de l'application. Ces mécanismes d'adaptation sont aussi très techniques, car ils font appels à des concepts de bas niveau de la plateforme d'exécution, et doivent maintenir la cohérence des états des composants et des services, même lors d'accès concurrents.

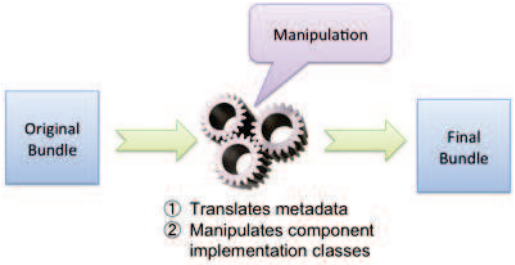

FIGURE 6.1 – Manipulation du code métier lors de la compilation

La suite de cette section va décrire la manière dont ces propositions ont été mises en œuvre, et la manière dont elles peuvent être mises à profit par les gestionnaires autonomiques des applications.

Interception du code métier des composants iPOJO

L'interception du code métier des composants iPOJO doit permettre aux gestionnaires autonomiques de modifier le comportement fonctionnel des composants. Les gestionnaires autonomiques spécifiques au composant doivent pouvoir, comme nous l'avons vu précédemment, intercepter les appels aux méthodes du composant, les accès en lecture et en écriture aux attributs du composant. Toutes ces interceptions doivent être effectuées selon le patron de conception en chaîne d'interception, présenté en détail dans la section 5.3.1.

État des lieux

Dans sa version de base (c'est-à-dire avant les modifications proposées dans cette thèse), iPOJO met en place un mécanisme d'interception rudimentaire qui permet aux différents *handlers* d'augmenter le comportement du code métier, par exemple en injectant des services requis dans un attribut. Pour cela, le code métier est manipulé lors de sa compilation, afin d'ajouter ces points d'interception, tel que le montre la figure 6.1.

Cette manipulation du code à la compilation permet l'exécution des composants dans des environnements embarqués, où il n'est pas envisageable, pour des raisons évidentes de performances, de manipuler ce code à l'exécution.

La figure 6.2 montre un exemple de code métier avant et après manipulation par la version iPOJO de base. Le code de la partie droite est en réalité un pseudocode : la manipulation n'est pas effectuée à partir du code source du composant, mais plutôt à partir du code intermédiaire compilé [1], tel qu'il est interprété par la machine virtuelle Java. Dans cet exemple, nous pouvons voir qu'un attribut `sensorInstanceManager`, qui est le conteneur de composants iPOJO, a été ajouté. La méthode métier `doSomething()` est renommée en `__doSomething()`, et les accès aux attributs qu'elle pouvait contenir sont remplacés par des appels au conteneur du composant. La méthode `doSomething()` finale appelle des méthodes de rappel [2] avant et après l'exécution du code métier de la méthode `__doSomething()`.

1. *Bytecode* Java : `http://en.wikipedia.org/wiki/Java_bytecode`
2. Appelées en anglais *callback*

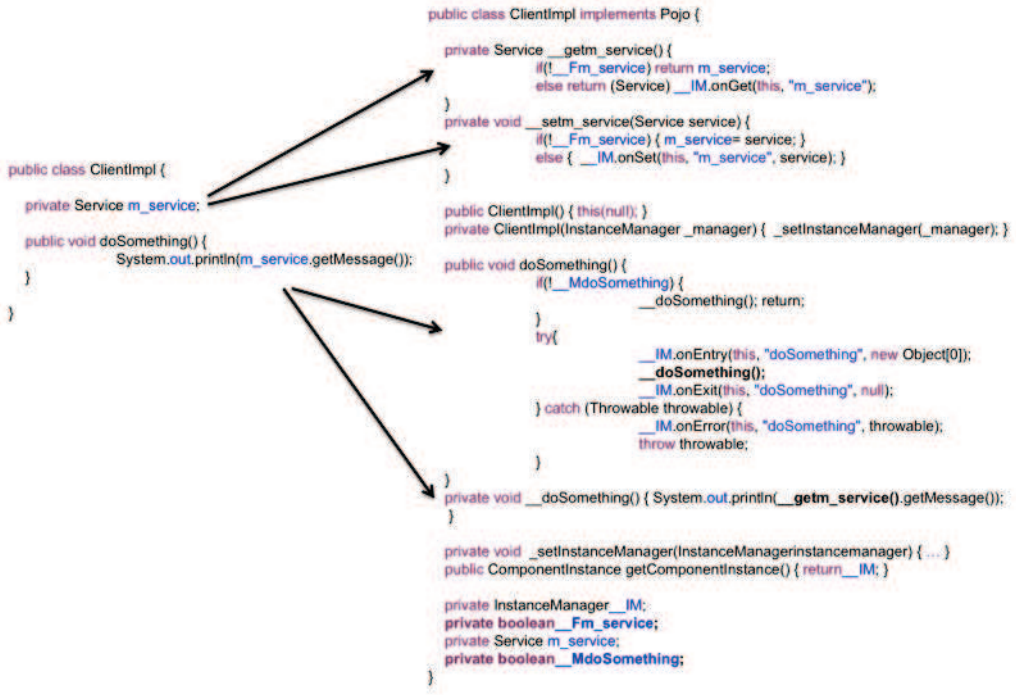

FIGURE 6.2 – Code metier avant et après manipulation

Ce mécanisme de manipulation de code métier par iPOJO est très technique, car la sémantique du code métier originel doit impérativement être maintenue, et les surcoûts à l'exécution du composant manipulé doivent être réduits au minimum [3].

La mise en œuvre des chaînes d'interception du code métier des composants nécessite de changer la manière avec laquelle iPOJO manipule ce code métier. En effet, tel qu'énoncé dans la section 5.1, la version iPOJO de base ne permet par exemple ni d'empêcher l'exécution d'une méthode métier, ni de modifier les paramètres d'appel.

API des chaînes d'interception au code métier

L'implantation des chaînes d'interception du code métier doit permettre aux gestionnaires autonomiques de composants d'ajouter leurs propres intercepteurs. Les *handlers* de composants doivent continuer à fonctionner. Les gestionnaires autonomiques doivent aussi pouvoir arbitrer les différents intercepteurs, qu'ils proviennent de *handlers* ou de gestionnaires autonomiques, en modifiant l'ordre de la chaîne d'interception, ou en ajoutant ou retirant des intercepteurs de la chaîne.

Remarque : Dans un souci de concision, la suite de cette section va montrer les modifications effectuées pour l'interception des méthodes métiers des composants. Les modifications apportées pour les autres accès interceptés (accès aux attributs et les constructions d'objets métiers) sont très semblables, et les différences mineures n'apportent pas de réel intérêt à ce chapitre.

3. C'est la principale raison qui a poussé iPOJO à adopter ce principe complexe de manipulation à la compilation, plutôt que d'utiliser des mécanismes beaucoup plus simples, mais désastreux en terme de performance, comme la réflexion.

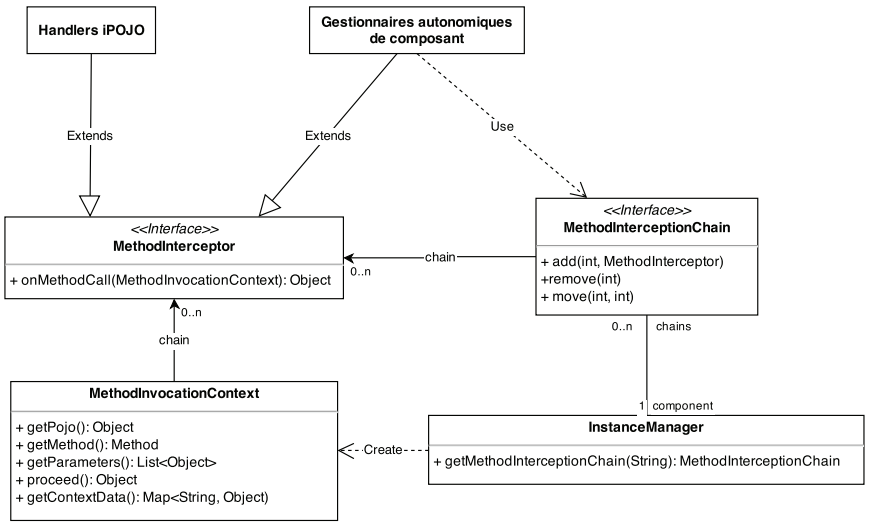

FIGURE 6.3 – Diagramme de classes des intercepteurs de méthodes métiers

La figure 6.3 montre comment le concept de chaîne d'interception de méthodes métiers a été ajouté à iPOJO. Les *handlers* iPOJO et les gestionnaires autonomiques de composants qui souhaitent intercepter les appels de méthodes du code métier du composant implémentent l'interface `MethodInterceptor`. Le conteneur de composant, `InstanceManager`, fournit une méthode qui permet d'accéder à la chaîne des intercepteurs pour chaque méthode du composant. Les gestionnaires autonomiques de composants peuvent accéder à cette chaîne, `MethodInterceptionChain`, et la manipuler en ajoutant, retirant ou déplaçant des intercepteurs.

Lorsqu'une méthode métier du composant est appelée, le conteneur va créer un contexte d'invocation de méthode, `MethodInvocationContext`, qui contient les paramètres d'appel de la méthode, l'objet cible, le nom de la méthode appelée. Ce contexte d'invocation contient aussi un instantané de la chaîne d'interception relative à la méthode appelée. Ensuite, ce contexte va appeler les intercepteurs constituant la chaîne.

La figure 6.4 illustre un intercepteur d'appel de méthodes métier fictif, tel qu'un gestionnaire autonomique de composant pourrait l'implémenter. Dans cet exemple, plusieurs comportements possibles sont montrés : l'intercepteur peut modifier la valeur des paramètres d'appel (ligne 6). Ces changements sont visibles immédiatement, et pour tous les intercepteurs suivants. L'appel peut être propagé au reste de la chaîne d'interception (ligne 10), à l'aide du contexte d'appel. Enfin, la chaîne peut être interrompue en retournant prématurément une valeur de retour (ligne 13). Dans ce dernier cas, les intercepteurs suivants sont ignorés, et la méthode métier du composant n'est pas exécutée.

Le comportement modifié de l'interception des accès au code métier des composants correspond à celui présenté dans la section 5.2 : chaque intercepteur a un contrôle total sur le déroulement de l'appel de la méthode métier. Les gestionnaires autonomiques, en utilisant la classe `MethodInterceptionChain` peuvent aussi modifier la

```
public class InterceptorExample implements MethodInterceptor {

  @Override
  public Object onMethodCall(MethodInvocationContext context) {
    // Modification de la valeur d'un paramètre.
    context.getParameters().set(0, null);

    if (context.getMethod().getName().contains("important")) {
      // Propagation de l'appel à l'intercepteur suivant.
      return context.proceed();
    } else {
      // Interruption de la chaîne d'interception.
      return aReturnValue;
    }
  }

}
```

FIGURE 6.4 – Exemple d'intercepteur de méthode métier

```
public class ClientImpl implements Pojo {

  ...

  public void doSomething() {
    if (!__MdoSomething) {
      // Cas où la classe est utilisée sans iPOJO.
      // ou bien la méthode n'est pas interceptée par iPOJO.
      __doSomething(); return;
    }
      // Transmission de l'appel au conteneur.
      // C'est lui qui construit le contexte d'invocation de la méthode,
      // et appelle la chaîne d'interception.
    __IM.onMethod(this, "doSomething", null);
  }

  ...

}
```

FIGURE 6.5 – Code métier transformé par le nouveau manipulateur d'iPOJO

chaîne d'interception dynamiquement. La gestion des accès concurrents est prise en charge par le conteneur de composant, qui évite par exemple d'un intercepteur ne soit retiré alors qu'il est présent dans une chaîne d'interception active.

Ce nouveau modèle d'interception nécessite aussi de modifier le mécanisme de manipulation du code métier lors de la compilation. Si l'on reprend l'exemple de la figure 6.2, le manipulateur de code métier va produire une classe dont le pseudocode ressemble à celui de la figure 6.5. Seule la méthode modifiée est montrée, le reste de la classe restant inchangé.

Interception des dépendances et des fournitures de services

Les autres mécanismes d'interception ajoutés au *framework* iPOJO concernent les services. Ils doivent permettre, comme décrit dans les sections 5.3.2 et 5.3.5, de modifier la façon de fournir, de sélectionner et d'utiliser les services. Afin de réaliser cela, il faut plonger dans le conteneur d'iPOJO, plus particulièrement dans les *handlers* qui sont en charge des fournitures de services, le *provider handler*, et des dépendances de services, le *dependency handler*.

Améliorations du *dependency handler*

Le *dependency handler* d'iPOJO est l'intermédiaire qui va, à partir des annotations présentes dans le code métier du composant, scruter le registre de services de la plateforme *OSGi* et obtenir le ou les services correspondants pour les injecter dans le composant. Les principales caractéristiques d'une dépendance de service ont été décrites dans la section 2.7.3. Le *dependency handler* doit aussi prendre en compte le contexte d'exécution des composants et de la plateforme, qui inclut notamment la compatibilité entre les différents modules et les droits d'accès aux services disponibles. Ces étapes de sélection des services candidats utilisables par le composant se déroulent de manière transparente pour ce composant.

L'ajout de mécanismes d'interception des dépendances de service doit conserver cette transparence : le code métier ou ses annotations descriptives ne doivent pas changer. Si depuis l'intérieur du composant rien ne change, le *dependency handler* fournit à l'extérieur une interface permettant à un gestionnaire autonomique d'ajouter différents intercepteurs :

- Des intercepteurs du registre de service, appelés `ServiceTrackingInterceptors`,
- Un intercepteur permettant l'ordonnancement des services sélectionnés par la chaîne d'interception précédente, appelé `ServiceRankingInterceptor`,
- Des intercepteurs de liaison aux services, appelés `ServiceBindingInterceptors`.

Pour chaque composant iPOJO, le *dependency handler* peut prendre en charge une ou plusieurs dépendances de service. Chaque dépendance est représentée par un objet de type `DependencyModel`. Ce modèle gère la relation entre la dépendance de service, telle que définie dans la description du composant, et le registre de service de la plateforme *OSGi*. Ce modèle de dépendance est déjà assez complexe, dans la version iPOJO de base. Afin de ne pas démultiplier cette complexité, nous avons ajouté une interface intermédiaire, nommée `ServiceReferenceManager`, qui prend en charge la gestion des intercepteurs sur une dépendance. C'est à partir de cette interface qu'un gestionnaire autonomique de liaison va pouvoir ajouter, retirer ou modifier les différentes chaînes d'interception.

Le diagramme de classe de la figure 6.6 montre les relations entre les différentes entités qui constituent une dépendance *Autonomic-Ready*. Le `ServiceReferenceManager`, en plus de contenir des listes d'intercepteurs, maintient une collection de références de services modifiables, les `TransformedServiceReference`s. Ces références, qui étendent les `ServiceReference` originelles de la plateforme *OSGi*, permettent d'enrichir un service avec des propriétés de contexte, ou au contraire d'enlever ou de modifier certaines propriétés.

Les figures suivantes montrent comment un gestionnaire autonomique de liaison peut utiliser ces nouveaux mécanismes mis en place. La figure 6.7 est un exemple de composant métier très simpliste. Ce composant déclare une dépendance sur un service de type `BinaryLight`. C'est sur cette dépendance que les gestionnaires autonomiques de liaisons des figures suivantes vont réaliser des adaptations.

Le premier exemple, illustré par la figure 6.8, est un gestionnaire autonomique de liaison réalisant une interception au registre de services. Son but est d'ajouter une information de localisation des dispositifs potentiellement utilisés par la dépendance de service. Á cette fin, le gestionnaire dispose d'informations de contexte, `deviceLocations`, qui contient ces connaissances de localisations des dispositifs. Le moyen utilisé pour obtenir ces informations n'est pas montré dans l'exemple, qui se focalise plutôt sur la manière d'exploiter ces informations en réalisant des adaptations. Deux types d'adaptation sont montrés dans cet exemple. La première est la suppression d'un service dont

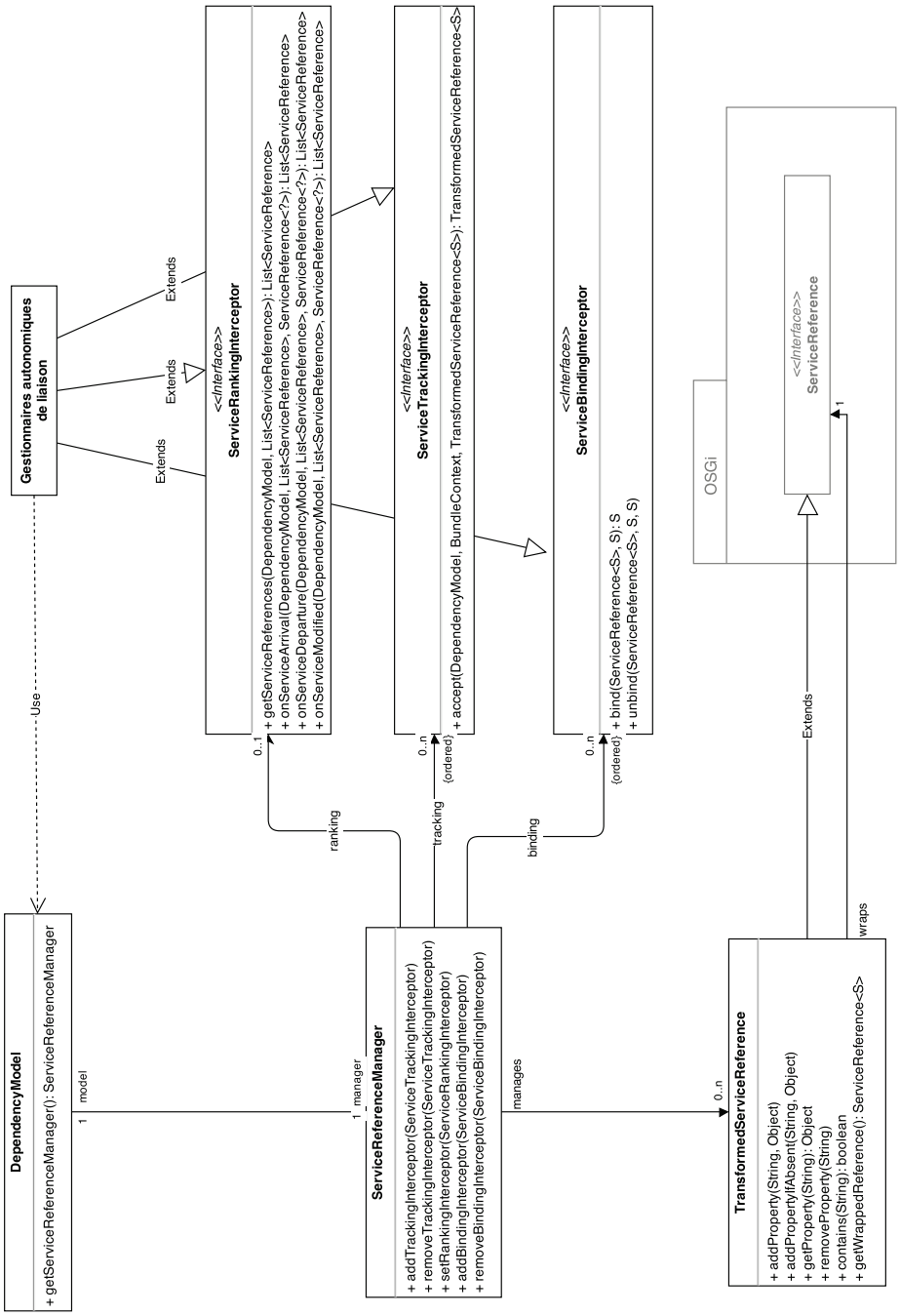

FIGURE 6.6 – Diagramme de classes de `DependencyModel`

```
@Component
public class SimpleComponent {

  @Requires(id="light")
  BinaryLight binaryLight;

  // ... suite de la logique métier du composant ...

}
```

FIGURE 6.7 – Exemple de composant avec une dépendance simple

```java
/**
 * Exemple de gestionnaire autonomique de liaison interceptant
 * le registre de services.
 */
@Component
public class Manager1 implements ServiceTrackingInterceptor {

    /**
     * Localisations des dispositifs.
     */
    private Map<String, String> deviceLocations;

    /**
     * Méthode interceptant les services provenant du registre.
     */
    @Override
    public <S> TransformedServiceReference<S> accept(
                                        DependencyModel dependency,
                                        BundleContext bundleContext,
                                        TransformedServiceReference<S> ref) {
        if (ref.contains("device.location")) {
            // Le service renseigne déjà sa localisation.
            // Il est retourné sans modification.
            return ref;
        }

        String id = (String) ref.get("device.id");
        if (id == null || !deviceLocations.containsKey(id)) {

            // Pas d'information de localisation sur ce dispositif.
            // Ce service est éliminé ; il ne sera pas visible pour
            // la dépendance de service.
            return null;

        } else {

            // Ajout de l'information de localisation sur le service.
            ref.addProperty("device.location", deviceLocations.get(id));
            return ref;

        }
    }

}
```

FIGURE 6.8 – Exemple d'interception du registre de services

nous ne disposons d'aucune information (ligne 33). Le service n'est pas supprimé du registre, mais il est complètement invisible pour la dépendance de service gérée. La deuxième adaptation (ligne 38) ajoute une propriété de localisation au service. Si le service contient déjà cette propriété, il n'est pas modifié (ligne 24).

