Gestion de conflits dans une plateforme ubiquitaire orientée services

Rania Ben Hadj

Communauté
UNIVERSITÉ Grenoble Alpes

THÈSE

Pour obtenir le grade de

DOCTEUR DE LA COMMUNAUTE UNIVERSITÉ GRENOBLE ALPES

Spécialité : **Informatique**

Arrêté ministériel du : 25 Mai 2016

Présentée par

Rania BEN HADJ

Thèse dirigée par **Philippe LALANDA**
et co-encadrée par **Catherine HAMON et Stéphanie CHOLLET**

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

Gestion des conflits dans une plateforme ubiquitaire orientée services

Thèse soutenue publiquement le **27 avril 2018**
devant le jury composé de:

Pr. Ernesto ESPOSITO
Professeur, Université de Pau et des Pays de l'Adour, Rapporteur
Dr. Frédéric WEIS
Maître de conférences (HDR), Université de Rennes 1, Rapporteur
Dr. Yoann MAUREL
Maître de Conférences, Université de Rennes 1, Examinateur
Pr. Claudia RONCANCIO
Professeur, Grenoble INP, Présidente
Pr. Philippe LALANDA
Professeur, Université Grenoble Alpes, Directeur de thèse
Dr. Catherine HAMON
Ingénieur de recherche, Orange Labs, Co-encadrante
Dr. Stéphanie CHOLLET
Maître de conférences, Grenoble INP, Co-encadrante

UNIVERSITÉ DE GRENOBLE

ÉCOLE DOCTORALE MSTII

Description de complète de l'école doctorale

T H È S E

pour obtenir le titre de

docteur en sciences

de l'Université de Grenoble-Alpes

Mention : Informatique et Mathématiques appliquées

Présentée et soutenue par

Rania BEN HADJ

Gestion des conflits dans une plateforme ubiquitaire orientée services

Thèse dirigée par Philippe LALANDA

préparée au Laboratoire d'Informatique de Grenoble (LIG)

soutenue le 30 avril 2018

Jury :

Rapporteurs :	Ernesto ESPOSITO	-	Université de Pau et des Pays de l'Adour
	Frédéric WEIS	-	Université de Rennes 1
Examinateur :	Yoann MAUREL	-	Université de Rennes 1
Présidente :	Claudia RONCANCIO	-	Grenoble INP
Directeur :	Philippe Lalanda	-	Université Grenoble Alpes
Co-encadrantes :	Catherine HAMON	-	Orange Labs
	Stéphanie CHOLLET	-	Grenoble INP

Contents

List of Figures

List of Tables

Introduction

Contexte et Motivation

À la lumière de l'évolution de l'informatique et de l'innovation technologique, une vague de dispositifs puissants, communicants et intelligents a balayé le monde entier : Des smartphones aux télévisions connectées, en passant par les nouveaux appareils plus miniaturisés, capteurs tant actionneurs, tous ces dispositifs présents dans nos environnements ont engendré une véritable refonte dans nos modes de vie. Ces nouveaux dispositifs ayant la capacité de collecter des données de nos environnements ou d'y agir ont entrainé une mutation profonde qui porte sur la nature des applications proposées pour l'utilisateur. Ces applications, nommées ubiquitaires, sont capables d'interagir avec les dispositifs qui l'entourent d'une manière transparente et invisible afin de lui fournir des services à valeur ajoutée. Aujourd'hui, ces applications suscitent l'intérêt et les attentes des consommateurs dans divers domaines tels que la domotique, la santé, le transport, etc. Toutefois, les applications ubiquitaires actuelles n'ont pas atteint une maturité optimale leur permettant de répondre à ces attentes. Elles présentent en effet des exigences strictes qui rendent leur développement complexe. Ces applications doivent en effet interagir avec des dispositifs proposés par des acteurs différents, ne parlant pas le même langage et pouvant apparaitre et disparaitre de l'environnement à tout moment d'une manière imprédictible.

Le domaine d'applications ubiquitaires que nous motive dans cette thèse est le domaine de la maison intelligente (Smart Home, en anglais). L'objectif principal de la maison intelligente est d'améliorer la qualité de vie de ses habitants. Dans un communiqué de presse publié en avril 2013, Xerfi, un institut spécialisé dans l'analyse sectorielle en France, affirme que « toutes les conditions sont enfin réunies pour faire de la maison intelligente un véritable marché de masse. ». Aujourd'hui, les circonstances technologiques permettent au marché de Smart Home de devenir un marché hyperactif promouvant une véritable concurrence avec les fournisseurs en amont et les clients en aval. Ce marché a séduit plusieurs acteurs puissants tels qu'Apple, Orange, Samsung, SFR qui y ont investi. En revanche, les solutions domotiques commercialisées actuellement par ces entreprises appariassent limitées. Par exemple, l'offre domotique proposée par Orange est limitée à un ensemble d'applications de sécurités développées par l'utilisateur final. Ce dernier n'étant pas un expert technique, il doit en effet programmer des applications simples sous forme des règles (si condition alors action). Certes, Orange vise dans la future à commercialiser une solution plus ouverte pour la maison intelligente, capable d'intégrer des applications plus riches développées par ses partenaires de confiance. Aujourd'hui, il est clair que tous les acteurs Smart Home envisagent d'adapter de plus en plus leurs solutions domotiques aux besoins et aux demandes des clients afin de les séduire. Ces solutions peuvent alors intégrer des applications ubiquitaires de divers domaines tels que la sécurité, la santé, la gestion d'énergie, le confort. D'une manière générale, les applications s'exécutent sur une

plate-forme ubiquitaire hébergée dans une box domotique.

Dans cette thèse, nous avons pu constater, à travers les travaux existants et la littérature, que l'informatique orientée services est au cœur des solutions domotiques. En effet, la plupart des plateformes ubiquitaires sont implantées en s'appuyant sur ce paradigme. Les dispositifs étant considérés comme les ressources primaires utilisées par les applications ubiquitaires, ne sont pas directement accessibles. Concrètement, une fois le dispositif est découvert, Il est généralement réifié sous forme de service dans la plate-forme. Il peut être par la suite recherché et consommé d'une manière explicite par les applications. Lorsque le dispositif n'est plus disponible, le service qui le représente est retiré de la plateforme. Aujourd'hui, certaines plateformes ubiquitaires proposent des services de plus en plus abstraits fournissant des fonctions de haut niveau construites à partir d'autres services. Tout comme les services représentant les dispositifs, les services abstraits permettent d'avoir des informations sur l'environnement (par exemple, la température dans une pièce) ou d'y agir (par exemple, allumer toutes les lumières dans une pièce). Ces services intéressent les développeurs et facilitent leur tâche en libérant les applications du code complexe permettant de calculer ou de créer ces informations. Toutefois, il est indispensable de noter que ces services permettent aux applications d'exploiter les dispositifs parsemés dans leur environnement d'une manière implicite.

Les services proposés par la plateforme sont partagés par les applications s'y exécutant. Ce type de partage pose naturellement le risque de conflits entre les applications. En effet, ces applications, appartenant à des domaines variés, consomment ces services afin de satisfaire des différentes exigences pouvant être incompatibles dans certaines situations. Dans ces situations, elles peuvent avoir des comportements désaccordés pouvant mettre la maison dans un état incohérent, voire dangereux, inadmissible par l'utilisateur. Il s'avère ainsi indispensable de traiter ces conflits afin de prévenir leurs effets indésirables et d'assurer un comportement cohérent des maisons intelligentes.

Dans ce travail, nous nous adressons à ce problème de conflits qui peuvent se produire entre les applications domotiques dans une plate-forme ubiquitaire orientée services. A cet égard, il faut noter que bien que les services abstraits facilitent le développement d'applications ubiquitaires, ils rendent la gestion de conflits un défi très difficile à relever. Ceci est particulièrement dû au fait que les applications consommant les services proposés par la plateforme peuvent agir sur les dispositifs d'une manière explicite ou implicite, ce qui peut sans doute multiplier les situations de conflits qui peuvent se produire. Dans cette thèse, nous avons pu constater, par le biais des travaux existants, qu'il est complexe de gérer les conflits à la phase de conception. Ceci nécessite de tout s'avoir non seulement

sur les applications (les applications s'exécutant sur la plateforme, ressources requis, condition d'utilisation, action, etc.) mais aussi sur l'environnement d'exécution (ressources disponibles, localisation, etc.). La dynamicité étant une caractéristique clé des environnements Smart Home, il s'avère impossible de décrire toutes ces informations au design time. Dans notre vision, Il est indispensable de définir une approche permettant de gérer les conflits à l'exécution via une représentation de l'environnement.

Contribution

Cette thèse s'inscrit dans le cadre d'une collaboration recherche/ industrie. L'objectif de cette collaboration est de fournir des outils pour gérer les conflits qui peuvent se produire entre les applications de la maison dans une plate-forme ubiquitaire basée sur un contexte orienté services. Ces outils doivent permettre de limiter l'intervention humaine (utilisateur/ développeur) dans le processus de gestion de conflits. De plus, il est indispensable qu'ils répondent à deux aspects complémentaires. Le premier aspect porte sur la prévention des conflits alors que le deuxième porte sur leur détection et résolution lorsque la prévention n'est pas abordable. À l'exécution, ces outils doivent assurer la continuité des services fournis par les applications en conflit.

Notre contribution s'articule autour de trois axes majeurs :

- Les conflits entre les applications sont définis dans un modèle qui représente l'environnement sur la plateforme ubiquitaire. Ce modèle, appelé contexte, fournit tous les services abstraits et simples pouvant être consommés par les applications afin d'interagir d'une manière implicite ou explicité avec les dispositifs installés dans leur environnement.
- Un modèle de développement d'applications étendu par des mécanismes de verrouillage et déverrouillage pour les services du contexte. Ces mécanismes permettent aux applications de détenir un verrou sur un service susceptible d'être au centre d'un conflit à l'exécution pendant une durée contextuelle.
- Une approche de gestion des conflits à trois phases (prévention, détection, résolution). Les conflits sont gérés pendant ces différentes phases par le biais des mécanismes de verrouillage/déverrouillage, de gestion de visibilité des services, d'adaptation et de priorisation des applications. L'approche que nous proposons étant optimiste, les conflits sont identifiés et résolus à l'exécution au plus tard possible permettant aux applications de s'exécuter le plus longtemps.

Notre proposition a été implantée et intégrée dans iCASA, une plateforme ubiquitaire domotique développée dans le cadre d'un projet collaboratif entre Orange Labs et l'équipe Adele du Laboratoire d'Informatique de Grenoble.

Organisation du manuscrit

Le présent manuscrit présente notre travail dans ses différentes phases. Il s'articule autour de deux grandes parties : l'état de l'art et la contribution.

L'état de l'art intègre trois chapitres :

Le Chapitre 1 présente les concepts clé de l'informatique ubiquitaire. Il examine les exigences majeures imposées par cette nouvelle vision et qui se rattachent principalement aux environnements ubiquitaires et aux applications y déployés. Nous nous concentrons particulièrement sur les Smart Home, les environnements pervasifs dans lesquelles s'intègre ce travail.

Le Chapitre 2 propose deux études : la première étant la gestion des conflits, nous détaillons son origine, sa définition ainsi qu'une analyse comparative des principales approches proposées permettant de les gérer. La deuxième étude est consacrée au contrôle d'accès. Nous présentons les principales techniques existantes et nous identifions la meilleure technique à appliquer pour gérer les conflits.

Le Chapitre 3 présente une étude autour des plateformes logicielles qui traitent les exigences imposées par les environnements pervasifs. Trois axes majeurs se dégagent de cette étude : les deux premiers sont les plus générales et présentent respectivement les plateformes dynamiques et ouvertes. Le troisième axe est dédié aux plateformes pervasives et se base sur une étude comparative des principales plateformes proposées.

La contribution comprend deux chapitres :

Le Chapitre 4 se divise en deux parties majeures. La première partie définit la problématique que nous adressons dans cette thèse et énonce les principales constatations dégagées de l'état de l'art. La deuxième partie présente notre approche générale, puis elle décrit d'une manière détaillée tous ses aspects clés.

Le Chapitre 5 expose l'implantation de notre proposition suivie d'une validation permettant de tester et d'évaluer la solution proposée.

Le Chapitre 6 présente une conclusion générale exposant le bilan du travail et dressant les perspectives concernant les voies d'amélioration de la solution proposée pour la gestion des conflits.

Chapter 1

Informatique Ubiquitaire

Sommaire

Le but de ce premier chapitre est de présenter le domaine de l'informatique ubiquitaire, aussi connu sous le nom d'informatique pervasive. Nous présenterons, dans une première partie, les concepts fondamentaux liés à ce domaine. Nous définirons ensuite les propriétés principales des environnements ubiquitaires : l'hétérogénéité, l'ouverture, la distribution et le dynamisme. La troisième partie de ce chapitre sera consacrée à une étude approfondie sur les besoins des applications informatiques ubiquitaires, leurs caractéristiques clés qui permettent de les distinguer des applications traditionnelles ainsi que leur cycle de vie. Nous nous intéressons, dans une dernière partie, à la maison intelligente, l'environnement ubiquitaire qui nous motive dans cette thèse. Nous présenterons donc les défis majeurs, tant les besoins techniques que les besoins humains, auxquels une maison intelligente doit répondre. Nous analyserons également les différents domaines d'applications pouvant y être déployées.

1.1 Introduction

L'informatique ubiquitaire est une nouvelle vision apparue suite à l'évolution progressive des environnements informatiques. Dans cette introduction, nous rappelons cette évolution et nous présentons certaines définitions de l'informatique ubiquitaire qui servent à clarifier cette vision et à dévoiler ses caractéristiques clés.

1.1.1 Evolution des environnements informatiques

Dans un monde où les technologies évoluent de plus en plus vite, la généralisation de l'informatique et des technologies numériques affectent fortement nos modes de vie. En outre, le nombre des systèmes informatiques et de dispositifs numériques parsemés dans nos environnements ne cesse de s'accroître tout en suivant le rythme intense des progrès technologiques. L'émergence avérée des dispositifs de plus en plus petits, communicants, intelligents et puissants a engendré de profondes mutations dans les services fournis à l'utilisateur. De plus, le progrès technologique et l'essor de l'informatique ont impacté profondément nos manières d'interagir avec les systèmes informatiques.

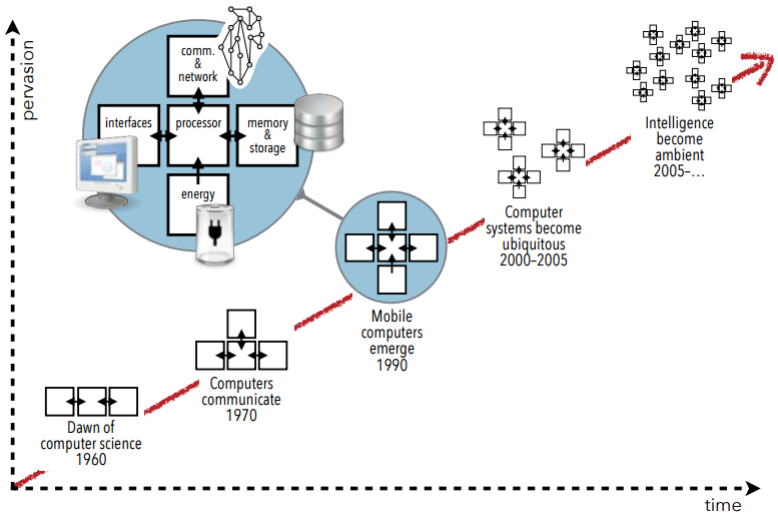

Figure 1.1: Evolution des systèmes informatiques [Wal07]

L'avancée ininterrompue de l'informatique et l'évolution des systèmes informatiques ont été le cœur de plusieurs analyses et études de recherche. Dans [WB96], Mark Weiser a introduit son analyse par les phrases suivantes :

« *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.* » [WB96]

La figure 1.1 montre les principaux changements technologiques qui ont marqué l'évolution de l'informatique. Elle met en lumière les différentes étapes clés de cette évolution.

La première ère de l'informatique est l'ère des mainframes où l'informatique centralisée se présentait comme le seul principe à suivre pour construire des systèmes numériques. Ces systèmes correspondaient à des machines volumineuses qui centralisaient toutes les données et les traitements. Ces mainframes (les systèmes centralisés) devaient être partagés entre plusieurs utilisateurs et leurs administrations exigeaient un niveau haut d'expertise. Un mainframe est ainsi considéré comme un ordinateur central qui présente un ensemble de ressources limitées et utilisées collectivement par plusieurs personnes. Ceci est présenté dans [WB96] par la manière suivante :

« *If lots of people share a computer, it is a mainframe computing.* » [WB96]

L'ère suivante est l'ère des ordinateurs personnels où l'adoption de l'informatique personnelle en tant que mode d'interaction entre les utilisateurs et le monde numérique est devenue prédominante. En adoptant ce mode d'interaction, un système ne peut appartenir qu'à un seul utilisateur et il ne peut être utilisé que par une seule personne à la fois. Chaque utilisateur est ainsi lié à un ordinateur personnel où il peut stocker ses données et installer ses logiciels. L'émergence de l'informatique personnelle a été présentée par Mark Weiser [WB96] par les phrases suivantes :

« *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.* » [WB96]

Comme les mainframes, les ordinateurs personnels possèdent un ensemble de ressources locales limitées. Toutefois, grâce aux nouvelles technologies de réseaux tels que l'Internet, les ordinateurs peuvent accéder à des ressources distantes, par exemple des services distants. Ceci est analysé dans [WB96] de la manière suivante :

« *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.* » [WB96]

La troisième ère est l'ère de l'informatique mobile née avec l'avènement des ordinateurs portables et des technologies sans fils. L'utilisateur peut ainsi déplacer son ordinateur et changer sa localisation et il peut ainsi accéder à ses données et utiliser ses applications de n'importe où et n'importe quand. Aujourd'hui, l'informatique mobile est favorisée par l'émergence des téléphones intelligents et des tablettes tactiles et elle est au cœur de plusieurs applications. L'exemple d'applications le plus connu dans le monde de l'informatique mobile est les applications sensibles à la position géographique de son utilisateur. Ces applications doivent adapter leurs réponses en fonction de la position géographique de leur utilisateur.

La dernière évolution illustrée par la figure 1.1 correspond à l'ère de l'informatique ubiquitaire. L'informatique ubiquitaire est une nouvelle vision ayant pour objectif d'intégrer les nouveaux dispositifs toujours plus miniaturisés, plus puissants, plus communicants et plus intelligents dans la vie quotidienne [Lal+10]. Cette vision a provoqué aussi des changements radicaux dans les capacités d'interaction de ces dispositifs. Ces interactions sont ainsi plus naturelles, transparentes et invisibles pour les utilisateurs.

1.1.2 Définition et propriétés

L'informatique ubiquitaire a été introduite en 1991 par Mark Weiser au Xerox PARC pour décrire sa vision futuriste de l'informatique du XXIe siècle. Dans son article fondateur [Wei91], il envisage un monde saturé de petits dispositifs informatisés (capteurs, actionneurs) embarqués dans les divers objets de la vie quotidienne. Ces appareils participent à un réseau d'information global et interagissent de manière autonome et transparente dans le but d'accomplir diverses tâches pour l'utilisateur.

Cette vison a inspiré de nombreux chercheurs et industriels et a donné naissance à une multitude de projets de recherche, ce qui a conduit à l'apparition de nombreux termes, généraux ou même relatifs aux objets dissimulés, tels que : « calm computing », « disappearing computer », « everyware » [Gre10] , « Internet of things » [Mat05] , « ambient intelligence » [Han+03] et « things that think » [HPT97]. Malgré les différences autour de l'utilisation de ces termes, les études de Daniel Ronzani [Ron09] ont conclu qu'ils reprennent tous la même idée de base proposée par Weiser [Wei91]. Dans cette thèse, nous

utilisons les termes informatique ubiquitaire ou informatique pervasive pour se réfèrer à tous les termes cités précédemment.

Plusieurs définitions ont été proposées selon ces divers concepts. Nous analysons dans la suite les principales définitions utilisées dans la littérature :

« *The most profound technologies are those that disappear, [...] They weave themselves into the fabric of everyday life until they are indistinguishable from it.* » [Wei91]

« *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.* » [Sat01]

Dans les définitions précédentes, les auteurs présentent la technologie d'une manière générale et ils se concentrent principalement sur l'aspect de l'intégration transparente des services fournis par l'informatique ubiquitaire. Les définitions que nous citons ci-dessous se concentrent davantage sur la connectivité des différents types de dispositifs.

« *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.* » [Mat01]

« *Pervasive computing calls for the deployment of a wide variety of smart devices throughout our working and living spaces. 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.* » [GB03]

« *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.* » [AIM10]

En se basant sur les définitions citées précédemment, nous pouvons distinguer trois propriétés fondamentales qui permettent de caractériser l'informatique ubiquitaire :

- l'informatique ubiquitaire est **invisible** : cette propriété est non seulement liée à l'invisibilité réelle grâce à la miniaturisation des systèmes numériques et leur intégration dans les objets quotidiens mais aussi à l'invisibilité de l'utilisation naissant d'une interaction homme-machine toujours plus naturelle et transparente.

- l'informatique ubiquitaire est **distribuée** : le service fourni à l'utilisateur est le résultat d'une coopération entre des équipements fixes et mobiles, ainsi que des services distants. En effet, chaque équipement doit interagir avec son environnement en coopérant avec les autres équipements qui l'entourent d'une manière invisible pour l'utilisateur.

- l'informatique ubiquitaire est **sensible au contexte** : le système doit s'adapter constamment à l'évolution du contexte d'exécution.

Ces propriétés permettent d'éclaircir la vision de l'informatique ubiquitaire. Elles révèlent les aspects fondamentaux de l'informatique ubiquitaire tels que l'invisibilité, la distribution et la sensibilité au contexte. Cependant, elles sont insuffisantes pour cerner les contours de l'informatique ubiquitaire. Les sections 1.2 et 1.3 de ce chapitre détaillent les caractéristiques attendues des systèmes informatiques ubiquitaires.

1.1.3 Contexte et sensibilité au contexte

La sensibilité au contexte est une propriété primordiale des systèmes informatiques ubiquitaires. Elle caractérise la capacité d'un système à réagir pour adapter sa réponse aux changements de son environnement. Selon Korkea-Aho [KA00] « un système est sensible au contexte s'il peut extraire, interpréter et utiliser des informations de contexte, et adapter sa fonctionnalité au contexte d'utilisation courant ».

Les systèmes sensibles au contexte dont l'interaction dépend des données physiques de l'environnement ont fait l'objet de recherches approfondies. En informatique mobile, l'exemple le plus connu de ces données mesurables a été introduit par la localisation de l'utilisateur, il s'agit de systèmes sensibles à la localisation. Ces systèmes sont capables de géolocaliser l'utilisateur et de lui apporter des réponses pertinentes en fonction de sa position, comme les cartes routières qui lui permettent de trouver son itinéraire vers sa destination.

La définition de contexte est une phase indispensable dans le processus de la construction d'un système sensible au contexte. Plusieurs définitions ont été proposées : Schilit et Theimer ont été les premiers à introduire la notion du contexte dans [ST94] en répondant aux questions suivantes : « Où suis-je ? », « Avec qui suis-je ? », « Quelles sont les ressources disponibles aux environs ? ». Ils définissent alors le contexte comme étant « la localisation de l'utilisateur, les identités et les états des personnes et des objets qui l'entourent ». Dans une définition semblable, Brown [BBC97] présente le contexte par « l'identité de l'utilisateur, des personnes et des objets qui l'entourent, sa localisation géographique, son orientation, la saison et la température où il évolue ». Cependant, ces définitions paraissent très limitées et très restrictives. En effet, bien que les auteurs aient essayé de présenter le contexte en énumérant les éléments du contexte qui peuvent modifier l'application, ces éléments ne peuvent pas être valables pour toutes les applications. Par exemple, une information, telle que la position géographique, n'est pas nécessairement significative pour tous les types d'applications ubiquitaires.

Anind Dey et Gregory Abowd [Abo+99] évitent de se baser sur une énumération d'exemples d'éléments contextuels et proposent une définition plus générale du contexte. Cette définition est la plus utilisée aujourd'hui par les développeurs de systèmes ubiquitaires sensibles au contexte :

« *Any information that can be used to characterize the situation of entities (i.e., whether a person, place or object) that are considered relevant to the interaction between a user and an application, including the user and the application themselves.* » [Abo+99]

1.2 Caractéristiques des environnements ubiquitaires

Dans la section précédente, nous avons étudié les différentes définitions de l'informatique ubiquitaire présentées dans la littérature et dans cette section nous présentons, en détail, les propriétés permettant de qualifier un environnement d'ubiquitaire.

1.2.1 Ouverture

Les environnements ubiquitaires sont des environnements ouverts permettant la cohabitation entre de multiples acteurs de marché qui deviennent de plus en plus actifs suivant la forte croissance technologique. En effet, ces environnements peuvent intégrer non seulement des dispositifs délivrés par des vendeurs et des fabricants différents mais aussi des services issus de fournisseurs multiples. Dans un environnement ouvert, ces dispositifs et services doivent être conçus pour être interopérables ; c'est-à-dire que leurs spécifications doivent être intégralement connues et accessibles dans le but de fonctionner et de communiquer avec d'autres dispositifs ou applications.

Bien que l'ouverture aux tiers soit une caractéristique primordiale des environnements ubiquitaires, de nombreux acteurs aujourd'hui limitent cette ouverture dans le but d'inhiber la concurrence et de dominer le marché. Ces derniers proposent des solutions où les dispositifs ne sont totalement accessibles que par leurs propres applications. Par ailleurs, ils restreignent l'accès à certaines ressources et à leurs fonctionnalités afin de limiter les possibilités offertes aux fournisseurs de services concurrents.

1.2.2 Distribution

Les environnements ubiquitaires intègrent un nombre de plus en plus important de dispositifs communicants : certains sont les dispositifs que nous utilisons tous les jours tels que les téléphones portables, les ordinateurs ou les tablettes ; d'autres sont des dispositifs qui permettent d'interagir avec l'environnement comme des capteurs qui permettent de capter des données de l'environnement et de connaître son état ou des actionneurs qui permettent d'agir sur l'environnement et de changer son état. En général, les applications ubiquitaires sont amenées à inter-opérer avec ces dispositifs, utiliser leurs fonctionnalités explicitement exposées et les coordonner pour rendre des services à l'utilisateur. Ces services sont ainsi le résultat d'une coopération entre les dispositifs dispersés dans son environnement, il s'agit donc d'un environnement distribué.