Le deuxième exemple, illustré par la figure 6.9, montre comment le gestionnaire autonomique de liaison peut réordonner la liste des services sélectionnés. Ces services sont préalablement filtrés par une chaîne contenant les intercepteurs du registre de services (comme par exemple l'intercepteur de la figure 6.8). Dans cet exemple, le gestionnaire définit un ordre total sur les services en comparant leur consommation énergétique. Cette consommation est supposée être déclarée par chaque service dans une propriété : `device.power`. L'ordre défini par `COMPARATOR` positionne en première place les services consommant le moins d'énergie. Si un service ne déclare pas sa consommation, il est fortement pénalisé et placé en fin de liste.

Cet intercepteur réagit aux événements relatifs aux services sélectionnés. Par exemple, quand un nouveau service est sélectionné pour la dépendance, la méthode `onServiceArrival` est appelée : elle insère le nouveau service dans la liste puis trie cette liste en utilisant l'ordre précédemment défini. Lorsqu'un service précédemment sélectionné disparaît (ou n'est plus sélectionné), la méthode `onServiceDeparture` est appelée : le service est retiré de la liste.

Le troisième et dernier exemple d'intercepteur sur une dépendance de service est le `ServiceBindingInterceptor`. La figure 6.10 montre un exemple simple d'un tel intercepteur qui ajoute au service utilisé des fonctions de journalisation. Au lieu d'utiliser le service qui a été publié dans le registre, la dépendance va utiliser celui fourni par cet exemple. Ainsi, les appels de méthode sur cet objet provoqueront la journalisation de messages de *log* en plus de l'appel des mêmes méthodes sur le service réel. Cet exemple est volontairement simpliste ; un gestionnaire autonomique de liaison plus réaliste devra plutôt générer du code à la volée selon le type de service utilisé.

Améliorations du *provider handler*

Afin de réaliser les interceptions sur les fournitures de services, le *provider handler* d'iPOJO a dû être modifié. De la même manière que le *dependency handler* est l'intermédiaire entre une dépendance de service et le registre de service *OSGi*, le *provider handler* est en charge de la publication des services fournis par un composant dans ce registre de services. La description d'un service fourni par un composant a été décrite dans la section 2.7.3, et comprend principalement la liste des interfaces fonctionnelles du service, ses propriétés descriptives et sa politique de fourniture.

En ajoutant les intercepteurs de fourniture de service, décrits dans la section 5.3.5, nous devons, comme dans le cas des dépendances de service, conserver la transparence pour le code métier des composants. En aucun cas la manière de décrire une fourniture de service ne doit changer ; seule la capacité d'un gestionnaire autonomique de liaison à intercepter cette fourniture doit être ajoutée.

Comme nous l'avons vu, une fourniture de service peut être interceptée de deux différentes manières :

- En affinant le service fourni, c'est-à-dire en l'enrichissant avec des propriétés de service supplémentaires, ou plus précises (cf. figure 5.20).
- En enrobant le service fourni par le composant dans un service mandataire (figure 5.21). Ce mandataire permet de gérer certains aspects non-fonctionnels du service de manière totalement transparente à la fois pour le composant

```java
/**
 * Exemple de gestionnaire autonomique de liaison réordonnant les services.
 */
@Component
public class Manager2 implements ServiceRankingInterceptor {

    /**
     * Comparateur utilisé pour trier les services selon leur puissance consommée.
     * Le service est "meilleur" s'il consomme moins d'énergie.
     * Les services ne renseignant pas leur consommation sont considérés
     * très consommateurs.
     */
    private static Comparator<ServiceReference> COMPARATOR = (s1, s2) -> {
        // Le service est "meilleur" s'il consomme moins d'énergie.
        Double c1 = (Double) s1.getProperty("device.power");
        if (c1 == null) {
            c1 = Double.POSITIVE_INFINITY;
        }
        Double c2 = (Double) s2.getProperty("device.power");
        if (c2 == null) {
            c2 = Double.POSITIVE_INFINITY;
        }
       return Double.compare(c1, c2);
    };

    @Override
    public List<ServiceReference> getServiceReferences(
                                        DependencyModel dependency,
                                        List<ServiceReference> matching) {
        // Trie la liste de services sélectionnés.
        Collections.sort(matching, COMPARATOR);
        return matching;
    }

    @Override
    public List<ServiceReference> onServiceArrival(
                                        DependencyModel dependency,
                                        List<ServiceReference> matching,
                                        ServiceReference<?> reference) {
        // Insère le nouveau service dans la liste, puis retrie la liste.
        matching.add(reference);
        Collections.sort(matching, COMPARATOR);
        return matching;
    }

    @Override
    public List<ServiceReference> onServiceModified(
                                        DependencyModel dependency,
                                        List<ServiceReference> matching,
                                        ServiceReference<?> reference) {
        // Un service de la liste a été modifié. La liste doit être retriée.
        Collections.sort(matching, COMPARATOR);
        return matching;
    }

    @Override
    public List<ServiceReference> onServiceDeparture(
                                        DependencyModel dependency,
                                        List<ServiceReference> matching,
                                        ServiceReference<?> reference) {
        // Retire le service partant de la liste. La liste n'a pas besoin
        // d'être retriée.
        matching.remove(reference);
        return matching;
    }

}
```

FIGURE 6.9 – Exemple de réordonnancement de services

```java
/**
 * Exemple de gestionnaire autonomique de liaison interceptant
 * les liaisons de services.
 */
@Component
public class Manager3 implements ServiceBindingInterceptor<BinaryLight> {

    /**
     * Dépendance sur un service journalisation.
     */
    @Requires
    Logger logger;

    /**
     * Classe ajoutant un aspect de journalisation à un dispositif BinaryLight.
     */
    class BinaryLightWithLogger implements BinaryLight {

        /**
         * Le dispositif original.
         */
        BinaryLight delegate;

        public BinaryLightWithLogger(BinaryLight service) {
            this.delegate = service;
        }

        @Override
        public void turnOn() {
            delegate.turnOn();
            logger.info(delegate.getSerialNumber() + " has been turned on.");
        }

        @Override
        public void turnOff() {
            delegate.turnOff();
            logger.info(delegate.getSerialNumber() + " has been turned off.");
        }

        // ...

    }

    @Override
    public BinaryLight bind(DependencyModel dependency,
                            ServiceReference<BinaryLight> reference,
                            BinaryLight original) {
        logger.info(original.getSerialNumber()
                + " is used by component "
                + dependency.getComponentInstance().getInstanceName());

        // Remplace le service BinaryLight original par un objet
        // gérant la journalisation.
        return new BinaryLightWithLogger(original);
    }

    @Override
    public void unbind(DependencyModel dependency,
                       ServiceReference<BinaryLight> reference,
                       BinaryLight original,
                       BinaryLight transformed) {
        logger.info(original.getSerialNumber()
                + " is no longer used by component "
                + dependency.getComponentInstance().getInstanceName());
    }

}
```

FIGURE 6.10 – Exemple d'interception de liaisons de services

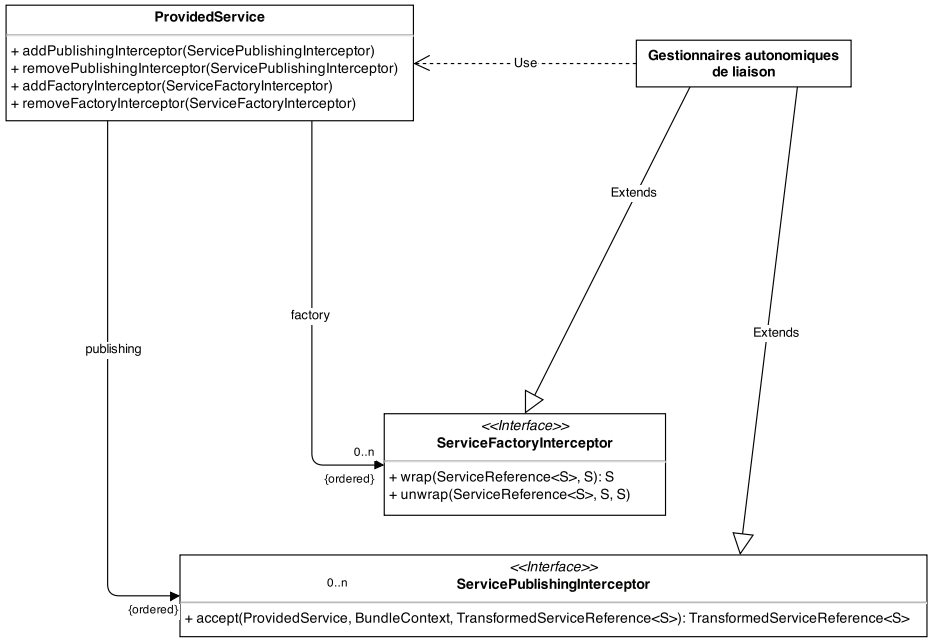

FIGURE 6.11 – Diagramme de classes de `ProvidedService`

fournissant le service mais aussi pour tous les composants utilisant ce service.

Un composant iPOJO peut fournir zéro, un ou plusieurs services. Dans le cas où il fournit un ou plusieurs services, le *provider handler* de ce composant fournit un objet de type `ProvidedService` pour chacun des services fournis. Ce modèle gère les relations entre le composant et le registre de service *OSGi* où le service est publié. Afin d'implanter l'interception des fournitures de services dans iPOJO, nous avons dû modifier la classe `ProvidedService` en lui ajoutant le concept d'intercepteur. C'est donc à partir de cette interface qu'un gestionnaire autonomique va pouvoir manipuler les deux chaînes d'interception (affinage et enrobage) en ajoutant, retirant ou réordonnant les intercepteurs.

Le diagramme de classe de la figure 6.11 montre les relations entre le *provider handler* et le ou les gestionnaires autonomiques de liaison et les différents intercepteurs impliqués dans la gestion de la fourniture de service. Le principe général est assez symétrique à celui mis en œuvre pour les dépendances de service :

- Á la manière des `ServiceTrackingInterceptor`, les `ServicePublishingInterceptor` permettent d'enrichir le service à fournir avec un ensemble de propriétés, en utilisant d'ailleurs la même interface, la `TransformedServiceReference`.
- De la même façon qu'un `ServiceBindingInterceptor` permet de modifier l'objet qui sera réellement utilisé par une dépendance de service, l'intercepteur de permettant l'enrobage du service fourni, le `ServiceFactoryInterceptor`, permet de changer le service qui sera réellement publié, en ajoutant par exemple des aspects non-fonctionnels.

Les figures suivantes montrent comment un gestionnaire autonomique de liaison va pouvoir utiliser les mécanismes mis en œuvre pour intercepter une fourniture de service. La figure 6.12 montre un exemple très simple de composant fournissant un service de type `SimulatedBinaryLight`. C'est sur ce composant que les exemples d'interception de fourniture de service se baseront.

```
1  /**
2   * Simple composant avec une fourniture de service.
3   */
4  @Component
5  @Provides(  // Fourniture de service :
6          specifications = {
7                  // Interfaces fournies
8                  SimulatedBinaryLight.class, BinaryLight.class
9          },
10         properties = {
11                 // Propriété statique de service
12                 @StaticServiceProperty(
13                         name = "binaryLight.maxPowerLevel",
14                         type = "double", value = "100.0"
15                 )
16         }
17 )
18 public class SimpleProvider implements SimulatedBinaryLight {
19
20     /**
21      * Propriété de service.
22      * La valeur de cette propriété est liée à la valeur de l'attribut.
23      */
24     @ServiceProperty(name = "binaryLight.powerStatus")
25     private boolean status;
26
27     // ... suite de la logique métier du composant...
28
29 }
```

FIGURE 6.12 – Exemple de composant avec une fourniture de service simple

La figure 6.13 est un exemple de gestionnaire autonomique de liaison qui enrichit le service fourni par le composant de la figure 6.12. En effet, ce gestionnaire a connaissance de la consommation énergétique requise pour certains dispositifs, et il peut donc ajouter cette propriété au service fourni avant qu'il ne soit publié dans le registre de services. Comme dans le cas d'un intercepteur de registre de service (figure 6.8, le service peut être étendu avec de nouvelles propriétés (ligne 36), laissé intact (ligne 23) ou même supprimé (ligne 31). Dans ce dernier cas, le gestionnaire empêche le service d'être publié ; il ne sera donc pas visible par les autres composants de la plateforme. Plus généralement, le principe de l'affinage de services fournis est symétrique au principe de l'interception du registre de service (pour une dépendance), à la différence près que les modifications apportées au service seront publiées dans le registre, et donc visibles par tous les consommateurs de ce service.

Dans la figure 6.14, le gestionnaire enrobe le service fourni par le composant de la figure 6.12. Le service de type `BinaryLight`, tel qu'il est fourni par le composant, est remplacé par un *wrapper*[4] qui ajoute des appels à un service de journalisation. C'est ce *wrapper* qui est publié dans le registre de service *OSGi*, et donc utilisé par les composants qui consomment ce service. Le principe d'enrobage de service est très similaire à celui de l'interception des liaisons de services dans une dépendance (`ServiceBindingInterceptor`, figure 6.14). L'objet fourni par le gestionnaire autonomique de liaison est ainsi utilisé de manière totalement transparente par tous les consommateurs de ce service, en lieu et place du service fourni par le composant `SimpleProvider`.

4. Littéralement « emballage » : nom technique donné à un objet qui en enveloppe un autre, dans le but de l'adapter (*Adapter Pattern*).

```java
/**
 * Exemple de gestionnaire autonomique de liaison affinant un service fourni.
 */
@Component
public class Manager4 implements ServicePublishingInterceptor<BinaryLight> {

    /**
     * Puissance électrique consommée pour chaque dispositif.
     */
    private Map<String, Double> wattage;

    /**
     * Méthode interceptant les services avant publication dans le registre.
     */
    @Override
    public TransformedServiceReference<BinaryLight> accept(
                                        ProvidedService providing,
                                        BundleContext bundleContext,
                                        TransformedServiceReference<BinaryLight> ref) {
        if (ref.contains("device.power")) {
            // Le service renseigne déjà sa puissance consommée..
            // Il est publié sans modification.
            return ref;
        }

        String id = (String) ref.get("device.id");
        if (id == null || !wattage.containsKey(id)) {

            // Pas d'information de consommation sur ce dispositif.
            // Ce service est éliminé ; il ne sera pas publié, et donc invisible.
            return null;

        } else {

            // Ajout de l'information de consommation sur le service.
            ref.addProperty("device.power", wattage.get(id));
            return ref;

        }
    }

}
```

FIGURE 6.13 – Exemple d'affinage de service fourni

```java
/**
 * Exemple de gestionnaire autonomique de liaison enrobant un service fourni.
 */
@Component
public class Manager5 implements ServiceFactoryInterceptor<BinaryLight> {

    /**
     * Dépendance sur un service journalisation.
     */
    @Requires
    Logger logger;

    /**
     * Classe ajoutant un aspect de journalisation à un dispositif BinaryLight.
     */
    class BinaryLightWithLogger implements BinaryLight {

        /**
         * Le service original.
         */
        BinaryLight delegate;

        public BinaryLightWithLogger(BinaryLight service) {
            this.delegate = service;
        }

        @Override
        public void turnOn() {
            delegate.turnOn();
            logger.info(delegate.getSerialNumber() + " has been turned on.");
        }

        @Override
        public void turnOff() {
            delegate.turnOff();
            logger.info(delegate.getSerialNumber() + " has been turned off.");
        }

        // ...

    }

    @Override
    public BinaryLight wrap(ServiceReference<BinaryLight> reference,
                            BinaryLight original) {
        logger.info("Start logging " + original.getSerialNumber());
        // Remplace le service BinaryLight original par un objet gérant la journalisation.
        return new BinaryLightWithLogger(original);
    }

    @Override
    public void unwrap(ServiceReference<BinaryLight> reference,
                       BinaryLight original,
                       BinaryLight wrapped) {
        logger.info("Stop logging " + original.getSerialNumber());
    }

}
```

FIGURE 6.14 – Exemple d'enrobage de service fourni

Amélioration proposée	Statut actuel
Interception des dépendances de services	**Intégré dans la version actuelle**
Interception des fournitures de services	**En attente d'intégration** Intégration avant la prochaine version majeure
Interception du code métier des composants	**Non rétrocompatible** Intégration dans la prochaine version majeure

TABLE 6.1 – État d'intégration des améliorations proposées

Bilan

La couche d'exécution iPOJO est déjà utilisée par un grand nombre de logiciels existants, et il est donc délicat de proposer directement ces améliorations sans altérer le comportement de ces logiciels ou d'en dégrader les performances. L'état des implantations des solutions proposées pour améliorer iPOJO est donc actuellement assez partagé :

- Toutes les améliorations proposées ont été implantées.
- Certaines de ces améliorations ont été intégrées dans la version *'mainline'* d'iPOJO (actuellement en version 1.12.0), et ont été publiées dans les dernières versions stables [5]. Ces versions sont actuellement utilisées dans des environnements de production, et les améliorations proposées dans cette thèse commencent petit à petit à être adoptées.
- D'autres améliorations plus expérimentales sont disponibles dans des branches non-officielles (*'fork'*) d'iPOJO, et attendent d'être intégrées [6]. Elles nécessitent d'être revues avant de pouvoir être intégrées dans le code d'iPOJO, car elles touchent à certains points sensibles, ou bien impliquent des compromis en terme de performance. Ces améliorations seront intégrées très prochainement, probablement avant la prochaine version majeure d'iPOJO.
- Parmi les améliorations du point précédent, certaines impliquent des changements plus profonds, et il devient extrêmement difficile de conserver une compatibilité descendante avec les versions précédentes d'iPOJO (v 1.x). Ces améliorations restent donc confinées dans des branches de développement séparées, jusqu'à la publication de la prochaine version majeure d'iPOJO (v 2.0). Le changement de version majeure permet, selon les conventions habituellement rencontrées, de s'affranchir du problème de rétrocompatibilité.

Le tableau 6.1 montre, pour chacune des améliorations proposées, l'état actuel de son intégration au sein du projet iPOJO. Seul le nouveau modèle d'interception du code métier ne peut pas être intégré dans la version actuelle d'iPOJO. En effet, la phase de manipulation du code à la compilation interdit toute rétrocompatibilité avec les composants existants.

À long terme, toutes les modifications proposées ont vocation à être intégrées dans le framework iPOJO, et donc exploitables par ses nombreux utilisateurs. La prochaine version majeure d'iPOJO a en effet été planifiée pour la fin de l'année 2014.

5. La version officielle d'iPOJO peut être trouvée à l'adresse suivante : `http://felix.apache.org/downloads.cgi`

6. Les modifications d'iPOJO non-intégrées dans la version officielle sont disponibles dans le *fork* à l'adresse suivante : `https://github.com/bourretp/felix`.

FIGURE 6.15 – Everest au sein de l'écosystème OW2 Chameleon

6.1.2 Implantation du modèle de représentation du système

L'implantation du modèle de représentation du système a été réalisée dans le cadre d'un nouveau projet logiciel, baptisé Everest. Il appartient au projet de plus grande envergure, OW2 Chameleon [7], dont le but est de fournir un écosystème de qualité permettant la conception et l'exécution d'applications dynamiques basées sur les *framework OSGi* et iPOJO. Ainsi, OW2 Chameleon fournit une plateforme *OSGi* incluant iPOJO et un ensemble de services techniques (persistance, planification de tâches, envoi de mail, de messages instantanés, gestion d'utilisateurs, *etc.*).

Le modèle de représentation du système a été implanté sous la forme d'un module annexe : ce modèle n'est pas inclus dans iPOJO, et iPOJO n'a pas besoin de ce modèle pour fonctionner. La figure 6.15 montre la relation entre la couche d'exécution *OSGi* + iPOJO, inclue dans OW2 Chameleon, et notre modèle de représentation du système, Everest.

Les caractéristiques générales qui ont guidé l'implantation d'Everest ont été préalablement détaillé dans la section 4.3.3. Notre approche consiste à fournir une représentation uniforme du système dans son ensemble sous la forme d'un ensemble de ressources et de relations entre elles. Les principes du style d'architecture *REST* ont été la base pour notre modèle de ressource. Les opérations *Create*, *Read*, *Update* et *Delete* permettent de manipuler ces ressources ainsi que leurs relations. Il est aussi possible d'observer une ressource, ou un ensemble de ressource afin de réagir lors d'évènements les concernant. Les ressources sont adaptables, c'est-à-dire qu'elles peuvent être traduites en des objets de plus bas niveau dont elles sont la représentation. Des extensions permettent d'enrichir le modèle en transformant certaines ressources (transformation dite globale), ou en en ajoutant de nouvelles. Un gestionnaire autonomique peut aussi choisir d'adapter la vision qu'il a de la représentation du système en effectuant des transformations locales.

Comme énoncé dans la section 4.3.3, dans Everest, la représentation du système

7. `http://chameleon.ow2.org`

est un graphe de ressources. Ce graphe est en fait la superposition de deux structures complémentaires :

- Une structure d'arbre représentant une organisation logique des ressources. Les nœuds de l'arbre sont des ressources, qu'il est possible d'identifier grâce à un chemin. Pour chaque ressource, il est possible d'accéder à sa ressource parente et à l'ensemble de ses ressources filles. Cette structure logique permet d'organiser les ressources. Les arbres sont des structures de données très utilisées, et facilement manipulables (par exemple les systèmes de fichiers, les *URLs*, *etc.*). Elle ne permet cependant pas de représenter la richesse sémantique des relations entre les ressources.
- Une structure de graphe orienté permettant de représenter les liens existants entre les ressources. Les ressources sont donc reliées entre elles par des relations unidirectionnelles. Chaque relation est nommée et contient une description sémantique, interprétable par un programme et compréhensible par un humain.

Une ressource contient une description de son état actuel, sous la forme d'un ensemble clé-valeur. Tel que décrit dans le chapitre précédent, les cinq types d'opérations sont supportés par les ressources : *CREATE*, *READ*, *UPDATE*, *DELETE* et *OBSERVE*. Une opération supplémentaire dite d'adaptation permet de traduire une ressource en un objet de plus bas niveau qu'elle représente. Everest, de base, fournit le modèle de ressources et de relations, mais ne fournit aucune représentation du système : c'est le rôle des domaines d'Everest.

Everest est implanté de manière extensible : le cœur fonctionnel ne fournit que le modèle de ressource. S'il est possible de créer et de manipuler des ressources à partir de ce modèle, elles ne sont pas liées au système à représenter. Les extensions du modèle d'Everest sont appelées des domaines. Chaque domaine est spécifique à un type d'objet du système à représenter. Un domaine a la capacité, en effectuant des transformations globales, de :

- Ajouter, modifier, supprimer des ressources, afin de refléter l'état courant des objets du système représenté.
- Étendre des ressources existantes, issues d'autres domaines, afin d'ajouter des informations ou des relations supplémentaires.

La figure 6.16 représente le diagramme de classes d'Everest, où apparaissent les notions de ressources, de relations entre ressources et les domaines d'extension. Le point d'accès initial est la classe `EverestService`, qui représente la ressource racine dans l'organisation logique, et permet d'effectuer des requêtes. Cette ressource peut être étendue par des `ResourceExtender`s. Lorsque l'`EverestService` exécute une requête, il va interroger les `ResourceExtender`s afin de leur permettre de modifier la ressource visée par la requête. Les domaines d'extension d'Everest utilisent ce mécanisme afin d'effectuer des transformations globales sur les ressources.

Chaque ressource a une ressource parente (sauf la ressource racine) et zéro, une ou plusieurs ressources filles. Ceci permet d'organiser logiquement les ressources, et de les identifier par leur chemin (`Path`). Les relations entre les ressources sont représentées par la classe `Relation`, qui contient le chemin vers la ressource ciblée et des informations permettant de parcourir la relation : une opération, des paramètres et une description.

L'opération *OBERVE* ne fait pas partie des actions traitées par `EverestService`. Ce dernier publie des événements relatifs aux ressources modifiées. Everest utilise le service `EventAdmin`, qui est le bus d'événements standardisé de la plateforme OSGi.

La figure 6.17 montre l'architecture globale d'Everest. Le service `EverestService`

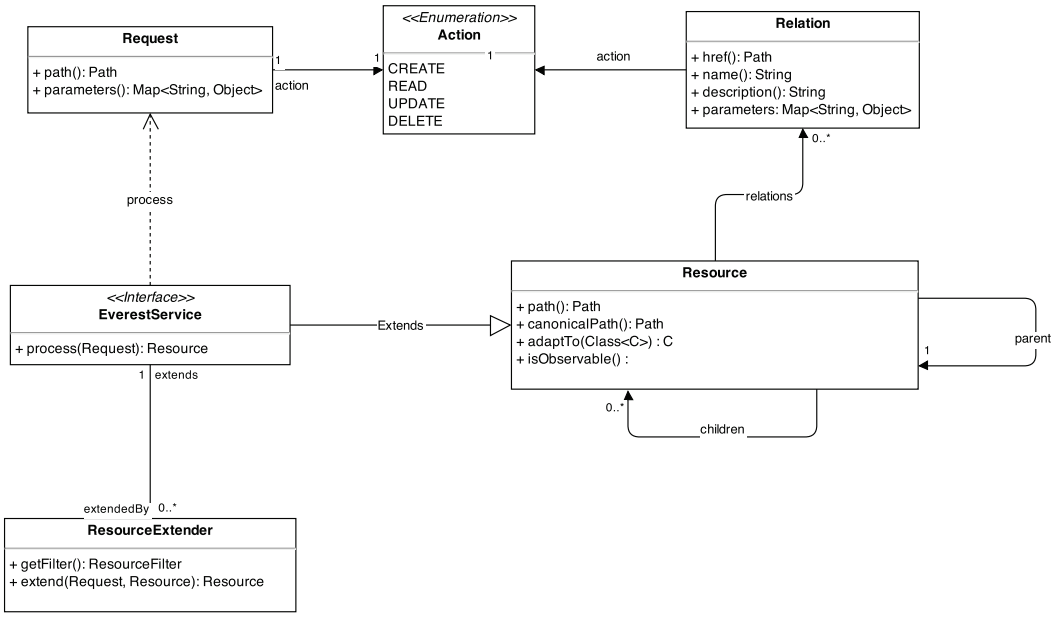

FIGURE 6.16 – Diagramme de classes d'Everest

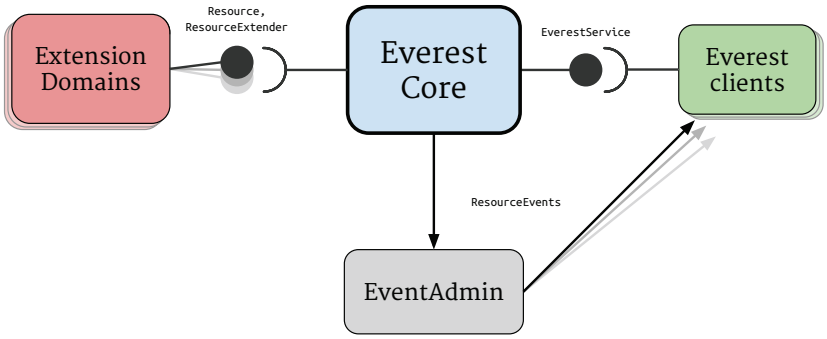

FIGURE 6.17 – Architecture globale d'Everest

est publié et est utilisable par les clients d'Everest, par exemple des gestionnaires autonomiques souhaitant consulter le modèle de représentation de la plateforme. Everest se lie à tous les services de type `Resource` et `ResourceExtender` afin d'enrichir le graphe des ressources et de transformer les ressources retournées par les requêtes qu'il traite. Enfin, les événements relatifs aux ressources ajoutées, modifiées ou retirées sont publiés par le biais de l'`EventAdmin`, auquel les clients d'Everest peuvent souscrire pour observer les changements du modèle.

Dans le cadre de notre proposition, la représentation du système doit contenir les informations pertinentes pour que les gestionnaires autonomiques puissent connaître l'état du système et le modifier. Le cœur d'Everest ne suffit donc pas : il faut ajouter les informations génériques concernant le système. Nous avons donc implanté quatre domaines qui viennent enrichir le modèle de représentation du système avec de nombreuses ressources :

- **everest-system** représente l'état de la plateforme matérielle et du système d'exploitation : mémoire, CPU, processus, threads, JVM, *etc.*
- **everest-fs** représente l'ensemble des systèmes de fichiers et leur arborescence.
- **everest-osgi** donne une vue des ressources de la plateforme *OSGi* : modules, *packages*, services, configurations, *logs*, *etc.*
- **everest-ipojo** est une représentation des entités de la plateforme iPOJO : types de composants, instances de composants, services utilisés et fournis, *etc.* Ce domaine étend activement les ressources *OSGi*. Par exemple une ressource représentant un service aura des relations supplémentaires vers les composants qui l'utilisent.

Enfin, *REST* étant un style d'architecture très lié au Web, nous avons voulu implanter un client d'Everest permettant de manipuler les ressources à partir d'un navigateur. Ce module interprète donc des requêtes *HTTP* provenant du navigateur pour les transformer en requêtes Everest. Les données, ressources et relations, sont représentées à l'aide d'objets *JSON*[8], mais il est possible d'ajouter d'autres types de représentations possibles[9]. Les évènements de modification de ressources peuvent être reçus par le client en utilisant les *WebSockets*.

La figure 6.18 montre un exemple d'utilisation du module Everest *HTTP*. Cette requête demande une lecture (*READ*) d'une ressource représentant une instance de composant iPOJO. La réponse de cette requête contient la représentation de cette ressource en *JSON*.

6.2 Validation de la proposition

La problématique générale de cette thèse consiste à simplifier la conception d'applications ubiquitaires. Ces applications, comme nous l'avons vu, nécessitent de pouvoir s'adapter aux changements inévitables de leur environnement d'exécution. L'approche de l'informatique autonomique permet une claire séparation entre la logique métier de l'application et sa logique d'adaptation. Notre proposition repose sur :

- une partie fonctionnelle à base de composants orientés services ;
- une couche d'exécution *Autonomic-Ready*, fournissant des points d'adaptations nécessaires à la gestion efficace de ces composants ;

8. *JavaScript Object Notation* : format de données textuelles, générique, dérivé de la notation des objets du langage JavaScript.

9. L'entête *HTTP Accept* permet de sélectionner le type souhaité pour la représentation des ressources et relations, par exemple `application/json`, ou bien `application/xml`.

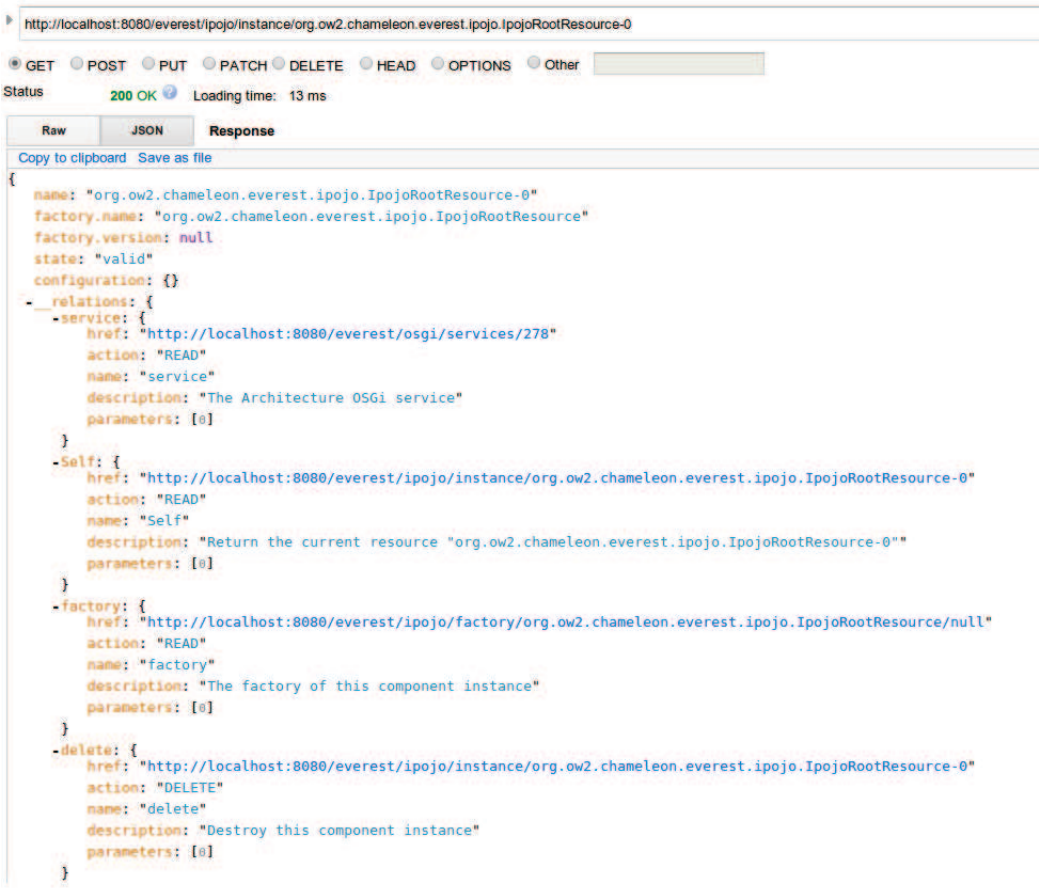

FIGURE 6.18 – Exemple de requête utilisant Everest avec *HTTP*

- des gestionnaires autonomiques locaux, qui réalisent des opérations d'adaptation de bas niveau, en s'appuyant sur les points d'adaptations de la couche d'exécution,
- une représentation globale du système, qui fournit un accès uniforme aux diverses informations concernant l'environnement d'exécution de l'application. Ce modèle permet aux gestionnaires autonomiques de collaborer en enrichissant cette représentation avec les informations dont ils disposent, et de partager ces informations ;
- un gestionnaire autonomique globale, qui a une vue d'ensemble sur le système, et, à partir de sa connaissance et de la représentation du système, coordonne les gestionnaires autonomiques locaux. Ce gestionnaire traduit les intentions de l'administrateur, ses buts de haut-niveau, en adaptations sur le modèle de l'application et sur les gestionnaires de plus bas niveau.

La validation de cette proposition repose sur deux évaluations. La première est qualitative : nous devons être en mesure de montrer qu'une application réalisée selon notre approche est de meilleure qualité qu'une application réalisée selon une approche « classique ». Dans notre cas, l'approche classique consiste à développer une application en utilisant les plateformes *OSGi* et iPOJO, mais sans utiliser les améliorations d'iPOJO ni le modèle de représentation du système, qui sont proposé par cette thèse. La deuxième évaluation est quantitative : les entités introduites par notre approche induisent, par indirection, un certain nombre de détérioration des performances de l'application. Nous mesurerons donc précisément ces détériorations afin de déterminer le coût global de notre approche.

Cette validation s'appuie sur un environnement logiciel dédié nommé iCASA [Escoffier 2014]. Cette environnement été conçu, parallèlement aux propositions de cette thèse, afin de fournir un atelier de conception et d'expérimentation d'applications ubiquitaires, plus particulièrement dans le domaine très fertile de la domotique. Après avoir brièvement présenté l'environnement iCASA qui servira de base technique à cette validation, les évaluations qualitatives et quantitatives de notre approche seront menées.

6.2.1 Présentation de l'environnement de validation : iCASA

Parallèlement aux travaux effectués durant cette thèse, et présentés dans les chapitres précédents, nous avons participé à la mise en place d'un projet relatif à l'informatique ubiquitaire : iCASA [10]. Ce projet fournit un environnement de développement et d'exécution d'applications ubiquitaires spécialisé dans le domaine de la domotique. iCASA se décompose en deux parties majeures : une plateforme d'exécution ainsi qu'un environnement de simulation domotique.

La plateforme d'exécution iCASA est un support pour le déploiement et l'exécution d'applications domotiques et fournit à ce titre :

- Des services techniques, utiles aux applications, comme par exemple un service de planification de tâches, d'enregistrement des préférences des utilisateurs, de persistance des données applicatives, ou de découverte et de réifications de dispositifs physiques sous forme de services.
- Un modèle de développement d'applications, à base de composants orientés service. En pratique, les frameworks *OSGi* et iPOJO sont les bases techniques

10. http://adeleresearchgroup.github.io/iCasa/

FIGURE 6.19 – Interface Web d'environnement simulé d'iCASA

de la plateforme iCASA, et ceci va nous permettre de directement utiliser notre proposition.
- Des outils d'analyse et d'introspection, permettant d'encadrer l'exécution d'applications et de détecter leurs éventuelles défaillances.
- Une interface Web de visualisation et d'administration de la plateforme ainsi que des applications et des services qui la compose.

L'environnement de simulation permet de tester une application en lui mimant un environnement d'exécution proche de la réalité. Pour cela, iCASA fournit :

- Un environnement domotique virtuel, représentant une maison ou un appartement. Cet environnement inclut un moteur physique permettant de mesurer certaines caractéristiques de l'environnement (luminosité, température, bruit, *etc.*). Ce monde virtuel permet aussi de représenter les habitants, leurs actions, leurs déplacements, *etc.*
- Une grande variété de dispositifs simulés qui peuvent être placés et déplacés dans l'environnement virtuel décrit ci-dessus. Ces dispositifs peuvent influer directement sur l'environnement virtuel en changeant ses caractéristiques physiques. Par exemple une lampe simulée qui est allumée va augmenter la luminosité de la pièce simulée dans laquelle elle est placée.
- Une interface Web de simulation, qui représente de façon graphique l'environnement simulé, les dispositifs qui y sont présents et les personnes qui y évoluent. Cette interface permet d'interagir directement avec l'environnement en ajoutant de nouveaux dispositifs simulés, en déplaçant des utilisateurs, en actionnant certains dispositifs, *etc.*

La figure 6.19 montre un exemple d'environnement simulé tel qu'il est représenté par cette interface. C'est dans cet environnement que va être réalisée l'application ubiquitaire *LightFollowMe*, qui va servir de base d'évaluation pour la validation de notre approche.

Afin de réaliser les évaluations qualitatives et quantitatives nécessaires à l'évaluation de notre approche, nous avons utilisé deux versions différentes de la plateforme iCASA :

- La version iCASA de base, qui contient l'ensemble des éléments décrits ci-dessus, et inclut la plateforme iPOJO de base, c'est-à-dire sans les améliorations proposées dans cette thèse.
- La version iCASA étendue, qui, en plus des éléments fournis par iCASA de base, inclut la version étendue d'iPOJO. Les améliorations d'iPOJO proposées dans cette thèse y sont dont inclues. Cette version d'iCASA contient également Everest, notre modèle de représentation uniforme.

6.2.2 Validation qualitative du modèle de développement

La validation qualitative de la proposition de cette thèse consiste à montrer que les applications ubiquitaires sont plus faciles à concevoir et à faire évoluer si l'on utilise le modèle de développement décrit dans cette thèse (chapitre 4). Ce type d'application est, comme nous l'avons montré, complexe à concevoir. L'évolution permanente de l'environnement dans lequel elles s'exécutent les oblige à prendre en compte les contraintes de dynamisme à l'exécution et d'évolution. La facilité d'adaptation et de maintenance de ces applications est donc un critère important, voire déterminant.

Les critères retenus pour évaluer notre approche sont issus du génie logiciel. Parmi la large étendue de critère de qualité existants, une grande part ne sont pas concernés par le domaine d'application de notre proposition, et n'illustrent donc pas les avantages et inconvénients de notre approche.

Après avoir défini les critères d'évaluation qualitative qui seront utilisés, l'application *LightFollowMe* qui servira de référence à cette évaluation sera présentée. Ensuite, cette application sera mise en œuvre en utilisant l'une et l'autre de deux versions d'iCASA. Enfin, les deux versions de ces applications seront comparées en utilisant les critères définis.