Les ressources disponibles pour l'environnement ubiquitaire ne sont pas restreintes aux dispositifs installés dans l'environnement de l'utilisateur mais elles intègrent aussi des serveurs ou des services distants présents, par exemple, dans le *cloud*. En général, dans le cas où l'application nécessite une grande capacité de stockage et une forte puissance de calcul, les données brutes issues des capteurs dispersés dans l'environnement ubiquitaire doivent être collectées, filtrées puis transmises à un serveur distant pour être par la suite analysées et traitées.

La conception des applications ubiquitaires est une tâche très complexe et difficile à cause de ces particularités de l'environnement d'exécution. En effet, les développeurs d'applications ubiquitaires doivent tenir compte de la nature distribuée de l'environnement.

1.2.3 Hétérogénéité

Avec l'essor de l'Internet et la forte croissance technologique, nos environnements deviennent de plus en plus riches en dispositifs fortement hétérogènes. Cette hétérogénéité résulte essentiellement d'une concurrence sévère entre les différents fabricants et vendeurs de dispositifs. La diversité des dispositifs dans les environnements ubiquitaires n'est pas limitée seulement au motif de leurs utilisations et leurs performances mais elle inclut aussi les technologies et les protocoles de communication, les modèles de données et les langages de programmation.

Le problème majeur qui résulte de cette hétérogénéité est la complexité de développement des applications ubiquitaires. En effet, ces applications sont amenées à inter-opérer avec un ensemble de ressources fortement hétérogènes. Les développeurs doivent ainsi intégrer une grande variété de types de ressources et de protocoles de communication et traiter des formats différents des données collectées de ces ressources, dans leurs applications. Cependant, plusieurs chercheurs se sont penchés sur cette problématique et ils ont proposé des solutions pour masquer cette hétérogénéité et faciliter la tâche de développement d'applications. Par exemple, certains travaux de recherche [Gar12] ont abordé le problème autour du format des données en s'appuyant sur un middleware de médiation. D'autres approches sont mises en œuvre afin de supporter un nombre extensible de protocoles pouvant être déployés en fonction des exigences des applications [Bar12].

1.2.4 Dynamisme

Le dynamisme est une propriété forte de l'environnement ubiquitaire. En effet, ces environnements évoluent continuellement et peuvent subir des changements imprédictibles. Cette particularité est souvent liée aux préférences de l'utilisateur. Ainsi, son environnement évolue en se pliant à ses exigences et à ses besoins qui varient au cours du temps. Ce dernier peut, par exemple, ajouter de nouveaux dispositifs et en retirer d'autres. Il peut aussi installer de nouvelles applications qui répondent à des nouveaux besoins ou en désinstaller d'autres qui ne lui apportent plus de fonctionnalités intéressantes.

En outre, l'aspect dynamique des environnements ubiquitaires peut être aussi accordé à la mobilité des dispositifs. En effet, les dispositifs mobiles tels que les ordinateurs portables, les tablettes ou les téléphones mobiles peuvent être déplacés par l'utilisateur. Ce dernier peut ainsi transporter ces appareils et changer leur localisation à tout moment. Par conséquent, cette localisation dépend fortement des envies et du rythme de vie de l'utilisateur. Cet aspect de mobilité des dispositifs soulève le problème de la connectivité au réseau. Ainsi, ces appareils mobiles peuvent apparaître et disparaître du réseau de manière imprévisible et en fonction de la localisation de leur utilisateur.

Par ailleurs, la variation de l'environnement peut être marquée par les propriétés des dispositifs. Ces dispositifs présentent souvent des caractéristiques techniques qui influencent généralement leurs fonctionnalités et leurs capacités de communication. Par exemple, le niveau de batterie d'un capteur de mouvement renseignant sur sa durée de vie est indispensable pour son bon fonctionnement. En-dessous d'un seuil, des dysfonctionnements sont possibles et le dispositif peut ne plus être accessible.

L'évolution est un principe clé de l'environnement ubiquitaire. Cette évolution n'est pas limitée à la volatilité des ressources matérielles et logicielles mais elle peut être engendrée par l'interaction de l'utilisateur avec son environnement. En effet, cette interaction va agir sur les mesures et les données collectées de l'environnement, les états des dispositifs qui entourent l'utilisateur ainsi que les comportements des applications. Etant donné que les applications ubiquitaires sont des applications sensibles au contexte, elles doivent s'adapter constamment à l'évolution de l'environnement. Cet aspect sensible au contexte des applications est aujourd'hui l'un des défis majeurs de recherche dans les environnements ubiquitaires qui sont des environnements dynamiques où le contexte change fréquemment [BDR07; Wan+92; Bet+10; CK00].

1.3 Caractéristiques des applications ubiquitaires

Comme nous l'avons déjà présenté dans la section 1.2, les environnements ubiquitaires imposent des exigences qui doivent être respectées lors de la construction d'applications pouvant y être déployées. La construction de ces applications s'avère complexe par rapport à celle d'applications traditionnelles à cause de plusieurs besoins devant être satisfaits. Dans cette section, nous présentons les caractéristiques clés des applications ubiquitaires permettant de les distinguer des applications traditionnelles. Une étude autour de ce sujet est déjà menée dans [Ban+00].

1.3.1 Gestion des ressources

Sacha Krakowiak [Kra07] définit une ressource par la phrase suivante :

« *The term resource applies to any identifiable entity (physical or virtual) that is used by a system for service provision.* » [Kra07]

Les environnements ubiquitaires intègrent des dispositifs hétérogènes capables d'apparaître et de disparaître d'une façon imprédictible. Ces dispositifs sont exploités par les applications ubiquitaires en tant que ressources externes afin de fournir des services à forte valeur ajoutée à leurs utilisateurs. En effet, chacun de ces dispositifs présente un ensemble de fonctionnalités qui sont explicitement exposées pour les applications ubiquitaires. Certaines fonctionnalités sont exclusives, elles ne peuvent être utilisées que par une seule application à la fois. D'autres fonctionnalités sont partagées et peuvent être utilisées par plusieurs applications simultanément. Un exemple de ces fonctionnalités est la configuration qui consiste à modifier certains paramètres de ces dispositifs. En effet, les paramètres configurés par les applications sont en général les paramètres qui sont associés aux services rendus par les dispositifs. La modification de la valeur d'un paramètre affecte ainsi toutes les applications qui l'utilisent. En conséquence, l'accès à ces dispositifs depuis des applications doit se faire dans un cadre contrôlé de manière à éviter les configurations et les ordres contradictoires nuisant au bon fonctionnement des services.

Toutefois, le nombre important des ressources dispersées dans les environnements ubiquitaires, leur aspect dynamique et leur nature hétérogène rendent la gestion des ressources l'un des enjeux majeurs dans le domaine de l'informatique ubiquitaire. Il s'avère ainsi difficile de gérer l'accès aux ressources, les autorisations et les restrictions, les accès illégaux ainsi que les utilisations abusives.

1.3.2 Orientation donnée

Les services fournis par les applications ubiquitaires sont marqués par l'état de l'environnement de l'utilisateur. En effet, ces applications sont conçues pour inter-opérer avec les dispositifs qui l'entourent : elles sont ainsi amenées à collecter les données et les mesures depuis les capteurs dispersés dans leur environnement, les filtrer et les traiter puis agir sur les actionneurs pour fournir des services à valeur ajoutée à l'utilisateur. Par conséquent, le comportement de ce type d'applications dépend fortement des données collectées par les capteurs dispersés dans leur environnement. Les applications ubiquitaires sont ainsi des applications sensibles aux événements qui comportent des données fournies par diverses sources. Ce type de dépendance des applications ubiquitaires impose un schéma de programmation orientée données où le service fourni par une application est une réponse à un événement qui contient des données.

1.3.3 Notion de contexte

La sensibilité au contexte est l'un des principes clé de l'informatique ubiquitaire où le contexte d'une application peut être considéré comme étant l'ensemble des informations nécessaires pour son exécution. D'après [SAW94], le contexte peut être représenté en trois classes :

- **le contexte utilisateur** représente toutes les informations liées à l'utilisateur d'un système : son identité, sa langue, ses préférences, sa position géographique, etc.
- **le contexte virtuel** contient toutes les propriétés qui caractérisent un système informatique, comme, par exemple, les ressources nécessaires, la bande passante de la connexion réseau, etc.
- **le contexte physique** inclut toutes les grandeurs physiques de l'environnement de l'utilisateur ; tels que le niveau de bruit, la température, la luminosité, etc.

Les contextes virtuel et/ou utilisateur ne sont pas des nouveaux concepts pour les systèmes informatiques. En effet, ils sont déjà utilisés dans les applications traditionnelles. Un bon exemple d'applications traditionnelles illustrant l'utilisation de contexte est les applications Web. Pour ce type d'applications, le contexte peut inclure les navigateurs utilisés pour manipuler l'application ainsi que les cookies qui permettent de collecter les pages Web visitées par l'utilisateur et de construire par conséquent son profil.

Par opposition aux applications traditionnelles, les applications ubiquitaires peuvent dépendre des attributs physiques de l'environnement de l'utilisateur tels que la température, le niveau sonore ou l'intensité lumineuse. Ces applications sont ainsi amenées à capter les mesures et les données brutes depuis les dispositifs présents dans leur environnement et à les agréger pour construire des données globales reflétant l'état physique de l'environnement.

Dans [Abo+99], les auteurs classent les fonctionnalités qu'une application sensible au contexte doit prendre en charge en trois catégories :

- la présentation d'informations ou de services à l'utilisateur : l'application doit être habilitée à fournir des informations ou des services à l'utilisateur d'une façon pertinente.
- l'exécution automatique d'un service : un service doit pouvoir exercer des actions qui peuvent impacter l'environnement. Il doit en effet pouvoir agir sur les actionneurs dispersés en réponse aux données collectées du contexte.
- marquer des informations avec des éléments de contexte pour un usage ultérieur : les données brutes remontées par les capteurs parsemés dans l'environnement ne sont pas les seules informations qui peuvent être utilisées par les applications ubiquitaires mais leur analyse, leur agrégation et leur interprétation peuvent aussi leur être très utile et les exonérer de ce raisonnement.

D'après cette classification, il est clair que les applications sensibles au contexte ont besoin d'extraire des données depuis ce contexte et d'y réagir en conséquence en exécutant des actions. Selon [VSB13] , il est alors nécessaire d'avoir un moyen qui supporte les interactions bidirectionnelles entre les applications ubiquitaires et leur contexte.

D'autres classifications des informations contextuelles se présentent. Par exemple, dans [Hen03], cette classification s'appuie sur les propriétés de la source de l'information. Un autre exemple de classification est présenté dans [Bet+10]. Celui-ci se base sur le niveau d'abstraction de l'information de contexte (voir la figure 1.2).

Selon ces classifications, il est clair que les informations de contexte peuvent être brutes issues directement des capteurs dispersés dans l'environnement ou bien plus abstraites et plus significatives déduites à partir des informations brutes à travers différents moyens de raisonnements.

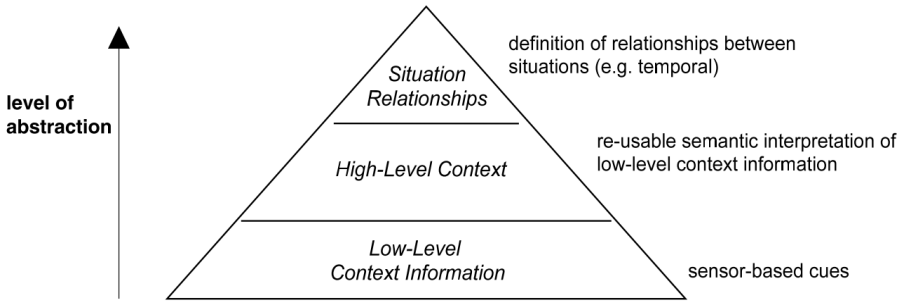

Figure 1.2: Niveau d'abstraction des informations de contexte [Bet+10].

1.3.4 Adaptabilité

Les applications ubiquitaires sont des applications sensibles au contexte par définition. En effet, une application ubiquitaire doit s'adapter dynamiquement à l'évolution constante de son contexte. Elle doit ainsi répondre automatiquement aux exigences de son utilisateur et satisfaire ses besoins face aux variations de son environnement. Il s'agit donc d'adapter son comportement en fonction de sa localisation, les capacités et la disponibilité des dispositifs et les caractéristiques de son environnement, etc. Pour supporter l'adaptation dynamique aux divers contextes d'utilisation, les applications ubiquitaires doivent être définies de manière flexible dans le but de pouvoir changer leurs configurations en fonction des variations du contexte.

Un exemple révélateur qui illustre l'adaptation d'applications ubiquitaires dans un contexte dynamique est le cas où la réponse de l'application doit suivre son utilisateur mobile. Dans cette situation, la réponse sonore ou visuelle de l'application doit migrer des dispositifs vers d'autres en suivant le déplacement de l'utilisateur. Cependant, cette migration doit être effectuée de façon autonome et surtout de manière transparente pour l'utilisateur.

L'évolution constante de contexte provoque une forte complexité à la construction des applications. En effet, elle exige une modification de certaines habitudes concernant le développement et l'exécution des applications informatiques. Lors de la phase de conception, il s'avère important de définir les différents contextes d'exécution puis de spécifier et de développer les différentes configurations possibles d'une application. Ce contexte peut inclure, par exemple, les différents types de dispositifs nécessaires pour l'application. A l'exécution, il est exigible de mettre en œuvre la reconfiguration dynamique de l'application dans le but de l'adapter à l'évolution de son contexte d'exécution. En outre, l'aspect fortement dynamique des environnements ubiquitaires exige que la reconfiguration de l'application doit être réalisée de manière autonome.

1.3.5 Sécurité

La sécurité est l'un des enjeux majeurs des systèmes ubiquitaires. Il s'avère important de définir des politiques et de mettre en place des mécanismes permettant d'assurer la sécurité de ces systèmes. L'authentification et la confidentialité sont deux aspects principaux pour la sécurité de tous les systèmes informatiques. Des protocoles d'authentification et des politiques d'autorisation et de contrôle d'accès peuvent être ainsi utilisés pour assurer la vérification de l'identité de l'utilisateur et contrôler l'accès aux ressources.

En outre, les systèmes ubiquitaires présentent des risques pour la vie privée de leurs utilisateurs. En effet, les informations fournies par les différentes sources peuvent contenir des données de nature personnelle. Il s'avère ainsi important de mettre en place des mécanismes de sécurité qui permettent de protéger la vie privée des utilisateurs et d'en assurer la confidentialité.

L'aspect dynamique et ouvert des environnements ubiquitaires rend la gestion d'accès très compliquée et peut conduire à gérer la sécurité et contrôler l'accès des applications aux ressources dans plusieurs niveaux du système ubiquitaire même aux niveaux des middleware.

1.3.6 Synthèse

Les développeurs d'applications ubiquitaires affrontent des défis difficiles pour concevoir et développer leurs applications. Ces difficultés sont issues principalement d'un ensemble de problèmes liés aux environnements ubiquitaires (hétérogénéité, dynamisme, distribution, ouverture) mais liés aussi aux applications (gestion des ressources, gestion de l'adaptabilité, gestion des données et de la sécurité). Il s'avère ainsi important de mettre en place des outils pour l'informatique ubiquitaire, particulièrement des plates-formes ubiquitaires, qui permettent de faciliter la tâche aux développeurs et de les guider tout au long des différentes phases de construction de leurs applications. Ces plates-formes ont pour but de traiter un ensemble d'aspects non-fonctionnels de l'informatique ubiquitaire tels que les caractéristiques des environnements et les propriétés des applications. De nombreux chercheurs et industriels se sont penchés sur la création de ces plates-formes ubiquitaires. Nous étudions certaines d'entre elles dans le troisième chapitre de ce manuscrit.

1.4 Cycle de vie d'application ubiquitaire

Le cycle de vie d'une application présente plusieurs étapes clés qui nécessitent des compétences différentes [Sch11]. Ces phases sont les suivantes :

Analyse des besoins : c'est une phase primordiale pour la réalisation d'un projet logiciel. Elle sert à identifier et à décrire sans ambiguïté l'ensemble des exigences qu'un système doit satisfaire. Il s'agit donc d'identifier tous les acteurs qui vont interagir avec le système et de définir les besoins fonctionnels ; c'est-à-dire l'ensemble des fonctionnalités qu'un système doit offrir en réponse aux demandes de son utilisateur ainsi que les besoins non-fonctionnels tels que la performance, la fiabilité, la portabilité et l'utilisabilité du système, etc. Pour les méthodologies de gestion de projet traditionnelles, la phase d'analyse et de spécification des besoins est première et unique. Toutefois, pour les méthodologies récentes telles que les méthode agiles, cette phase est découpée en plusieurs parties relativement indépendantes afin d'adapter les fonctionnalités du système en fonction des besoins évolutifs du client.

Conception : la phase de conception permet de décrire d'une manière non ambigüe l'architecture d'un système afin d'en faciliter la réalisation. Elle sert ainsi à décrire la structure générale du système, l'organisation de ses différentes parties (composants ou modules) ainsi que les relations entre eux. Le choix de l'architecture du système pendant cette phase de cycle de vie va guider sa réalisation. En général, les concepteurs visent à rendre les constituants du système aussi indépendants que possible afin d'assurer le développement parallèle et de faciliter les maintenances futures.

Développement : cette phase constitue le cœur du processus de développement du système. Elle inclut principalement l'activité de programmation. Elle consiste ainsi à réaliser tous les constituants (composants ou modules) du système qui sont définis lors de la conception. Ces constituants doivent être conformes aux spécifications décrites pendant la phase de conception.

Déploiement : cette phase vise à placer les différents constituants d'un système dans leur environnement et préparer leur exécution. Contrairement aux systèmes dédiés au grand public, cette phase fait partie du processus de développement logiciel pour les systèmes dédiés par exemple aux grandes entreprises ; ce type de système nécessite une livraison ainsi qu'une installation sur site.

Exécution : dans cette phase, le système doit répondre impérativement aux exigences décrites au cours de la phase d'analyse et de spécification des besoins. Par ailleurs, à l'exécution, le système doit avoir une architecture valide par rapport à son architecture de conception. Toutefois, des dysfonctionnements peuvent se produire et le

système peut même cesser de fonctionner. Le traitement de ces problèmes ainsi que la réparation du système doivent être effectués pendant la phase de maintenance.

Tests : plusieurs tests peuvent être effectués permettant ainsi de valider les différentes phases du cycle de vie du système :

- **des tests fonctionnels** sont effectués afin de prouver que le système réalisé accomplit bien toutes les fonctionnalités décrites pendant la phase d'analyse et de spécification des besoins.
- **les tests unitaires** consistent à tester individuellement les différents composants du système réalisé pendant la phase de développement afin de vérifier leur bon fonctionnement.
- **les tests d'intégration** permettent de valider l'intégration des différents composants entre eux, suivant l'architecture globale du système définie lors de la conception.

Le génie logiciel propose plusieurs méthodes, techniques et outils (bibliothèques partagées, compilateurs et interpréteurs de code, outils de tests, etc.) qui couvrent les différentes phases du cycle de vie d'applications, dans le but de faciliter leurs constructions. Ces moyens diffèrent principalement selon les étapes du cycle de vie ainsi que les approches adoptées pour construire le système.

Dans cette thèse, nous nous intéressons particulièrement aux approches basées sur des middleware. Ces approches ont pour objectif de traiter un ensemble de problématiques qui peuvent être liées aux différentes phases clés du cycle de vie de l'application. Les middleware visent donc à réduire le cycle de vie des applications en rendant leur conception, leur développement et leur exécution plus facile et plus rapide. Ils peuvent, par exemple, assister le développeur durant le développement de ses applications en proposant des modèles de développement simples.

Les applications ubiquitaires présentent un ensemble d'exigences et de propriétés qui rendent leur conception, leur développement et leur exécution extrêmement complexes. Cette complexité exige des compétences particulières et un haut niveau d'expertise. Il provoque aussi l'allongement du processus de développement. Il s'avère ainsi important de mettre en place un ensemble d'outils de développement et de middleware qui permettent de faciliter la tâche aux développeurs et de les guider tout au long des différentes phases de construction de leurs applications.

1.5 Maison intelligente

La maison intelligente ou en anglais *Smart Home* est l'exemple d'environnement ubiquitaire auquel nous nous intéressons dans cette thèse. Son objectif principal est d'améliorer la qualité de vie des habitants en fournissant des services à haute valeur ajoutée. Dans cette section, nous présentons les définitions de la maison intelligente proposées dans la littérature. Ensuite, nous discutons ses défis et ses domaines d'applications.

1.5.1 Définition

D'après [Can92; Gal12], Ken Sakamura a énoncé un ensemble de critères d'exclusion afin de définir la maison intelligente :

« Une maison sera disqualifiée au regard du classement dans la catégorie des maisons intelligentes si :

- l'information ne peut pas circuler librement de l'intérieur de la maison vers le monde extérieur, et vice-versa ;
- si la maison fonctionne avec des ordinateurs intégrés qui ne peuvent pas se parler entre eux ;
- si sa domotisation consiste en un « patchwork » de « gadgets » ;
- si elle est équipée avec des fonctions sophistiquées difficiles à utiliser. »

Les critères énoncés par Ken Sakamura mettent en lumière deux types de concepts fondamentaux pour définir une maison intelligente. Le premier est purement technique qui consiste en la mise en place d'une infrastructure de communication permettant de faire circuler les informations entre les dispositifs de la maison d'une part et entre la maison et le monde extérieur d'autre part. Le second porte principalement sur la simplicité et la facilité de l'utilisation.

Aldrich [Ald03] définit une maison intelligente comme « une résidence équipée de technologies informatiques qui anticipent et répondent aux besoins de ses occupants en essayant de promouvoir leur confort, leur bien-être, leur sécurité et leur détente grâce à la gestion de la technologie à l'intérieur de la maison et des connexions avec le monde extérieur. »

Selon [Jeu05], une maison intelligente est « une maison qui dispose de fonctionnalités susceptibles de simplifier la vie de ses habitants au quotidien, de réaliser des économies d'énergie et d'apporter un certain niveau de confort et de sécurité. Elle est ouverte aux évolutions futures par la nature même de ses infrastructures de câblage et par son ouverture au monde numérique. »

1.5.2 Défis

Au cours des dernières décennies, plusieurs travaux de recherches se sont penchés sur le développement des maisons intelligentes afin d'améliorer la qualité de vie. Toutefois, avant que le concept de la maison intelligente soit implanté en situation réelle, un ensemble de défis doivent être relevés. Ces défis ont été au cœur des travaux de plusieurs chercheurs.

Edwards, dans son article [EG01], présente sept défis majeurs qui doivent être surmontés pour une maison intelligente. Ces défis couvrent principalement les problèmes qui résultent de la manière attendue des maisons intelligentes à être déployées et habitées. Selon [BBL], les défis proposés par Edwards peuvent être divisés en deux classes :

- **besoins techniques :** la maison intelligente doit relever tous les défis associés à son aspect technique tels que la fiabilité, l'interopérabilité et la capacité de fonctionner en présence d'ambiguïtés.

- **besoins humains :** une maison intelligente doit intégrer un ensemble de fonctionnalités et de services qui doivent être faciles à comprendre et à utiliser par son habitant. Aussi, ces fonctionnalités doivent avoir des interfaces attractives respectant ainsi l'ergonomie qui doit être adaptée à un environnement domestique. De plus, la maison intelligente doit respecter le contexte social de ses habitants.

Dans leur article [Cha+08], Chan et al. présentent un ensemble de défis qu'une maison intelligente doit résoudre, dans un contexte de télémédecine et de maintien à domicile. Ces défis portent non seulement sur des critères utilisateur tels que les besoins, l'acceptabilité, la satisfaction et l'ergonomie mais aussi sur des critères technologiques qui sont liés principalement à la fiabilité et l'efficacité des infrastructures, des capteurs portés par les habitants ainsi que les applications dédiées à leur surveillance.

1.5.3 Domaines d'application

Avec la forte croissance technologique, les maisons sont devenues intelligentes s'adaptant de plus en plus aux rythmes, aux besoins et aux envies de leurs habitants. Les maisons intelligentes permettent d'améliorer les conditions de vie en augmentant le confort de l'habitant. Grâce aux technologies intégrées dans les maisons, il est possible aujourd'hui de piloter l'éclairage, la température ainsi que différents appareils électroniques.

En plus du confort, la maison intelligente apporte une réponse à un autre défi majeur : l'économie d'énergie. L'habitant est aujourd'hui habilité à contrôler sa consommation d'énergie et à optimiser son budget énergétique. Il est désormais possible de contrôler les différentes ressources énergétiques telles que les lumières et les chauffages et d'adapter leurs utilisations en fonction des besoins et des rythmes des habitants dans le but de réduire les dépenses d'énergie.

De plus, assurer la sécurité et la protection des habitants est l'un des objectifs principaux de la maison intelligente. Il est désormais possible de sécuriser de façon efficace la maison et de protéger les habitants ainsi que leurs biens. Grâce aux solutions domotiques dédiées à la sécurité, il est possible de protéger la maison contre les différents dangers et menaces tels que les intrusions, les incendies, les inondations, etc. L'habitant est ainsi capable de surveiller sa maison à distance et il est averti immédiatement en cas de problème.

Le vieillissement démographique peut réorienter les technologies existantes dans les maisons intelligentes pour répondre aux besoins particuliers de cette catégorie de la population. Les innovations technologiques intégrées dans les maisons aujourd'hui permettent de redonner une véritable autonomie aux personnes âgées et handicapées, de suivre leur santé et d'améliorer leur qualité de vie

Actuellement, plusieurs projets de recherche se sont consacrés à étudier les besoins des personnes âgées fragilisées qui souffrent d'une perte d'autonomie. De nombreuses études se sont penchées sur les technologies dédiées à la santé des personnes âgées [MBT08]. Le défi majeur de ces études de recherche est de construire un habitat qui permet de garantir le confort, la sécurité, la bonne santé et surtout l'indépendance et l'autonomie à ces personnes. Grâce aux nouvelles technologies, les personnes âgées, ayant une autonomie limitée, peuvent garder un lien avec le monde extérieur et communiquer à tout moment avec leurs proches et les personnes de leur entourage, ce qui facilite ainsi leur assistance et leur maintien à domicile [Rom+12].

Les solutions pour les maisons intelligentes sont actuellement en plein essor et s'étendent de plus en plus à une grande variété de domaines afin d'améliorer la qualité de vie de l'habitant. Le marché du *smart home* est un marché diversifié qui intègre des solutions domotiques dédiées pour le confort, l'économie d'énergie, la sécurité et même pour la santé. Ces solutions peuvent être automatiques ou programmables par l'utilisateur, utilisées en présence et en l'absence de l'habitant, et aussi contrôlées depuis la maison ou à distance.