Critères qualitatifs d'évaluation de l'approche

Le but de notre proposition est de fournir une approche permettant de réduire la complexité de conception, d'administration et d'évolution des applications ubiquitaires. Afin d'évaluer de manière la plus précise possible les apports de cette approche, nous devons définir des critères qui vont permettre de la comparer avec un approche plus classique, qui servira de référence. Ces critères d'évaluation se répartissent en deux catégories, qui correspondent aux deux phases du cycle de vie du logiciel concernées par notre approche.

Phase de développement

1. **Couplage** : ce critère évalue le mélange des préoccupations. Dans le cadre de cette thèse, il s'agit plus précisément du couplage entre la logique fonctionnelle d'une application et sa gestion des aspects autonomiques.
2. **Nombre de composants** : le nombre de composants formant une application est un critère permettant d'estimer sa complexité architecturale. Un nombre de composants plus élevé indique une architecture plus complexe. Le nombre de liaisons est aussi un autre indicateur de la complexité architecturale. Cependant,

vu le nombre relativement faible de composants considérés dans notre exemple, le nombre de liaisons est considéré comme directement lié à ce nombre de composants.

3. **Nombre moyen de lignes de code par composant** : le nombre de ligne de code permet, pour chaque composant, d'estimer sa complexité. Plus un composant contient de lignes de code, plus difficile est sa conception, sa mise au point et son évolution.

La séparation d'une application en composants est une méthode permettant de réduire son couplage (critère 1). Des composants trop complexes (critère 3) seront très difficiles à entretenir et à faire évoluer. Un trop grand nombre de composants, même si il favorise le découplage, peut être difficile à comprendre, car il faut appréhender toutes les interactions entre ceux-ci. En pratique un compromis peut être trouvé entre les critères 2 et 3, afin de réaliser une application ayant une architecture suffisamment simple, avec des composants de petites tailles.

Phase de développement

4. **Type d'adaptations** : les adaptations vont permettre à un administrateur de changer l'application, afin de prendre en compte une modification des politiques de gestion par exemple. Nous distinguons ici deux types possibles d'adaptations. Les **adaptations spécifiques** demandent à l'administrateur d'interagir avec le code de l'application (composant, service, *etc.*). Les **adaptations génériques** laissent l'administrateur effectuer des modifications de manière générique, sans interagir directement avec le code applicatif. Les adaptations génériques permettent de réduire considérablement la complexité d'administration, car les actions de l'administrateur restent très semblables quelle que soit l'application ciblée.
5. **Facilité d'évolution** : ce critère permet de juger l'aptitude d'une application à accepter des modifications, des évolutions ou à intégrer de nouveaux aspects.
6. **Mise à jour à la volée** : la mise à jour à la volée est la capacité de modifier une application en cours d'exécution sans devoir l'arrêter. Ce critère est particulièrement important dans l'optique de réduire les interruptions de service.

Présentation de l'application de référence : *LightFollowMe*

L'application ubiquitaire qui va servir à valider les critères précédents se nomme *LightFollowMe*. Cette application a un rôle très simple : elle doit détecter la présence et les mouvements de l'utilisateur dans sa maison afin d'allumer et d'éteindre les lumières de différentes pièces, le long de son parcours. La figure 6.20 montre un exemple d'appartement muni de capteurs de présence et de lampes. Les lampes sont allumées dans la pièce où se situe l'utilisateur, et éteinte dans les autres pièces.

La première mise en œuvre de l'application *LightFollowMe*, qui va servir de référence, sera réalisée selon une approche dite classique, c'est-à-dire sans l'utilisation des propositions de cette thèse. Elle est basée sur les *frameworks OSGi* et iPOJO, mais nous limiterons à n'utiliser que les fonctionnalités d'iPOJO dans sa version de base.

La deuxième mise en œuvre de l'application *LightFollowMe* va être réalisée selon les principes proposés dans cette thèse. Plus particulièrement, l'architecture sera

FIGURE 6.20 – Application *LightFollowMe* exécutée dans la plateforme iCASA

conforme à l'architecture de référence proposée dans la section 4.3.7, et les gestionnaires autonomiques vont utiliser les deux mécanismes de la plateforme d'exécution : les intercepteurs de composants disponibles dans iPOJO étendu et le modèle de représentation du système fourni par Everest.

Dans les deux sections qui vont suivre, nous allons faire la distinction entre la logique fonctionnelle de l'application et sa gestion autonomique :

- La fonctionnalité de l'application est l'allumage et l'extinction des lampes.
- La gestion autonomique consiste à déterminer la localisation de l'utilisateur, et à sélectionner les lampes à actionner selon cette localisation.

Les annexes A et B de ce manuscrit contiennent l'intégralité du code de mise en œuvre de ces deux implantations.

Première mise en œuvre : implantation de référence

La figure 6.21 montre l'architecture utilisée pour l'implantation de référence de l'application *LightFollowMe*. Celle-ci ne contient qu'un seul composant, qui fournir un service de type `LightFollowMe`, et utilise tous les services de type `PresenceSensor` et `BinaryLight`.

Le composant, à partir de l'ensemble des détecteurs de présence (`presenceSensors`) détermine la localisation courante de l'utilisateur (`currentLocation`). Lorsque cette localisation change, c'est-à-dire lorsqu'une présence est détectée par un capteur (`devicePropertyModified`), le composant se charge de sélectionner les lampes de la zone concernée (`getAllLightsAt`) pour les allumer ou les éteindre.

Le service exposé, `LightFollowMe`, permet de modifier les informations de localisation des dispositifs. C'est cette interface que l'administrateur doit utiliser pour adapter

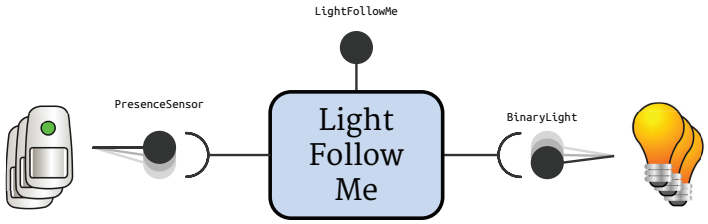

FIGURE 6.21 – Architecture de l'implantation de référence de l'application *LightFollowMe*

Implantation de référence	
Critère	**Évaluation**
Couplage	Fort
Nombre de composants	1
Lignes de code par composant	Élevé
Type d'adaptation	Spécifique
Facilité d'évolution	Difficile
Mise à jour à la volée	Très difficile

TABLE 6.2 – Évaluation de l'implantation de référence

l'application lorsque, par exemple, un dispositif (détecteur de présence ou lampe) est ajouté, ou déplacé.

Le tableau 6.2 montre l'évaluation, pour chacun des critères définis dans la section 6.2.2, de cette implantation de référence. L'application est fortement couplée, puisque le même composant comporte des parties fonctionnelles et des parties autonomiques. Le composant peut être considéré comme complexe du fait de son nombre important de lignes de code. L'architecture est simple, car ne comportant qu'un seul composant. Les adaptations sont spécifiques : l'administrateur doit utiliser l'interface d'administration spécifique `LightFollowMe` pour changer le contexte de l'application. L'application est difficile à faire évoluer : pour cela il faut modifier le code de l'unique composant, qui est comme nous l'avons vu complexe. Ce composant unique rend quasiment impossible la mise à jour à la volée : le composant doit être arrêté, mis à jour et redémarré.

Deuxième mise en œuvre : iCASA étendu

La deuxième mise en œuvre de l'application *LightFollowMe* utlise l'architecture de référence proposée, telle qu'elle est décrite dans la section 4.3.7. Cette application est constituée d'un composant métier, d'un gestionnaire autonomique de liaison et d'un gestionnaire autonomique global.

Le composant métier est très simple : il utilise tous les services de type `BinaryLight`. Lorsqu'un tel service apparaît, le composant allume la lampe, et lorsqu'il disparaît, le composant l'éteint. Le gestionnaire autonomique de liaison a pour rôle de filtrer les services visibles par le composant : seules les lampes localisées dans la même pièce que l'utilisateur seront visibles par le composant métier. Cette information de localisation

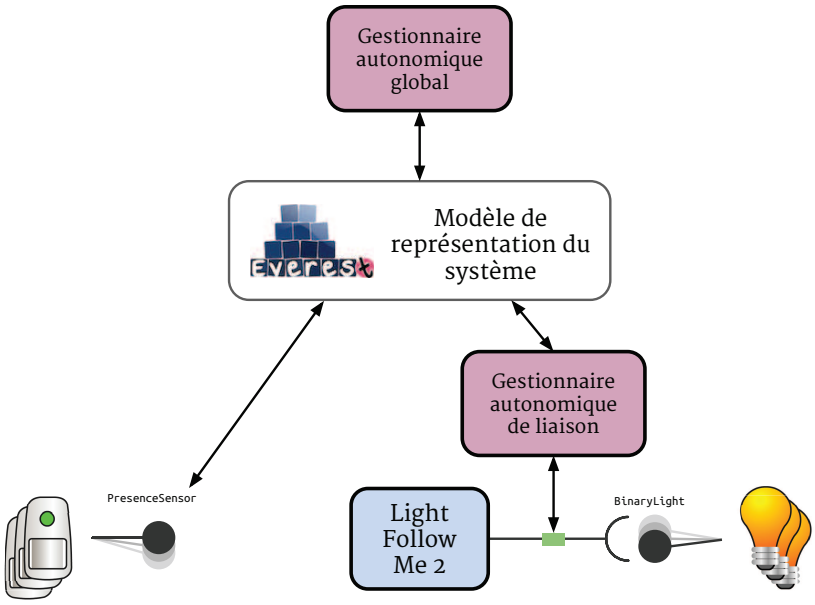

FIGURE 6.22 – Architecture de l'implantation étendue de l'application *LightFollowMe*

n'est pas calculée par le gestionnaire local, mais est directement issue du modèle de représentation du système.

Ce modèle représente l'ensemble des services de la plateforme, en particulier les détecteurs de présence. Le gestionnaire autonomique global, en utilisant ces informations sur les détecteurs de présence, va déterminer la localisation de l'utilisateur, et inscrire cette information dans le modèle. Le gestionnaire autonomique de liaison observe cette information de localisation et modifie la liste des services visibles par le composant métier, en simulant des départs et arrivées de services.

Sur le plan architectural, cette implantation est sans aucun doute plus complexe que l'implantation de référence, puisqu'elle compte trois composants. Cependant chacun des composants est beaucoup plus simple : le nombre de lignes de code par composant est considérablement réduit. Le couplage de l'application est aussi très faible, car la gestion autonomique est réalisée par des composants dédiés ; le composant métier ne contient que la partie fonctionnelle de l'application (allumage et extinction de lampes). Les composants, bien que plus nombreux, sont donc plus faciles à entretenir, car plus simples et moins couplés.

Pour adapter l'application, l'administrateur utilise le modèle de représentation. C'est à partir de ce modèle qu'il peut changer la localisation des dispositifs par exemple. Les adaptations sont donc rendues plus génériques. Si l'administrateur doit adapter d'autres applications conçus selon la même approche, il n'aura pas à changer ses méthodes d'interventions : il utilisera le même modèle de représentation.

Du fait du découpage en plusieurs composants, et de leur découplage, l'application est plus facile à faire évoluer. Il est par exemple possible d'ajouter ou de changer un gestionnaire autonomique sans devoir intervenir sur les autres composants. Il est de plus possible de la mettre à jour sans l'arrêter, si les changements concernent les gestionnaires autonomiques. Un gestionnaire autonomique supplémentaire pourrait ainsi être ajouté afin de prendre en compte des informations provenant d'autres types de capteurs (capteur de mouvement, d'ouverture de portes, *etc.*). Ce nouveau

Implantation étendue	
Critère	**Évaluation**
Couplage	Faible
Nombre de composants	3
Lignes de code par composant	Faible
Type d'adaptation	Générique
Facilité d'évolution	Facile
Mise à jour à la volée	Possible

TABLE 6.3 – Évaluation de l'implantation étendue

gestionnaire peut être ajouté dynamiquement, lors de l'exécution de l'application, sans pour autant nécessiter d'interruption de service.

Le tableau 6.3 récapitule les évaluations de l'implantation étendue en fonction des critères établis dans la section 6.2.2.

6.2.3 Validation quantitative de la couche d'exécution

La validation quantitative va consister à mesurer les performances de la couche d'exécution et de l'application *LightFollowMe*. Ces mesures vont servir à déterminer quel est le coût des propositions de cette thèse, et de leurs implantations. Les applications ubiquitaires s'exécutant bien souvent dans des dispositifs embarqués, aux ressources matérielles limitées, il est primordial que les solutions proposées aient des performances acceptables.

Les mesures de performance sont réalisées à partir des deux applications *LightFollowMe* conçues dans les sections précédentes, afin de comparer l'approche proposée dans cette thèse à l'approche classique. Nous avons donc procédé à ces mesures dans les deux situations suivantes :

1. Plateforme iCASA standard et application *LightFollowMe* classique. La plateforme contient une version d'iPOJO n'incluant aucune des améliorations proposées et implantées dans le cadre de cette thèse. Le modèle de représentation Everest n'est pas compris dans cette version de la plateforme d'exécution.
2. Plateforme iCASA étendue et application *LightFollowMe* étendue. Cette plateforme contient iPOJO étendu par nos propositions et le modèle de représentation Everest.

Ces deux plateformes ont été comparées selon plusieurs critères de performance classiques :

1. La consommation mémoire : mesure de la moyenne de la quantité de mémoire consommée par la *JVM* sur une durée fixée de l'exécution de l'application.
2. La consommation du processeur : pics de d'activité enregistrés lors d'une réaction à un changement du contexte, en l'occurrence un changement de la localisation de l'utilisateur.
3. Le temps de réaction : mesure du délai entre changement de situation du contexte (changement de localisation de l'utilisateur) et l'application des adaptations relatives à ce changement (allumage des lampes correspondantes).

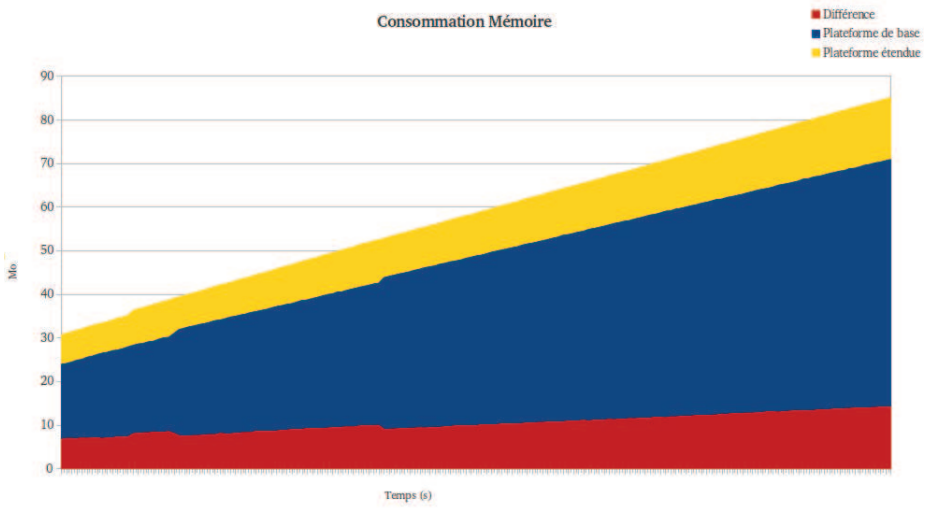

FIGURE 6.23 – Mesures de consommation mémoire instantanée

La figure 6.23 montre les relevés de consommation en mémoire lors des exécutions des deux différentes versions de l'application *LightFollowMe* sur les deux versions des plateformes d'exécution iCASA. Ces mesures ont été effectuées durant 10 minutes, en phase de repos des applications, sans aucun événement de reconfiguration. Au cours du temps, les consommations mémoire relevées augmentent. Ce phénomène est normal, car la plateforme Java ne permet pas de récupérer explicitement la mémoire inutilisée [11] : le ramasse-miette [12], intégré dans la machine d'exécution, se charge de repérer quels sont les objets inutilisés et de les détruire afin de recycler la mémoire qu'ils occupaient. Donc lorsque la consommation mémoire dépasse un certain seuil (paramétrable), le ramasse-miette entre en action et supprime les objets inutilisés. Ce seuil n'a pas été dépassé lors des mesures effectuées, et la récupération de mémoire par le ramasse-miette n'est donc pas visible sur la figure.

On peut observer que la plateforme étendue, et l'application *LightFollowMe* étendue, consomment plus de mémoire que leurs versions de bases. La différence de consommation (partie rouge du graphique) progresse de façon linéaire par rapport à la mémoire consommée par la plateforme et l'application de base. Cette surconsommation est d'environ 22 %(soit environ 10 Mo), ce qui est considérable. Cependant cette surconsommation a une progression linéaire, quelque soit la consommation mémoire de la plateforme de base. Il n'y a donc pas d'explosion de la consommation induite par les modifications proposées, qui reste, à un facteur près, proportionnelle à la consommation mémoire de la plateforme iCASA de base et de l'application *LightFollowMe* développée avec une approche classique.

Le facteur de surconsommation mémoire mesuré expérimentalement pour l'application *LightFollowMe* est substantiel. Nous avons donc cherché à en déterminer l'origine. Pour cela, nous avons effectué plusieurs mesures des consommations mémoires de différentes configurations de plateformes iCASA, comme le montre la figure 6.24. Ces mesures ont été effectuées en faisant varier le nombre de composants iPOJO présents sur la plateforme, qui est un indicateur direct de la complexité globale du système. Pour chaque situation, la consommation mémoire moyenne a été mesurée sur une

11. Le langage C par exemple, plus précisément la bibliothèque standard C, permet d'allouer (`malloc`) et de libérer (`free`) explicitement de la mémoire.
12. *Garbage Collector* en anglais

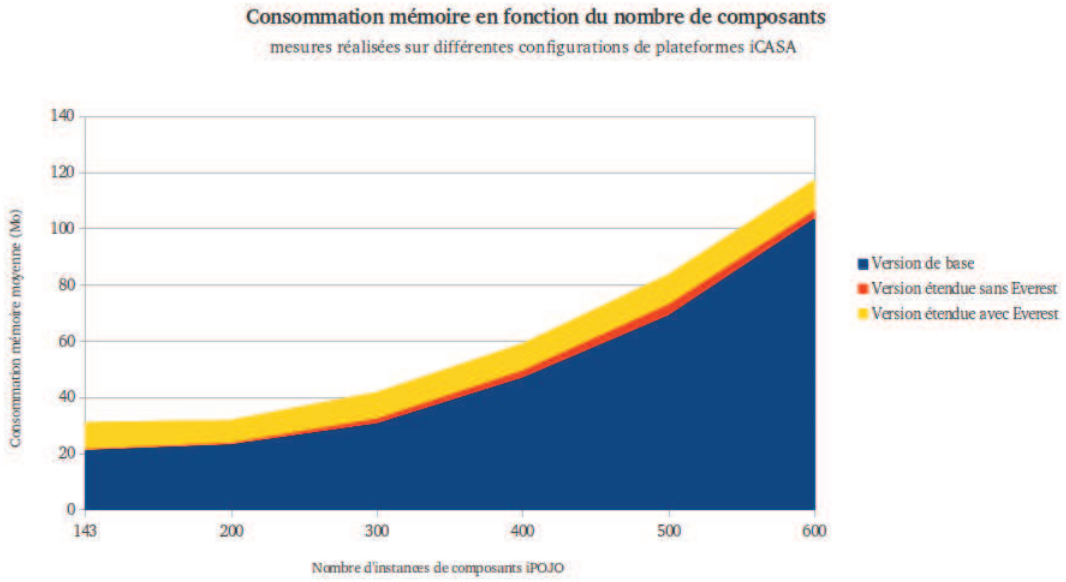

FIGURE 6.24 – Influence de la complexité du système sur la consommation mémoire

période de 10 minutes.

La première constatation que l'on peut tirer de ces mesures est que la grande majorité de la surconsommation mémoire est due au modèle de représentation du système Everest. Les extensions d'iPOJO ajoutées dans le cadre de cette thèse ont un coût très faible en terme de consommation mémoire. De plus, on observe que le surcoût moyen induit par Everest, amorti sur une période de 10 minutes, est relativement constant (environ 10 Mo). La complexité du système à représenter n'a donc qu'un très faible impact sur la consommation mémoire moyenne d'Everest.

Ces mesures peuvent sembler contredire celles effectuées dans la figure 6.23, où le coût en mémoire de notre approche avait été jugé proportionnel à la consommation mémoire du système de base, n'incluant pas ces contributions. Il faut cependant noter que cette tendance a été déduite à partir d'une application très simple. Les applications *LightFollowMe* proposées ne contiennent tout au plus qu'une poignée de composants, et ne contribue donc pas à augmenter significativement la complexité du système.

Le surcoût quasi-constant déterminé à partir de la figure 6.24 tend à montrer que le modèle Everest, bien qu'ayant un impact non-négligeable, peut supporter aisément une montée en complexité du système qu'il doit représenter. Le système représenté est donc très léger par rapport au système réel. Cependant l'ajout de composants supplémentaires et d'une couche de représentation intermédiaire peut avoir d'autres effets sur les performances globales du système et des applications, notamment en consommation du processeur. Les propositions implantées n'apportent vraisemblablement pas de surcharge de calcul lorsqu'une application est en phase dite de repos. C'est plutôt pendant les phases d'adaptation qu'il faut mesurer les impacts de notre approche, en terme de surconsommation processeur, et en délai supplémentaire de réaction aux changements.

La figure 6.25 montre comment les deux versions des plateformes iCASA et des applications *LightFollowMe* réagissent en terme de consommation de processeur face à un changement du contexte d'exécution. Le changement de contexte concerne en l'occurrence un mouvement de l'utilisateur d'une pièce à une autre. On peut constater

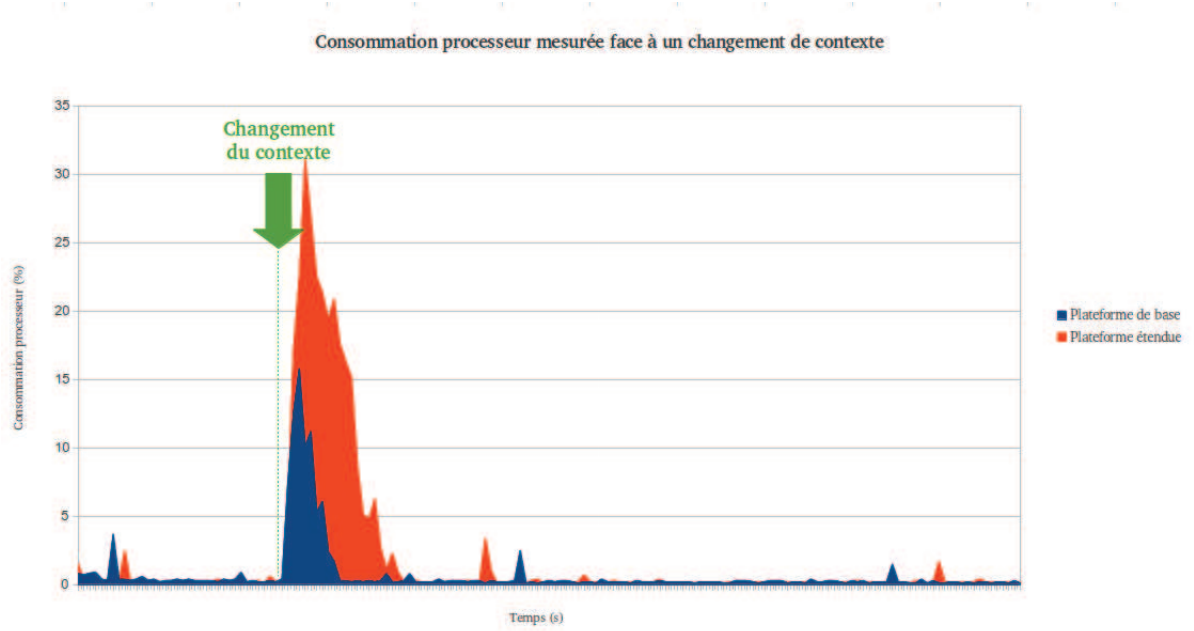

FIGURE 6.25 – Mesures de la consommation processeur face à un changement de contexte

que la plateforme étendue ne consomme apparemment pas plus de processeur que la plateforme de base en phase de repos. Après un changement de contexte, le pic d'activité de la plateforme étendue est toutefois plus haut (d'un facteur 2 environ) et dure plus longtemps.

Le fait de décomposer une application en plusieurs composants et de devoir mettre à jour le modèle Everest lors d'un changement implique donc une surconsommation. Cette surconsommation n'est cependant visible que lorsque survient un changement de contexte, c'est-à-dire quand l'application ou la plateforme doit être reconfigurée. On peut donc considérer que les adaptations étant des phénomènes occasionnels, voire rares, dans la vie d'une application, le surcoût de consommation processeur engendré par notre approche est acceptable.

La figure 6.26 montre un autre aspect relatif aux performances de notre approche : le temps de réaction aux changements de contexte. Pour cela, nous avons mesuré, pour les deux versions des applications *LightFollowMe*, le temps séparant un changement de contexte à l'application de la réaction appropriée. Les changements de contexte considérés sont le mouvement de l'utilisateur et le changement de localisation des dispositifs (détecteurs de présence et lampes). La réaction est considérée achevée dès que toutes les lampes se situant dans la même pièce que l'utilisateur sont allumées.

Le temps moyen de réaction de la plateforme étendue est, sans surprise, plus élevé (d'un facteur d'environ 2,5) que celui de la plateforme de base. Ce temps de réaction supplémentaire est causé par une propagation des évènements d'observation de l'état du système et de reconfiguration de celui-ci à travers un plus grand nombre de composant. L'architecture de l'application étant plus complexe, les interactions entre composants sont multipliés, et les délais de réaction allongés. L'ordre de grandeur des délais supplémentaires occasionnés est néanmoins comparable aux temps de réaction de la plateforme de base. Les évènements d'adaptations étant occasionnels, cette perte de réactivité est donc à peine perceptible.

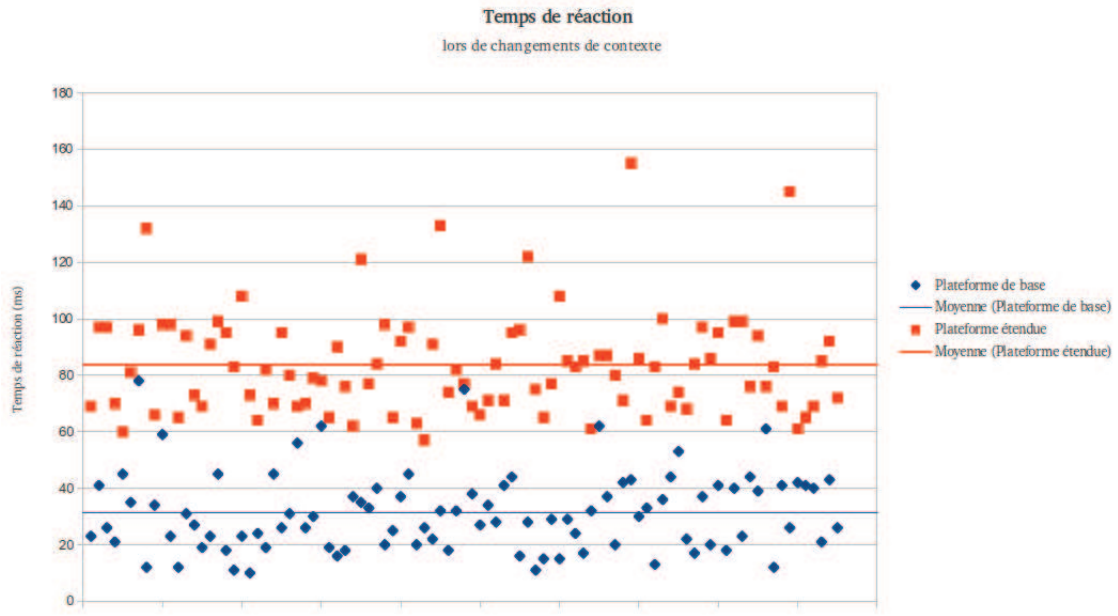

FIGURE 6.26 – Quelques temps de réaction constatés lors de changements de contexte

6.3 Utilisation des solutions proposées

Les modifications apportées à la plateforme iPOJO permettent une gestion beaucoup plus précise des composants. Il devient en effet possible de modifier leur comportement pendant leur exécution, sans toucher au code métier. Ces possibilités ont apporté de nouveaux cas d'utilisation. Ainsi, les mécanismes d'interception des services ont pu servir à l'élaboration d'un gestionnaire autonomique particulier prenant en charge la qualité de service. Les services utilisés sont triés selon les propriétés non-fonctionnelles qui les décrivent. L'approche retenue par ce gestionnaire est l'utilisation du réordonnancement des services (`ServiceRankingInterceptor`). L'ordre est déterminé par l'application de l'analyse de concepts formels [13] [Chollet 2014].

D'autre part, les dernières versions de la plateforme iCASA utilisent les mécanismes d'interception au registre de services (`ServiceTrackingInterceptor`) pour ajouter des propriétés de services contextuelles, comme la localisation des dispositifs. Ces informations proviennent du modèle de représentation du système Everest qui est en cours d'intégration dans la plateforme (domaine d'extension *everest-icasa*). L'interception des liaisons aux services (`ServiceBindingInterceptor`) est aussi intéressante pour la gestion des droits d'accès aux dispositifs. Les composants dépendant de dispositifs utilisent alors un *wrapper* qui détermine si l'utilisation (appel de méthode) est autorisée par ce composant ou non. Cette gestion des droits d'accès est effectuée de manière totalement transparente pour les utilisateurs des services. En pratique, les droits d'accès sont déduits du composant qui utilise le dispositif, mais surtout de l'utilisateur qui a initié l'accès à ce dispositif. C'est ici une illustration de la fusion de sources de contexte de natures différentes (cf. figure 2.2) qui détermine un comportement d'adaptation.

Le modèle de représentation Everest est utilisé dans un domaine assez différent,

13. *FCA : Formal Concept Analysis*

relatif au déploiement de plateformes et d'applications dans des environnements dynamiques. Le modèle de la plateforme est utilisé pour décrire son état courant, et un domaine d'extension spécifique permet d'y effectuer des opérations transactionnelles de déploiement de modules, de services ou d'applications et d'effectuer des reconfigurations. Les domaines d'extension *everest-osgi* et *everest-ipojo* permettent de décrire les relations entre les modules, composants et services de la plateforme et de calculer un graphe permettant de déployer une application avec l'ensemble de ses dépendances.

6.4 Conclusion

Au vu des résultats obtenus lors des différentes évaluations, tant qualitatives que quantitatives, l'approche proposée par cette thèse a été validée par les constats suivants :

- Cette approche permet de réduire considérablement le mélange des préoccupations au sein de l'application. Le *framework* iPOJO favorise déjà la séparation entre code métier et la gestion d'aspects non-fonctionnels (dépendances de services, dynamisme, *etc.*). Notre approche va plus loin en séparant clairement la logique fonctionnelle de l'application et sa partie autonomique. Le code métier est significativement simplifié ; sa conception et sa maintenance s'en trouvent d'autant simplifiées.
- La gestion des aspects autonomiques de l'application est plus structurée. La gestion autonomique est décomposée selon le niveau d'abstraction utilisé. Cette décomposition permet d'avoir des gestionnaires de plus petite taille, et donc plus faciles à concevoir et faire évoluer.
- Le système est plus lisible. Á partir d'un point d'accès universel, Everest, il est possible d'obtenir des informations diverses sur chacune des parties composant une application, et des relations entre elles. Cette représentation facilite à la fois la prise en charge de l'application par un gestionnaire autonomique mais aussi sa surveillance et sa maintenance par un administrateur humain.
- L'application est plus facile à faire évoluer. Son découpage métier/autonomique permet de changer les politiques de gestion autonomique sans induire d'arrêt de la partie fonctionnelle. De même, il est possible de changer une partie de la gestion autonomique sans arrêter le reste de l'application. Les temps d'arrêt de l'application sont réduits, et sa disponibilité améliorée.
- Les performances de l'application sont dégradées par l'introduction des améliorations d'iPOJO proposées. Cependant, les performances globales de l'application sont très peu affectées. Les surcoûts concernent principalement les adaptations de l'application, par exemple lors de la sélection d'un nouveau service. Ces surcoûts sont relativement faibles, voire même négligeables si l'on considère les adaptations comme étant des évènements rares.
- Le maintien d'un modèle de représentation globale du système représente un coût en mémoire et en temps *CPU* non-négligeable. En effet, chaque entité du système doit être surveillée, chaque changement doit être analysé pour être reflété par le modèle. Nous considérons cependant que les avantages, en termes de génie logiciel surpassent les inconvénients.
- Les approches définies dans cette thèse ont été réutilisées par ailleurs. Les travaux concernés, dans le domaine de la recherche, confirment l'intérêt notre approche et l'utilisabilité de son implantation.

Chapitre 7

Conclusion et perspectives

Dans les chapitres précédents, nous avons présenté notre proposition : une architecture de référence pour le développement d'applications ubiquitaires, une modification du modèle à composants orientés services ainsi qu'un modèle de représentation du système. Ces propositions ont été implantées : les améliorations proposées ont été intégrées (ou sont en cours d'intégration) dans le projet Apache Felix iPOJO, et le projet OW2 Chameleon Everest permet de représenter un système à l'exécution de manière extensible. Ces briques logicielles commencent à être réutilisées dans d'autres projets internes, mais aussi à l'extérieur.

Dans ce chapitre nous faisons une synthèse du contexte et des propositions effectuées, et analysons ensuite quels sont les axes de recherches qui sont soulevés par ce travail.

Sommaire

7.1 Contexte

Dans le chapitre 2, nous avons décrit l'émergence de l'informatique ubiquitaire. Cette nouvelle vision des environnements informatiques met l'utilisateur au centre de l'attention des systèmes, qui doivent s'effacer, se fondre dans le monde environnant. Les défis soulevés par cette vision sont nombreux, et couvrent un vaste ensemble de disciplines variées, telles la micro-électronique, les communications sans-fils, les interactions homme-machine, *etc.*

Les programmes devant s'exécuter dans des environnements ubiquitaires sont soumis à de très fortes contraintes, et doivent traiter des aspects non-fonctionnels de plus en plus nombreux : distribution, hétérogénéité, interaction avec des systèmes tiers, *etc.* La gestion de ces aspects n'est pas nouvelle, et de nombreux outils et *middleware* permettent déjà de les intégrer. Il y a cependant deux aspects cruciaux qui conditionnent les caractéristiques des environnements ubiquitaires, et des logiciels qui s'y exécutent :

- Le **dynamisme** : l'environnement ubiquitaire est vivant : il bouge, change, évolue en fonction des utilisateurs qui l'utilisent. Il doit aussi être en mesure de gérer des défaillances, comme l'usure des dispositifs, les pertes de connectivités, les manques d'énergie, les pannes logicielles, *etc.* Une application ubiquitaire doit donc pouvoir détecter l'ensemble de ces situations et pouvoir agir en conséquence. Pour cela, elle doit être **sensible au contexte** d'exécution, et être **adaptable** à ses changements.
- L'**autonomie** : afin de s'intégrer de manière transparente dans un environnement ubiquitaire, une application doit être relativement autonome.
  - D'une part l'utilisateur ne doit pas avoir conscience qu'il utilise un environnement informatique. Les applications doivent donc induire les intentions de l'utilisateur en fonction de ses actions, et déduire les actions concrètent à réaliser en fonction de ses intentions. C'est au système de s'adapter à l'utilisation, et non l'inverse.
  - D'autre part la large diffusion des environnements ubiquitaires pose un problème pour leur administration. Il n'est plus possible de mettre un administrateur humain en permanence derrière chaque système ou application. Afin de réduire le coût d'exploitation et d'administration et de maintenance de ces systèmes, ces derniers doivent être autonomes, c'est-à-dire être en mesure de réagir de manière adaptée aux divers dysfonctionnements de l'environnement augmenté. Cette autonomie ne supprime pas le besoin d'administration humaine, mais réduit grandement cette dépendance.

La gestion du dynamisme est très complexe, et demande la plus grande attention dés la phase de conception des logiciels. Afin de réduire la complexité de ceux-ci, il existe des *middleware*, comme iPOJO, permettant de gérer de façon transparente les aspects très techniques de cohérence et de synchronisation des applications. Nous remarquons cependant que cette gestion du dynamisme au niveau applicatif est encore peu répandue, et que nombre d'approches existantes dédiées à l'informatique ubiquitaire reposent sur des bases très statiques, générées par avance au développement, et immuables à l'exécution.

La gestion de l'autonomie des systèmes est un sujet vaste, transversal, qui n'est pas spécifique à l'informatique ubiquitaire. *IBM* a ainsi, dès 2001, proposé l'approche de l'informatique autonomique afin de résoudre le problème de maintenance de larges systèmes, de plus en plus répandus. Cette approche repose sur une décomposition du

système en deux parties : le système à gérer, qui contient la partie fonctionnelle, et le gestionnaire autonomique, qui surveille la partie fonctionnelle, son environnement d'exécution, et adapte le système en fonction du contexte et de ses changements.

L'approche de l'informatique autonomique permet en théorie la couverture de nombreux problèmes liés à l'administration des systèmes en général, et des logiciels en particulier. Parmi les points particulièrement importants, on retrouve notamment l'évolution des systèmes, la complexité de leur intégration, les interruptions de services, l'optimisation du fonctionnement des systèmes, et les inévitables erreurs humaines liées à l'administration de systèmes de plus en plus complexes.

La boucle autonomique permet d'interpréter des buts de haut niveau, définis par l'administrateur, et de les transformer en adaptations spécifiques sur le système géré. Le gestionnaire autonomique est lui aussi découpé en plusieurs entités, chacune ayant spécifique : c'est la boucle *MAPE-K* (section 3.2.4).

Les interactions entre le système à gérer et le gestionnaire autonomique sont limités par le niveau de réflexivité de la plateforme d'exécution. Il est donc très difficile de gérer de manière autonomique un système ne permettant pas de communiquer son état courant, ou de reconfigurer cet état à l'exécution.

En résumé, l'approche de l'informatique autonomique offre des perspectives très intéressantes pour la gestion transparente des nombreuses contraintes imposées par la vision de l'informatique ubiquitaire. Nous remarquons cependant que cette gestion ne peut se faire sans une plateforme d'exécution suffisamment réflexive, qui fournit des *touchpoints* permettant de connaître l'état d'une application et de la reconfigurer.

7.2 Contributions

En réponse aux problématiques introduites par la vision de l'informatique ubiquitaire, et en application des principes généraux de l'informatique autonomique, nous avons proposé dans cette thèse un ensemble de contributions. Celles-ci s'articulent autour de deux axes majeurs :

- proposer un modèle architectural pour les applications ubiquitaires, tiré des principes de l'informatique autonomique.
- rendre la plateforme d'exécution plus réflexive, en y ajoutant des procédés permettant de connaître précisément son état, des scruter les changements de cet état et de modifier le comportement
- représenter le système à l'exécution, ce qui permet aux gestionnaires autonomiques de raisonner non plus sur le système lui-même, ce qui nécessite une gestion de son hétérogénéité, et de ses aspects techniques, mais sur une représentation simplifiée, uniforme et extensible.

7.2.1 Architecture de référence

Dans le chapitre 4, nous avons proposé une architecture de référence pour la conception d'applications ubiquitaires. Cette architecture, conformément à la vision de l'informatique autonomique, découpe l'application en une partie fonctionnelle et une partie de gestion autonomique.

- La partie fonctionnelle s'appuie sur un modèle à composants orientés services. Ce découpage permet de diviser la complexité des composants de l'application. L'approche orientée services réduit le couplage entre composants, qui ne dépendent plus directement les uns des autres, mais plutôt d'une description abstraite de

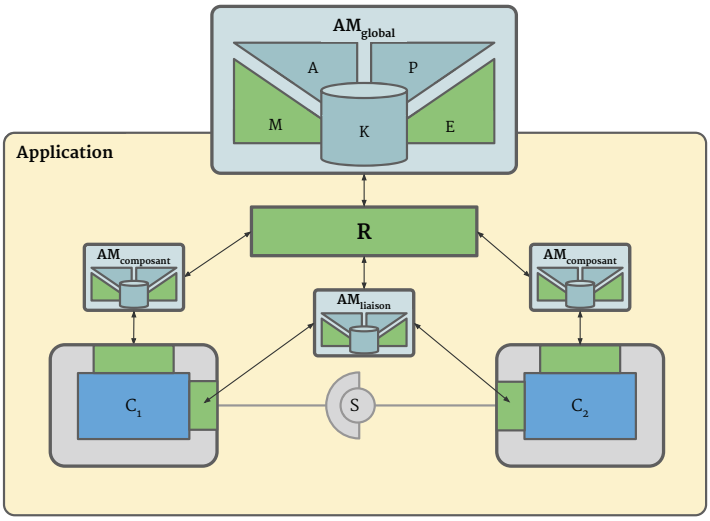

FIGURE 7.1 – Rappel de l'architecture de référence proposée

leurs fonctionnalités. Il est possible de modifier l'architecture de l'application à l'exécution, par exemple en ajoutant ou retirant des services. Cette architecture à géométrie variable est la base du principe d'adaptabilité au contexte, géré par la couche de gestion autonomique.