Cependant, la maturité actuelle des offres domotiques n'a pas atteint son optimum. En effet, il existe actuellement un frein technique dû aux solutions fermées qui dominent le marché du *smart home*. Ces solutions proposées aujourd'hui reposent essentiellement sur le principe de couplage fort entre l'application et les dispositifs (mono fournisseur, le même protocole et la même marque).

Aujourd'hui, la révolution technologique favorise le terrain pour l'entrée de puissants acteurs tels que les opérateurs de télécommunications et les groupes de l'informatique, ce qui impose un jeu concurrentiel intense entre ces différents acteurs. Dans ce contexte concurrentiel actuel, il s'avère important de respecter la propriété d'ouverture du *smart home* et de permettre la cohabitation entre les différents fournisseurs des services. Désormais, une passerelle domotique devra être ouverte non seulement aux marques et aux différents protocoles mais elle devra aussi héberger diverses applications délivrées par plusieurs tiers afin de fournir des services plus riches à l'utilisateur final.

1.6 Conclusion

Dans l'étude présentée dans ce premier chapitre, nous pouvons conclure que :

- l'informatique ubiquitaire est une nouvelle vision apparue suite à l'évolution de l'informatique. Elle vise en effet à fusionner les environnements informatiques avec le monde réel. L'objectif principal de cette vision est de fournir aux utilisateurs des services à forte valeur ajoutée de manière naturelle et transparente.
- les développeurs d'applications ubiquitaires doivent répondre à des exigences qui rendent leurs tâches particulièrement difficiles et complexes. Ces exigences résultent de la nature des environnements ubiquitaires (hétérogène, dynamique, distribué, ouvert) imposant de nouvelles propriétés pour les applications déployées.
- les *smart home* est un environnement ubiquitaire en plein essor où des applications appartenant à des domaines divers (la sécurité, le confort, l'énergie et la santé, etc.) peuvent être installées et exécutées afin d'améliorer la qualité de vie des habitants.

La plupart des solutions proposées aujourd'hui dans le marché du *smart home* reposent sur une plateforme domotique qui peut héberger ces applications ubiquitaires. Cette plateforme a pour objectif de traiter un ensemble des aspects non fonctionnels (par exemple, les problèmes liés à l'environnement ubiquitaire tels que la gestion de l'hétérogénéité et de dynamisme) afin de faciliter le développement de ces applications ubiquitaires et d'assurer aussi leur cohabitation. Une étude approfondie sur ces plateformes sera présentée dans le troisième chapitre.

L'aspect non fonctionnel que motive ce travail de thèse est la gestion des conflits qui peuvent se produire entre les applications hébergées dans les plateformes domotiques. En effet, les applications appartenant à des domaines différents vont partager les dispositifs, tant capteurs que actionneurs, présents dans leur environnement. Elles peuvent agir de façon contradictoire sur des actionneurs partagés et elles risquent ainsi d'être en conflit. Dans certaines situations de conflit, elles risquent de mettre la maison, les habitants et leurs biens dans un danger. Une étude minutieuse qui s'articule autour des solutions existantes de la gestion des conflits sera le cœur du chapitre suivant de ce manuscrit.

Chapter 2

Gestion des conflits et d'accès

Sommaire

Ce chapitre sera divisé en deux grandes parties. La première partie sera consacrée à une étude sur la gestion des conflits. Nous commencerons, tout d'abord, par présenter l'origine de ce problème et analyser les différentes définitions utilisées dans la littérature. Nous exposerons, ensuite, les différentes classes des approches proposées pour la gestion des conflits. Trois classes seront identifiées : la prévention qui consiste à chercher les moyens pour prévenir les conflits, la détection qui consiste à identifier et localiser les conflits, puis la résolution qui se penche sur les mécanismes pour minimiser leurs effets potentiels. Afin de limiter le périmètre de notre contribution, nous présenterons une étude comparative des différentes solutions existantes pour la gestion des conflits dans les environnements ubiquitaires multi-applications. Le contrôle d'accès étant indispensable pour parvenir à gérer les conflits entre les applications, nous consacrons la deuxième partie de ce chapitre à une étude détaillée des techniques proposées pour les gérer. Nous conclurons que le mécanisme de verrouillage est la technique la plus adaptée pour contrôler les accès dans les environnements ubiquitaires grâce à leurs caractéristiques. Nous nous appuierons alors sur cette technique pour traiter notre problème de gestion des conflits.

2.1 Introduction

Les applications traditionnelles se centrent sur les machines sur lesquelles elles sont exécutées dans la mesure où elles sont bien limitées par les ressources que ces machines fournissent. Ces applications s'exécutent alors en utilisant un ensemble de ressources spécifiées à l'avance pendant la phase de conception. Toutefois, l'émergence de l'informatique ubiquitaire a entraîné une refonte globale de cette vision. Dans la nouvelle vision, les applications sont plutôt centrées sur l'utilisateur et se concentrent surtout sur ses besoins. Afin de les satisfaire, les applications ubiquitaires sont amenées à découvrir et à utiliser des ressources hétérogènes telles que les dispositifs dispersés dans leur environnement ou des services distants.

Les applications traditionnelles s'exécutent généralement sur des systèmes d'exploitation qui font l'abstraction des différentes ressources réelles par des ressources virtuelles (par exemple, la mémoire physique est abstraite par la mémoire virtuelle). L'abstraction de ces ressources est un aspect important permettant de simplifier leur utilisation. De plus, ces systèmes fournissent un ensemble de fonctionnalités contrôlant leurs utilisations afin d'éviter les conflits d'accès entre les applications. Par exemple, le système de fichiers, étant une abstraction du disque, intègre des fonctionnalités pour contrôler les accès simultanés dans le but de gérer les éventuels conflits.

Typiquement, les dispositifs peuvent être considérés comme des ressources primaires utilisées par les applications pervasives. Les plateformes logicielles sur lesquelles ces applications s'exécutent s'avèrent le lieu le plus convenable où ces ressources doivent être abstraites et les conflits doivent être traités. Le contrôle d'accès est l'approche la plus privilégiée pour parvenir à gérer ces conflits. Comme les systèmes d'exploitation, ces plateformes doivent alors présenter cette capacité de gestion d'accès afin de parvenir à diriger l'utilisation des ressources et à gérer les conflits pouvant se produire à l'exécution. Toutefois, les politiques de gestion d'accès à intégrer dans ces plateformes doivent être adaptées à la nature hétérogène et dynamique des ressources.

L'étude présentée dans ce chapitre s'articule autour de deux axes : le premier axe se concentre sur les conflits qui peuvent se présenter entre les applications ubiquitaires et les approches proposées pour les gérer. Le deuxième axe est mené autour des techniques de contrôle d'accès.

2.2 Gestion des conflits

2.2.1 Définitions

Le problème des conflits est un sujet de plus en plus important de nos jours. La première notion de conflit, appelée en anglais *feature interaction*, a été introduite dans le domaine des télécommunications où un service est défini comme une collection de fonctionnalités. Le problème de conflit se manifeste lorsqu'une fonctionnalité, qui fonctionne correctement en isolation, se comporte d'une manière incorrecte en présence d'autres fonctionnalités. D'une manière générale, un conflit se produit lorsque le comportement d'une fonctionnalité est modifié, du fait de sa coexistence avec d'autres fonctionnalités. La gestion des conflits a été au cœur de plusieurs travaux de recherche où de multiples définitions des conflits ont été proposées. Dans ce qui suit, nous citons les définitions qui font aujourd'hui figure de référence dans le domaine des télécommunications :

« *A feature interaction occurs when the behavior of one feature is affected by the behavior of another feature or another instance of the same feature.* » [KV95]

« *Feature interaction is the term used to describe interference between services or features.* » [CM00]

L'exemple de conflit le plus connu dans le domaine des télécommunications est celui qui peut se produire entre les fonctions de transfert d'appel et l'appel en attente. Ces deux fonctions ont deux effets différents sur le deuxième appel reçu par un client déjà au téléphone. En effet, le transfert d'appel permet de renvoyer les appels entrants vers un autre numéro de téléphone spécifié à l'avance, tandis que l'appel en attente permet de notifier le client qu'il y a un deuxième appel s'il est déjà au téléphone et lui permet de passer d'un appel à l'autre. Un conflit se produit ainsi entre ces deux fonctions lorsqu'elles sont activées sur la même ligne téléphonique et un deuxième appel (ressource partagée) arrive alors que le premier appel n'est pas encore terminé. Dans ce cas, le système doit impérativement prendre une décision pour résoudre ce conflit : soit que le deuxième appel sera renvoyé vers un numéro secondaire (transfert d'appel) ou que le client sera averti qu'un autre appel est arrivé (appel en attente). La décision optimale qui doit être prise par ce système téléphonique dépend principalement des besoins du client.

Pendant les dernières décennies, le problème de conflit a été particulièrement aigu dans le domaine télécommunications où le nombre et la complexité des services intégrés dans les systèmes a été de plus en plus accentué. Toutefois, ce problème a dépassé le spectre des télécommunications et il s'est manifesté dans d'autres domaines. D'une manière générale, le problème des conflits peut être identifié dans tous les domaines où certaines ressources peuvent être partagées par plusieurs consommateurs (entités logicielles) qui doivent cohabiter. Ceci est résumé dans les phrases suivantes :

« *These interactions can usually be traced down to the fact that "two 'features' manipulate the same entities in the base system, and in doing so violate some underlying assumptions about these entities that the other 'features' rely on* » [PR98]

« *The subject has relevance to any domain where separate software entities control a shared resource.* » [CM00]

Aujourd'hui, le problème de conflit est devenu l'un des enjeux majeurs dans les environnements informatiques ubiquitaires où le nombre d'applications et leur complexité sont en plein essor. En effet, les environnements ubiquitaires sont des environnements ouverts par définition. Ils doivent, par conséquent, assurer la cohabitation des applications fournies par des acteurs différents. Ces applications partagent les dispositifs, tant capteurs que actionneurs, présents dans leur environnement. Cependant, elles peuvent agir de manière contradictoire sur les actionneurs partagés et elles risquent d'être en conflit. De par le nombre important des ressources dispersées dans les environnements ubiquitaires, leur aspect dynamique et leur nature hétérogène, la gestion des conflits est devenu un défi très complexe et très difficile à relever dans le domaine de l'informatique ubiquitaire. Certes, la gestion des conflits dans les environnements ubiquitaires multi-applications est aujourd'hui un véritable défi qui suscite de nombreux projets industriels et travaux de recherche. Dans le cadre de ces travaux, plusieurs définitions de conflit ont été proposées. La plupart de ces définitions sont plus précises que celles déjà présentées pour donner une vision générale des conflits utilisée dans tous les domaines. Ces définitions sont particulièrement dédiées à définir la notion de conflit dans les environnements ubiquitaires. Dans ce qui suit, nous analysons les principales définitions présentées dans la littérature :

« *A conflict occurs for an application or a user, if a context change – caused by an application – leads to a state of the environment which is considered inadmissible by the application or user.* » [TSB07]

« *A resource usage raises a potential conflict when two or more controllers may access it. These controllers may be defined within an application or across applications.* » [JCL11]

D'après les définitions citées précédemment , nous pouvons identifier, dans les environnements ubiquitaires multi-applications, deux types de conflits. Le premier type de conflit se manifeste au niveau des actionneurs dispersés dans l'environnement. Il se produit lorsque deux applications agissent sur le même actionneur de manière différente, par exemple allumer et éteindre la même lampe. Le deuxième type de conflit se présente au niveau de l'environnement. Il se produit lorsque deux applications agissent sur deux actionneurs différents mais leurs effets sur l'environnement sont contradictoires ou s'interfèrent. Par exemple, un conflit d'environnement peut avoir lieu lorsque deux applications agissent sur la température de la même pièce de manière contradictoire (augmenter et réduire la température).

A titre d'illustration, nous considérons une plateforme domotique qui héberge deux applications de sécurité : une application de détection d'incendie et une application de détection d'intrusion. Dans le cas d'un incendie, la porte d'entrée de la maison doit être déverrouillée par l'application de détection d'incendie alors que la nuit, la porte doit être verrouillée par l'application de détection d'intrusion. La nuit, dans le cas d'un incendie, ces deux applications agissent sur la porte de façon contradictoire. Un conflit sera ainsi détecté entre ces deux applications au niveau de la porte d'entrée.

Considérons un deuxième exemple de conflit où la plateforme domotique intègre deux applications dédiées au divertissement. La première application est *Movies Application* qui permet à l'utilisateur de regarder un film lancé sur le téléviseur installé par exemple dans le salon. La deuxième application est *Music Application* qui s'exécute en utilisant un haut-parleur installé dans la même pièce. Dans cet exemple, les deux applications n'utilisent pas les mêmes dispositifs. Néanmoins, elles affectent le niveau sonore de du salon et les sons issus de ces applications s'interfèrent. Il s'agit donc de conflit de l'environnement.

En conclusion, le problème de conflits est devenu aujourd'hui l'un des problèmes majeurs dans les environnements informatiques ubiquitaires particulièrement dans la maison intelligente qui doit assurer la cohabitation d'un ensemble d'applications appartenant à un champ croissant de domaines. Certes, plusieurs travaux de recherche se sont penchés sur ce problème et ils ont proposé des solutions pour prévenir, détecter et résoudre ces conflits. Une étude des différentes approches proposées pour la gestion des conflits dans le domaine de l'informatique ubiquitaire sera menée dans la sous-section 2.2.3 de ce chapitre.

2.2.2 Les classes des approches existantes

Comme nous l'avons déjà évoqué dans la section précédente, le problème des conflits a été identifié pour la première fois dans le domaine des télécommunications où plusieurs chercheurs se sont penchés sur ce problème proposant ainsi plusieurs approches pour l'analyse et l'identification des conflits. La plupart de ces approches ont été divisées par Cameron et Velthuijsen [CV93], puis affinées par Bouma et Velthuijsen [BV94] en trois classes : la prévention, la détection et la résolution. La prévention consiste à chercher les moyens pour prévenir les conflits. La détection suppose que les conflits seront présents et permet de déterminer les méthodes pour les identifier et les localiser. La résolution suppose que les conflits seront présents et détectés et se penche sur les mécanismes pour minimiser leurs effets négatifs potentiels.

- **La prévention :** la prévention est une étape importante pour la gestion des conflits qui se manifeste par la recherche de moyens et de techniques pour les prévenir. En général, les approches de prévention des conflits visent à développer des plateformes ou des environnements de création logicielle qui mènent à l'implantation des services moins exposés aux conflits.

- **La détection :** la détection est essentielle pour la gestion des conflits qui permet d'identifier et de localiser les conflits. En effet, étant donné qu'il est peu probable d'éviter tous les conflits, il s'avère primordial de fournir des techniques qui permettent de les détecter. Les approches de détection de conflits peuvent être appliquées *offline* (pendant la spécification, la conception, l'implémentation) et/ou *online* (pendant la phase de tests ou après le déploiement). La détection des conflits *offline* permet d'identifier et localiser les conflits potentiels qui peuvent se produire entre les services. La complexité de détection dépend du nombre de services impliqués et des situations dans lesquels ils peuvent entrer en conflit. En revanche, la détection *online* des conflits permet de considérer seulement les services invoqués dans des situations bien particulières.

- **La résolution :** Une fois que les conflits sont détectés, il faut les résoudre. En effet, la résolution des conflits se penche sur les mécanismes et les stratégies qui permettent de minimiser ou d'éliminer leurs effets indésirables potentiels. Tout comme la détection, la résolution des conflits peut être effectué pendant le processus de création des services (*offline*) ou lorsque les services sont opérationnels (*online*).

Cette classification des approches proposées pour la gestion de conflits en trois classes a été largement utilisée dans plusieurs domaines où le problème des conflits a été au cœur de plusieurs travaux de recherches. Ces approches peuvent intervenir dans les différentes

étapes du cycle de vie logiciel. En effet, elles peuvent être appliquées *offline* (pendant la spécification, la conception, l'implémentation) ou*online* (pendant la phase de tests ou après le déploiement) et même elles peuvent être des approches hybrides (combinaison de*offline* et *online*).

Dans [JD12], l'auteur définit les trois classes de gestion des conflits d'une manière plus générale et indépendamment du domaine : Le processus d'identification des conflits est appelé détection. Le processus qui permet de minimiser ou d'éliminer les effets indésirables d'un conflit est appelé résolution. La prévention est présentée comme étant une autre approche permettant d'éliminer les conflits et qui repose principalement sur les directives (*guidelines*) et les architectures pour les prévenir.

2.2.3 Les solutions existantes

Le problème de conflits est l'un des principaux défis dans les environnements *smart home* où le nombre d'applications et leur complexité sont en plein essor. En fait, ces environnements sont par définition ouverts. Il s'avère nécessaire d'assurer la coexistence d'applications fournies par différents acteurs. Néanmoins, le nombre élevé de ressources dispersées dans ces environnements, leur aspect dynamique et leur nature hétérogène font de la gestion des conflits un défi très complexe. Certes, de nombreuses recherches ont déjà porté sur cette problématique et plusieurs approches ont été proposées.

Parmi les solutions proposées, Nakamura [NIM05] présente une approche pour gérer les conflits au *design time*. Il définit les applications comme une séquence de méthodes de dispositifs et détecte les conflits entre elles par l'analyse du code. Il identifie alors deux types de conflits. (1) *Appliance interaction* se produit lorsque deux applications différentes invoquent des méthodes effectuant des mises à jour incompatibles pour les mêmes propriétés d'un dispositif. (2) *Environment interaction* se produit entre deux méthodes de deux dispositifs différents via l'objet de l'environnement présentant un ensemble de propriétés globales telles que la température, la luminosité ou le volume sonore. Cette solution ne peut être applicable que pour des applications simples, mais elle ne s'applique pas aux applications complexes comme les applications Java utilisées dans notre cas. L'approche proposée pour la résolution consiste à éviter l'installation d'applications susceptibles d'entraîner des conflits. En ce sens, cette solution peut être considérée comme pessimiste.

Dans [YIH15], Yagita présente une approche pour gérer les conflits au moment de l'installation (c'est-à-dire lorsque l'utilisateur installe des nouvelles applications alors qu'il

y a un ensemble d'applications déjà installées). Les conflits sont en effet détectés lors de l'installation. La stratégie utilisée pour les résoudre est de fournir à l'utilisateur toutes les informations nécessaires afin de le soutenir pour attribuer des priorités à ses applications selon les différentes situations des conflits. Les conflits détectés sont alors classés selon un certain nombre de situations significatives pour l'utilisateur afin qu'il puisse définir l'ordre de priorité de ses applications en fonction des informations fournies. Par conséquent, lorsque des nombreuses applications sont installées sur la plateforme, le nombre de situations de conflits devant être traitées par l'utilisateur peut être important. Nous croyons que la limite majeure de cette approche est que la résolution des conflits est la responsabilité de l'utilisateur. Nous pensons alors qu'il nécessaire de limiter l'intervention de l'utilisateur dans le processus de la gestion des conflits.

Dans son article [Par+05], Park propose une solution de gestion de conflits pouvant se produire entre applications sensibles au contexte qui desservent plusieurs utilisateurs. L'approche présentée permet en effet de détecter les conflits entre les applications à l'exécution et leur appliquer une politique de résolution afin de continuer à servir leurs utilisateurs sans distraire les unes des autres. Pour détecter les conflits, une représentation sémantique des comportements des applications en termes d'effets sur les attributs de contexte (la température, la luminosité, etc.) est proposée. Les conflits sont ensuite résolus par une politique d'adaptation générée dynamiquement en se basant sur les préférences de l'utilisateur. L'approche proposée se concentre sur la gestion des conflits survenus. Il est cependant souhaitable que les conflits soient détectés et résolus avant qu'ils ne surviennent.

Dans [MS14], les développeurs doivent spécifier des informations sur les dépendances de leurs applications sous la forme de métadonnées (les dispositifs utilisés, les effets sur l'environnement, les conditions déclenchant des actions, etc.). Ces métadonnées sont ensuite utilisées pour détecter et résoudre les conflits à l'installation et à l'exécution. Les applications sont classées en quatre groupes : énergie, santé, sécurité et divertissement. Les développeurs d'applications spécifient les groupes auxquels leurs applications appartiennent. Par défaut, les priorités des différentes catégories d'applications sont définies dans l'ordre suivant : santé> sécurité> divertissement> énergie. La stratégie utilisée par le système pour résoudre les conflits qui se produisent entre deux applications du même groupe n'est pas très précise. Il n'y a pas vraiment d'importance à savoir quelle application prend le contrôle. Toutefois, l'ordre de priorité entre ces applications peut être spécifié. En effet, les applications de la même catégorie peuvent être réparties dans des classes. La politique utilisée pour définir l'ensemble des classes et spécifier leur ordre de priorité n'est pas claire. De plus, le système proposé gère les conflits entre les applications agissant sur les actionneurs avec des conditions temporelles (par exemple, entre 20h et 21h). Cependant, ce système n'aborde pas les conflits qui peuvent survenir si les conditions qui

déclenchent les actions sont basées sur des événements. Il s'agit d'une limite majeure, notamment parce que les applications ubiquitaires sont sensibles au contexte et réagissent sur l'environnement en fonction des informations de contexte collectées depuis les dispositifs.

Kolberg et al. [KMW03] proposent une solution basée sur OSGi^{TM}. Ils définissent un certain nombre de conflits pouvant survenir dans les maisons intelligentes. Ils traitent ces conflits à travers un modèle de l'environnement et les effets potentiels des applications sur l'environnement. Cependant, cette proposition se concentre principalement sur les propriétés physiques (température ou luminosité, par exemple) et ne propose pas de mécanismes spécifiques pour aider les développeurs.

2.2.4 Synthèse

Le problème des conflits a été au cœur de plusieurs travaux de recherche dans le domaine de l'informatique ubiquitaire. Dans la sous-section précédente, nous avons étudié les solutions les plus représentatives qui ont été proposées pour résoudre ce problème. Bien que ces solutions visent à gérer les conflits entre les applications dans les environnements ubiquitaires multi-applications, elles le font à partir de points de vue différents. Par exemple, certaines approches traitent les situations de conflits où plusieurs utilisateurs se concurrencent pour utiliser la même ressource via des applications différentes ou la même application pour répondre à des besoins différents. Toutefois, d'autres approches se concentrent plutôt sur le traitement des conflits entre les applications sans considérer ceux pouvant se présenter entre les utilisateurs. Dans ce cadre, nous pensons que la *smart home*, même si elle sert plusieurs utilisateurs, elle doit avoir un seul administrateur local qui dirige le comportement de la maison. Il s'avère alors primordial de traiter le problème de base qui consiste à gérer les conflits pouvant se produire entre les applications, et non pas entre les utilisateurs.

Nous avons aussi pu constater que certaines solutions sont préventives visant à anticiper les conflits et à les traiter a priori alors que d'autres les détectent dès leurs occurrences et les résolvent en éliminant leurs effets indésirables présentés dans l'environnement. Cependant, l'apparition effective des conflits est considérée comme une situation inadmissible par l'utilisateur. Il est alors souhaitable de les gérer avant leur occurrence.

Le tableau 2.1 présente les solutions décrites précédemment en les positionnant par rapport aux catégories des approches utilisées pour la gestion des conflits ainsi que les différentes phases de cycle de vie des applications. La détection et la résolution des conflits avant leur occurrence peuvent être considérées comme une stratégie de prévention. Pour

cela, ce tableau permet de classer les approches proposées en deux classes : la détection (les techniques qui permettent d'identifier et de localiser les conflits) et la résolution (les techniques qui permettent d'éviter ou d'éliminer les effets indésirables potentiels des conflits). En outre, ces approches interviennent dans les différentes étapes du cycle de vie des applications. Certaines approches sont pessimistes et se concentrent sur la détection et la résolution des conflits au design time. Ceci est une tâche très complexe parce que d'une manière générale, le développeur est sollicité pour spécifier des nombreuses informations sur l'application (par exemple, les ressources requises, actions, etc.). Toutefois, ces conflits peuvent ne pas se présenter à l'exécution. D'autres approches sont optimistes et se focalisent sur leur résolution à l'exécution. Par exemple, dans [TSB07], la résolution est principalement basée sur l'adaptation de l'application à l'origine du conflit. Cette approche repose sur la stratégie du premier arrivé, premier servi. Contrairement à [TSB07], [JCL11] utilise des priorités pour résoudre les conflits lors de l'exécution. Cependant, lorsqu'un conflit est détecté, l'application avec le niveau de priorité le plus bas ne peut pas satisfaire les exigences de l'utilisateur. En outre, toutes ces solutions présentent une limite majeure : elles nécessitent l'intervention humaine. En effet, certaines solutions exigent l'intervention du développeur d'applications pour gérer les conflits. Par exemple, dans [TSB07] les développeurs doivent faire un travail supplémentaire pour la gestion des conflits. Pour chaque application, ils doivent décrire les effets sur l'environnement et spécifier les conflits. D'autres nécessitent l'intervention de l'utilisateur pour résoudre les conflits. Par exemple, dans [YIH15], la résolution est la responsabilité de l'utilisateur.

	Conflicts Detection	Conflicts Resolution
Design time	[NIM05; JCL11]	[NIM05]
Installation	[YIH15; MS14]	[YIH15; MS14]
Runtime	[Par+05; TSB07; MS14; KMW03]	[Par+05; JCL11; TSB07; MS14; KMW03]

Table 2.1: Classification des approches proposées pour la gestion des conflits

La plupart des solutions proposent d'ajouter dans la plate-forme une couche responsable d'analyser et de contrôler l'accès aux ressources afin de parvenir à gérer les conflits. Le contrôle d'accès s'avère ainsi une nécessité dans le processus de gestion des conflits. Dans la partie suivante de ce chapitre, nous présenterons une étude minutieuse autour des techniques existantes pour le contrôle d'accès.

2.3 Gestion d'accès

2.3.1 Définitions

Le contrôle d'accès a été défini dans [SS94] puis dans [SV00] par les phrases suivantes :

« *Access control constrains what a user can do directly, as well what programs executing on behalf of the user are allowed to do.* » [SS94]

« *Access control is the process of mediating every request to resources and data maintained by a system and determining whether the request should be granted or denied.* » [SV00]

Le contrôle d'accès est alors le processus qui permet de décider si l'accès à une ressource doit être autorisé ou refusé. Généralement, la mise en œuvre de ce processus peut s'appuyer sur les trois concepts suivants [SV00] :

- **les modèles** fournissent une représentation formelle de la politique de contrôle d'accès;
- **les politiques de contrôle d'accès** définissent l'ensemble des règles selon lesquelles le contrôle d'accès doit être réglementé;
- **les mécanismes** permettent d'implémenter les contrôles imposés par la politique de de contrôle d'accès.

Le contrôle d'accès se base principalement sur quatre entités clés : **(1) les objets** constituent l'ensemble des ressources à contrôler, par exemple des fichiers, des périphériques matériels, etc.; **(2) les sujets** représentent l'ensemble des entités qui veulent effectuer des actions sur les objets, par exemple des utilisateurs ou des applications; **(3) les actions** représentent les modes d'accès aux objets, par exemple lecture, écriture ou exécution; **(4) les contraintes** constituent l'ensemble des critères gouvernant l'accès à un objet. Dans [Far13], l'auteur propose une classification des différentes contraintes de contrôle d'accès, inspirée de la classification présentée dans [KLS07]. Les contraintes sont ainsi divisées en contraintes statiques et contraintes dynamiques.

- **Les contraintes statiques** portent sur des éléments qui n'évoluent pas dans le temps. Elles sont divisées en deux sous-groupes. Le premier sous-groupe est celui

des contraintes structurelles portant sur la catégorisation des sujets et des objets. En général, les catégories sont nommées « rôles ». Un rôle reflète la catégorie à laquelle appartient une entité (un sujet ou un objet). Par exemple, dans un système hospitalier, un sujet doit appartenir à la catégorie Médecin, pour pouvoir décrire une prescription médicale. Le deuxième sous-groupe des contraintes porte sur l'authentification qui consiste à vérifier l'identité d'un sujet pour lui autoriser l'accès à des ressources. Un système d'authentification, par exemple un login et un mot de passe, permet alors de valider l'authenticité d'un sujet.

- **Les contraintes dynamiques** sont liées à des informations qui peuvent changer (par exemple, la localisation). Elles sont aussi divisées en deux sous-groupes. Le premier sous-groupe est celui des contraintes sur l'environnement. Ces contraintes peuvent porter sur des attributs de l'environnement, des attributs des sujets ou bien des attributs des objets. Le deuxième sous-groupe est celui des contraintes sur les activités. En effet, ces contraintes peuvent porter sur la liaison ou la séparation des privilèges (un sujet ne peut pas avoir deux groupes différents de droits en même temps) ou bien la cardinalité. Par exemple, une contrainte de cardinalité peut préciser le nombre de fois qu'un sujet a exécuté une action.

2.3.2 Le contrôle d'accès discrétionnaire

Discretionary access control (DAC) ou, en français le contrôle d'accès discrétionnaire est présenté par TCSEC (*Trusted Computer System Evaluation Criteria*) comme : « un moyen de restriction d'accès aux objets basé sur l'identité des sujets et/ou groupes auxquels ils appartiennent. Les contrôles sont discrétionnaires dans le sens où le sujet est capable de transférer les permissions d'accès à d'autres sujets. » [Li11]

Dans le contrôle d'accès discrétionnaire, les politiques de contrôle d'accès sont représentées par des règles d'autorisation exprimées sous la forme d'un triplet <utilisateur, objet, action>. Une règle d'autorisation décrit ainsi l'action (par exemple, lire, écrire, exécuter) que l'utilisateur peut effectuer sur l'objet spécifié. Deux types de règles d'autorisation sont définis dans le DAC : règle d'autorisation positive et règle d'autorisation négative. Une règle d'autorisation est dite positive si elle définit l'ensemble des utilisateurs qui sont autorisés à accéder aux objets tandis qu'une règle d'autorisation est dite négative si elle définit l'ensemble des utilisateurs qui sont interdits d'accéder aux objets.

Dans les systèmes adoptant DAC, une politique de contrôle d'accès peut être ouverte ou fermée. Dans un système qui fonctionne avec une politique de contrôle d'accès ou-

verte, l'accès aux objets est par défaut autorisé puis les interdictions sont spécifiées via la définition des règles d'autorisation négatives (règles d'interdictions); c'est-à-dire que l'utilisateur est autorisé d'accéder à tous les objets sauf dans le cas de présence d'une règle d'interdiction spécifiant les objets dont l'accès est interdit. En revanche, dans un système qui utilise une politique de contrôle d'accès fermée, l'accès aux objets est par défaut interdit et les permissions sont par la suite spécifiées via la définition des règles d'autorisation positives; c'est-à-dire que l'utilisateur ne peut accéder à aucun objet si aucune permission n'est définie d'une manière explicite via une règle d'autorisation.

La modélisation des règles d'autorisations sous la forme d'une matrice d'accès a été proposée pour la première fois en 1974 par Lampson [Lam74]. En 1976, Harrison, Ruzzo et Ullman [HRU76] l'ont généralisé en donnant naissance au célèbre modèle matriciel HRU. Dans ce dernier, une matrice d'accès M représente l'ensemble des droits que des utilisateurs possèdent sur des objets du système. Les lignes de la matrice réfèrent les droits attribués à un utilisateur alors que les colonnes présentent les droits appliqués sur un objet. Chaque cellule de la matrice M spécifie les droits qu'un utilisateur possède sur un objet. Ces droits correspondent généralement aux actions qu'un utilisateur peut effectuer sur un objet comme par exemple lire, écrire ou exécuter. Un exemple d'une matrice d'accès est illustré par la figure 2.1.

	File 1	**File 2**	**File 3**	**Program 1**
Ann	own read write	read write		execute
Bob	read		read write	
Carl		read		execute read

Figure 2.1: Exemple de matrice d'accès [SV00].

Comme le montre la figure 2.1, une matrice d'accès présente généralement plusieurs cellules vides. Pour cette raison et dans le but d'éviter le gaspillage de l'espace mémoire, une matrice de contrôle d'accès ne peut pas être stockée sous la forme d'un tableau à deux dimensions. Il existe plusieurs techniques d'implantation des matrices de contrôle d'accès parmi lesquelles nous citons :

- **une liste de contrôle d'accès (ACL pour *Access Control List*) :** la matrice

est stockée par colonne. Chaque objet est attaché à une liste de règles spécifiant pour chaque utilisateur les actions qui peuvent être effectuées par ce dernier sur cet objet.

- **une liste de capacité :** la matrice est stockée par ligne. A chaque utilisateur est associé une liste de capacité, spécifiant pour chaque objet les actions que ce dernier peut effectuer sur cet objet.

Comme nous l'avons déjà mentionné précédemment, l'accès aux objets, dans le contrôle d'accès discrétionnaire repose principalement sur l'identité de l'utilisateur. C'est pour cette raison que ce modèle a été nommé aussi contrôle d'accès basé sur l'identité ou IBAC (pour *Identity Based Access Control*). Toutefois, le principe clé de ce modèle présente une importante faille qui fragilise le système en rendant le contrôle d'accès vulnérable aux chevaux de Troie. Un exemple de cheval de Troie est illustré par la figure 2.2.

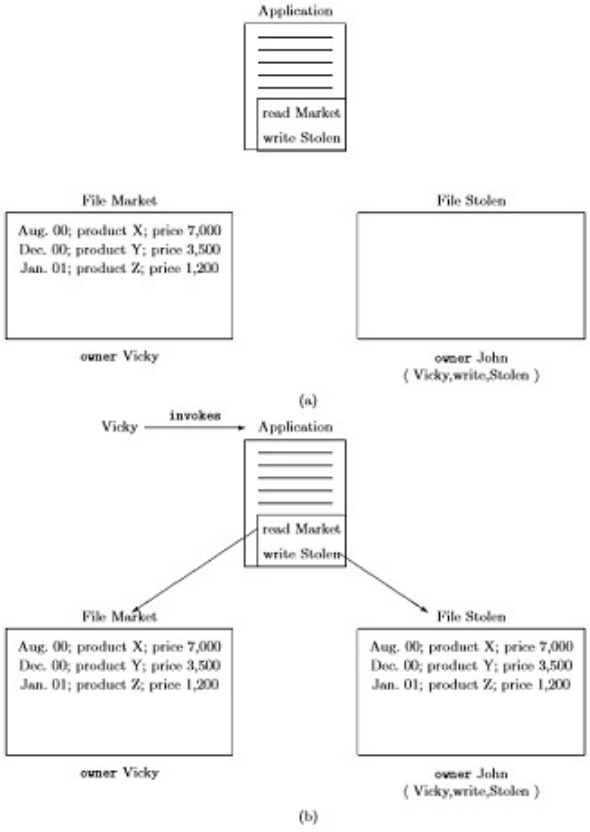

Figure 2.2: Vulnérabilité du cheval de Troie au contrôle d'accès [SV00].

Supposons dans une entreprise, que Vicky, le directeur, crée un fichier secret nommé 'Market'. Ce fichier contient des informations très importantes sur les nouveaux produits.

D'après la politique de l'entreprise, le directeur est la seule personne qui peut accéder à ces informations. Supposons maintenant que John, un adjoint de Vicky, veut récupérer ces informations délicates pour une utilisation malveillante. Pour ce faire, il crée un fichier 'Stolen' puis attribut l'autorisation d'écriture à son directeur. En outre, John ajoute deux opérations cachées, lire sur le fichier Market et écrire dans Stolen, dans une application généralement utilisée par Vicky. Une fois que cette application est exécutée par Vicky, ces deux opérations sont autorisées. John peut donc récupérer toutes les informations en accédant à son fichier Stolen.

En conclusion, le modèle de contrôle d'accès discrétionnaire ignore la différence entre l'utilisateur et le processus qu'il exécute. Ainsi, un processus possède tous les droits de son utilisateur. En outre, les flots d'informations ne sont pas contrôlés, ce qui peut provoquer une fuite d'informations. Dans la sous-section suivante, nous présentons le contrôle d'accès obligatoire qui a pour objectif de traiter ces problèmes.

2.3.3 Le contrôle d'accès obligatoire

Dans le but de traiter les problèmes liés aux fuites d'information qui peuvent être soulevés dans le contrôle d'accès discrétionnaire, le contrôle d'accès obligatoire (ou MAC pour *Mandatory Access Control*) définit un ensemble de règles pour imposer le respect des exigences de contrôle d'accès. En effet, dans MAC, les règles de contrôles d'accès ne sont pas définies par le propriétaire des objets et elles sont plutôt imposées par une autorité centrale [SV00]. Le pouvoir de la gestion d'accès des utilisateurs est ainsi limité : ces derniers ne peuvent pas, par conséquent, modifier les règles de contrôle d'accès.

Dans le contrôle d'accès discrétionnaire, la non distinction entre les sujets et les utilisateurs peut amener à des problèmes de fuite d'informations. Afin de traiter ce problème, le contrôle d'accès obligatoire différencie les sujets et les utilisateurs. L'utilisateur désigne la personne qui peut se connecter au système tandis que le sujet désigne le processus qui agit au nom de l'utilisateur.

Les exemples les plus connus d'utilisation de MAC sont les modèles multi-niveaux dont le principe de base est la classification des sujets et des objets. Une classe d'accès est ainsi associée à chaque objet et à chaque sujet. Une classe d'accès comprend généralement deux composants : un niveau de sécurité et un ensemble de catégorie.

- **Un niveau de sécurité** est un élément d'un ensemble ordonné. Nous pouvons, par

exemple, définir les quatre niveaux de sécurités suivants : Top Secret (TS), Secret (S), Confidential (C) et Unclassified (U) tel que TS>S>C>U.

- **Une catégorie de sujet** est un élément d'un ensemble non ordonné. Une catégorie peut, par exemple, représenter une compétence, une fonction, une région ou un département. Par exemple, les catégories Armée et Nucléaire peuvent appartenir à l'ensemble des catégories définis pour les systèmes militaires.

Les classes d'accès sont ordonnées par une relation d'ordre partiel notée ">=", il s'agit d'une relation de dominance. Nous considérons c1 et c2 deux classes d'accès, c1 domine c2 ou c1 >= c2 si et seulement si : le niveau de sécurité de c1 est plus grand ou égal au niveau de sécurité de c2 et que les catégories de c2 sont inclues dans les catégories de c1. L'ensemble de classes d'accès qui sont partiellement ordonnées par une relation de dominance présente deux bornes : une borne inférieure et une borne supérieure. D'une manière générale, un ensemble de classes d'accès peut être structuré sous la forme d'un treillis des concepts [Den76; GW12]. Un exemple de treillis est illustré par la figure 2.3. L'exemple présente un treillis avec l'ensemble des catégories Armée, Nucléaire et deux niveaux de sécurité (Top Secret(TP) , Secret (S)), tel que TS >= S.

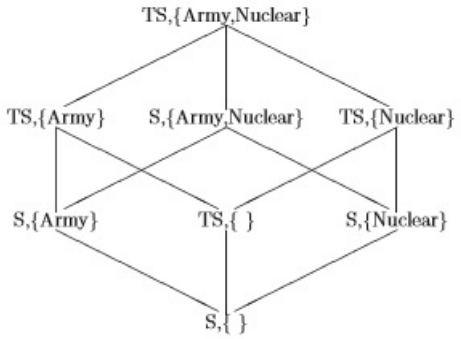

Figure 2.3: Exemple de treillis de sécurité [SV00].

2.3.4 Le contrôle d'accès à base de rôles

Le contrôle d'accès à base de rôles ou, en anglais,*Role-Based Access Control* (RBAC) [Fer+01] [Fer+03] est l'un des modèles les plus connus pour la gestion du contrôle d'accès. Il a été proposé par Ferraiolo et Kuhn comme une nouvelle alternative pour simplifier la gestion des politiques de contrôles d'accès présentées par les modèles DAC et MAC. Dans RBAC, la décision d'autorisation d'accès s'appuie principalement sur le rôle attribué à l'utilisateur. En effet, un ensemble de rôles est défini dans le système et des permissions

leurs sont attribuées. Chaque utilisateur est, par ailleurs, attaché à un ou plusieurs rôles. Les permissions sont ensuite attribuées aux utilisateurs en fonction de leurs rôles.

En 2004, ANSI (*The American National Standard Institute*) a proposé le standard ANSI RBAC [SFK04]. Ce standard définit le RBAC de référence et fournit les caractéristiques que tout système RBAC doit avoir. Il propose des définitions des termes suivants :

- **utilisateur :** un utilisateur est défini comme un être humain. Ce terme d'utilisateur peut être étendu pour représenter les machines, les agents autonomes intelligents et les réseaux.
- **rôle :** un rôle représente la fonction d'un utilisateur dans une organisation et il peut aussi représenter une responsabilité.
- **objet :** un objet représente chaque ressource dans le système telle qu'un fichier, une imprimante, une base de données, etc.
- **opération :** une opération est une commande exécutable d'un programme qui exécute des fonctions pour l'utilisateur après son invocation.
- **permission :** une permission est un accord pour exécuter une opération sur un ou plusieurs objets.

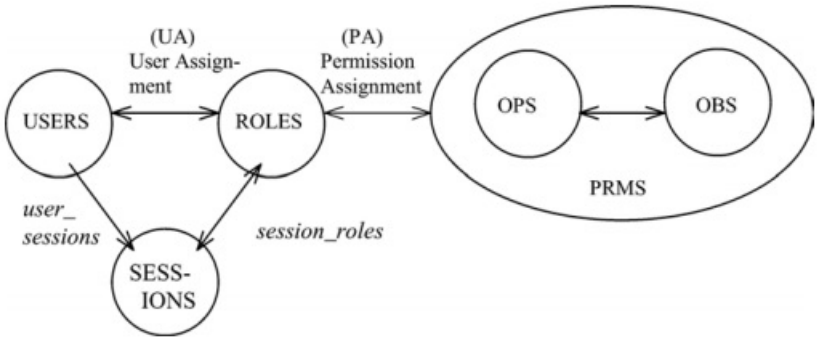

Figure 2.4: Le contrôle d'accès à base de rôles [SFK04].

La figure 2.4 décrit les éléments et les relations du modèle RBAC. Ce modèle comprend un ensemble de cinq éléments de base : les utilisateurs (USERS), les rôles (ROLES), les objets (OBS), les opérations (OPS) et les permissions (PRMS). Dans le modèle RBAC, le rôle n'est qu'un moyen pour présenter des relations de type plusieurs-à-plusieurs entre les utilisateurs et les permissions. En effet, un utilisateur peut être attaché à plusieurs rôles et

un rôle peut être accordé à plusieurs utilisateurs. De plus, un rôle peut présenter plusieurs permissions et une permission peut être accordée à plusieurs rôles. Par conséquent, les permissions sont attribuées uniquement aux rôles. Aucune permission n'est attribuée directement à l'utilisateur mais l'utilisateur possède les permissions en fonctions de ses rôles. En outre, le modèle RBAC présente une entité optionnelle appelée session (SESSION) qui représente un mappage entre un utilisateur et l'ensemble de ses rôles activés. En effet, pour accéder au système, l'utilisateur peut ouvrir une session durant laquelle il ne possède que les droits qui correspondent aux rôles activés.

RBAC permet aussi d'avoir des hiérarchies de rôles. Les hiérarchies sont en effet un moyen de structuration des rôles pour refléter l'ordre de responsabilité dans une organisation. Ceci permet, par exemple, de refléter la hiérarchie dans une entreprise. De plus, RBAC permet d'introduire la notion des contraintes pouvant être appliquées sur l'affectation des rôles aux sujets.

2.3.5 Le contrôle d'accès à base d'attributs

Le contrôle d'accès basé sur les attributs, ou ABAC (pour *Attribute-Based Access Control*) a été développé, en 2005, par Eric Yuan et Jin Tong [YT05]. Il s'appuie principalement sur des caractéristiques de chaque entité (sujet, ressource, environnement), appelés attributs. La figure 2.5 présente les entités de base du modèle ABAC et leurs relations.

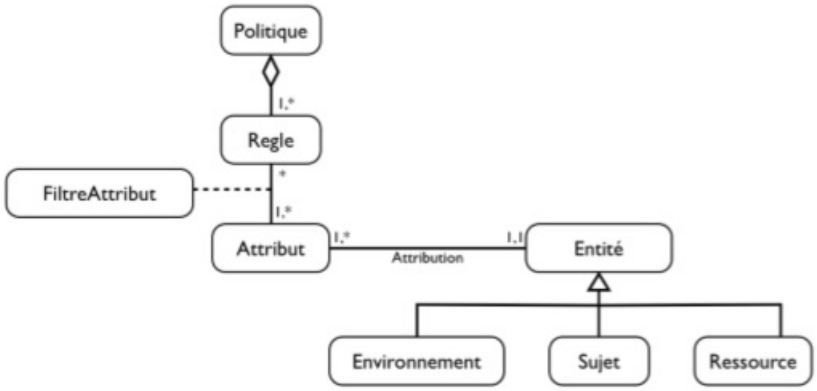

Figure 2.5: Le contrôle d'accès basé sur les attributs [Far13].

- Un attribut est une propriété, exprimée sous forme d'une paire (nom, valeur), sur laquelle se basent les autorisations d'accès. Chaque entité peut être caractérisée par

un ensemble d'attributs qui peuvent avoir une valeur. Trois groupes d'attributs sont ainsi identifiés selon le type de l'entité à laquelle ils sont associés :

- **les attributs des sujets :** un sujet peut être défini comme une entité qui peut agir sur une ressource. A chaque sujet est accordé un ensemble d'attributs qui définit son identité et décrit ses propriétés. Les attributs du sujet peuvent être, par exemple, son nom, son prénom et même son rôle, etc.
- **les attributs des ressources :** une ressource est une entité sur laquelle un sujet peut agir. A chaque ressource est accordé un ensemble d'attributs qui la décrit et sert à contrôler les accès. Ces attributs peuvent être, par exemple, son type, son propriétaire, sa date de modification, etc.
- **les attributs d'environnement :** la particularité clé du modèle de contrôle d'accès à base d'attribues est que les décisions de contrôle d'accès peuvent être prises en fonction du contexte d'exécution du système, via la définition des attributs d'environnement. Ces attributs décrivent l'état de l'environnement comme, par exemple, la date, l'heure, etc.

- Une règle de contrôle d'accès permet de limiter l'ensemble des valeurs autorisées pour les attributs auxquels elle s'applique. Une règle est ainsi vérifiée si et seulement si l'ensemble d'attributs auxquels elle se rapporte ont une valeur parmi l'ensemble des valeurs autorisées.

- Une politique est constituée par un ensemble de règles. Généralement, les règles qui composent une politique de contrôle d'accès, sont reliées par une idée commune.

Le modèle de contrôle d'accès ABAC est un modèle très flexible qui permet de définir des règles d'autorisation avec une granularité très fine via la définition des attributs qui peuvent être associés aux sujets, aux ressources ou aux environnements.

Le modèle ABAC peut ainsi remplacer le modèle RBAC en représentant les rôles par des attributs des sujets. De plus, le modèle de contrôle d'accès basé sur des attributs permet de représenter les politiques de contrôle d'accès dans une organisation avec des critères plus complexes que le modèle RBAC, en définissant des règles qui portent sur un nombre illimité d'attributs.

La prise de décision de contrôle d'accès en fonction du contexte, via la définition des attributs environnementaux, est l'un des avantages principaux d'ABAC. Ceci permet

d'obtenir un modèle plus souple et surtout plus adapté à la problématique liée au dynamisme d'accès aux ressources. Les décisions de contrôle d'accès peuvent s'adapter à l'évolution du contexte et ainsi aux différents états de l'environnement.

2.3.6 Le verrouillage

Un verrou est un mécanisme de synchronisation qui permet d'imposer des contrôles d'accès à une ressource partagée. Ce mécanisme est souvent utilisé pour contrôler les accès aux fichiers dans les systèmes d'exploitation. Un verrou est caractérisé par deux propriétés importantes :

- **la portée (la granularité) :** elle constitue l'ensemble des éléments sur lesquels porte la pose d'un verrou. Par exemple, en système d'exploitation, le verrou peut être appliqué sur l'ensemble du fichier ou seulement sur une partie limitée.

- **le type :** décrit la possibilité de cohabitation de différents verrous. D'une manière générale, deux types de verrous sont disponibles : (1) Verrou partagé : ce type de verrou a pour objectif de protéger l'accès en lecture. Des lectures simultanées par plusieurs processus sont autorisées mais aucun accès en écriture. (2) Verrou exclusif : il permet seulement au processus propriétaire du verrou d'accéder à la ressource en lecture et en écriture. Tout accès par un autre processus est interdit.

Un verrou présente deux états : il peut être libre (ou déverrouillé) donc aucun processus ne détient le verrou ou occupé (ou verrouillé); c'est-à-dire il existe un processus qui le maintient. Deux opérations sont disponibles pour changer l'état de verrou :

- verrouiller() permet au processus de détenir le verrou s'il est dans un état libre. Le processus en question nommé ainsi le propriétaire du verrou.

- déverrouiller() permet au processus de relâcher le verrou qu'il détenait.

Les verrous sont aussi utilisés dans les Système de Gestion de Base de Données (SGBD) afin de garantir la synchronisation des transactions. En effet, une transaction, unité d'exécution pour un SGBD, est définie comme une suite d'opérations de lecture ou d'écriture limitée par une instruction de validation (commit) ou d'annulation (rollbak). En base de données, il existe deux stratégies de verrouillage :

- **le verrouillage optimiste :** ce verrouillage consiste à poser un verrou lors de la validation de la transaction. Son avantage principal est que les données ne sont verrouillées que pendant une courte durée qui correspond à la phase de validation. Entretemps, les données manipulées restent accessibles aux transactions concurrentes. Par conséquent, la transaction peut échouer et les opérations effectuées peuvent être annulées si les données ont été modifiées par d'autres transactions. Ce risque d'échec constitue alors la limite majeure de cette stratégie de verrouillage.

- **le verrouillage pessimiste :** un verrou est posé pendant toute la durée d'utilisation des données par la transaction. Aucune autre transaction n'est alors autorisée à manipuler ces données jusqu'à la libération du verrou initialement posé. Cette stratégie permet d'éviter la perte des modifications effectuées par les transactions. Toutefois, son inconvénient principal est que les autres transactions concurrentes peuvent être bloquées durant une longue période.

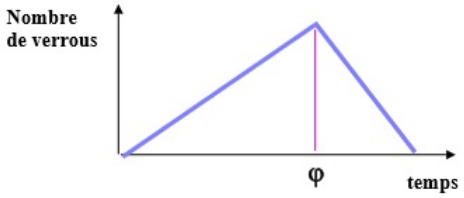

Figure 2.6: Comportement des transactions deux phases [Gar03].

Une transaction peut être en verrouillage à deux phases (*Two phase locking*) [Esw+76], comme la montre la figure 2.6 : pendant la première phase (phase d'acquisition des verrous), la transaction peut poser des verrous mais elle n'est pas habilitée à les relâcher. En revanche, pendant la deuxième phase (phase de relâchement), la transaction ne peut poser aucun verrou et elle ne peut qu'en libérer.

	L	E
L	V	F
E	F	F

Figure 2.7: Compatibilité des opérations de lecture/écriture [Gar03].

La matrice illustrée par la figure 2.7 présente les compatibilités entre opérations de lecture (L) et écriture (E). Les opérations incompatibles ne sont pas autorisées et les verrous conflictuels ne peuvent pas alors être posés sur le même objet à la fois. En effet, toute transaction demandant un verrou sur un objet déjà verrouillé doit attendre la fin des transactions incompatibles

2.3.7 Synthèse

Pour parvenir à gérer les conflits, le contrôle d'accès semble le moyen le plus efficace. Pour cela, dans cette partie de ce chapitre, nous avons mené une étude approfondie sur les différentes techniques existantes de contrôle d'accès. Nous pouvons alors conclure que chaque technique a été ainsi conçue pour répondre à des défis du contrôle d'accès dans des situations bien particulières.

Les contrôles d'accès discrétionnaire et obligatoire sont des techniques classiques qui ont été définies pour répondre aux contraintes d'accès traditionnelles. Toutefois, dans le domaine de l'informatique ubiquitaire, les techniques doivent satisfaire des exigences d'accès très pointues. Le contrôle d'accès basé sur les rôles ne peut pas être appliqué forcément à ce domaine. En effet, le nombre de rôles augmente selon le nombre d'attributs à considérer. En revanche, le contrôle d'accès basé sur les attributs semble plus adapté au domaine ubiquitaire. Il présente la capacité de définir plusieurs contraintes pouvant être statiques (liées aux sujets ou aux ressources) ou dynamiques (relatives à l'environnement). Toutefois, les règles de contrôle d'accès définies par ce modèle restent difficiles à interpréter et certaines peuvent aboutir à des décisions contradictoires. Le verrouillage est une technique qui peut être adaptée parfaitement. Il garantit qu'une ressource ne peut être utilisée que par une seule application à la fois. De plus, gérer les conflits nécessite de savoir le début et la fin de l'utilisation d'une ressource susceptible d'être au centre de conflits. Les applications ubiquitaires étant contextuelles, ces informations ne peuvent pas être spécifiées pendant la phase de conception. Toutefois, grâce à cette technique, elles peuvent être inférées à l'exécution en faisant appel aux méthodes de verrouillage et de déverrouillage. Afin de gérer les conflits, cette technique nous a alors apparu la plus appropriée.