- La partie autonomique est elle aussi décomposée en plusieurs gestionnaires, qui gravitent autour d'un élément central : le modèle de représentation du système.
  - Des gestionnaires locaux vont alimenter cette représentation en remontant l'état des composants et des services de l'application ; ils ont aussi la charge de modifier le comportement de ces composants et services en réponse à des changements de cette représentation. Les gestionnaires locaux peuvent gérer et reconfigurer les composants fonctionnels de l'application mais aussi les liaisons entre ces composants.
  - Au sommet se trouve le gestionnaire autonomique global. Ce dernier a une vision du système dans son ensemble, tel que représenté par le modèle. À partir des données disponibles, et des buts de haut niveau définis par l'administrateur, le gestionnaire global va adapter le système en coordonnant les différents gestionnaires autonomiques locaux.

7.2.2 Une couche d'exécution réflexive

La couche d'exécution d'un système est une pièce maîtresse, dans le sens où elle va permettre ou limiter le pouvoir des applications qui s'y exécutent. À ce titre, dans le cadre de l'informatique ubiquitaire, les environnements sont très dynamiques, et les applications doivent donc être en mesure de se réorganiser face aux nombreux types de changements pouvant survenir. Nous avons choisi d'utiliser la plateforme d'exécution iPOJO, qui permet de spécifier une application sous la forme d'un ensemble de composants liés entre eux par des services. iPOJO permet de décrire les aspects non-fonctionnels des composants à côté du code fonctionnel, et réduit grandement la complexité de gestion de tous les aspects techniques, comme le dynamisme des services, le configuration des composant, la garantie de la cohérence de l'état des composants, *etc.*

La plateforme d'exécution iPOJO est adaptée pour le développement d'applications en général, car elle promeut la séparation des préoccupations, le découplage des composants et leur dynamisme, et favorise, dans une certaine mesure, leur ré-utilisabilité. Il a cependant fallu ajouter des fonctionnalités à cette plateforme afin de fournir les propriétés requises pour sa gestion autonomique, à savoir :

- la **réflexivité des composants** : un gestionnaire autonomique local doit être en mesure de connaître précisément l'état des composants métiers constituant une application. Les modifications apportées permettent non seulement de connaître l'état des composants et de leurs attributs, mais aussi d'intercepter les accès aux composants, et ainsi d'être averti de leurs changements, voire même de modifier leur comportement à l'exécution.
- la **réflexivité des dépendances de services** : un gestionnaire autonomique de liaison doit pouvoir connaître facilement l'ensemble des services utilisés par une dépendance. Il peut aussi avoir à s'interposer, et filtrer, exclure, enrichir et modifier les services utilisés par ce composant, et ce de manière totalement transparente pour le code métier du composant.
- la **reflexivité des services fournis** : un composant fournissant un service peut être étendu par un gestionnaire autonomique de liaison. ce dernier a le pouvoir d'enrichir le service fourni avec des propriétés non-fonctionnelles, comme la qualité de service, mais aussi de fournir un service métier modifié, qui gère d'autres aspects non-fonctionnels non-prévus par le composant originel.

L'ensemble de ces améliorations permettent une intégration beaucoup plus naturelle de gestionnaires autonomiques sur des composants iPOJO. La plateforme d'exécution supportant cette intégration de gestionnaire est qualifiée d'***Autonomic-Ready***.

7.2.3 Un système auto-représenté et auto-descriptif

Les gestionnaires autonomiques doivent d'interfacer avec les différents constituants de la plateforme d'exécution : les composants principalement, mais aussi les services, le système d'exploitation, et d'autres services ou sources de données externes. La prise en compte de toutes ces entités, souvent très hétérogènes, est complexe et très technique.

Afin de simplifier la conception de gestionnaires autonomiques, mais aussi de permettre une lecture de l'état du système par un administrateur, nous avons proposé un modèle de représentation du système, implanté par le projet Everest. Ce modèle permet un accès uniforme aux différentes données du systèmes, représentées sous forme de ressource, et masque don la complexité de gestion de l'hétérogénéité.

Ce modèle de représentation prend en compte le dynamisme des entités représentées, et permet de les surveiller individuellement afin de réagir lorsqu'elles sont modifiées. Conformément à la philosophie *REST*, les ressources représentées sont auto-descriptives, à la fois par un programme et par un humain. Les liens entre les ressources sont représentées manière explicite par des relations elles aussi auto-descriptives.

Les gestionnaires autonomiques peuvent consulter ce modèle de représentation du système afin de consulter son état, mais peuvent aussi l'enrichir de diverse manières :

- en ajoutant, retirant ou modifiant des ressources existantes.
- en fournissant un nouveau domaine de ressources, qui représentent de nouveaux concepts (services, composants, modules, dispositifs, *etc.*)
- en transformant le graphe de ressources existant, soit de façon locale, soit de façon globale et visible par tous les autres gestionnaires.

7.3 Perspectives

Avec notre approche, nous avons montré qu'il est effectivement plus facile d'écrire des applications ubiquitaires autonomiques de meilleur qualité. Les applications ainsi créées sont fortement découplées, et donc plus faciles à concevoir, entretenir et à faire évoluer. Le surcoût occasionné par ces contributions a été jugé globalement satisfaisant, voire quasiment imperceptible dans le cadre du fonctionnement normal de l'application. Les adaptations sont plus faciles à réaliser, car exprimées à plus haut niveau, et abstraites de considérations techniques.

Il y a cependant certains points importants soulevés par ces travaux, notamment dans le cadre de la conception détaillées des gestionnaires autonomiques, et plus généralement dans la conception de systèmes sensibles au contexte, voire « conscients ».

7.3.1 Vers une formalisation de la notion de contexte

La notion de contexte a été abordée de nombreuses fois dans cette thèse, mais nous n'avons pas chercher à le modéliser de façon formelle. Le modèle de représentation du système permet une représentation des trois types de contextes ubiquitaires (environnement, utilisateur, informatique), mais cette représentation n'a pas de structure bien définie, il n'existe pas non plus de schéma sémantique entre les entités représentées.

Si cette représentation délibérément « relâchée » du contexte peut permettre la conception d'applications autonomiques, une définition plus pragmatique de cette notion de contexte ubiquitaire peut être nécessaire. Il peut paraître difficile de modéliser un contexte d'exécution qui est, par nature, très volatil, dynamique et hétérogène. Cependant, cette modélisation est nécessaire si l'on veut fournir des outils de développement efficace permettant de s'appuyer sur le contexte, dès la phase de conception des applications ubiquitaires.

Le modèle de représentation du système proposé, Everest, est actuellement la base de recherches visant à formaliser de manière très précise ce qui constitue le contexte ubiquitaire, et comment l'utiliser dans une application ubiquitaire.

7.3.2 Outils de conception pour gestionnaires autonomiques

L'approche décrite dans cette thèse a proposé un modèle de développement, centré sur une architecture de référence, afin de guider la conception d'applications ubiquitaires. Toutefois, la méthodologie proposée et les techniques associées peuvent paraître n'être que des recommandations, et un concepteur d'application ubiquitaire aura peut être du mal à les appliquer littéralement.

Ce problème est le même pour les environnements d'exécution en général : beaucoup de fonctionnalités existent, mais elles ne sont utilisées que lorsque des outils de développement commencent à les prendre en compte, souvent bien plus tard.

La plateforme iCASA dispose d'un environnement de développement assez complet, permettant de composer à l'aide d'assistants un composant d'application ubiquitaire. Par exemple il est possible de rechercher une interface de service parmi la liste des dispositifs simulés. Cet assistant, intégré dans l'*IDE Eclipse*, a permis l'utilisation d'iCASA par des étudiants, qui ont aussi pu développer leurs propres applications ubiquitaires dans cet environnement. En étendant cet outil de développement afin de favoriser l'utilisation de l'architecture proposée, ces étudiants pourraient être guidées dans la conception de gestionnaires autonomiques structurés pour les applications ubiquitaires qu'ils développent.

7.3.3 Écriture de gestionnaires autonomiques

Tout d'abord, l'approche proposée, si elle favorise la gestion autonomique des application, et simplifie l'intégration de gestionnaires autonomiques, ne résout pas le problème fondamental de la conception concrète de ces gestionnaires. Les parties *Monitoring* et *Execution* des gestionnaires a été grandement simplifiée, et la connaissance *Knownledge* est plus accessible, grâce au modèle de représentation du système. Les parties centrales du gestionnaire (*Analysis* et *Planning*) n'ont pas été abordées ; il s'agit pourtant du cœur de la logique de gestion autonomique.

La conception de gestionnaires autonomiques est un vaste sujet, et des ouvrages entiers y sont consacrés [Murch 2004, Lalanda 2013]. Les techniques utilisées pour l'élaboration de gestionnaires autonomiques évolués dépassent largement le cadre de ce thèse. Nous nous sommes en fait inspiré de l'architecture des systèmes autonomiques afin de proposer une solution pratique, dans le cadre du génie logiciel, à la conception d'applications ubiquitaires. Nous avons passé sous silence la partie la plus pointue et la plus stratégique des gestionnaires autonomiques, en considérant qu'elle était très spécifique au système à gérer.

Il existe cependant une classe de gestionnaires autonomiques qui peut être qualifiée de générique. Ces gestionnaires permettent d'interpréter des buts de haut niveau qui ne dépendent pas de l'application à gérer, comme par exemple « diminuer la consommation mémoire », ou « augmenter la vitesse de fonctionnement ». Si les buts de haut niveau exprimés sont génériques, les actions entreprises par ce type de gestionnaires sont par contre très spécifiques et ciblés : ils dépendent fortement de l'application visée. De tels gestionnaires doivent donc « comprendre » comment fonctionne une application inconnue, à moins que l'application elle-même ne décrive comment effectuer ces adaptations (ce qui est extrêmement rare).

Notre approche, si elle ne résout en rien la complexité de conception de gestionnaires autonomiques, permet d'extraire cette gestion de la partie fonctionnelle des applications. Le problème n'est que reporté en plus haut, mais débarrassé des considérations techniques de bas niveau.

7.3.4 Vers des systèmes plus conscients

Lorsque l'on évoque les principes et buts de l'informatique autonomique, spécialement parmi une audience profane, beaucoup s'imaginent des systèmes « intelligents », qui deviennent indépendants de leur administrateurs et évoluent à la manière de consciences vivantes. Il est certain que les ouvrages avant-gardistes et films de science-fiction montrent de façon de plus en plus réaliste de tels systèmes dotés d'une conscience propre, et ont pu véhiculer une image très romancée et hollywoodienne des recherches en informatique. À tel point qu'on ne se pose parfois plus la question de savoir si cela va arriver, mais quand cela arrivera [1].

Bien loin de cette imagination débordante, les systèmes autonomiques évoqués dans ces travaux sont très éloignés de ce que certains peuvent se figurer. Il y a cependant une (petite) once de vérité dans ce que peuvent penser les non-initiés. Nous avons d'ailleurs fait un parallèle entre les système autonomiques et la psyché humaine (section 3.3.2). Cette allégorie, qui n'avait certainement pas pour but d'alimenter l'imaginaire collectif, montre cependant une certaine similarité de comportement

1. Dans l'actualité récente, une intelligence artificielle russe, du nom d'Eugène Gootsman, aurait passée avec succès le fameux test de Turing.

entre des systèmes informatiques de plus en plus complexes capables de réagir, même de façon limitée, à des situations non-prévues et aux mécanismes encore relativement incompris du raisonnement et de la conscience, bases de la formidable capacité d'adaptation de l'espèce humaine.

Il est sûr que les approches abordées dans cette thèse ne vont apporter aucune réponse à la manière de construire une « machine qui pense ». Nous ne proposons pas de résoudre les problèmes soulevés par la vision de Mark Weiser en utilisant des gestionnaires autonomiques conscients, mais il paraît probable que de tels systèmes, si hypothétiques soient-ils, puissent être en mesure de relever les défis de l'informatique ubiquitaire, et bien plus encore. . .

Annexe A

Implantation de référence de *LightFollowMe*

Cet annexe montre le code complet de l'implémentation de référence de l'application *LightFollowMe*. Cette implantation n'utilise aucune contribution énoncée dans cette thèse. Cette implantation, comme mentionné dans la section 6.2.2, n'est formée que d'un seul composant qui fournit un service de type `LightFollowMe`. C'est cette interface que l'administrateur doit utiliser pour adapter l'application lorsque, par exemple, un dispositif (détecteur de présence ou lampe) est ajouté, ou déplacé.

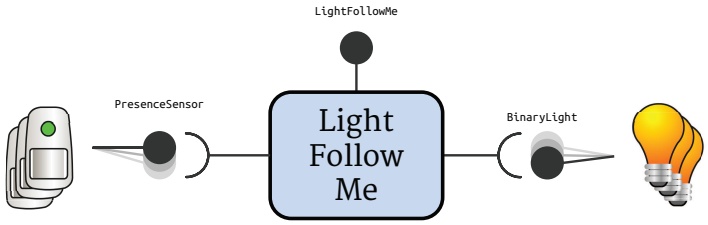

FIGURE A.1 – Architecture de l'implantation de référence de *LightFollowMe* (rappel)

Listing A.1 – Interface `LightFollowMe`

```
1  package fr.liglab.adele.icasa.valid.pierre.refimpl;
2
3  /**
4   * Specific configuration interface of the LightFollowMe application.
5   */
6  public interface LightFollowMe {
7
8      /**
9       * Define the location of the {@code serialNumber} device.
10      * @param serialNumber the serial number of the device.
11      * @param location the new location of the device. May be {@code null}.
12      */
13     void setDeviceLocation(String serialNumber, String location);
14
15 }
```

Listing A.2 – Classe `LightFollowMeRefImpl`

```
1  package fr.liglab.adele.icasa.valid.pierre.refimpl;
2
3  import fr.liglab.adele.icasa.device.DeviceListener;
4  import fr.liglab.adele.icasa.device.light.BinaryLight;
5  import fr.liglab.adele.icasa.device.presence.PresenceSensor;
6  import org.apache.felix.ipojo.annotations.Bind;
7  import org.apache.felix.ipojo.annotations.Component;
8  import org.apache.felix.ipojo.annotations.Instantiate;
9  import org.apache.felix.ipojo.annotations.Unbind;
10
11 import java.util.ArrayList;
12 import java.util.HashMap;
13 import java.util.List;
14 import java.util.Map;
15
16 /**
17  * Reference implementation of the LightFollowMe application.
18  */
19 @Component
20 @Instantiate(name = "LightFollowMe-1")
21 public class LightFollowMeRefImpl implements LightFollowMe, DeviceListener<PresenceSensor> {
22
23     /**
24      * Current location of the user, or {@code null} if unknown.
25      */
26     private String currentLocation;
27
28     /**
29      * Device locations, indexed by device serial number.
30      */
31     private Map<String, String> deviceLocations = new HashMap<>();
32
33     /**
34      * The presence sensors, indexed by device serial number.
35      */
36     private Map<String, PresenceSensor> presenceSensors = new HashMap<>();
37
38     /**
39      * The binary lights, indexed by device serial number.
40      */
41     private Map<String, BinaryLight> binaryLights = new HashMap<>();
42
43     @Override
44     public synchronized void setDeviceLocation(String serialNumber, String location) {
45         String previousLocation = deviceLocations.put(serialNumber, location);
46
47         if (previousLocation != null && previousLocation.equals(currentLocation)) {
48             currentLocation = location;
49             turnOnLightsAt(location);
50             turnOffLightsAt(previousLocation);
51         }
```

```
52      }
53
54
55      private synchronized List<BinaryLight> getAllLightsAt(String location) {
56          List<BinaryLight> result = new ArrayList<>();
57          for (Map.Entry<String, BinaryLight> e : binaryLights.entrySet()) {
58              if (location.equals(deviceLocations.get(e.getKey()))) {
59                  result.add(e.getValue());
60              }
61          }
62          return result;
63      }
64
65      private void turnOnLightsAt(String location) {
66          for (BinaryLight binaryLight : getAllLightsAt(location)) {
67              binaryLight.turnOn();
68          }
69      }
70
71      private void turnOffLightsAt(String location) {
72          for (BinaryLight binaryLight : getAllLightsAt(location)) {
73              binaryLight.turnOff();
74          }
75      }
76
77      // Service @Bind and @Unbind callbacks.
78
79      @Bind(id = "presenceSensors", aggregate = true)
80      private synchronized void bindPresenceSensor(PresenceSensor presenceSensor) {
81          presenceSensors.put(presenceSensor.getSerialNumber(), presenceSensor);
82          presenceSensor.addListener(this);
83
84          String location = deviceLocations.get(presenceSensor.getSerialNumber());
85          if (location != null) {
86              if (presenceSensor.getSensedPresence()) {
87                  currentLocation = location;
88                  turnOnLightsAt(location);
89              } else {
90                  turnOffLightsAt(location);
91              }
92          }
93      }
94
95      @Unbind(id = "presenceSensors", aggregate = true)
96      private synchronized void unbindPresenceSensor(PresenceSensor presenceSensor) {
97          presenceSensor.removeListener(this);
98          presenceSensors.remove(presenceSensor.getSerialNumber());
99      }
100
101     @Bind(id = "binaryLights", aggregate = true)
102     private synchronized void bindBinaryLight(BinaryLight binaryLight) {
103         binaryLights.put(binaryLight.getSerialNumber(), binaryLight);
104
105         if (currentLocation != null) {
106             String location = deviceLocations.get(binaryLight.getSerialNumber());
107             if (currentLocation.equals(location)) {
108                 binaryLight.turnOn();
109             } else {
110                 binaryLight.turnOff();
111             }
112         }
113     }
114
115     @Unbind(id = "binaryLights", aggregate = true)
116     private synchronized void unbindBinaryLight(BinaryLight binaryLight) {
117         binaryLights.remove(binaryLight.getSerialNumber());
118     }
119
120     // Device listener callbacks
121
```

```
122     @Override
123     public void deviceAdded(PresenceSensor device) {
124         // Nothing to do!
125     }
126
127     @Override
128     public void deviceRemoved(PresenceSensor device) {
129         // Nothing to do!
130     }
131
132     @Override
133     public synchronized void devicePropertyModified(PresenceSensor device,
134                                                     String propertyName,
135                                                     Object oldValue,
136                                                     Object newValue) {
137         if (propertyName.equals(PresenceSensor.PRESENCE_SENSOR_SENSED_PRESENCE)) {
138             boolean sensedPresence = (boolean) newValue;
139
140             String location = deviceLocations.get(device.getSerialNumber());
141             if (location != null) {
142                 if (sensedPresence) {
143                     currentLocation = location;
144                     turnOnLightsAt(location);
145                 } else {
146                     turnOffLightsAt(location);
147                 }
148             }
149         }
150     }
151
152     @Override
153     public void devicePropertyAdded(PresenceSensor device, String propertyName) {
154         // Nothing to do!
155     }
156
157     @Override
158     public void devicePropertyRemoved(PresenceSensor device, String propertyName) {
159         // Nothing to do!
160     }
161
162     @Override
163     public void deviceEvent(PresenceSensor device, Object data) {
164         // Nothing to do!
165     }
166
167 }
```

Annexe B

Implantation étendue de *LightFollowMe*

Cet annexe montre le code complet de l'implémentation étendue de l'application *LightFollowMe*. Cette implantation utilise certaines des contributions définies dans cette thèse, dans les chapitres 4 et 5. Cette implantation, détaillée dans la section 6.2.2, est formée d'un composant métier et d'un ensemble de gestionnaires autonomiques. Ces gestionnaires autonomiques se basent sur le modèl de de représentation du contexte Everest.