2.4 Conclusion

Dans la première partie de ce chapitre, nous avons constaté que les approches de gestion des conflits visent à anticiper les conflits et à les résoudre avant leur apparition effective. Cette stratégie est intéressante pour assurer une exécution sans conflit. Ces approches peuvent être mises en œuvre à différentes étapes du cycle de vie de l'application, mais essentiellement, au moment de la conception. Ici, tous les conflits potentiels peuvent être identifiés. Pour ce faire, il est nécessaire de spécifier beaucoup d'informations, souvent d'une manière formelle sur l'application. Cela peut compliquer considérablement la tâche du développeur. En outre, les approches de résolution de conflits pendant la conception sont généralement pessimistes car certains conflits détectés peuvent ne jamais se produire lors de l'exécution. De l'autre côté, d'autres approches sont plus « optimistes » et visent la résolution des conflits à l'exécution. En pratique, les approches proposées nécessitent souvent une intervention humaine pour gérer les conflits. En revanche, nous visons à gérer les conflits entre les applications de la maison de manière plus autonome, tout en limitant l'intervention humaine.

En général, les applications traditionnelles s'exécutent sur un système d'exploitation qui est responsable d'exploiter et de contrôler l'accès aux ressources. D'une façon similaire, les applications ubiquitaires s'exécutent sur une plate-forme ubiquitaire, une suite logicielle qui satisfait un ensemble de contraintes imposées par la nature de l'environnement. D'une manière analogue aux systèmes d'exploitation traditionnels, les plateformes ubiquitaires doivent présenter cette capacité de contrôle d'accès et de gestion des ressources. Cette capacité s'avère indispensable pour la gestion des conflits entre les applications ubiquitaires. Pour cette raison, la deuxième section de ce chapitre a été consacrée à une étude détaillée autour des techniques de contrôle d'accès. Nous avons constaté que dans chaque technique, plusieurs concepts ont été définis pour répondre à des besoins particuliers. Selon cette étude, le verrouillage nous semble la technique la plus convenable permettant de gérer les conflits.

Chapter 3

Plateformes Logicielles

Sommaire

Comme nous l'avons vu dans le premier chapitre, les environnements pervasifs imposent un ensemble de contraintes sur la manière de concevoir et d'exécuter les applications déployées. Ces environnements étant ouverts et dynamiques, ce qui rend la gestion des conflits entre ces applications l'une des contraintes les plus complexes à satisfaire. Le but de ce chapitre est alors de présenter une étude autour des plateformes logicielles permettant de répondre à ces contraintes. Nous présenterons, dans un premier temps, les plateformes généralistes pour le dynamisme. L'informatique orientée services étant au cœur de ces solutions, nous définirons alors tous les concepts et les mécanismes clé liés à ce paradigme. Puis, dans un second temps, nous présenterons les différentes plateformes ouvertes existantes, principalement les plateformes mobiles. Enfin, nous exposerons d'une manière comparative, certaines plateformes ubiquitaires qui satisfont des exigences relatives à la nature de leur environnement.

3.1 Plateformes généralistes pour le dynamisme

Les applications ubiquitaires s'exécutent sur des plateformes logicielles qui doivent répondre à un ensemble d'exigences liées principalement à la nature de l'environnement. Dans cette section, nous nous concentrons sur l'une d'entre elles, le dynamisme. Nous présentons, dans un premier volet, une étude sur les approches proposées pour construire les plateformes logicielles répondant à cette exigence. Puis, dans un second volet, nous présentons des exemples de plateforme généraliste implantant ces approches pour traiter le dynamisme.

3.1.1 Introduction

3.1.1.1 Approche orientée composants

L'objectif principal de l'approche à composant est de traiter les limites de l'approche orientée objet [Tay97]. Pour cela, elle propose un ensemble de mécanismes qui facilite le développement et l'administration des applications. Cette approche vise en outre à améliorer la réutilisation grâce à la construction d'applications à partir d'assemblages de briques logicielles bien définies et indépendantes nommées composants.

La définition du composant logiciel la plus utilisée dans la littérature est celle proposée par Szyperski :

« *A software component is a unit of composition which contractually specified interfaces and explicit context dependencies only. A software component can be deployed independently and is subject to composition by third parties.* » [SGM02]

Cette définition met en lumière les concepts clés qui caractérisent un composant : **(1)** Un composant est une unité de composition. **(2)** Un composant expose ses interfaces fonctionnelles. **(3)** Il possède des dépendances explicites. Une dépendance peut être logique, c'est-à-dire que le composant requiert des fonctionnalités fournies par d'autre composants pour réaliser ses propres fonctionnalités, ou physique, par exemple des bibliothèques. **(4)** Un composant est une unité de déploiement qui peut être déployée de manière indépendante.

« *A software component is a software element that conforms to a component model and can be independently deployed and composed without modification according to a composi-*

tion standard. » [CH01]

Heineman et Councill [CH01] proposent une autre définition de composant dans laquelle ils précisent qu'un composant logiciel doit être conforme à un modèle de composants définissant la structure des composants et la façon d'effectuer des assemblages. D'où, il s'avère nécessaire d'ajouter à la notion de composant un mécanisme de composition et une plateforme d'exécution. Cette plateforme d'exécution propose les mécanismes nécessaires pour l'exécution d'instances de composants constituant l'application. Elle fournit aussi des mécanismes pour gérer un ensemble d'aspects non fonctionnels tels que la sécurité, la persistance, les transactions ou encore la gestion du cycle de vie des instances.

Dans l'approche à composants, l'assemblage de composants peut être défini par un langage de description d'architecture (ADL pour *Architecture Description Language*). Un ADL se base principalement sur deux éléments clés :

- **les instances de composants :** à l'exécution, une instance présente les fonctionnalités de son type de composant. Les instances sont ainsi les éléments indispensables pour la réalisation de l'application ; ils constituent donc la logique applicative.
- **les connecteurs :** ce sont les liaisons effectuées entre les différentes instances de composants constituants l'application. Ces connecteurs sont déterminés selon les dépendances spécifiées entre les différents types de composants utilisés.

L'approche orientée composant se base sur le principe de la séparation des préoccupations entre les aspects fonctionnels et non fonctionnels. En suivant ce principe, le code métier lié aux fonctionnalités offertes par le composant est clairement séparé du code traitant ses besoins et ses propriétés non fonctionnelles. Ainsi, le développeur réalise uniquement le code fonctionnel de son composant et l'ensemble des aspects non fonctionnels est pris en charge par l'infrastructure d'exécution. Toutefois, l'ensemble des besoins extra fonctionnels étant extensible, le développeur d'un composant peut ainsi en traiter d'autres en implantant le code associé.

Une des approches les plus fréquemment appliquées pour effectuer la séparation des préoccupations entre les aspects fonctionnels et non fonctionnels est l'utilisation du concept de conteneur. Un conteneur, comme le montre la figure 3.1, peut être présenté comme une membrane qui englobe le contenu (le code fonctionnel) d'une instance de composant. Ce dernier prend en charge la gestion d'un ensemble d'aspects non fonctionnels tels que

la sécurité, la persistance ou bien la gestion dU cycle de vie de l'instance ainsi que ses interactions avec d'autres instances de composants.

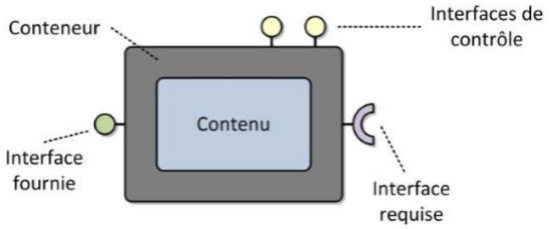

Figure 3.1: Une instance de composant et son conteneur [MG13].

3.1.1.2 Approche orientée services

L'objectif principal de l'approche à services [Pap03] est de construire des applications en utilisant des entités logicielles faiblement couplées, appelées services. La notion de service est ainsi le concept de base de l'approche à services. Cette approche repose sur trois types d'acteurs principaux, comme les montre la figure 3.2 :

- **les fournisseurs de services :** les fournisseurs de services offrent des services spécifiés par des contrats (descriptions des services). Le contrat de service est un élément clé dans l'approche à services. Il est la seule information partagée par un consommateur et un fournisseur de service. Il décrit les propriétés fonctionnelles ainsi que les propriétés non fonctionnelles présentées par le service.
- **les consommateurs de services :** les consommateurs de services sont les clients qui utilisent des services exposés par des fournisseurs.
- **le registre de services :** le registre de services joue le rôle de l'intermédiaire entre les fournisseurs et les consommateurs de services. Il permet, d'une part, aux fournisseurs d'enregistrer les contrats de leurs services. Il permet, d'autre part, aux consommateurs de chercher et de sélectionner les services qu'ils requièrent.

La figure 3.2 illustre aussi trois primitives d'interactions entre les différents acteurs de l'approche à service [MG13] :

- **la publication de services :** les fournisseurs de service s'enregistrent auprès du registre de services afin de publier les contrats de leurs services.

- **la découverte de services :** les consommateurs interrogent le registre de services pour découvrir les services qu'ils requièrent.
- **la liaison et l'invocation des services :** une fois le service choisi, son invocation consiste à effectuer une liaison entre le fournisseur et le consommateur de service pour assurer son utilisation.

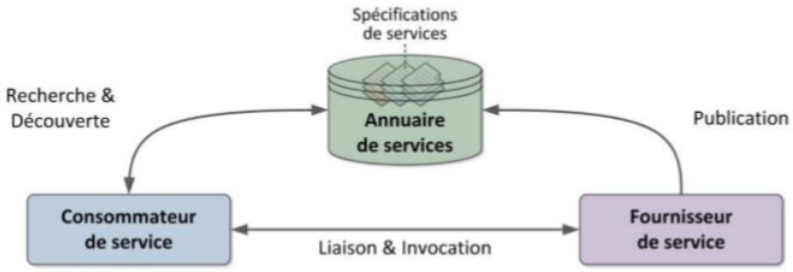

Figure 3.2: Principe d'interaction de l'approche à services [MG13].

L'un des avantages clés de l'approche orientée service est le faible couplage entre le fournisseur et le consommateur de service. En effet, la seule information partagée entre ces deux acteurs est le contrat de service. Ainsi, un consommateur utilise un service sans connaître la technologie utilisée pour son implémentation. Par conséquent, l'hétérogénéité peut être masquée, grâce à ce faible couplage, en cachant les détails d'implémentation de service. De plus, l'approche à services présente un autre avantage important qui se manifeste par la liaison tardive entre le fournisseur et le consommateur du service. En effet, la liaison entre ces deux acteurs est établie seulement lorsqu'un fournisseur est trouvé et lorsque le consommateur le demande.

Le principe d'interaction entre les différents acteurs de l'approche à services peut avoir lieu dans plusieurs phases du cycle de vie de l'application : au développement, au déploiement ou à l'exécution. Cependant, retarder ce principe d'interaction à l'exécution favorise la construction d'applications dynamiques. Deux nouvelles primitives, comme les montre la figure 3.3, ont été introduites dans le but de supporter le dynamisme [Esc08] :

- **le retrait de service :** lorsqu'un service n'est plus disponible, il doit être retiré du registre de services.
- **la notification :** les consommateurs de service doivent être notifiés de l'arrivée ou du départ d'un fournisseur qui implémente une spécification de service correspondant à leurs besoins.

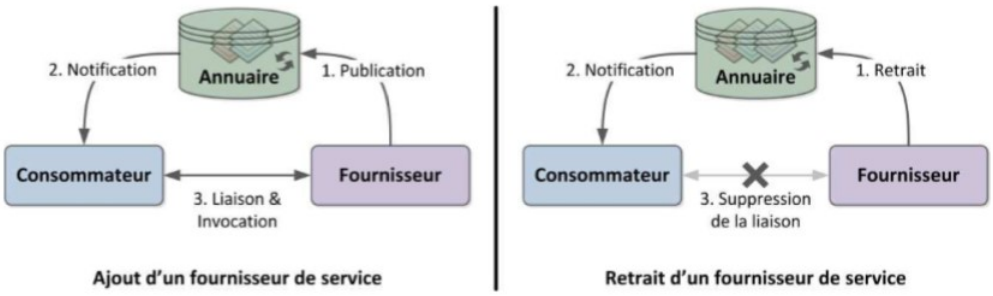

Figure 3.3: Interactions de l'approche à services dynamique [MG13].

3.1.1.3 Approche à composants orientés services

L'objectif principal de l'approche à composants orientées service [Cer04] est de combiner les avantages de l'approche à composants et de l'approche à services afin de faciliter la construction d'applications dynamiques. Ainsi, cette approche hérite de l'approche à composant l'idée de séparation des préoccupations entre les aspects fonctionnels et non fonctionnels dans le but de déléguer tous les mécanismes liés à l'approche à service (publication, découverte, liaison et invocation, etc.) aux conteneurs des composants.

Le modèle à composants orientés service propose d'utiliser les composants pour implémenter les services. Les principes de ce modèle ont été décrits dans [CH04]. Ces principes sont :

- **un service est une fonctionnalité fournie.**
- **un service est caractérisé par une spécification de service.** Cette spécification comporte des informations syntaxiques, comportementales et sémantiques ainsi que des dépendances sur d'autres spécifications de service.
- **un composant implémente une ou plusieurs spécifications de service et peut aussi dépendre d'autres services.** Une spécification de service définit un ensemble de contraintes qui doivent être respectées par le composant.
- **le principe d'interaction de l'approche à services dynamique est appliqué pour résoudre les dépendances de services.** Les services doivent être publiés dans un registre de services qui sera utilisé, à l'exécution, pour assurer la découverte des services et la résolution des dépendances.
- **les compositions sont décrites en termes de spécifications de services.** Une composition est un ensemble de spécifications permettant de sélectionner des com-

posants concrets. Les liaisons seront déterminées à l'exécution à partir des dépendances des services

- **les spécifications de services sont les éléments de base de la substitution.** Tout composant, faisant partie d'un assemblage de composant, peut être remplacé par un composant alternatif respectant la même spécification de service.

En conclusion, l'approche à composants à services vise à faciliter le développement d'applications à service dynamiques. Elle présente des avantages de l'approche orientée composant (une description de l'architecture et un modèle de développement simple) et des avantages de l'approche orientée service (un faible couplage et le dynamisme).

3.1.2 OSGi™

OSGi™ [All] (*Open Services Gateway Initiative*) est une spécification de plateforme à services définie par le consortium OSGi Alliance qui réunit plusieurs entreprises telles qu'IBM, Oracle, Nokia, Sun Microsystems, Ericsson, etc. Cette spécification fut originellement destinée aux passerelles résidentielles mais aujourd'hui utilisée dans de nombreux domaines tels que le secteur de l'automobile, des téléphones portables et des serveurs d'applications.

La spécification OSGi™ suit le style architectural proposé par l'approche à services. Elle présente un modèle de développement d'applications à base de services [Hal+11]. Les services sont spécifiés sous forme d'interfaces Java. Dans OSGi™, les unités de déploiement de services sont appelées *bundles*. Un *bundle* est implanté sous la forme d'une archive Java, qui est un fichier JAR, contenant du code Java et des ressources (fichiers de configuration, images, etc.). Chaque *bundle* doit spécifier les packages exportés ainsi que les packages importés. Une application peut être ainsi composée d'un ensemble de bundles interconnectés.

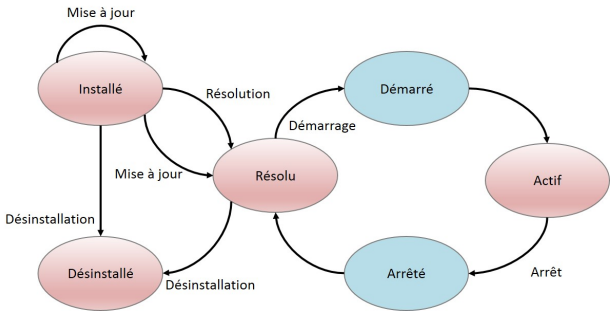

Figure 3.4: Le cycle de vie d'un *bundle*.

Dans OSGi^{TM}, le *bundle* suit le cycle de vie représenté par le diagramme d'états de la figure 3.4. Un *bundle* correctement installé sera dans l'état *Installed*. Une fois toutes ses dépendances résolues, il passera dans l'état *Resolved* et sera prêt à être démarré. A son démarrage, le *bundle* passera à l'état *Starting*, puis *Active* une fois qu'il sera en cours de fonctionnement. Une demande d'arrêt du bundle le basculera dans l'état *Stopping* puis *Resolved*. Le *bundle* pourra ainsi être redémarré, mis à jour ou désinstallé, le mettant alors dans l'état *Uninstalled*.

OSGi^{TM} spécifie une plateforme à services centralisée avec un registre de services. Les fournisseurs enregistrent leurs services exposés dans un registre de services. Afin de trouver un service, le consommateur interroge le registre pour trouver les fournisseurs de services disponibles qui répondent à ses critères. En réponse, le registre de services renvoie les références de services compatibles à la demande du consommateur. Ce dernier peut ainsi choisir un service et peut demander par la suite au registre l'objet de service associé à la référence choisie. Le registre de services renvoie une référence sur le fournisseur qui peut être invoqué directement par le consommateur.

OSGi^{TM} est aujourd'hui une plateforme très connue pour la construction d'applications dynamiques. En effet, en plus des mécanismes de déploiement et d'administration, OSGi^{TM} définit des mécanismes de notification des changements associés aux services. Le consommateur doit ainsi être notifié lorsqu'un nouveau fournisseur apparaît, un service en cours d'utilisation disparaît ou un contrat d'un service utilisé subit des changements.

En conclusion, OSGi^{TM} traitre l'aspect dynamique des environnements où les dispositifs peuvent apparaître et disparaître à tout moment. Grâce à l'ensemble des mécanismes qu'il fournit, il s'est imposé comme la plateforme de référence pour la construction des applications dynamiques. Toutefois, la spécification OSGi^{TM} présente un défaut fondamental qui rend la construction d'applications dynamiques une tâche très complexe pour les développeurs non experts. En effet, OSGi^{TM} ne propose pas un modèle de composition. Le développeur doit ainsi définir et gérer la composition de son application.

De plus, il doit gérer d'une manière explicite le dynamisme afin de valider ou d'invalider une composition. Ce dernier doit donc gérer manuellement les interactions prévues par l'architecture orientée service telles que la publication et le retrait des services ainsi que vérifier le relâchement des références vers les services désenregistrés. Ces tâches sont non seulement très critiques et peuvent être des sources d'erreurs mais aussi elles sont très complexes, délicates et fastidieuses pour les développeurs d'applications.

3.1.3 Apache Felix iPOJO

La technologie Apache Felix iPOJO (*injected Plain Old Java Object*) [Esc08] est une implantation des principes de l'approche à composants orientés service, développée au sein de l'équipe Adèle du Laboratoire d'Informatique de Grenoble et intégrée au projet Apache Felix [1].

iPOJO propose un modèle de développement simple qui permet d'implémenter les services sous la forme de composants. Ce modèle, basé sur l'idée de POJO (*Plain Old Java Object*) [Fow00], permet de masquer la complexité du l'architecture orientée service dynamique ainsi que toute la complexité inhérente au dynamisme de OSGi™. Pour cela, iPOJO fournit une machine d'exécution qui propose des mécanismes d'introspection et de reconfiguration dynamique permettant de gérer ces préoccupations d'une façon transparente pour les développeurs d'applications.

iPOJO utilise le concept de conteneur de l'approche orientée composant pour effectuer une séparation entre les propriétés fonctionnelles et non fonctionnelles des composants. Les conteneurs iPOJO ne sont pas monolithiques, mais ils sont constitués par un ensemble de *handlers*. Chaque *handler* gère un ou plusieurs aspects non fonctionnels.

iPOJO propose un ensemble de *handlers* de base pour traiter certains besoins non fonctionnels tels que le *handler* du cycle de vie chargé de la gestion du cycle de vie des instances de composants, le *handler* des dépendances de services permettant la réalisation des dépendances entre les services et le *handler* de configuration responsable de la configuration et de la reconfiguration dynamique des instances, etc. En plus des *handlers* prédéfinis, d'autres *handlers* peuvent être ajoutés, grâce à des mécanismes d'extension, pour traiter d'autres aspects non fonctionnels comme la sécurité, la persistance ou la qualité de service.

La séparation claire entre les aspects fonctionnels et non fonctionnels dans le modèle iPOJO facilite la tâche du développeur. Ce dernier peut se concentrer seulement sur la logique métier des composants constituants son application alors que les conteneurs se chargent de gérer tous les aspects liés au dynamisme et aux primitives de l'approche à service dynamique (publication, recherche, sélection et liaison). En outre, iPOJO est utilisé aujourd'hui pour développer certaines plateformes ubiquitaires domotiques (iCasa [Lal+15; IMA]).

[1] http://felix.apache.org/site/index.html

3.2 Plateformes ouvertes

Pour définir les plateformes ouvertes, Balland dans [BC10] décrit trois propriétés clés :

- elles fournissent des interfaces de programmation publiques que des développeurs tiers utilisent pour programmer des nouvelles applications.
- elles fournissent des ressources partagées aux applications qui peuvent être soit matérielles (appareil photo, etc.), logicielles (email, navigateur web, etc.) ou des données (profil d'utilisateur, carnet d'adresses, etc.).
- elles offrent un environnement d'exécution pour les applications hébergées dans un dispositif donné (par exemple, un smartphone) où les ressources sont locales ou en réseau.

D'après ces caractéristiques, Balland identifie trois acteurs: le propriétaire de la plateforme ouverte, le développeur d'applications et l'utilisateur de la plateforme. **(1)** Le propriétaire de la plateforme vise à faciliter le processus de développement d'applications afin d'encourager l'adoption de sa plateforme. Il est responsable de fournir aux développeurs des interfaces de programmation pour construire leurs applications et assurer aux utilisateurs l'accès à un magasin d'applications où les développeurs tiers peuvent déposer leurs applications. Ces applications peuvent être directement téléchargées à partir de ce magasin et installées sur l'appareil de l'utilisateur. Aujourd'hui, Google et Apple sont les leaders dans ce domaine d'industrie avec leurs plateformes respectives Android et iOS. **(2)** Le développeur s'intéresse particulièrement à la popularité de la plateforme ainsi qu'à la facilité de développement d'applications. **(3)** L'utilisateur de la plateforme souhaite contrôler l'impact de ses applications sur son appareil. Il souhaite ainsi être informé non seulement par les ressources (par exemple, appareil photo, carnet d'adresses) utilisées par ses applications mais aussi par comment elles sont utilisées.

Récemment, les plateformes ouvertes pouvant intégrer des applications tierces sont devenues très populaires, notamment sous l'impact de l'explosion du marché des smartphones. En effet, l'industrie des plateformes mobiles a été commencée en 2007 avec le système iOS proposé par Apple. L'iPhone a été le premier smartphone qui a permis d'installer des applications proposées par des développeurs tiers. Aujourd'hui, Android est le premier système d'exploitation utilisé pour les appareils mobiles dans le monde.

Android est une plateforme ouverte très populaire et très intéressante pour les développeurs d'applications. Elle propose un ensemble d'API (*Application Programming*

Interface) de haute qualité qui permet d'interagir avec les ressources de l'appareil, par exemple son GPS ou son appareil photo, etc. Elle fournit en outre un kit de développement logiciel (SDK: *Software Development Kit*) qui contient les outils nécessaires pour programmer, tester et débugger les applications Android.

Le magasin d'applications Google Play d'Android est aujourd'hui le plus gros magasin d'applications dans le monde. Ceci reflète le succès incontestable du concept des plateformes ouvertes qui a attiré non seulement les développeurs mais aussi les consommateurs par le large choix d'applications.

Google Play joue le rôle d'intermédiaire entre les développeurs d'applications et les consommateurs qui utilisent la plateforme Android. Ainsi, les développeurs tiers peuvent publier leurs applications et les utilisateurs peuvent les chercher et les télécharger.

Le système Android propose un ensemble de ressources sensibles aux applications installées par l'utilisateur sur son smartphone telles que la caméra, le GPS, le Bluetooth, etc. Afin de pouvoir utiliser ces ressources, une application doit déclarer les permissions qu'elle a besoin dans un fichier AndroidManifest.xml. Par exemple, la permission doit impérativement figurer dans le fichier de déclaration d'une application souhaitant lire les SMS.

Le système Android propose un modèle de sécurité qui repose sur le principe suivant: Une application ne peut accéder que seulement aux ressources explicitement autorisées par l'utilisateur. Avec ce principe de sécurité, Android délègue à l'utilisateur la responsabilité d'accepter ou non les permissions demandées par l'application pour lui permettre l'accès aux ressources. Dans l'optique d'aider l'utilisateur à prendre une décision, les permissions demandées sont présentées à l'utilisateur lors de l'installation de l'application. Ce dernier a ainsi le choix d'accepter ou de refuser l'application. Si l'utilisateur refuse les permissions sollicitées, l'application ne peut être installée. Toutefois, une fois l'application installée, les permissions accordées ne peuvent pas être révoquées.

Android applique le contrôle d'accès obligatoire (MAC) [EOM09]. Le mécanisme MAC est basé sur les permissions que l'utilisateur et/ou le système accorde à l'application lors de l'installation. A l'exécution, le système vérifie si une application présente les autorisations nécessaires pour effectuer une certaine action.

Android est une plateforme ouverte qui est capable d'héberger un ensemble d'applications développées par des tierces parties. Toutefois, ces applications peuvent partager les ressources proposées par cette plateforme et risque d'être en conflit. L'exemple de conflit le plus connu dans cette plateforme mobile est celui qui peut se produire entre un appel GSM (*Global System for Mobile Communications*) et un appel en VoIP (*Voice over IP*). En effet, un appel GSM permet au client de communiquer à travers un réseau de téléphonie mobile, tandis qu'un appel en voix sur IP permet de communiquer via un réseau Internet. Aujourd'hui, avec l'essor de l'Internet, plusieurs applications (Skype, Viber, Messenger, etc.) permettant de profiter de ce deuxième type de service sont disponibles dans les magasins d'applications.

En cas de conflits, les appels GSM sont généralement prioritaires par rapport aux appels en VoIP. En effet, si un appel vocal GSM est reçu pendant qu'un appel en VoIP est en cours, ce dernier sera prioritaire. Ainsi, l'appel en VoIP sera mis en attente et il ne sera rétabli que lorsque l'appel GSM sera terminé. D'autre part, si un appel en VoIP est tenté pendant qu'un appel vocal GSM est en cours, il sera automatiquement ignoré. Cette fonction va ainsi se comporter comme si l'appel n'a pas été répondu.

3.3 Plateformes ubiquitaires

Comme nous l'avons déjà mentionné dans le chapitre précédent, les applications ubiquitaires doivent exposer de nombreuses propriétés et satisfaire plusieurs contraintes qui rendent leur développement très complexe et nécessitent un haut niveau de compétence. Il s'avère ainsi indispensable de fournir des environnements de développement et d'exécution pour l'informatique ubiquitaire afin d'assister les développeurs et de les guider tout au long des différentes phases de construction d'applications, il s'agit de plateformes ubiquitaires. L'objectif principal de ces plateformes est de gérer un ensemble d'aspects non-fonctionnels notamment les propriétés de l'environnement ubiquitaire (hétérogénéité, dynamisme, distribution et ouverture) ainsi que les problèmes liés aux applications (gestion de la sécurité, gestion de l'adaptabilité, gestion des données, etc.). Ceci est résumé dans [ECL14] par les phrases suivantes :

« *The purpose of such platforms is to free developers from complex, technical code and allows them to concentrate on domain-specific code.* » [ECL14]

« *The purpose of a pervasive platform is to manage a set of non-functional properties on behalf of the [pervasive] applications.* » [ECL14]

Plusieurs chercheurs et industriels se sont penchés sur la création des plateformes ubiquitaires dans le but de gérer la complexité croissante des applications ubiquitaires. Dans la suite de ce chapitre, nous présentons les plateformes les plus populaires dans le domaine de l'informatique ubiquitaire.

3.3.1 DiaSuite

DiaSuite [Ber+14] est une suite logicielle qui couvre tout le cycle de vie de développement des applications ubiquitaires. DiaSuite fournit un langage de description d'applications et de leurs environnements. Ce langage, appelé DiaSpec [Cas11], est basé sur le modèle architectural *Sense/Compute/Control (SCC)* [MT10] généralement utilisé dans le domaine de l'informatique ubiquitaire. Ce modèle présente trois catégories de composants, comme les montre la figure 3.5 : **les ressources**, tant matérielles que logicielles, qui sont capables de capter des données de l'environnement ou d'agir sur l'environnement ; **les composants contextes** qui sont responsables d'interpréter et d'agréger des données brutes collectées de l'environnement et **les contrôleurs** qui reçoivent des informations de contextes puis invoquent les actionneurs.

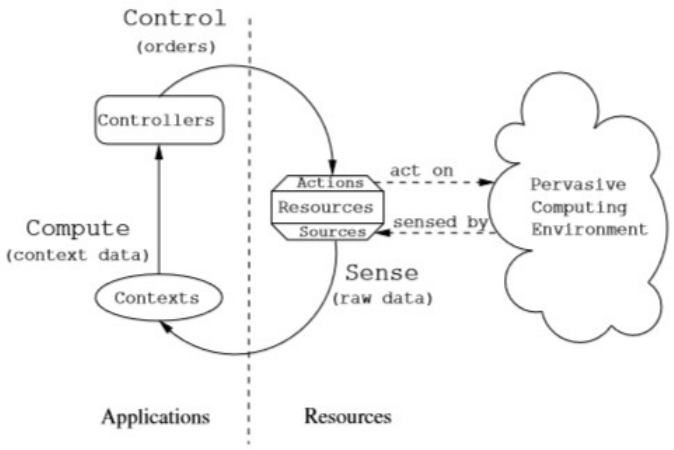

Figure 3.5: Le modèle de conception de DiaSpec [Jak11].

Le langage de conception DiaSpec est utilisé, dans un premier volet, pour définir une taxonomie d'entités décrivant les capacités (capter des données ou agir sur l'environnement) des ressources matérielles ou logicielles. Puis, dans un second volet, il est utilisé pour décrire l'architecture d'une application ubiquitaire. L'utilisation de ce langage concerne deux acteurs principaux :

- **expert du domaine :** DiaSpec permet aux experts du domaine de décrire les ressources qui peuvent être trouvées dans l'environnement ubiquitaire. Les experts doivent décrire alors tous les dispositifs tant capteurs que actionneurs, leurs propriétés, les données et les informations qu'ils peuvent fournir et les actions qu'ils peuvent effectuer.
- **architecte logiciel :** DiaSpec est utilisé par les architectes logiciels pour concevoir leurs applications par rapport aux ressources déclarées. Ils doivent ainsi décrire la logique applicative décomposée essentiellement par les composants contextes et les contrôleurs. De plus, ils doivent spécifier les interactions entre les ressources, les composants contextes et les contrôleurs.

La suite logicielle DiaSuite présente un modèle à base de composants afin de faciliter la conception et le développement d'applications ubiquitaires. En effet, dans DiaSuite, la spécification d'une application peut être représentée par un graphe orienté dont les nœuds sont les composants et les arcs indiquent le sens d'échange des données entre ces composants. Le graphe de flux de données peut être structuré en quatre couches:

- **les capteurs** envoient les données brutes et les informations collectées à partir de

l'environnement à la couche d'opérateurs de contexte.

- **les opérateurs de contexte** prennent en charge l'interprétation et l'agrégation des données brutes fournies par les capteurs. Ils sont ainsi responsables de transformer les données brutes en des données plus significatives. Ces informations de contexte sont envoyées par la suite vers les contrôleurs.
- **les contrôleurs** transforment les informations fournies par les opérateurs de contexte en des ordres vers les actionneurs.
- **les actionneurs** déclenchent des actions sur l'environnement.

La figure 3.6 montre une représentation graphique de la conception de l'application *Hello World*. Les flèches indiquent le sens des flux d'informations. Deux types de ressources sont identifiés : la ressource en bas du diagramme est un capteur qui fournit des informations au composant du contexte tandis que la ressource en haut du diagramme est un actionneur sur lequel agit le contrôleur. Dans cette figure, le composant du contexte est nommé *Presence* (il permet de signaler la détection d'une présence) et le contrôleur correspond au composant *DisplayController* (il permet d'afficher des messages sur les écrans lorsqu'une présence est signalée).

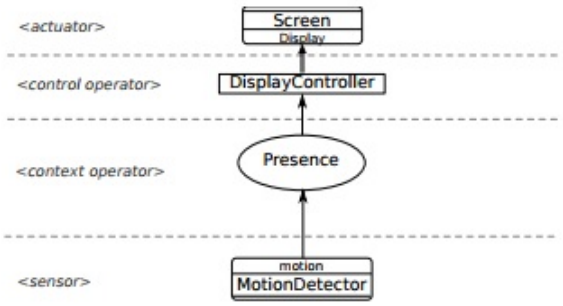

Figure 3.6: Architecture de l'application *Hello World* dans DiaSuite [Ber+14].

Bien que la suite logicielle DiaSuite introduise la notion du contexte à travers un composant responsable de l'interprétation et l'agrégation de données brutes fournies par les différents capteurs, elle ne traite pas les aspects liés à la gestion du contexte. Dans DiaSuite, la notion de contexte est ainsi limitée aux composants de type opérateurs de contexte. Un composant de ce type peut appartenir à une ou plusieurs applications. Il permet de collecter les informations brutes depuis des capteurs dispersés dans l'environnement ubiquitaire, les transformer en des informations plus significatives, puis les envoyer vers les contrôleurs.

A partir d'une description en DiaSpec, un compilateur, appelé DiaGen [Cas+09], génère un framework de programmation Java. Ce framework est utilisé afin de guider et supporter les développeurs durant la phase d'implantation de leurs applications. Plus particulièrement, une classe abstraite est générée pour chaque composant déjà déclaré. Les classes générées sont des classes abstraites qui présentent des méthodes abstraites pour guider la programmation de la logique applicative (par exemple, déclencher des actions de l'entité) et des méthodes concrètes pour soutenir le développement (par exemple, une entité de découverte). L'implantation de différents composants se fait ainsi par des classes qui étendent des classes abstraites déjà générées.

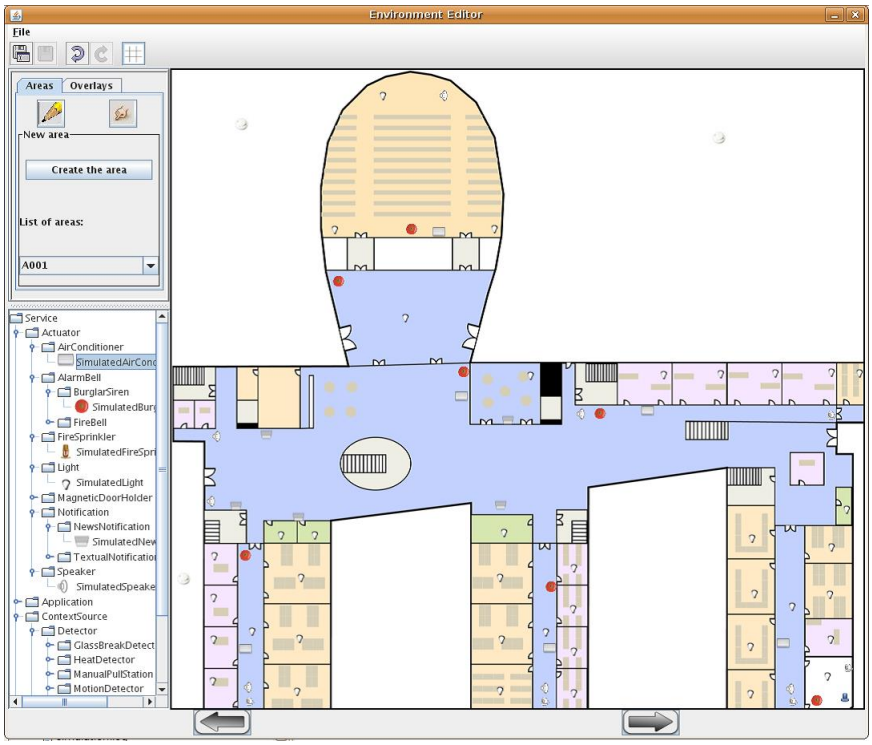

Figure 3.7: Éditeur de simulations DiaSim [BJC09].

Afin de faciliter le test et la validation de l'application, la suite logicielle DiaSuite intègre un outil de simulation, nommé DiaSim [BJC09]. Ce simulateur supporte la modélisation d'un environnement informatique ubiquitaire ainsi que les différentes entités définies dans la description DiaSpec. DiaSim fournit un éditeur qui permet de définir un ensemble de scénarii permettant de tester l'application développée. Grâce à un outil de visualisation 2D, l'application simulée peut être surveillée et son comportement peut être validé.

En outre, DiaSim supporte la simulation hybride, c'est-à-dire il est possible de tester l'application dans un environnement hybride avec des entités physiques réelles et des entités simulées. La figure 3.7 illustre un exemple d'environnement ubiquitaire simulé par DiaSim.

En conclusion, DiaSuite offre un support complet au développeur pendant les différentes phases de construction d'une application ubiquitaire. La figure 3.8 résume les différentes étapes de développement d'une application ubiquitaire avec DiaSuite que nous avons déjà présenté en détail :

- un expert du domaine déclare la taxonomie des ressources qui peuvent être trouvées dans un environnement ubiquitaire.
- un architecte décrit les interactions entre les ressources, les composants du contexte et les contrôleurs.
- à partir de la description de la taxonomie et de l'architecture de l'application, le compilateur DiaGen génère un framework de programmation Java qui sera utilisé pour aider et supporter le développeur pour implanter son application
- le code de l'application peut être testé en utilisant le simulateur DiaSim dédié aux environnements informatiques ubiquitaires.
- un administrateur du système peut déployer l'application dans un environnement ubiquitaire réel.

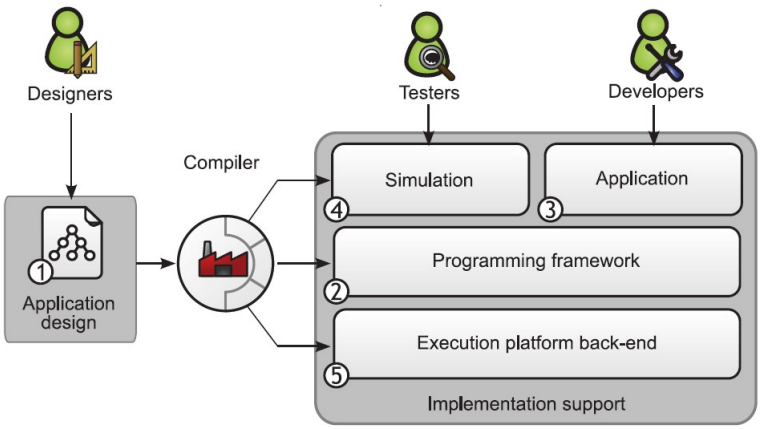

Figure 3.8: Cycle de développement d'une application avec DiaSuite.

DiaSuite propose une solution pour gérer les conflits. Cette solution décrite dans [JCL11] se base principalement sur deux étapes fondamentales : la détection des conflits au design time et leur résolution à l'exécution. En effet, les conflits potentiels sont détectés au design time depuis les descriptions d'applications. Pour résoudre ces conflits, Jacob introduit la notion d'état, par exemple état d'incendie et il leur attribue des priorités. A l'exécution, les conflits sont ainsi résolus en attribuant aux applications des droits d'accès en fonction des états activés et des priorités.

Toutefois, cette solution présente des limites importantes : la résolution des conflits nécessite l'intervention du développeur d'application qui doit implémenter le composant état. De plus, Jacob propose seulement une solution pour la gestion de conflits qui peut se produire au niveau des actionneurs utilisés par les applications ubiquitaires et il ne traite pas les conflits de l'environnement.

3.3.2 PCOM

PCOM [Bec+04] est un système à base de composants pour l'informatique ubiquitaire. Il est développé au-dessus de BASE [Bec+03], un middleware de communication dédié pour les environnements ubiquitaires. BASE fournit un ensemble de mécanismes assurant la découverte et la communication avec des dispositifs hétérogènes.

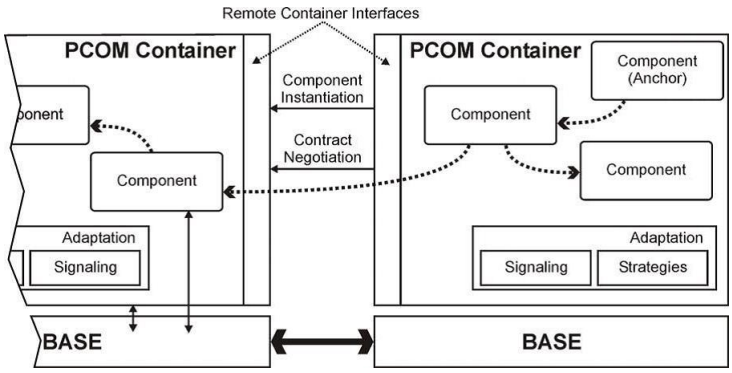

Figure 3.9: Architecture de PCOM [Bec+04].

PCOM fournit un modèle distribué d'applications ainsi que des mécanismes et des stratégies pour l'adaptation d'applications, comme les montre la figure 3.9. Une application est ainsi une composition de composants dont les dépendances sont explicitement spécifiées par des contrats. Ces composants sont hébergés dans un conteneur PCOM. Ce dernier

fournit un environnement d'exécution pour les composants d'une application et gère leurs dépendances ainsi que leurs cycles de vie.

PCOM fournit un modèle d'application à base de composants. Une application est représentée sous la forme d'un arbre de composants où le composant racine identifie l'application ainsi que son cycle de vie. L'arborescence de l'application, comme la montre l'exemple dans la figure 3.10, décrit les dépendances (interface de service ou événement) entre les composants. Une dépendance a ainsi une direction et peut présenter deux types de communication : une communication est de type pull lorsque le composant exige certaines interfaces de service et elle est de type push si le composant écoute certains événements fournis par un autre composant. En outre, ces dépendances sont spécifiées par des contrats où chaque contrat définit les fonctionnalités offertes aux autres composants, les fonctionnalités requises d'autres composants et les ressources requises de la plateforme d'exécution telles que les bibliothèques et l'espace mémoire nécessaire, etc.

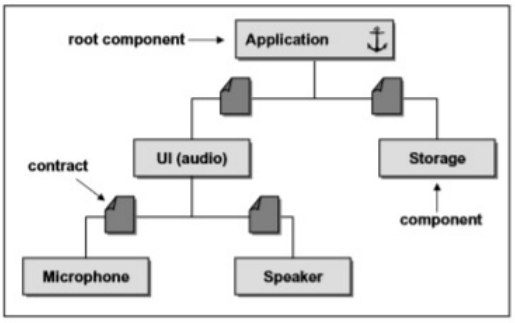

Figure 3.10: Architecture d'application dans PCOM [TSB07].

En plus de dépendances fonctionnelles, des paramètres non fonctionnels peuvent être spécifiés pour exprimer d'autres propriétés, comme par exemple le niveau de la batterie. Contrairement à la spécification fonctionnelle bien connue au moment de la compilation, les paramètres non-fonctionnels peuvent varier à l'exécution. En effet, ces paramètres peuvent dépendre des composants qui sont utilisés pour satisfaire les dépendances. Ces derniers peuvent être ainsi statiques ou dynamiques.

PCOM est capable de composer dynamiquement les applications à partir des composants disponibles et de les adapter automatiquement à l'évolution de leur environnement en utilisant des mécanismes et des stratégies d'adaptation. En effet, pour exécuter l'application, le système recherche des candidats pour chaque dépendance, en commençant par le com-

posant racine. Une fois qu'une configuration complète est trouvée, l'application peut être exécutée.

Si un composant devient indisponible, l'application doit être adaptée. Pour ce faire, le conteneur est notifié. Une fois notifié, ce dernier arrête l'application et commence à rechercher un remplaçant approprié. Si un remplaçant est trouvé, il reprend l'application. Dans le cas contraire, le conteneur marque le composant comme indisponible et il répète le processus pour son composant parent. Si le composant racine est atteint et aucune configuration n'est trouvée, le conteneur peut arrêter l'application.

En outre, PCOM propose une solution pour la gestion des conflits qui peuvent se produire entre les applications dans un environnement ubiquitaire. Cette solution s'articule principalement sur la gestion de conflits à l'exécution. Ces conflits sont spécifiés à l'avance par les développeurs d'applications et par l'utilisateur. Pour gérer les conflits, [TSB07] propose un gestionnaire de conflits qui est responsable de la détection et la résolution de conflits à l'exécution. Avant qu'une application soit exécutée, elle doit communiquer ses effets sur le contexte au gestionnaire de conflits. Ce dernier utilise les spécifications des conflits ainsi que l'état actuel de l'environnement physique (comment l'environnement est actuellement affecté par les applications actives) pour déterminer ensuite si ces effets vont conduire à un conflit. Si un conflit est détecté, le système doit le résoudre automatiquement. La stratégie utilisée pour résoudre ce conflit est d'interdire l'exécution de l'application qui va le provoquer. Cependant, une configuration alternative de cette application (adaptation de l'application) peut être proposée permettant ainsi son exécution. Toutefois, cette stratégie de « premier arrivé, premier servi » utilisée pour résoudre les conflits n'est pas souhaitable pour les applications de sécurité et risque de mettre les utilisateurs en danger. De plus, cette solution proposée par Tutlies présente une autre limite majeure : en effet, la gestion des conflits nécessite l'intervention du développeur d'applications qui doit déclarer pour chaque application ses impacts sur l'environnement et ce qu'elle considère comme conflits.

3.3.3 Gaia

Gaia [Rom+02b; Rom+02a] est un middleware dédié à la construction d'applications dans les environnements ubiquitaires de type *active space* défini comme étant une extension des espaces physiques. Un espace physique est en effet considéré comme une région physique avec des frontières bien délimitées, contenant des objets connectés et des utilisateurs qui effectuent des activités. Un *active space* est alors un environnement physique contenant une infrastructure logicielle qui permet d'une part à l'utilisateur d'interagir d'une manière transparente avec son environnement et de soutenir d'autre part le développement et

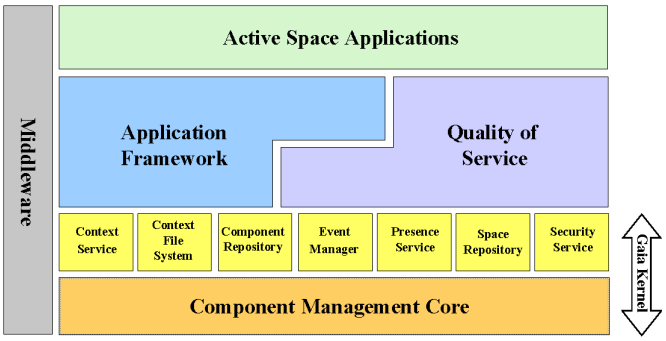

Figure 3.11: Architecture de Gaia [Gai].

l'exécution d'applications ubiquitaires. Dans ce contexte, Gaia est présenté comme une couche d'abstraction à cet environnement et à ses ressources.

l'architecture de Gaia est illustrée par la figure 3.11. Son noyau (*Component Management Core*) est responsable de gérer dynamiquement le chargement, le déchargement, le transfert, la création et la destruction des composants de Gaia. Ces composants étant distribués sur différents noeuds (c'est-à-dire les dispositifs qui proposent un service exportant leurs fonctionnalités vers Gaia) présents dans un *active space*, le noyau s'appuie sur l'architecture à service CORBA (*Common Object Request Broker Architecture*) [Gro08] afin de prendre en charge leurs interactions. De plus, le noyau Gaia propose un ensemble de services afin de faciliter la construction d'applications ubiquitaires.

Component Repository : c'est un registre responsable de stocker tous les composants Gaia qui peuvent s'exécuter dans un *active space* ainsi que les informations relatives à ces composants tels que le nom et l'OS requis, etc. Ce registre propose aussi les fonctionnalités nécessaires pour le déploiement de ces composants permettant par exemple de les télécharger (ou de les mettre à jour) sur les nœuds Gaia.

Space Repository : c'est une base de données qui contient toutes les informations sur les dispositifs et services actifs dans l'*Active Space*. Toutes ces ressources ont une description en XML spécifiant un ensemble de propriétés (par exemple, le nom, la localisation, le type, etc.). Les applications peuvent alors interroger cette base pour chercher une ressource requise.

Event Manager : ce service permet de notifier les applications à travers des événements envoyés à travers des canaux suite à des changements effectués dans l'*active space* (par exemple, apparition ou disparition d'une ressource, ajout ou retrait d'un composant, etc.). Les applications doivent s'abonner alors à ces canaux afin d'acquérir

les événements publiés. Elles sont aussi capables de définir leurs propres canaux et transmettre leurs propres événements.

Context File System : il permet d'organiser les données et les ressources d'une façon hiérarchique. Ces données peuvent être alors utilisées par les applications pouvant à leur tour ajouter d'autres données. Un exemple de requête est le suivant : /**type**:/parasol/**location**:/North-Garden/**weather**:/sunny. Cette requête permet de récupérer la liste des parasols du jardin Nord lorsqu'il fait un temps ensoleillé.

Context Service : ce service propose des informations contextuelles concernant l'*active space*. Les applications peuvent consommer ce service afin d'inférer des informations de plus haut niveau sur leur contexte. Gaia propose en effet un modèle de règle et un langage associé afin de permettre à ses applications d'utiliser leur contexte. **Context**(number_of_person, RoomF018, >, 0) => **Context**(is_free, RoomF018, =, false) : est un exemple de règle permettant de conclure l'état de la pièce F018 (libre ou occupée) .

Presence Service : ce service permet de connaître l'état actuel (actif ou inactif) de toutes les ressources de l'*active space*. Ces ressources doivent envoyer d'une manière régulière un signal (*heartbeat*) à ce service pour signaler leur présence. Toutefois, la non-acquisition de ce signal pendant un certain délai implique que cette ressource est inactive. Cela entraîne alors son retrait du registre. Gaia permet aussi de suivre la présence des personnes en installant un ensemble de capteurs spécifiques dans l'*active space*.

Security Service : ce service de sécurité prend en charge l'authentification de l'utilisateur et le contrôle d'accès aux ressources. Il implante un protocole de communication qui sert à préserver l'identité de l'utilisateur ainsi que des stratégies pour contrôler les accès à différentes ressources dans un *active space*.

[Ran+05] propose un langage Olympus pour développer à un haut niveau d'abstraction des applications ubiquitaires dédiés à des *active space* basés sur Gaia. Le modèle de programmation proposé présente deux propriétés importantes :

- **la découverte sémantique des entités :** les besoins (services, dispositifs, etc.) de chaque application sont décrits d'une façon abstraite par le développeur. Ces besoins sont liés aux différentes entités de l'*active space* selon le contexte courant à l'exécution.
- **des opérations de haut niveau relatives au domaine de l'*active space* :** Olympus définit un ensemble d'opérations relatives aux *Active Space* tels que le

démarrage ou l'arrêt d'un dispositif, etc. intégrées dans le modèle de programmation proposé, ces opérations peuvent être utilisées par les développeurs, ce qui n'incite pas à se préoccuper de leur exécution. Toutefois, il n'est pas envisageable d'ajouter de nouvelles opérations.

Gaia propose aussi une solution pour la gestion des conflits [RC03] entre les applications ubiquitaires sensibles au contexte. Pour chaque application, le développeur doit décrire l'ensemble des expressions contextuelles ayant un impact sur son comportement et spécifier les actions à effectuer sur l'environnement lorsqu'une expression devient vraie à l'exécution. Les conflits peuvent alors se produire entre ces actions déclenchées par des expressions devenant varies simultanément. Ces conflits sont gérés à l'exécution grâce aux priorités attribuées aux expressions du contexte par le développeur d'applications. Certes, une approche basée sur des priorités est indispensable pour traiter les conflits à l'exécution. Toutefois, il s'avère complexe pour le développeur de considérer toutes les situations de conflits pendant la phase de conception. En effet, il lui est inabordable de connaitre toutes les actions définies par les autres applications. De plus, les priorités attribuées peuvent ne pas répondre aux préférences de l'utilisateur.

3.3.4 Synthèse

Certes, toutes les plateformes ubiquitaires que nous avons présentées dans les sections précédentes proposent des fonctionnalités permettant de cacher certaines complexités du développement d'applications ubiquitaires dans le but de faciliter leur programmation et leur exécution. Toutefois, elles présentent une grande diversité dans les fonctionnalités proposées. Toutes ces plateformes ubiquitaires sont des plateformes multi-applications. Elles sont ainsi capables d'héberger des applications appartenant à des domaines différents tels que la sécurité, la santé, le confort et l'énergie, etc.