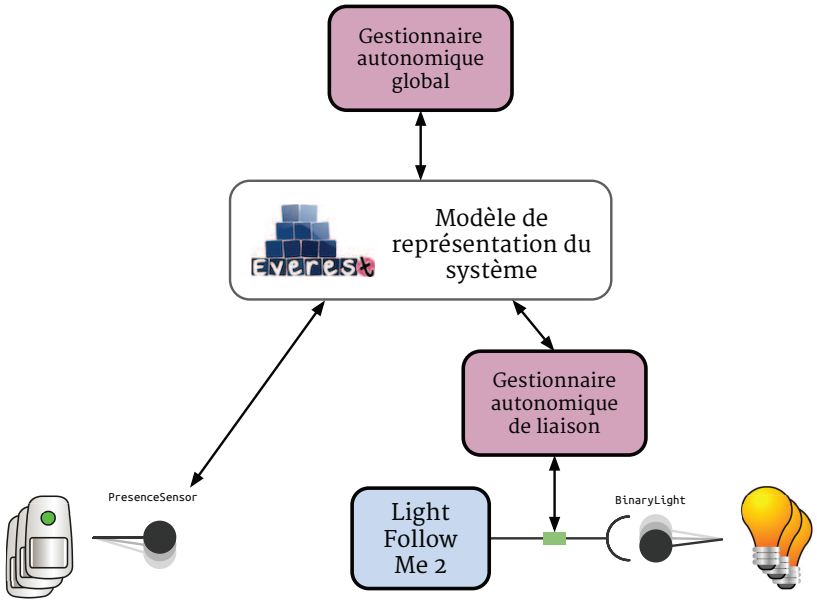

FIGURE B.1 – Architecture de l'implantation étendue de *LightFollowMe* (rappel)

Listing B.1 – Classe `LightFollowMeExtImpl`

```
package fr.liglab.adele.icasa.valid.pierre.extendedimpl;

import fr.liglab.adele.icasa.device.light.BinaryLight;
import org.apache.felix.ipojo.annotations.Bind;
import org.apache.felix.ipojo.annotations.Component;
import org.apache.felix.ipojo.annotations.Instantiate;
import org.apache.felix.ipojo.annotations.Unbind;

/**
 * Extended implementation of the LightFollowMe application.
 */
@Component
@Instantiate(name = "LightFollowMe-2")
public class LightFollowMeExtImpl {

    @Bind(id = "binaryLights", aggregate = true)
    private void bindBinaryLight(BinaryLight binaryLight) {
        // As soon as a binary light is bound, we turn it on.
        // The local autonomic manager selects the appropriated set of services,
        // according to the context.
        binaryLight.turnOn();
    }

    @Unbind(id = "binaryLights", aggregate = true)
    private void unbindBinaryLight(BinaryLight binaryLight) {
        // As soon as a binary light is unbound, we turn it off.
        // The local autonomic manager selects the appropriated set of services,
        // according to the context.
        binaryLight.turnOff();
    }

}
```

Listing B.2 – Classe `LocalManagerImpl`

```
package fr.liglab.adele.icasa.valid.pierre.extendedimpl;

import fr.liglab.adele.icasa.device.GenericDevice;
import org.apache.felix.ipojo.annotations.*;
import org.apache.felix.ipojo.dependency.interceptors.ServiceTrackingInterceptor;
import org.apache.felix.ipojo.dependency.interceptors.TransformedServiceReference;
import org.apache.felix.ipojo.handlers.event.Subscriber;
import org.apache.felix.ipojo.util.DependencyModel;
import org.osgi.framework.BundleContext;
import org.osgi.service.event.Event;
import org.ow2.chameleon.everest.impl.DefaultRequest;
import org.ow2.chameleon.everest.services.*;

import java.lang.reflect.UndeclaredThrowableException;
import java.util.HashMap;
import java.util.Map;

import static fr.liglab.adele.icasa.valid.pierre.extendedimpl.GlobalManagerImpl.*;

/**
 * Local autonomic manager for the extended LightFollowMe implementation.
 * <p>
 * This manager listens to resource events related to the binary lights services and the
 * user location.
 * </p>
 */
@Component
@Provides(
        // This property set the scope of the ServiceTrackingInterceptor.
        // We intercept the dependency "binaryLights" of component instance "LightFollowMe-2"
        properties = @StaticServiceProperty(
                name = "target",
                type = "java.lang.String",
                value = "(&(instance.name=LightFollowMe-2)(dependency.id=binaryLights))")
```

```
)
@Instantiate(name = "LightFollowMe-2.LocalManager")
public class LocalManagerImpl implements ServiceTrackingInterceptor {

    /**
     * The Everest resource model.
     */
    @Requires
    EverestService everest;

    /**
     * The dependency being managed.
     */
    private DependencyModel managedDependency;

    /**
     * The current user location, retrieved from the "/user" Everest resource.
     */
    private String currentLocation;

    /**
     * The device locations, retrieved from the "/device/location" resource.
     */
    private Map<String, Object> deviceLocations;

    /**
     * Called when the interceptor starts to manage a dependency.
     *
     * @param dependency the dependency being managed.
     */
    @Override
    public synchronized void open(DependencyModel dependency) {
        // Save the dependency.
        managedDependency = dependency;

        // Get the current location of the devices and the user.
        try {
            deviceLocations = new HashMap<>(
                    everest
                            .process(new DefaultRequest(
                                    Action.READ,
                                    DEVICE_LOCATION_PATH,
                                    null))
                            .getMetadata()
            );

            currentLocation = everest
                    .process(new DefaultRequest(Action.READ, USER_PATH, null))
                    .getMetadata()
                    .get(USER_CURRENT_LOCATION_PROPERTY, String.class);
        } catch (IllegalActionOnResourceException | ResourceNotFoundException e) {
            throw new UndeclaredThrowableException(e, "unexpected exception");
        }
    }

    @Override
    public synchronized <S> TransformedServiceReference<S> accept(
            DependencyModel dependency, BundleContext context,
            TransformedServiceReference<S> ref) {
        // The binary light service is accepted only if its location matches the current
        // user location.

        if (currentLocation == null) {
            // No user location information, hide the service.
            return null;
        }

        String binaryLightLocation = (String) deviceLocations.get(
                ref.getProperty(GenericDevice.DEVICE_SERIAL_NUMBER));

```

```
105         if (binaryLightLocation == null || !binaryLightLocation.equals(currentLocation)) {
106             // No device location information
107             // Or the location of the binary light does not math the user location
108             // => Hide the service.
109             return null;
110         }
111 
112         // BinaryLight and user are located in the same place.
113         return ref;
114     }
115 
116     /**
117      * Called when the "/user" resource has changed.
118      *
119      * @param e the resource event.
120      */
121     @Subscriber(name = "userEvents", topics = "/everest/user")
122     public synchronized void userEvent(Event e) {
123         if (managedDependency == null) {
124             // Not activated!
125             return;
126         }
127 
128         // Get the new value of the user location from the resource event.
129         String newLocation = ((ResourceMetadata) e.getProperty("metadata"))
130                 .get(USER_CURRENT_LOCATION_PROPERTY, String.class);
131 
132         // Compare with old value.
133         if (currentLocation == null && newLocation == null
134                 || currentLocation != null && currentLocation.equals(newLocation)) {
135             // Nothing has changed!
136             return;
137         }
138 
139         // Context has changed. The matching set of services is invalidated.
140         // iPOJO will automatically recompute it, calling our accept() method.
141         managedDependency.invalidateMatchingServices();
142     }
143 
144     /**
145      * Called when the "/device/location" resource has changed.
146      *
147      * @param e the resource event.
148      */
149     @Subscriber(name = "deviceLocationEvent", topics = "/everest/device/location")
150     public synchronized void deviceLocationEvent(Event e) {
151         if (managedDependency == null) {
152             // Not activated!
153             return;
154         }
155 
156         // We could compute what location of which device has changed,
157         // and check if the device is currently in the matching set of the managed dependency.
158         // However it is mush simpler here to invalidate the whole matching set and let iPOJO
159         // recompute the set, calling out interceptor with the updated context.
160 
161         // Get the changed context and invalidate the matching service set!
162         deviceLocations = new HashMap<>((ResourceMetadata) e.getProperty("metadata"));
163         managedDependency.invalidateMatchingServices();
164     }
165 
166     /**
167      * Called when the interceptor stops to manage a dependency.
168      *
169      * @param dependency the dependency no longer being managed.
170      */
171     @Override
172     public synchronized void close(DependencyModel dependency) {
173         // Release the dependency.
174         managedDependency = null;
```

```
175     }
176
177 }
```

Listing B.3 – Classe `GlobalManagerImpl`

```
1  package fr.liglab.adele.icasa.valid.pierre.extendedimpl;
2
3  import fr.liglab.adele.icasa.device.GenericDevice;
4  import fr.liglab.adele.icasa.device.presence.PresenceSensor;
5  import org.apache.felix.ipojo.annotations.Component;
6  import org.apache.felix.ipojo.annotations.Instantiate;
7  import org.apache.felix.ipojo.annotations.Requires;
8  import org.apache.felix.ipojo.annotations.Validate;
9  import org.apache.felix.ipojo.handlers.event.Subscriber;
10 import org.osgi.service.event.Event;
11 import org.ow2.chameleon.everest.impl.DefaultRequest;
12 import org.ow2.chameleon.everest.services.*;
13
14 import java.util.Arrays;
15 import java.util.HashMap;
16 import java.util.List;
17 import java.util.Map;
18
19 /**
20  * Global autonomic manager for the extended LightFollowMe implementation.
21  * <p>
22  * This manager listens to the PresenceSensor services and determine the current
23  * user location.
24  * </p>
25  */
26 @Component
27 @Instantiate(name = "LightFollowMe-2.GlobalManager")
28 public class GlobalManagerImpl {
29
30     /**
31      * The path of the service resources.
32      */
33     private static Path SERVICES_PATH = Path.from("/osgi/services");
34
35     // The following constants are shared with the local autonomic manager.
36
37     /**
38      * The path of the resource that contains the locations of the devices.
39      */
40     public static Path DEVICE_LOCATION_PATH = Path.from("/device/location");
41
42     /**
43      * The path of the user resource.
44      */
45     public static Path USER_PATH = Path.from("/user");
46
47     /**
48      * The property of the user resource that contains its current location
49      * (or null if unknown).
50      */
51     public static String USER_CURRENT_LOCATION_PROPERTY = "currentLocation";
52
53     /**
54      * The Everest resource model.
55      */
56     @Requires
57     EverestService everest;
58
59     /**
60      * Called when the component is started.
61      */
62     @Validate
63     private synchronized void start()
64             throws ResourceNotFoundException, IllegalActionOnResourceException {
```

```
65          // Get all the service resources.
66          Resource allServices = everest.process(
67                  new DefaultRequest(Action.READ, SERVICES_PATH, null));
68
69          // Filter out the resources that do not represent PresenceSensor devices.
70          List<Resource> presenceSensorServices = allServices.getResources(
71                  r -> Arrays.asList(r.getMetadata().get("objectClass", String[].class))
72                          .contains(PresenceSensor.class.getName())
73          );
74
75          // Create the device location resource if not present.
76          ResourceMetadata deviceLocations = getOrCreateResource(DEVICE_LOCATION_PATH)
77                  .getMetadata();
78
79          // Create the user resource, if not present.
80          Map<String, Object> userMetadata = new HashMap<>(getOrCreateResource(USER_PATH)
81                  .getMetadata());
82
83          // Determine the currentLocation of the user: it is the location of the first
84          // occurrence of a presenceSensor with "sensedPresence" property set to true.
85          String userLocation = null;
86          for (Resource r : presenceSensorServices) {
87              ResourceMetadata m = r.getMetadata();
88              if (m.get(PresenceSensor.PRESENCE_SENSOR_SENSED_PRESENCE, Boolean.class)) {
89                  userLocation = deviceLocations.get(
90                          m.get(GenericDevice.DEVICE_SERIAL_NUMBER, String.class),
91                          String.class);
92                  break;
93              }
94          }
95
96          // Write the user location. First create an empty resource if not present.
97          userMetadata.put(USER_CURRENT_LOCATION_PROPERTY, userLocation);
98          everest.process(new DefaultRequest(Action.UPDATE, USER_PATH, userMetadata));
99      }
100
101     /**
102      * Called invoked when a service resource event is received.
103      *
104      * @param event the received EventAdmin event.
105      */
106     @Subscriber(name = "serviceEvent", topics = "/everest/osgi/services/*")
107     private synchronized void serviceEvent(Event event)
108             throws ResourceNotFoundException, IllegalActionOnResourceException {
109         // Filter out non-PresenceSensor services.
110         ResourceMetadata metadata = (ResourceMetadata) event.getProperty("metadata");
111         if (!Arrays.asList(metadata.get("objectClass", String[].class))
112                 .contains(PresenceSensor.class.getName())) {
113             return;
114         }
115
116         // Is the presence sensor sensing someone?
117         if (metadata.get(PresenceSensor.PRESENCE_SENSOR_SENSED_PRESENCE, Boolean.class)) {
118             // Yes!
119             // Get the location of the presence sensor.
120             String location = everest.process(
121                     new DefaultRequest(Action.READ, DEVICE_LOCATION_PATH, null))
122                     .getMetadata().get(
123                             metadata.get(GenericDevice.DEVICE_SERIAL_NUMBER, String.class),
124                             String.class);
125
126             // Update current user location.
127             Map<String, Object> userMetadata = new HashMap<>();
128             userMetadata.putAll(
129                     everest.process(
130                             new DefaultRequest(Action.READ, DEVICE_LOCATION_PATH, null))
131                             .getMetadata()
132             );
133             userMetadata.put(USER_CURRENT_LOCATION_PROPERTY, location);
134             everest.process(new DefaultRequest(Action.UPDATE, USER_PATH, userMetadata));
```

```
135         }
136     }
137 
138     /**
139      * Utility method. Get the resource at following path, or create it if it does not exist.
140      *
141      * @param path the path of the resource to retrieve or create.
142      * @return the retrieved or created resource.
143      */
144     private Resource getOrCreateResource(Path path)
145             throws IllegalActionOnResourceException, ResourceNotFoundException {
146         Resource r;
147         try {
148             r = everest.process(new DefaultRequest(Action.READ, path, null));
149         } catch (ResourceNotFoundException e) {
150             // Not present! Create it.
151             r = everest.process(new DefaultRequest(Action.CREATE, path, null));
152         }
153         return r;
154     }
155 
156 }
```

Annexe C

Liste des publications

- [Escoffier 2013] Clément Escoffier, Pierre Bourret et Philippe Lalanda (2013). *Describing dynamism in service dependencies Industrial experience and feedbacks*. IEEE International Conference on Services Computing (SCC'13), pages 328–335, IEEE.
- [Garcia 2011] Isaac Noé Garcia Garza, Denis Morand, Bassem Debbabi, Philippe Lalanda, Pierre Bourret (2011). *A Reflective Framework for Mediation Applications*. International Middleware Conference (Middleware'11), pages 22–28, ACM.
- [Yu 2011] Jianqi Yu, Pierre Bourret, Philippe Lalanda, Johann Bourcier (2011). *Building and Deploying Self-Adaptable Home Applications*. Pervasive Computing and Communications Design and Deployment : Technologies, Trends and Applications, pages 49–73, IGI Global.
- [Yu 2010] Jianqi Yu, Philippe Lalanda, Pierre Bourret (2010). *An Approach for Dynamically Building and Managing Service-Based Applications*. Asia-Pacific Services Computing Conference (APSCC'10), pages 51–58, IEEE.

Bibliographie

[Abowd 1998] Gregory Abowd, Chris Atkeson et Irfan Essa. *Ubiquitous smart spaces*. Rapport technique, Georgia Institute of Technology, College of Computing, 1998.
Page 19.

[Aiello 2005] Marco Aiello, Ganna Frankova et Daniela Malfatti. *What's in an Agreement ? An Analysis and an Extension of WS-Agreement*. In International Conference on Service Oriented Computing, 2005.
Page 47.

[Atzori 2010] Luigi Atzori, Antonio Iera et Giacomo Morabito. *The Internet of Things : A survey*. Computer Networks, vol. 54, no. 15, pages 2787–2805, 2010.
Page 16.

[Baldauf 2007] Matthias Baldauf, Schahram Dustdar et Florian Rosenberg. *A survey on context-aware systems*. International Journal of Ad Hoc and Ubiquitous Computing, vol. 2, no. 4, pages 263–277, 2007.
Page 17.

[Baldoni 2003] R. Baldoni, C. Marchetti et L. Verde. *CORBA request portable interceptors : analysis and applications*. Third International Symposium of Distributed Object Applications (DOA'01), vol. 15, no. 6, pages 551–579, Mai 2003.
Page 88.

[Bouchenak 2006] Sara Bouchenak, Noel Palma, Daniel Hagimont et Christophe Taton. *Autonomic Management of Clustered Applications*. In 2006 IEEE International Conference on Cluster Computing, pages 1–11. IEEE, 2006.
Page 44.

[Bouix 2005] Emmanuel Bouix, Marc Dalmau, Philippe Roose et Franck Luthon. *A multimedia oriented component model*. In 19th International Conference Advanced Information Networking and Applications, 2005. AINA 2005., pages 3–8. IEEE, 2005.
Page 43.

[Bouix 2008] Emmanuel Bouix, Philippe Roose et Marc Dalmau. *The korrontea data modeling*. In Proceedings of the 1st international conference on Ambient media and systems, page 6. ICST (Institute for Computer Sciences, Social-Informatics and Telecommunications Engineering), 2008.
Page 43.

[Bourcier 2008] Johann Bourcier. *Auto-Home : une plate-forme pour la gestion automatique d'applications pervasives*. PhD thesis, Université Joseph Fourier, 2008.
Page 89.

[Bourcier 2011] Johann Bourcier, Ada Diaconescu, Philippe Lalanda et Julie A. McCann. *Autohome : An autonomic management framework for pervasive home*

applications. ACM Transactions on Autonomous and Adaptive Systems (TAAS), vol. 6, no. 1, pages 1–9, 2011.
Page 89.

[Bradbury 2004] Jeremy S. Bradbury, James R. Cordy, Juergen Dingel et Michel Wermelinger. *A survey of self-management in dynamic software architecture specifications*. In Proceedings of the 1st ACM SIGSOFT workshop on Self-managed systems, pages 28–33. ACM, 2004.
Page 72.

[Brown 2010] Nanette Brown, Ipek Ozkaya, Raghvinder Sangwan, Carolyn Seaman, Kevin Sullivan, Nico Zazworka, Yuanfang Cai, Yuepu Guo, Rick Kazman, Miryung Kim, Philippe Kruchten, Erin Lim, Alan MacCormack et Robert Nord. *Managing technical debt in software-reliant systems*. In Proceedings of the FSE/SDP workshop on Future of software engineering research - FoSER '10, page 47, New York, New York, USA, 2010. ACM Press.
Page 68.

[Bruneau 2009] Julien Bruneau, Wilfried Jouve et Charles Consel. *DiaSim : A parameterized simulator for pervasive computing applications*. In Mobile and Ubiquitous Systems : Networking & Services, (MobiQuitous' 09). 6th Annual International, pages 1–3. IEEE, Mars 2009.
Pages 41 et 42.

[Bruneton 2006] Eric Bruneton, Thierry Coupaye, Matthieu Leclercq, Vivien Quéma et Jean-Bernard Stefani. *The FRACTAL component model and its support in Java*. Software : Practice and Experience, vol. 36, no. 11-12, pages 1257–1284, Septembre 2006.
Page 44.

[Chess 2004] David M. Chess, Alla Segal, Ian Whalley et S.R. White. *Unity :experiences with a prototype autonomic computing system*. In International Conference on Autonomic Computing, 2004. Proceedings., pages 140–147. IEEE, 2004.
Page 89.

[Cheung 2003] Daniel Cheung, Jérôme Fuchet, Florent Grillon, Gabriel Joulie et Jean-Yves Tigli. *Wcomp : rapid application development toolkit for wearable computer based on Java*. SMC'03 Conference Proceedings. 2003 IEEE International Conference on Systems, Man and Cybernetics. Conference Theme - System Security and Assurance (Cat. No.03CH37483), vol. 5, pages 4198–4203, 2003.
Page 41.

[Chollet 2008] Stéphanie Chollet et Philippe Lalanda. *Security Specification at Process Level*. In 2008 IEEE International Conference on Services Computing, pages 165–172. IEEE, Juillet 2008.
Page 45.

[Chollet 2014] Stéphanie Chollet, Vincent Lestideau, Denis Morand, Yoann Maurel et Philippe Lalanda. *Auto-réparation et auto-optimisation des applications pervasives - Un gestionnaire de sélection de dépendances de services basé sur l'Analyse de Concepts Formels*. TSI Informatique autonomique, vol. 33, no. 1-2, pages 7–30, 2014.
Page 187.

[Cunningham 1993] Ward Cunningham. *The WyCash portfolio management system*. ACM SIGPLAN OOPS Messenger, vol. 4, no. 2, pages 29–30, Avril 1993.
Page 68.