Toutes ces plateformes proposent des modèles de développement d'applications. DiaSuite, PCOM présentent des modèles à base de composants dans le but de faciliter la conception et le développement d'applications. Dans Diasuite, les composants d'une application peuvent être structurés en quatre couches : les capteurs qui transmettent les informations captées de l'environnement vers la couche contexte, les opérateurs de contexte qui interprètent et agrègent les informations fournies par les capteurs, les contrôleurs qui traduisent les informations fournies par les opérateurs de contexte en des ordres vers les actionneurs et, enfin, la couche des actionneurs qui déclenchent les actions sur l'environnement. Dans PCOM, une application est représentée sous la forme d'un arbre de composants où l'arborescence de l'application décrit les dépendances entre les composants

et le composant racine identifie l'application ainsi que son cycle de vie. Dans Gaia, les applications sont décrites par un modèle de programmation de haut niveau d'abstraction, nommé Olympus. Toutefois, la conception de ces applications reste une tâche difficile parce qu'aucune assistance n'est proposée par Gaia (ou Olympus) pour guider cette phase. PCOM est la seule plateforme, parmi celles que nous avons étudiées, qui permet de définir la notion d'application grâce à un composant racine qui identifie l'application et gère son cycle de vie.

Les trois plateforme présentées proposent des solutions pour la gestion de conflits. Cependant, ces solutions sont limitées et incomplètes. En effet, les conflits pouvant se produire entre les applications ont été traités dans les trois plateformes ubiquitaires présentées précédemment à partir de points de vue différents. En effet, ces points de vue diffèrent principalement selon les réponses aux cinq questions suivantes :

- à quelle phase du cycle de vie d'application faut-il gérer les conflits ?
- quels sont les acteurs pouvant intervenir dans la gestion des conflits (le développeur, l'administrateur ou l'utilisateur).
- à quel moment de leurs occurrences (avant l'occurrence effective des conflits ou a posteriori) ?
- quels sont les intervenants dans les situations de conflit (c'est-à-dire, gérer les situations de conflit entre plusieurs utilisateurs ou entre les applications d'un seul utilisateur) ?
- quelle approche faut-il utiliser afin de gérer les conflits (la prévention, la détection et la résolution des conflits a priori ou leur détection et résolution a posteriori) ?

Les approches de gestion des conflits peuvent être appliquées dans les différentes étapes du cycle de vie de l'application : *offline*(avant l'exécution) ou *online* (pendant la phase d'exécution). La prévention des conflits *offline* se concentre généralement sur le développement des plateformes permettant l'implantation des applications moins exposées aux conflits alors que la prévention des conflits *online* permet de les éviter en utilisant des mécanismes proposés par la plateforme.

Plusieurs travaux se focalisent sur la détection des conflits *offline* (par exemple, via l'analyse de code [NIM05] ou grâce à la description d'applications [JCL11]). D'après ces travaux, nous avons pu constater que les conflits détectés *offline* sont potentiels et peuvent ne jamais se produire à l'exécution. De plus, cette détection *offline* nécessite de connaitre de

nombreuses informations sur l'environnement (ressources disponibles, localisation, etc.) et sur les applications ubiquitaires (par exemple, les applications s'exécutant sur la plateforme, les ressources requises, les conditions d'utilisation, les actions, etc.). La détection des conflits *offline* est alors une tâche complexe surtout que la maison intelligente est un environnement dynamique rendant son évolution imprévisible *offline* (ajout ou retrait de dispositif, installation des nouvelles applications, etc.).

D'autres approches se focalisent plutôt sur la détection des conflits *online*. Par exemple, dans [TSB07], les conflits sont détectés à l'exécution en utilisant les spécifications des conflits et l'effet de l'application sur l'environnement. Ceci permet de considérer seulement les applications invoquées dans des situations bien particulières.

Les solutions proposées pour résoudre les conflits *offline* sont souvent considérées pessimiste. Par exemple, [NIM05] propose d'éviter l'installation d'applications susceptibles d'entraîner des conflits. À l'opposé, d'autres approches sont plutôt optimistes et se focalisent sur la résolution des conflits *online*. Par exemple, dans [TSB07], la résolution repose principalement sur la stratégie du premier arrivé, premier servi. Cette stratégie n'est pas conseillée pour les applications de sécurité et peut mettre les utilisateurs en danger. Contrairement à [TSB07], les approches proposées pour [JCL11] et [RC03] utilisent des priorités pour résoudre les conflits lors de l'exécution.

À ce niveau, nous pouvons constater que la gestion des conflits *offline* est une tâche difficile vu la nature dynamique de l'environnement : Il est alors nécessaire de mettre en place des mécanismes pour la gestion des conflits à l'exécution

Un autre critère de classification des approches proposées est le moment de gestion de conflit. Certaines solutions sont préventives visant à anticiper les conflits et à les traiter a priori, c'est-à-dire avant que leurs effets indésirables se présentent dans l'environnement (Par exemple, les solutions proposées dans [RC03] et [JCL11] visent à gérer les conflits potentiels alors que la solution proposée pour PCOM [TSB07] permet les gérer juste avant leurs occurrences). Cette vision peut être appliquée *offline* et *online* et elle permet d'exécuter les applications sans conflits. D'autres approches, comme celle proposée par [Par+05], détectent les conflits *online* dès leurs occurrences et les résolvent en éliminant leurs effets indésirables présents dans l'environnement. Cependant, l'apparition effective des conflits est considérée comme une situation inadmissible par l'utilisateur. Il est alors souhaitable de les gérer avant leur occurrence effective. À ce niveau, nous pouvons aussi constater que la gestion des conflits, c'est-à-dire leur prévention ou leur détection et résolution a priori est une mesure préventive.

Les solutions proposées peuvent être aussi classés selon leurs intervenants. En effet, certaines approches comme celles proposées par [Par+05; TSB07] traitent les situations de conflits entre intervenants, c'est-à-dire lorsque plusieurs intervenants utilisent simultanément la même ressource via des applications différentes ou la même application pour répondre à des besoins différents. Toutefois, d'autres approches, comme celle proposée dans [JCL11], se concentrent plutôt sur la gestion des conflits entre les applications sans considérer ceux pouvant se présenter entre les utilisateurs. Dans ce cadre, nous pensons que la maison intelligente peut être utilisée par plusieurs intervenants mais elle doit avoir un seul administrateur local qui dirige le comportement de la maison. Il s'avère alors primordial de traiter le problème de base qui consiste à gérer les conflits pouvant se produire entre les applications, et non pas entre les utilisateurs.

Les approches proposées aujourd'hui pour la gestion des conflits nécessitent généralement une intervention humaine. C'est l'une des limites majeures de ces solutions. En effet, certaines solutions exigent l'intervention du développeur d'applications. Par exemple, dans la solution proposée pour gérer les conflits dans la plateforme PCOM [TSB07], les développeurs doivent faire un travail supplémentaire. Pour chaque application, ils doivent décrire leurs effets sur l'environnement et spécifier les conflits. Dans [JCL11], les développeurs doivent non seulement détecter les conflits intra application mais ils doivent aussi implanter les composants états nécessaires pour la gestion des conflits. D'autres solutions nécessitent l'intervention de l'utilisateur pour résoudre les conflits. Par exemple, dans l'approche proposée par [YIH15], la résolution est de la responsabilité de l'utilisateur. D'autres solutions nécessitent des tâches complexes pour l'administrateur, par exemple, dans la solution proposée dans [JCL11], l'administrateur doit identifier les conflits entre les applications à partir de leurs descriptions. Contrairement à ces approches, nous pensons qu'il est nécessaire de limiter l'intervention de ces acteurs dans le processus de gestion des conflits mais de ne pas les exclure.

3.4 Conclusion

D'après l'étude de l'état de l'art, nous avons pu constater que les plateformes ubiquitaires sont aujourd'hui au centre des solutions proposées pour la maison intelligente. Pour gérer les conflits entre les applications s'exécutant sur ces plateformes, les approches existantes se différent selon plusieurs points :

- **la phase de cycle de vie de l'application :** la maison intelligente est un environnement dynamique par définition. Ceci permet à la gestion des conflits *offline* d'être

une tâche difficile. Il est nécessaire de mettre en place des mécanismes pour les gérer *online* via une représentation de l'environnement.

- **moment de gestion des conflits :** l'utilisateur n'accepte pas l'apparition effective des conflits. Il est alors nécessaire de les gérer avant leur occurrence.
- **intervenant :** la maison intelligente peut être utilisée par plusieurs intervenants mais doit avoir un seul administrateur local.
- **acteurs :** Nous pensons qu'il faut limiter l'intervention de différents acteurs. Pour faire ceci, il est nécessaire de déléguer la complexité de la gestion des conflits à la plateforme. Ceci permet d'éviter de trop surcharger les applications et de faciliter la tâche du développeur et de l'administrateur et de diminuer leurs efforts.

Pour conclure, d'après l'étude de l'état de l'art, nous avons dégagé un ensemble d'exigences pour la gestion de conflit et des limites des solutions existantes auxquelles cette thèse doit répondre. Cette thèse doit en effet :

- gérer les conflits entre les applications d'une maison de plusieurs intervenants mais ayant un seul administrateur local qui gère son comportement ;
- gérer les conflits a priori avant leurs occurrences effectives ;
- mettre en place des mécanismes pour la gestion des conflits *online* via une représentation de l'environnement ; .

L'une des limites majeures des solutions existantes est la nécessité de l'intervention humaine. Cette thèse vise à limiter l'intervention des différents acteurs (développeur, utilisateur final et administrateur) dans le processus de gestion des conflits en déléguant la tâche complexe de gestion des conflits à plateforme ubiquitaire.

Chapter 4

Proposition

Sommaire

L'objectif principal de ce quatrième chapitre est de présenter notre proposition de recherche. À cet égard, nous commencerons par décrire d'une façon concise la problématique abordée dans ce manuscrit, suivie d'un constat, présenté d'une manière synthétique, autour des chapitres de l'état de l'art. Puis, nous décrirons l'approche générale de notre proposition de thèse qui va répondre à notre problématique de gestion des conflits entre les applications ubiquitaires de la maison. Les différentes parties de cette proposition ainsi que les discussions appropriées seront ensuite présentées d'une manière détaillée.

4.1 Problématique et objectifs

Dans une première partie de ce manuscrit, nous avons présenté l'informatique ubiquitaire. Nous nous sommes, en particulier, concentrés sur les caractéristiques des applications pervasives et des plateformes qui sont nécessaires à leur bon fonctionnement. Ces applications s'exécutent dans un environnement ouvert, distribué, hétérogène et dynamique. Elles doivent s'adapter aux ressources disponibles au moment de l'exécution, aux préférences et aux habitudes des utilisateurs et aussi, comme toute application, tenir compte des contraintes de sécurité spécifiques au domaine d'application (i.e., maison intelligente, ville connectée, santé connectée). Ces applications ont aujourd'hui un succès grandissant et sont de plus en plus répandues mais il reste encore de nombreux défis théoriques et techniques tels que la gestion des conflits et le contrôle des accès aux différentes ressources.

Dans une seconde partie, nous avons étudié les différentes méthodes existantes pour la gestion des conflits ainsi que les modèles de contrôle d'accès. Cette étude a été faite d'un point de vue généraliste puisque les problèmes de gestion des conflits et de contrôle d'accès sont des problèmes récurrents en informatique dans de nombreux domaines comme, par exemple, en système d'exploitation ou encore en base de données. Pour la gestion des conflits, nous avons classé les différentes solutions en fonction de leur approche pessimiste ou optimiste. Les modèles pour le contrôle d'accès sont nombreux et prennent en considération différents éléments comme des contraintes statiques ou dynamiques.

La troisième partie est une analyse des différentes plateformes logicielles existantes pour réaliser des applications pervasives. Nous avons présenté des plateformes dédiées aux applications pervasives et d'autres plus généralistes qui permettent également le développement de ces applications. Pour la suite de notre travail, nous avons fait le choix de travailler avec la plateforme iCasa qui est le résultat de travaux de recherche communs entre l'équipe Adèle du Laboratoire d'Informatique de Grenoble et Orange Labs. Pour rappel, cette plateforme se base sur un modèle généraliste de composants orientés services qui sont des composants iPOJO.

Voici une vision synthétique des constatations que nous avons faites tout au long de ces chapitres dédiés à l'état de l'art :

- **les besoins en gestion de conflits entre les applications sont importants.** Aujourd'hui, les utilisateurs veulent que leurs équipements (capteurs et actionneurs) soient partagés par leurs applications pour ne pas acheter des équipements en dou-

blon inutilement, ce qui évite également des installations multiples généralement non intuitives pour un non informaticien.

- **des comportements désagréables ou dangereux ne peuvent pas être acceptés par les utilisateurs.** Les applications pervasives ont des comportements autonomes qui doivent s'intégrer dans la vie de l'utilisateur ; elles ne doivent pas être source de gêne ou même de mise en danger.

- **l'informatique orientée service est très utilisée aujourd'hui.** Cette approche a été adoptée dans de nombreux domaines tels que la maison connectée, la ville intelligente ou l'industrie manufacturière.

- **l'informatique orientée service permet une grande souplesse mais soulève des problèmes de partage de ressources.** Dès lors que le système décide lui-même de certaines adaptations, il devient difficile de maintenir une cohérence pour partager les ressources entre les consommateurs de ces ressources.

- **le contexte d'exécution des applications influence le partage des ressources.** Il est absolument nécessaire d'avoir une connaissance du contexte d'exécution des applications pour s'adapter au mieux à l'environnement physique de l'utilisateur mais aussi à ses habitudes. De plus, le contexte permet d'adapter l'utilisation des différentes ressources en fonction de priorités prédéfinies ou opportunistes.

L'objectif de cette thèse est de fournir des outils pour gérer les conflits entre applications dans une plateforme ubiquitaire basée sur le contexte tout en limitant l'intervention humaine (développeur, opérateur ou utilisateur final). Plus précisément, nous portons notre attention sur les futures applications du marché de la maison intelligente qui consomment et se partagent les équipements découverts dans les maisons. Nous savons qu'en l'absence de contrôle, la maison peut se comporter de manière incohérente avec des effets perceptibles désagréables pour les occupants, voire dangereux dans certaines situations.

Notre proposition reprend des solutions présentées dans l'état de l'art et les étend pour raisonner à un plus haut niveau avec toute la flexibilité nécessaire. Tout d'abord, nous prenons en compte le contexte de la plateforme représentant l'environnement physique à un instant donné pour identifier et résoudre les conflits, ensuite, nous considérons la capacité d'adaptation de la plateforme et des applications pour sélectionner la stratégie à appliquer en cas de conflits. Dans notre approche, l'équipement au centre d'un conflit est toujours attribué à une seule application à la fois. Nous adoptons enfin une approche optimiste où les conflits sont détectés et résolus à l'exécution des applications.

Le contexte est un élément clé de notre proposition. Il repose sur un modèle causal qui se modifie en fonction des évolutions de l'environnement physique et pousse également vers cet environnement les modifications demandées par les applications, lorsque celles-ci sont réalisables. C'est justement parce que le contexte reçoit les demandes de modification des applications que nous avons choisi de détecter à ce niveau les demandes d'accès aux équipements de la maison.

Le contexte n'est pas une connaissance sur l'environnement strictement interne à la plateforme. Elle contient des informations de contexte qui sont publiées à l'attention des développeurs d'applications, en l'occurrence des informations représentant directement ou indirectement des équipements de la maison. Parce que l'utilisation des équipements et la durée d'utilisation sont dépendantes de la sémantique de chaque application, nous laissons aux développeurs le soin d'identifier dans leur code métier les ressources critiques dont ils ont besoin pour réaliser une tâche ou l'application dans son ensemble.

La plateforme ubiquitaire est le lieu par excellence où s'exécutent des tâches de supervision et de gestion. Elle découvre ainsi les équipements de la maison, crée et réactualise des informations de contexte et met en relation les applications avec les informations de contexte qu'elles consomment. Nous choisissons ainsi de nous appuyer sur la plateforme pour identifier les besoins en accès à des ressources critiques et exécuter la stratégie de résolution de conflits. Cette stratégie trouve son prolongement dans chaque application, car nous laissons les développeurs décider du comportement de chaque application, lorsque celle-ci doit céder (provisoirement) un équipement.

4.2 Approche générale

Dans cette section, nous présentons, dans un premier volet, une vision globale de l'approche proposée pour traiter les conflits pouvant se produire entre les applications d'une maison dans une plateforme ubiquitaire orientée services et nous résumons les principes et les éléments clés de cette approche. Puis, dans un second volet, nous décrivons les rôles de tous les acteurs intervenant dans le processus de gestion des conflits.

4.2.1 Présentation générale

Notre objectif est de faciliter la gestion des conflits entre les applications pervasives qui accèdent aux mêmes ressources en même temps tout en évitant les interventions humaines. Notre approche consiste à fournir des outils pour la gestion des conflits dans une plateforme ubiquitaire basée sur des composants orientés services. Afin de gérer les conflits, nous nous sommes inspirés des mécanismes de verrouillage utilisés, notamment, dans les systèmes d'exploitation et présentés dans le chapitre 2 sous-section 2.3.6. Avant de définir quels mécanismes de verrouillage nous proposons, il est important de formaliser ce qu'est un conflit dans le contexte des applications pervasives et, par conséquent, nous pourrons déterminer ce qu'est un service critique qui pourra utiliser les mécanismes de verrouillage. Dans notre approche, les applications n'interagissent pas directement avec les équipements ; elles interrogent le contexte présent dans la plateforme (figure 4.1). Le contexte comprend une réification des équipements sous forme de composants logiciels mais aussi des informations contextuelles. Le propriétaire de la plateforme construit ce contexte pour prendre en charge certains types d'applications (gestion d'énergie, gestion de la lumière, gestion de la température, etc.). Les développeurs utilisent les services définis dans le contexte pour simplifier leur code applicatif et se concentrer seulement sur la logique métier de leurs applications. Certains services du contexte présentent des méthodes potentiellement conflictuelles (par exemple, allumer et éteindre, ouvrir et fermer, etc.), nous les appelons des services critiques ou encore ressources critiques. Ces services ne peuvent pas être utilisés par plus d'une application simultanément.

L'aspect temporel est très important pour les applications ubiquitaires. D'une manière générale, ces applications agissent sur les équipements dispersés dans leur environnement afin de modifier leurs états qui peuvent être maintenus pendant une certaine durée. Cette durée peut être spécifiée à l'avance par le concepteur selon les besoins fonctionnels de l'application. Un exemple consiste à allumer les lumières dès la présence d'une personne pendant 10 secondes. En revanche, cette durée peut être contextuelle et ne peut être déduite qu'à l'exécution. Par exemple, une application permet d'allumer les lumières pendant

la présence d'une personne dans une zone. Dans cette situation, le début et la fin de cette période sont déclenchés par des événements du contexte. Par conséquent, si une autre application agit d'une manière contradictoire sur les même lumières pendant cette période, un conflit doit être identifié.

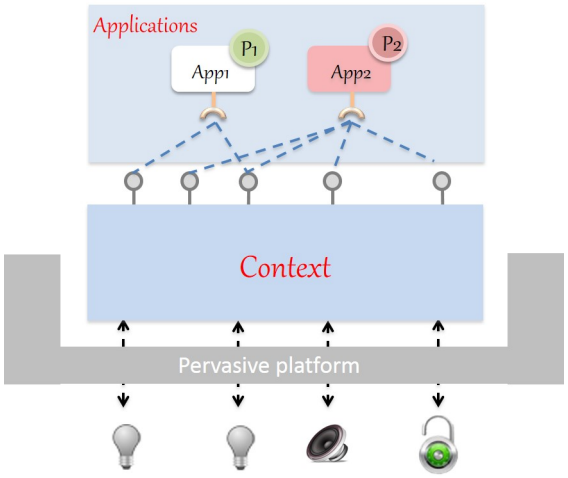

Figure 4.1: Principes de notre approche.

Notre proposition de gestion des conflits se base essentiellement sur les trois axes suivants :

- **une formalisation des conflits** direct et indirect dans un modèle de contexte d'une plateforme ubiquitaire ;
- **un mécanisme de verrouillage** pour les composants orientés services d'une plateforme ubiquitaire ;
- **une approche optimiste en trois phases** pour la gestion des conflits avec un mécanisme de verrouillage et de priorisation des applications qui se base sur :
  - **la prévention des conflits** en modifiant, notamment, la visibilité des services disponibles et la priorité des applications ;
  - **la détection des conflits** à l'exécution en fonction du modèle de programmation proposé avec les verrous ;
  - **la résolution des conflits** identifiés à l'exécution qui se fait en fonction de la criticité des applications.

Ces trois axes de notre proposition sont présentés en détail dans les trois sections suivantes de ce manuscrit. La section 4.3 est consacrée à définir les conflits directs et indirects dans un contexte orienté service. La section 4.4 présente le mécanisme de verrouillage, ses principes ainsi que les concepts clé pour les gérer. La section 4.5 décrit les trois phases de notre approche.

4.2.2 Acteurs

Pour ces travaux de recherche, nous faisons l'hypothèse d'applications logicielles développées par les partenaires de confiance de l'opérateur et d'une plateforme ubiquitaire reliant d'une façon dynamique et transparente ces applications et les équipements installés dans une maison. L'utilisateur de la plateforme peut chercher et installer les applications en fonction de ses besoins à partir d'un magasin d'applications ouvert seulement à l'opérateur de la plateforme et ses partenaires. Ces applications doivent être développées en utilisant les informations du contexte définies par un expert du domaine tout en respectant le modèle de programmation proposé. Nous distinguons trois acteurs principaux qui peuvent intervenir dans le processus de gestion des conflits : l'opérateur de la plateforme, l'utilisateur et le développeur. Nous définissons le rôle de chaque acteur dans la suite.

4.2.2.1 Opérateur de la plateforme

D'une manière générale, l'opérateur propose une solution *smart home* qui comprend une plateforme domotique, un ensemble d'équipements (détecteur de présence, détecteur de fumée, station météo, lumière communicante, alarme sonore, etc.) et d'applications logicielles développées par ses partenaires de confiance. Concrètement, ces applications sont implantées en s'appuyant sur les informations contextuelles publiées au niveau de la plateforme.

L'opérateur de la plateforme exerce la gouvernance nécessaire pour garantir la cohabitation des applications de la maison. Pour ce faire, il décide quelles informations du contexte peuvent être partagées par les différents acteurs métiers. Notre solution de gestion de conflits porte sur ces informations partagées et consommées par les applications proposées par ces acteurs. De plus, l'opérateur peut jouer un rôle principal dans le cadre de la prévention des conflits. Il peut spécifier des règles restrictives ou aussi exclusives qui permettent d'assurer une cohabitation optimisée des applications de la maison. Ces règles peuvent porter par exemple sur la visibilité des informations contextuelles par catégorie et/ou par application. Par exemple, l'utilisation d'une information du contexte peut être exclusive

à une seule application (par exemple, une seule application de sécurité). Ceci réduit les situations de conflits qui peuvent se présenter à l'exécution.

En outre, l'opérateur, en tant que gestionnaire de la plateforme *smart home*, doit décrire la loi générale de comportement des maisons de ses clients. Il est ainsi sa responsabilité d'insérer les applications dans un schéma qui fixe les niveaux de priorités des applications. Ces priorités vont servir à la résolution des conflits qui peuvent se produire entre les applications à l'exécution. Un niveau de priorité est accordé à une seule application à la fois.

Les applications sont classées selon des catégories (par exemple, sécurité, santé, confort, énergie) de niveaux de priorités différents. Par exemple, les applications de sécurité ont un niveau de priorité plus élevé que les applications de confort. De plus, chaque application doit appartenir à une seule catégorie à la fois. Aussi, les applications appartenant à la même catégorie peuvent être classées par ordre de priorité. Par exemple, une application d'incendie doit avoir un niveau de priorité plus élevé que celui attribué à une application d'intrusion.

4.2.2.2 Développeur d'application

D'une manière générale, le développeur implante son application comme si elle s'exécutera toute seule sur la plateforme et elle satisfera toutes ses exigences en ressources. Toutefois, cette vision peut affecter la logique métier de l'application qui peut ne pas satisfaire ses besoins fonctionnels dans les différentes situations de conflits pouvant avoir lieu.

La maison intelligente est un environnement ouvert par définition. Elle peut alors héberger des applications fournies par des acteurs différents. Ces applications vont partager les équipements installés dans leur environnement et y accéder potentiellement au même temps. Il s'avère ainsi important que le développeur d'application soit conscient du cadre conflictuel et concurrentiel d'applications à l'exécution. C'est pour cela que nous proposons un modèle de programmation étendu par des mécanismes de verrouillage pour gérer l'accès concurrent aux ressources et permettre ainsi l'identification des conflits avant qu'ils se produisent à l'exécution. Dans notre approche, le développeur d'application doit déclarer dans son code l'accès exclusif aux ressources critiques. Plus précisément, l'application doit acquérir un verrou avant d'utiliser une ressource critique et de le libérer lorsqu'elle n'en a plus besoin.

De plus, le développeur peut ajouter un code qui spécifiera le comportement de l'application lorsqu'elle n'a pas réussi à récupérer des ressources critiques. L'exécution de ce code permet l'adaptation de l'application et peut assurer la continuité du service attendu par l'utilisateur.

4.2.2.3 Utilisateur final

L'utilisateur final est considéré comme l'administrateur local de la plateforme domotique. Clairement, ce dernier ne peut pas être exclu du processus de gestion des conflits ainsi que de gouvernance du comportement de sa maison. Toutefois, il est nécessaire de noter que la gestion des conflits ne doit pas être de sa responsabilité. Dans notre proposition, nous visons à limiter sa tâche et à le faire intervenir que si nécessaire. En effet, l'utilisateur peut modifier les priorités associées aux applications selon ses préférences. Ce dernier peut changer l'ordre de priorité associé aux catégories d'applications. Par exemple, il peut accorder à la catégorie confort une priorité supérieure à celle d'énergie. L'utilisateur peut aussi changer l'ordre des applications de la même catégorie. Il peut, par exemple, modifier les priorités des applications de confort selon ses besoins.

De plus, l'utilisateur peut définir une stratégie de résolution parmi celles proposées par la plateforme. Dans cette situation, les priorités des applications sont calculées en fonction de la stratégie choisie. Par exemple, il peut spécifier la stratégie « Premier arrivé, premier servi » pour résoudre les conflits qui peuvent se produire entre les applications de la catégorie confort.

4.3 Formalisation des conflits dans un modèle de contexte

Dans l'état de l'art, nous avons pu constater que pour traiter les conflits, il est indispensable de savoir certaines informations sur l'environnement. Pour cela, dans cette thèse, nous proposons de gérer les conflits à travers une représentation de l'environnement sur la plateforme nommée contexte. Avant de définir les types majeurs de conflits et les illustrer par des exemples, nous présentons dans la sous-section suivante le modèle de contexte sur lequel nous nous sommes appuyés dans ce travail.