[Da 2014] Keling Da, Marc Dalmau et Philippe Roose. *Kalimucho : Middleware for Mobile Applications*. In Proceedings of the 29th Annual ACM Symposium on Applied Computing, SAC'14, pages 413–419. ACM, 2014.
Page 43.

[David 2005] Pierre-charles David et Thomas Ledoux. *WildCAT : a generic framework for context-aware applications*. In Proceedings of the 3rd international workshop on Middleware for pervasive and ad-hoc computing - MPAC '05, pages 1–7, New York, New York, USA, 2005. ACM Press.
Page 44.

[de Palma 2008] Noël de Palma, Sara Bouchenak, Fabienne Boyer, Daniel Hagimont, Sylvain Sicard et Christophe Taton. *Jade, un environnement d'administration autonome*. Techniques et sciences informatiques, vol. 27, no. 8, pages 1225–1252, Octobre 2008.
Page 89.

[Dey 1999] Anind K Dey et Gregory D Abowd. *Towards a better understanding of context and context-awareness*. In Handheld and ubiquitous computing, pages 304–307. Springer, 1999.
Page 17.

[Diaconescu 2008] Ada Diaconescu, Yoann Maurel et Philippe Lalanda. *Autonomic Management via Dynamic Combinations of Reusable Strategies*. In Proceedings of the 2nd International ICST Conference on Autonomic Computing and Communication Systems. ICST, 2008.
Page 89.

[Dijkstra 1974] Edsger W. Dijkstra. *Self-stabilizing systems in spite of distributed control*. Communications of the ACM, vol. 17, no. 11, 1974.
Page 23.

[Dijkstra 1986] Edsger W. Dijkstra. *A belated proof of self-stabilization*. Distributed Computing, vol. 1, no. 1, pages 5–6, Mars 1986.
Page 72.

[Dowling 2001a] Jim Dowling et Vinny Cahill. *Dynamic Software Evolution and The K-Component Model*. In Workshop on Software Evolution (OOPSLA 2001), 2001.
Page 45.

[Dowling 2001b] Jim Dowling et Vinny Cahill. *The k-component architecture meta-model for self-adaptive software*. In The Third International Conference on Metalevel Architectures and Separation of Crosscutting Concerns (REFLECTION 2001), volume 3, pages 81–88, Kyoto, Japan, 2001. Springer Berlin Heidelberg.
Page 45.

[Druilhe 2013] Rémi Druilhe. *L'Efficience Énergétique des Services dans les Systèmes Répartis Hétérogènes et Dynamiques : Application à la Maison Numérique*. PhD thesis, Université Lille 1, 2013.
Page 23.

[Escoffier 2008] Clement Escoffier. *iPOJO : Un modèle à composant à service flexible pour les systèmes dynamiques*. PhD thesis, Université Joseph Fourier, 2008.
Pages 21, 93, et 96.

[Escoffier 2013] Clément Escoffier, Pierre Bourret et Philippe Lalanda. *Describing dynamism in service dependencies Industrial experience and feedbacks*. In IEEE Computer Society, editeur, IEEE 10th International Conference on Services Computing, page 8, Santa Clara, CA, USA, 2013. IEEE Computer Society.
Pages 52, 96, 105, 125, et 211.

[Escoffier 2014] Clement Escoffier, Stéphanie Chollet, Philippe Lalanda*et al.* *Lessons Learned in Building Pervasive Platforms*. In The 11th Annual IEEE Consumer Communications and Networking Conference, 2014.
Page 176.

[Fassino 2002] Jean-Philippe Fassino, Jean-Bernard Stefani, Julia Lawall et Gilles Muller. *Think : A Software Framework for Component-based Operating System Kernels*. Usenix Annual Technical Conference, pages 73–86, 2002.
Page 44.

[Fielding 2000] Roy Thomas Fielding. *Architectural Styles and the Design of Network-based Software Architectures*. PhD thesis, University Of California, Irvine, 2000.
Page 140.

[Fouquet 2012] François Fouquet, Erwan Daubert, Olivier Barais, Noël Plouzeau, Johann Bourcier, Jean-Émile Dartois et Arnaud Blouin. *Kevoree : une approche model@runtime pour les systèmes ubiquitaires*. In UbiMob2012, volume 26, page 12, Décembre 2012.
Page 45.

[Ganek 2004] A.G. Ganek, C.P. Hilkner, J.W. Sweitzer, B. Miller et J.L. Hellerstein. *The response to IT complexity : autonomic computing*. In Third IEEE International Symposium on Network Computing and Applications, 2004. (NCA 2004). Proceedings., pages 151–157. IEEE, 2004.
Page 86.

[Garcia Garza 2012] Issac Noé Garcia Garza. *Modèles de conception et d'exécution pour la médiation et l'intégration de services*. PhD thesis, Université de Grenoble, 2012.
Page 63.

[Garcia 2011] Issac Garcia, Denis Morand, Bassem Debbabi, Philippe Lalanda et Pierre Bourret. *A Reflective Framework for Mediation Applications*. In Adaptive and Reflective Middleware on Proceedings of the International Workshop, ARM '11, pages 22–28. ACM, 2011.
Page 211.

[Garlan 2001] David Garlan, Bradley Schmerl et Jichuan Chang. *Using gauges for architecture-based monitoring and adaptation*. In Working Conference on Complex and Dynamic Systems Architecture, Brisbane, Australia, 2001.
Page 87.

[Garlan 2004] David Garlan, Shang-Wen Cheng, An-Cheng Huang, Bradley Schmerl et Peter Steenkiste. *Rainbow : architecture-based self-adaptation with reusable infrastructure*. Computer, vol. 37, no. 10, pages 46–54, Octobre 2004.
Page 89.

[Grimm 2004] Robert Grimm, Janet Davis, Eric Lemar, Adam Macbeth, Steven Swanson, Thomas Anderson, Brian Bershad, Gaetano Borriello, Steven Gribble et David Wetherall. *System support for pervasive applications*. ACM Transactions

on Computer Systems (TOCS), vol. 22, no. 4, pages 421–486, 2004.
Page 16.

[Hallsteinsen 2010] Svein Hallsteinsen. *MUSIC vision and solutions*. Rapport technique, IST Music, Janvier 2010.
Page 39.

[Heineman 2001] George T. Heineman et William T. Councill. Component-Based Software Engineering : Putting the Pieces Together. Addison-Wesley Longman Publishing, 2001.
Page 45.

[Helal 2005] Sumi Helal, William Mann, Hicham El-Zabadani, Jeffrey King, Youssef Kaddoura et Erwin Jansen. *The gator tech smart house : A programmable pervasive space*. Computer, 2005.
Page 20.

[Horn 2001] Paul Horn. *Autonomic computing : IBM's Perspective on the State of Information Technology*, 2001.
Page 71.

[Huebscher 2004] Markus C. Huebscher et Julie A. McCann. *Adaptive middleware for context-aware applications in smart-homes*. In Proceedings of the 2nd workshop on Middleware for pervasive and ad-hoc computing, pages 111–116, New York, New York, USA, 2004. ACM.
Page 31.

[Huebscher 2008] Markus C. Huebscher et Julie A. McCann. *A survey of autonomic computing—degrees, models, and applications*. ACM Computing Surveys, vol. 40, no. 3, pages 1–28, Août 2008.
Page 75.

[IBM 2006] IBM. *An architectural blueprint for autonomic computing*. Rapport technique, IBM, International Business Machines Corporation, 2006.
Page 79.

[IEE 1998] IEEE, Institute of Electrical and Electronics Engineers, Inc. *IEEE Standard for Software Maintenance*, Juin 1998.
Page 57.

[Jiang 2008] Nanyan Jiang, Andres Quiroz, Cristina Schmidt et Manish Parashar. *Meteor : a middleware infrastructure for content-based decoupled interactions in pervasive grid environments*. Concurrency and Computation : Practice and Experience, vol. 20, no. 12, pages 1455–1484, Août 2008.
Page 89.

[Kephart 2003] Jeffrey O Kephart et David M. Chess. *The vision of autonomic computing*. Computer, vol. 36, no. 1, pages 41–50, Janvier 2003.
Pages 72, 75, 76, et 81.

[Kiczales 1991] Gregor Kiczales, Jd Riveres et Daniel G Bobrow. The Art of the Metaobject Protocol. 1991. MIT press, 1991.
Page 138.

[Koomey 2007] Jonathan G. Koomey. *Estimating total power consumption by servers in the US and the world*, Février 2007.
Page 66.

[Kramer 1990] Jeff Kramer et Jeff Magee. *The evolving philosophers problem : dynamic change management*. In IEEE Transactions on Software Engineering, volume 16, pages 1293–1306, 1990.
Page 88.

[Kramer 1998] Jeff Kramer et Jeff Magee. *Analysing dynamic change in software architectures : a case study*. In Configurable Distributed Systems, 1998. Proceedings. Fourth International Conference on, pages 91–100. IEEE, 1998.
Page 72.

[Lalanda 2013] Philippe Lalanda, Julie A. McCann et Ada Diaconescu. Autonomic Computing Principles, Design and Implementation. Undergraduate Topics in Computer Science. Springer London, London, 2013.
Page 196.

[Leclercq 2004] Matthieu Leclercq, Vivien Quéma et Jean-Bernard Stefani. *DREAM : a Component Framework for the Construction of Resource-Aware, Reconfigurable MOMs*. In Proceedings of the 3rd workshop on Adaptive and reflective middleware, pages 250–255, New York, New York, USA, 2004. ACM Press.
Page 44.

[Lehman 1980] Meir M. Lehman. *Programs, life cycles, and laws of software evolution*. Proceedings of the IEEE, vol. 68, no. 9, pages 1060–1076, Septembre 1980.
Pages 34, 57, 59, et 68.

[Li 2004] Zhen Li et Manish Parashar. *Rudder : a rule-based multi-agent infrastructure for supporting autonomic grid applications*. In International Conference on Autonomic Computing, 2004. Proceedings., pages 278–279. IEEE, 2004.
Page 89.

[Lientz 1980] Bennett P. Lientz et E. Burton Swanson. Software maintenance management. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1980.
Page 57.

[Liu 2006] Hua Liu et Manish Parashar. *Accord : a programming framework for autonomic applications*. IEEE Transactions on Systems, Man and Cybernetics, Part C (Applications and Reviews), vol. 36, no. 3, pages 341–352, Mai 2006.
Page 89.

[Louberry 2011] Christine Louberry, Philippe Roose et Marc Dalmau. *Kalimucho : Plateforme d'Adaptation des Applications Mobiles*. In Conférence Internationale sur les Nouvelles Technologies de la Répartition (NOTERE'11), Paris, FRANCE, 2011.
Page 43.

[Mattern 2001] Friedemann Mattern. *The vision and technical foundations of ubiquitous computing*. Upgrade European Online Magazine, vol. II, no. 5, pages 2–5, 2001.
Page 16.

[Maurel 2010a] Yoann Maurel. *CEYLAN : Un canevas pour la création de gestionnaires autonomiques extensibles et dynamiques*. PhD thesis, Université de Grenoble, 2010.
Page 89.

[Maurel 2010b] Yoann Maurel, Ada Diaconescu et Philippe Lalanda. *CEYLON : A Service-Oriented Framework for Building Autonomic Managers*. In 2010 Seventh

IEEE International Conference and Workshops on Engineering of Autonomic and Autonomous Systems, pages 3–11, Oxford, United Kingdom, Mars 2010. IEEE.
Page 89.

[Melville 2010] NP Melville. *Information systems innovation for environmental sustainability*. MIS Quarterly, vol. 34, no. 1, pages 1–21, 2010.
Page 67.

[Morand 2013] Denis Morand. *Cilia : un framework pour le développement d'applications de médiation autonomiques*. PhD thesis, Université de Grenoble, 2013.
Page 3.

[Morin 2006] Alain Morin. *Levels of consciousness and self-awareness : A comparison and integration of various neurocognitive views.* Consciousness and cognition, vol. 15, no. 2, pages 358–71, Juin 2006.
Page 84.

[Morin 2009] Brice Morin, Olivier Barais, Jean-Marc Jezequel, Franck Fleurey et Arnor Solberg. *Models@Runtime to Support Dynamic Adaptation*. Computer, vol. 42, no. 10, pages 44–51, Octobre 2009.
Page 45.

[Murch 2004] Richard Murch. Autonomic Computing. IBM Press, 2004.
Pages 83, 84, et 196.

[Oreizy 1999] Peyman Oreizy, Michael M. Gorlick, Richard N. Taylor, Dennis Heimbigner, Gregory Johnson, Nenad Medvidovic, Alex Quicili, David S. Rosenblum et Alexander L. Wolf. *An architecture-based approach to self-adaptive software*. Intelligent Systems and their Applications, IEEE, vol. 14, no. 3, pages 24–62, 1999.
Pages 72 et 96.

[Papazoglou 2003] Michael P Papazoglou et Dimitrios Georgakopoulos. *Service-oriented computing*. Communications of the ACM, vol. 46, no. 10, pages 25–28, 2003.
Pages 4 et 45.

[Papazoglou 2008] Michael P Papazoglou, Paolo Traverso, Schahram Dustdar, Frank Leymann et Bernd J. Krämer. *Service-Oriented Computing Research Roadmap*. International Journal of Cooperative Information Systems, vol. 17, no. 02, pages 223–255, Juin 2008.
Page 96.

[Parashar 2005] Manish Parashar et Salim Hariri. *Autonomic Computing : An Overview*. In International Workshop UPP 2004, pages 257–269, Le Mont Saint Michel, France, 2005. Springer Berlin Heidelberg.
Page 73.

[Parashar 2006] Manish Parashar, H. Liu, Z. Li, V. Matossian, C. Schmidt, G. Zhang et Salim Hariri. *AutoMate : Enabling Autonomic Applications on the Grid*. Cluster Computing, vol. 9, no. 2, pages 161–174, Avril 2006.
Page 89.

[Patterson 2002] David A. Patterson. *A Simple Way to Estimate the Cost of Downtime*. In Sixteenth Systems Administration Conference (LISA '02), pages 185–188, Berkeley, CA, 2002.
Page 70.

[Ranganathan 2005] Anand Ranganathan, Shiva Chetan, Jalal Al-Muhtadi, Roy H Campbell et M Dennis Mickunas. *Olympus : A high-level programming model for pervasive computing environments*. In Third IEEE International Conference on Pervasive Computing and Communications, 2005 (PerCom 2005), pages 7–16. IEEE, 2005.
Page 36.

[Román 2002] Manuel Román, Christopher Hess, Renato Cerqueira, Anand Ranganathan, Roy H. Campbell et Klara Nahrstedt. *Gaia : a middleware platform for active spaces*. ACM SIGMOBILE Mobile Computing and Communications Review, vol. 6, no. 4, pages 65–67, 2002.
Page 35.

[Romero 2010] Daniel Romero, Gabriel Hermosillo, Amirhosein Taherkordi, Russel Nzekwa, Romain Rouvoy et Frank Eliassen. *RESTful integration of heterogeneous devices in pervasive environments*. In Distributed Applications and Interoperable Systems, volume 6115, pages 1–14. Springer Berlin Heidelberg, 2010.
Page 44.

[Ronzani 2009] Daniel Ronzani. *The battle of concepts : Ubiquitous Computing, pervasive computing and ambient intelligence in Mass Media*. Ubiquitous Computing and Communication Journal, vol. 4, no. 2, pages 9–19, 2009.
Page 15.

[Rudolph 2001] Larry Rudolph. *Project Oxygen : Pervasive, human-centric computing–an initial experience*. In Advanced Information Systems Engineering, pages 1–12. Springer, 2001.
Page 38.

[Satyanarayanan 2001] Mahadev Satyanarayanan. *Pervasive computing : Vision and challenges*. Personal Communications, IEEE, vol. 8, no. 4, pages 10–17, 2001.
Pages 15 et 16.

[Schilit 1994] Bill Schilit, Norman Adams et Roy Want. *Context-aware computing applications*. In Mobile Computing Systems and Applications, 1994. WMCSA 1994. First Workshop on, pages 85–90. IEEE, 1994.
Pages 17 et 75.

[Seinturier 2006] Lionel Seinturier, Nicolas Pessemier, Laurence Duchien et Thierry Coupaye. *A component model engineered with components and aspects*. In 9th International Symposium on Component-Based Software Engineering (CBSE 2006), pages 139–153, Västerås, Sweden, 2006. Springer Berlin Heidelberg.
Page 44.

[Sicard 2008] Sylvain Sicard, Fabienne Boyer et Noel De Palma. *Using components for architecture-based management*. In Proceedings of the 13th international conference on Software engineering - ICSE '08, pages 101–110, New York, New York, USA, 2008. ACM Press.
Page 89.

[Singh 1996] Raghu Singh. *International Standard ISO/IEC 12207 Software Life Cycle Processes*. Software Process Improvement and Practice, vol. 2, no. 1, pages 35–50, 1996.
Page 57.

[Soules 2003] Craig A.N. Soules, Jonathan Appavoo, Kevin Hui, Dilma Da Silva, Gregory R. Ganger, Orran Krieger, Michael Stumm, Robert W. Wisniewski, Marc Auslander, Michael Ostrowski, Bryan Rosenburg et Jimi Xenidis. *System Support for Online Reconfiguration*. In USENIX Annual Technical Conference, General Track (USENIX'03), pages 141–154, 2003.
Page 88.

[Sousa 2002] João Pedro Sousa et David Garlan. *Aura : an architectural framework for user mobility in ubiquitous computing environments*. In Software Architecture, pages 29–43. Springer, 2002.
Page 37.

[Sprott 2004] David Sprott et Lawrence Wilkes. *Understanding service-oriented architecture*. The Architecture Journal, vol. 2004, no. January, 2004.
Page 45.

[Sterritt 2005] Roy Sterritt. *Autonomic computing*. Innovations in Systems and Software Engineering, vol. 1, no. 1, pages 79–88, Mars 2005.
Page 73.

[Szyperski 2002] Clemens Szyperski. Component software : beyond object-oriented programming. Pearson Education, 2002.
Page 3.

[Szyperski 2011] Clemens Szyperski. Component Software : Beyond Object-Oriented Programming. Addison-Wesley Professional, 2011.
Page 45.

[Tesauro 2004] Gerald Tesauro, David M Chess, William E Walsh, Rajarshi Das, Alla Segal, Ian Whalley, Jeffrey O Kephart et Steve R White. *A multi-agent systems approach to autonomic computing*. In Proceedings of the Third International Joint Conference on Autonomous Agents and Multiagent Systems (AAMAS '04), pages 464–471, New York, NY, USA, 2004.
Page 89.

[Tigli 2010] Jean-Yves Tigli, Stéphane Lavirotte, Gaëtan Rey, Vincent Hourdin et Michel Riveill. *Lightweight Service Oriented Architecture for Pervasive Computing*. International Journal of Computer Science Issues (IJCSI), vol. 7, no. 4, pages 1–9, 2010.
Page 42.

[Vandewoude 2006] Yves Vandewoude, Peter Ebraert, Yolande Berbers et Theo D'Hondt. *An alternative to Quiescence : Tranquility*. In 2006 22nd IEEE International Conference on Software Maintenance, pages 73–82, Philadelphia, PA, United States of America, Septembre 2006. IEEE.
Pages 72, 88, et 105.

[Waldner 2007] Jean-Baptiste Waldner. Nano-informatique et intelligence ambiante : inventer l'ordinateur du xxie siècle. Hermès science publications, 2007.
Page 13.

[Weiser 1991] Mark Weiser. *The computer for the 21st century*. Scientific american, vol. 265, no. 3, pages 94–104, 1991.
Pages 15 et 16.

[Weiser 1996] Mark Weiser et John Seely Brown. *Designing calm technology*. PowerGrid Journal, vol. 1, no. 1, pages 75–85, 1996.
Pages 13 et 14.

[Wiederhold 1992] Gio Wiederhold. *Mediators in the architecture of future information systems*. Computer, vol. 25, no. 3, pages 38–49, Mars 1992.
Pages 30 et 63.

[Yu 2010] Jianqi Yu, Philippe Lalanda et Pierre Bourret. *An Approach for Dynamically Building and Managing Service-Based Applications*. In Services Computing Conference (APSCC), 2010 IEEE Asia-Pacific, pages 51–58, Dec 2010.
Page 211.

[Yu 2011] Jianqi Yu, Pierre Bourret, Philippe Lalanda et Johann Bourcier. *Building and Deploying Self-Adaptable Home Applications*. Pervasive Computing and Communications Design and Deployment : Technologies, Trends, and Applications, pages 49–73, 2011.
Page 211.