4.3.1 Services du modèle de contexte

Nous définissons dans cette thèse une approche où les conflits sont gérés à l'exécution par la plateforme ubiquitaire à travers un modèle du contexte. Plusieurs plateformes incluent déjà la notion du contexte, dont le but est de fournir des informations pertinentes du point de vue de l'application. Comme nous l'avons déjà présenté dans la sous-section 1.3.3 du chapitre 1, ces informations contextuelles peuvent décrire l'environnement informatique (par exemple, les ressources disponibles), l'environnement utilisateur (position géographique, besoins, préférences, etc.) et l'environnement physique (température, luminosité, bruit, etc.). Le contexte permet aux applications ubiquitaires de réagir aux évolutions et aux changements qui peuvent avoir lieu dans leurs environnements (par exemple, la disponibilité des ressources, le comportement de l'utilisateur, etc.). D'une manière générale, le contexte et les applications sont séparés. Cette approche architecturale facilite le développement des applications ubiquitaires sensibles au contexte. De plus, elle permet aux développeurs d'applications de se concentrer sur la logique métier de leurs applications, tout en déléguant la complexité de la gestion de contexte à la plateforme ubiquitaire.

Le contexte est un élément clé de notre approche. Il est définit comme un modèle causal représentant l'environnement physique sur la plateforme [Col+16; Ayg+16]. Ce modèle est causal, dans le sens où il est synchronisé avec l'environnement. En effet, les actions sur le modèle sont transmises à l'environnement et, inversement, les évolutions sont reflétées sur le modèle. Le module de contexte fait partie de la plateforme ubiquitaire, comme le montre la figure 4.2. Les applications s'exécutant sur une plateforme basée sur le contexte ne peuvent pas accéder directement aux dispositifs physiques installés dans leur environnement. Le contexte constitue ainsi le seul moyen pour ces applications d'interagir avec ces dispositifs.

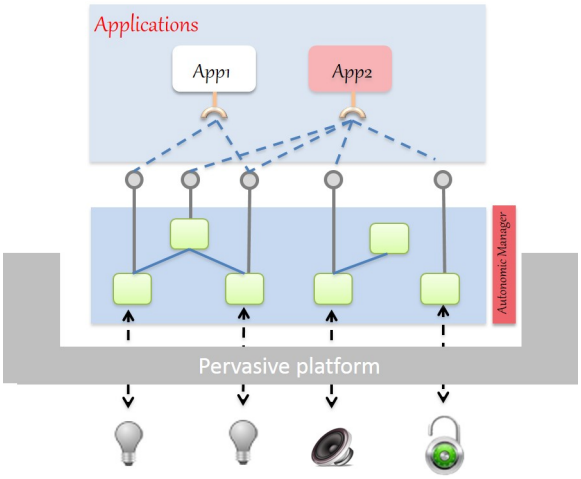

Figure 4.2: Modèle causal du contexte.

Notre proposition est fondée sur les plateformes orientées services où le contexte est modélisé sous la forme d'un graphe des composants orientés service. Certains de ces composants fournissent des services contextuels. Ils sont publiés au niveau de la plateforme et peuvent être consommés par les applications. D'autres composants fournissent des services qui ne sont pas publiés au niveau de la plateforme mais utilisés seulement dans le contexte. Les services peuvent représenter des dispositifs (capteurs et actionneurs) ou bien ils peuvent être abstraits et fournir des fonctions de plus haut niveau comme, par exemple, la luminosité moyenne dans une pièce. Ils peuvent également permettre d'agir sur l'environnement comme pour augmenter la température de la maison qui implique de modifier la température de plusieurs pièces. Les services abstraits sont très utiles pour les développeurs d'applications qui ne sont plus concernés par la construction et le maintien d'informations complexes et dynamiques.

Le module de contexte fait partie de la plateforme ubiquitaire. Il est construit par l'opérateur de la plateforme afin de supporter un ensemble bien défini d'applications. À l'exécution, il fournit dynamiquement les services requis par les applications en cours d'exécution. Cette adaptation est réalisée par un gestionnaire autonomique qui a pour but de créer et mettre à jour des services contextuels en fonction des besoins de la plateforme et des ressources disponibles dans l'environnement. Afin de faciliter le développement et la gestion du contexte, un langage orienté service spécifique au domaine (DSL: *Domain Specific Language*) a été développé par Colin et al. [Col+16] pour permettre la définition du module du contexte et de ses capacités d'autonomie.

4.3.2 Définition des conflits

Comme présenté précédemment, le contexte est modélisé comme un ensemble de services dynamiques mis à disposition dans la plateforme. Ces services peuvent correspondre à des dispositifs ou à des entités fournissant des fonctions utiles. Un des aspects importants du contexte est que les services publiés peuvent être partagés par les applications installées sur la plateforme. Ces applications consomment les services fournis selon leurs besoins afin d'atteindre des objectifs différents et parfois incompatibles. Cela peut causer des conflits qui doivent être résolus afin de maintenir les maisons dans des états cohérents. Nous distinguons deux types de conflits au niveau des services du contexte :

- **le conflit direct**, qui survient lorsque plusieurs applications invoquent le même service de manière incompatible. Nous comprenons dans le terme incompatible des actions contradictoires (ouvrir vs. fermer une porte), des configurations différentes (chauffer à 18 °C et chauffer à 20 °C), etc.
- **le conflit indirect**, qui se produit lorsque plusieurs applications invoquent des services différents en entraînant des actions incompatibles sur un service partagé.

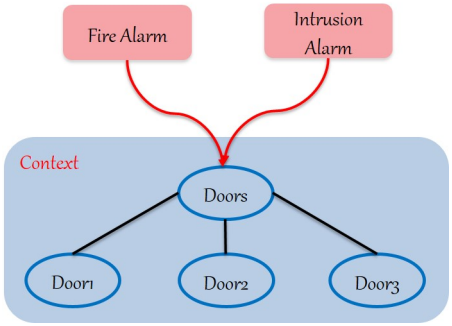

Figure 4.3: Conflit direct entre applications.

Pour illustrer ces deux types de conflits, nous examinons deux exemples simples. Tout d'abord, considérons une plateforme ubiquitaire qui héberge deux applications de sécurité, appelées « Application d'intrusion » et « Application d'incendie ». Leurs exigences sont les suivantes : l'application d'intrusion verrouille les portes d'entrée la nuit lorsque les habitants dorment ; l'application d'incendie s'assure que toutes les portes sont déverrouillées lorsqu'un incendie est détecté pour permettre l'évacuation des habitants. Dans le cas d'un incendie la nuit, les deux applications agissent sur les portes (par le même service du contexte) de façon contradictoire. Ce conflit doit être détecté et, évidemment, la priorité doit être accordée à l'application d'incendie (c'est l'application la plus critique). Cet exemple

de conflit direct où deux applications utilisent le même service (le service représentant les portes d'entrée d'une maison) est illustré par la figure 4.3.

Considérons une deuxième situation où la plateforme ubiquitaire fournit plusieurs services contextuels pour gérer les lumières dans une maison. Plus précisément, les services sont disponibles pour contrôler les lampes d'une manière individuelle et aussi pour contrôler la luminosité plus globalement dans une pièce, une zone ou une maison entière. Ces services sont très pratiques pour les développeurs d'applications et sont fournis par la plupart des plateformes ubiquitaires actuelles. Considérons maintenant deux applications : une application de gestion de la consommation énergétique et une application de gestion d'éclairage. Le but de la première application est d'optimiser la consommation d'énergie de la maison. Son implémentation fait appel à des services contrôlant des zones entières telles que le service luminosité fourni par la plateforme. L'objectif de la deuxième application est d'allumer les lampes installées dans une pièce lorsqu'une présence est détectée. Plus précisément, cette application utilise des services qui représentent les lampes dans la plateforme. Il apparaît clairement que ces deux applications ont des objectifs contradictoires : l'application de gestion de la consommation énergétique veut éteindre les lumières tandis que l'application de gestion d'éclairage cherche à les allumer. Bien qu'elles n'utilisent pas les mêmes services contextuels, le conflit doit être détecté et résolu. Cet exemple de conflit indirect où deux applications utilisent les mêmes ressources mais pas les mêmes services est illustré par la figure 4.4.

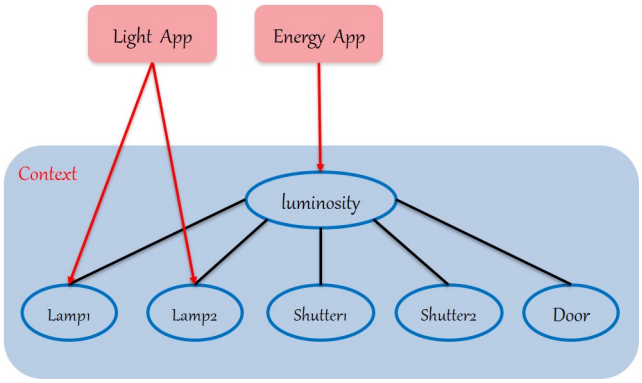

Figure 4.4: Conflit indirect entre applications.

Pour conclure, d'une manière générale, les conflits directs peuvent se produire lorsque deux ou plusieurs applications invoquent directement le même service du contexte. Ce service peut être un service abstrait ou tout simplement un service qui représente un dispositif dans la plateforme. En outre, les conflits indirects peuvent se présenter lorsque

deux ou plusieurs applications invoquent indirectement le même service. Deux situations de conflits indirects peuvent avoir lieu au niveau d'un contexte orienté services :

- une application utilise un service abstrait qui agit sur un service correspondant à un dispositif consommé par une autre application ;
- deux ou plusieurs applications consomment des services abstraits différents agissant sur le même service représentant un dispositif.

4.4 Mécanismes de verrouillage

Notre solution de gestion des conflits à travers le modèle du contexte s'appuie sur le mécanisme de verrouillage, présenté dans la sous-section 2.3.6 du chapitre 2, pour accéder aux services critiques partagés. En effet, un verrou est un mécanisme qui impose du contrôle d'accès à une ressource protégée. Plus précisément, nous l'utilisons dans notre proposition pour attribuer un service critique à une seule application à la fois. Un service est verrouillé lorsqu'il est utilisé par une application demandant un accès exclusif. Il est libre et peut être utilisé quand aucune application n'a pas encore posé un verrou. Si deux applications tentent d'acquérir un verrou sur une même période de temps, le verrou est accordé à celle qui a la priorité la plus élevée. Afin de gérer les verrous, nous avons associé un contrôleur à chaque service critique du contexte. Le rôle de ce contrôleur est présenté en détail dans la sous-section 4.4.2 de ce chapitre.

4.4.1 Verrou – les principes

Un verrou est caractérisé par deux propriétés importantes :

- **son type :** dans notre proposition, le verrou est exclusif ; il permet seulement à l'application possédant le verrou d'invoquer le service. Tout autre accès par une autre application est interdit.
- **sa granularité :** le verrou est défini et géré au niveau du service (simple ou abstrait) de contexte.

Un verrou a une durée de vie. Cette durée correspond à l'intervalle de temps pendant lequel un service critique est alloué à l'application détenant le verrou à l'exécution. En général, cette durée dépend de la logique métier de l'application. Dans notre proposition, elle peut être explicitement spécifiée par le programmeur au moment du développement ou peut être associée à une condition dépendant de services présents dans le contexte. Dans la deuxième situation, cette durée est contextuelle, donc elle ne peut pas être définie par le développeur d'application à la conception.

Cependant, le début et la fin de cette période ne peuvent être déduits qu'à l'exécution. A titre d'illustration, nous considérons l'application de détection d'incendie qui a pour objectif de maintenir la porte d'entrée déverrouillée pendant que la maison est en état d'incendie. Cette application doit poser le verrou dès qu'un incendie est détecté et le libérer lorsque la maison revient à son état normal, comme le montre la figure 4.5.

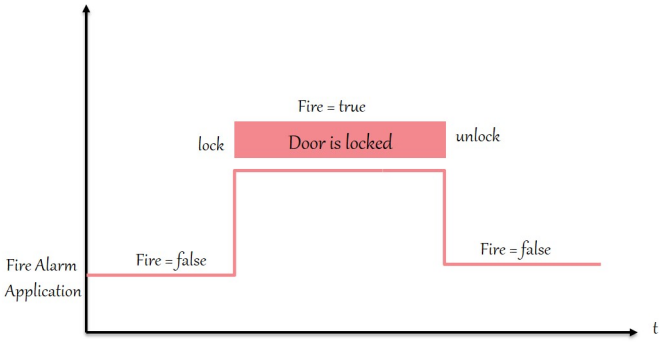

Figure 4.5: Exemple d'utilisation d'un verrou.

Le code de verrouillage et de déverrouillage d'un service ne peut pas être généré automatiquement par la plateforme ubiquitaire car l'acquisition du verrou et sa libération dépendent fortement de la sémantique des applications. Pour cette raison, nous proposons des méthodes pour obtenir et libérer le verrou. Ces méthodes doivent être utilisées par le développeur au moment de la conception selon les exigences fonctionnelles de ses applications. Pour gérer les verrous, un gestionnaire de conflits est associé à chaque application. Ceci est illustré par la figure 4.6 où ACM (*Application Conflict Manager*) désigne le gestionnaire de conflits d'application. Ce dernier fournit donc les API nécessaires pour verrouiller et déverrouiller les services critiques requis par l'application. De plus, il influence la visibilité des services. Son objectif est de cacher dynamiquement les services verrouillés aux applications de priorité inférieure que l'application possédant le verrou.

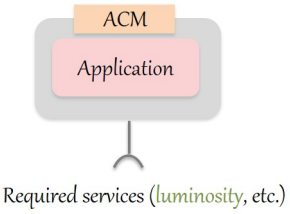

Figure 4.6: Gestionnaire de conflits pour une application.

Le code 4.1 présente un extrait du code d'une application de gestion d'éclairage où le gestionnaire de conflits est appelé pour verrouiller et déverrouiller les services représentant les lampes avec les méthodes **lockService()** et **unlockService()** qui prennent en paramètre la lampe du contexte sur laquelle nous appliquons le verrou.

Listing 4.1: Extrait du code utilisant les verrous fournis par le gestionnaire de conflits.

```
1  private void managelight(PresenceService presenceService){
2     String zoneName = presenceService.sensePresenceIn();
3     Set<BinaryLight> lightInZone = getLightInZone(zoneName);
4     if
          (presenceService.havePresenceInZone().equals(PresenceService.PresenceSensing.YES))
          {
5        // lock all Binary Lights in the zone
6        lightInZone.stream().forEach((light)
             ->conflictManager.lockService(services.get(light) ));
7        lightInZone.stream().forEach((light) ->light.turnOn() );
8     }
9     else {
10       lightInZone.stream().forEach((light) ->light.turnOff() );
11       // unlock all Binary Lights in the zone
12       lightInZone.stream().forEach((light)
             ->conflictManager.unlockService(services.get(light)) );
13    }
14 }
```

La figure 4.7 illustre le diagramme d'états associé à tous les services contextuels critiques de la plateforme (au niveau du composant). Lorsqu'un service est disponible, il est libre et peut être utilisé. Lorsqu'une méthode de verrouillage est appelée, le service devient verrouillé. Un attribut spécifique conserve l'identité de l'application demandant le verrou. Cet attribut correspond donc à l'application propriétaire du service à un instant donné. À tout moment, le service peut devenir indisponible pour différentes raisons liées au dynamisme de l'environnement telles qu'un dysfonctionnement ou une durée de vie limitée d'une batterie. De plus, le service peut être préempté par une application de priorité plus élevée. Dans les deux cas, une méthode de callback est appelée permettant l'adaptation de l'application.

Plusieurs situations de mauvaise pratique de notre modèle de programmation peuvent avoir lieu. A titre d'illustration, nous pouvons considérer le cas où un service critique est invoqué directement dans le code de l'application ; c'est-à-dire la méthode spécifique pour verrouiller le service n'a pas été appelée. Dans cette situation, lorsque l'application invoque le service à l'exécution, une exception est lancée et l'application ne peut plus utiliser ce service. De plus, l'administrateur de la plateforme peut être aussi notifié afin d'assurer la correction du bug.

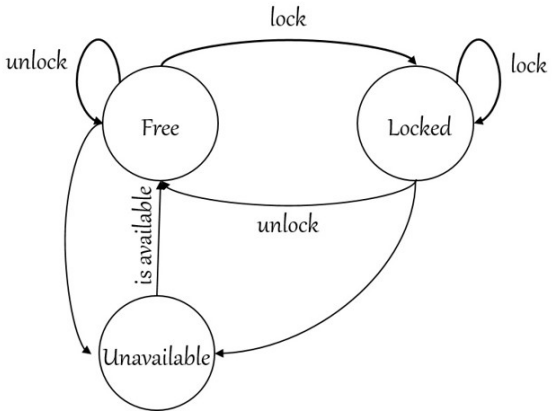

Figure 4.7: Diagramme d'états des services contextuels.

4.4.2 Les contrôleurs

Un conflit est identifié lorsqu'une application appelle la méthode spécifique pour acquérir le verrou correspondant à un service déjà verrouillé par une autre application. Pour résoudre les conflits, une stratégie de résolution est définie par un contrôleur associé à tous les services critiques fournis par le contexte. Ceci est illustré par la figure 4.8 où C désigne le contrôleur (*Controller*). Le contrôleur a pour but d'intercepter les invocations de service, vérifier l'existence d'un verrou et accorder l'accès au service si possible. Pour ce faire, le contrôleur retient les informations concernant l'état du service (libre ou verrouillé) et l'identité de son propriétaire (l'application qui a posé le verrou). Par conséquent, la décision concernant l'application qui peut acquérir le verrou est prise en fonction de ces informations (l'état du service et l'application propriétaire du service) ainsi que les priorités associées aux applications. Par exemple, lorsqu'une application demande de verrouiller un service qui représente un dispositif sur la plateforme, le contrôleur associé doit vérifier si le service est déjà verrouillé ou non. Dans le cas contraire, le service est attribué à l'application demandant l'acquisition du verrou. Cette application devient alors son propriétaire. Sinon, pour prendre sa décision, le contrôleur doit vérifier que l'application demandant le verrou est plus prioritaire que l'application propriétaire du service. Le verrou est ainsi accordé à l'application qui possède le niveau de priorité le plus élevé.

Dans notre proposition, le verrou présente une propriété clé : il peut être propagé, comme le présente la figure 4.9. En effet, lorsqu'une application pose un verrou sur un service abstrait, la plateforme propage le verrou jusqu'aux services feuilles représentant les dispositifs concrets. Par exemple, lorsqu'une application verrouille le service de luminosité, le verrou est transmis à tous les services représentant les lampes utilisées par le service luminosité. Dans le cas où une lampe requise est verrouillée par une autre application

avec un niveau de priorité plus élevé, l'application ne peut pas verrouiller le service de haut niveau (luminosité) requis. La propagation du verrou est ainsi de la responsabilité du contrôleur associé aux services abstraits.

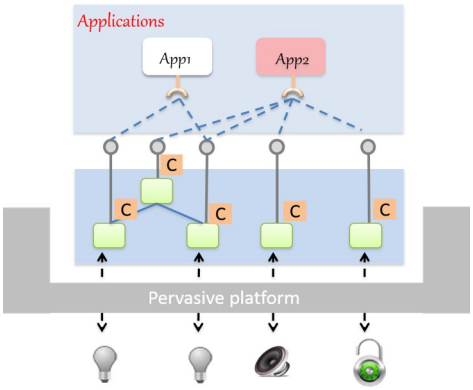

Figure 4.8: Les contrôleurs des services critiques.

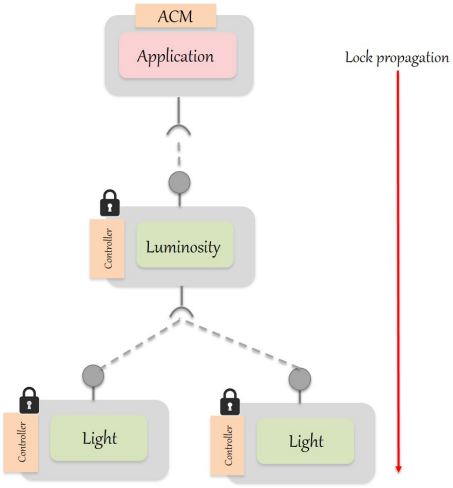

Figure 4.9: Propagation des verrous par les contrôleurs.

4.4.3 Visibilité des services

Notre stratégie de résolution de conflits est fondée sur la notion d'adaptation des applications. En effet, nous pensons que l'interruption de service est mal acceptée par l'utilisateur final. La résolution des conflits permet à l'environnement de rester dans une situation

cohérente mais exige aussi aux applications qui n'ont pas les droits d'utiliser un service de s'adapter. Toutefois, ceci peut être très complexe surtout dans certaines situations. C'est le cas lorsqu'un code a été exécuté avant d'invoquer le service demandé et que les actions doivent être annulées ou compensées. La notion de visibilité de service est l'un des aspects de notre approche de gestion des conflits. Lorsqu'un service est verrouillé par une application, il n'est plus visible pour toutes les applications ayant une priorité inférieure. Par conséquent, ces applications n'essayent plus d'invoquer le service verrouillé et doivent s'adapter immédiatement aux changements effectués dans leur contexte d'exécution. Ces applications exécutent le code écrit en fonction de la disponibilité du service verrouillé. Ceci est ainsi une stratégie pour éviter les conflits entre l'application détenant le verrou et les applications moins prioritaires et pour réduire de ce fait le nombre de situations de conflits qui peuvent se produire à l'exécution. Il s'avère ainsi nécessaire que l'application n'ayant pas les droits d'accès aux ressources continue à répondre aux besoins de son consommateur et à fournir les fonctionnalités demandées. L'adaptation est un processus permettant à l'application de satisfaire en continu ses exigences fonctionnelles dans un contexte changeant. Dans notre approche, l'adaptation est déclenchée par un stimulus local qui correspond précisément à l'invisibilité d'un service verrouillé par une autre application.

Dans notre proposition, le développeur d'application doit respecter le modèle de programmation proposé. Il peut ainsi implanter les méthodes de *callback*. Plus précisément, il peut spécifier le comportement de l'application lorsqu'un service requis n'est plus disponible. Au moment de l'exécution, lorsqu'un conflit est identifié, le service devient indisponible pour l'application avec le niveau de priorité le plus faible. Toutefois, cette application est notifiée et la méthode de *callback* est appelée permettant l'adaptation de l'application. Par exemple, pour s'adapter, une application peut utiliser un autre dispositif installé dans la même zone ou tout simplement, une notification peut être envoyée à l'utilisateur final pour l'informer de l'indisponibilité du dispositif.

4.5 Approche optimiste en trois phases

Afin de gérer les conflits au niveau du contexte orienté service, notre proposition est organisée sur trois axes : le premier axe consiste à prévenir les conflits entre les applications à l'exécution, le deuxième axe vise à détecter les conflits lorsque leur prévention est hors de portée et le dernier axe se penche sur la résolution des conflits déjà identifiés. Nous détaillons ces trois étapes dans les sous-sections suivantes.

4.5.1 Prévention des conflits

La prévention est une étape primordiale dans le processus de gestion des conflits qui peuvent se présenter entre les applications de la maison. Nous pensons qu'il est nécessaire que la plateforme domotique recherche des approches de prévention surtout que le nombre de situations de conflits s'accroît en fonction du nombre d'applications installées sur la plateforme.

Afin de réduire ces situations et de garantir une exécution d'applications moins exposée aux conflits, nous proposons une stratégie de prévention. Contrairement aux approches présentées dans l'état de l'art (cf. sous-section 2.2.3 du chapitre 2), notre stratégie s'applique à la phase d'exécution du cycle de vie de l'application. Cette stratégie s'appuie sur les mécanismes de verrouillage, de visibilité des services, de priorités des applications et d'un code d'adaptation. Concrètement, lorsqu'une application pose un verrou sur un service, la plateforme rend invisible ce service pour toutes les autres applications moins prioritaires. Dans notre proposition, les applications peuvent présenter des capacités d'adaptation. Ces capacités leur permettent de s'adapter dynamiquement en fonction des services disponibles afin d'assurer la continuité de leur fonctionnement. Cette mesure préventive permet aux applications de ne pas exécuter les lignes de code écrites pour un service déjà verrouillé. Ceci permet alors de réduire les situations de conflits en évitant celles qui peuvent se présenter entre l'application possédant le verrou et les applications les moins prioritaires.

La figure 4.10 illustre notre stratégie de prévention où *App1* et *App2* désignent les applications 1 et 2 (*App2* possède une priorité P2 supérieure à la priorité P1 associée à *App1*) et Si désigne un service i. L'application *App2* pose un verrou sur le service S2 initialement libre. En conséquence, *App1* moins prioritaire ne voit plus ce service jusqu'à ce que ce dernier soit libre de nouveau. Cette stratégie permet d'éviter le conflit qui peut se produire entre les applications *App1* et *App2*.

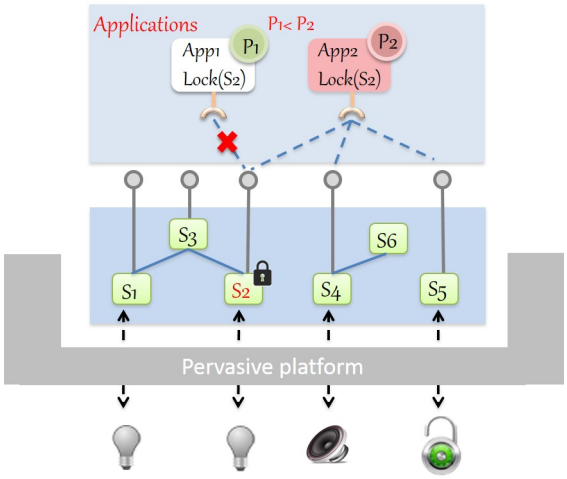

Figure 4.10: Exemple de prévention de conflits entre applications.

4.5.2 Détection et résolution des conflits

4.5.2.1 Détection des conflits

Comme nous l'avons déjà discuté dans la sous-section 2.2.3 du chapitre 2 de l'état de l'art, les conflits potentiels peuvent être identifiés en phase de conception. Pour ce faire, il est nécessaire de spécifier beaucoup d'informations concernant l'application (par exemple, les ressources requises, les effets sur l'environnement, les conditions déclenchant les actions, etc.) souvent d'une manière formelle. Cela peut compliquer considérablement la tâche du développeur qui ne peut pas être au courant de toutes les applications potentielles. Pour cette raison, nous pensons que les plateformes ubiquitaires doivent fournir des mécanismes permettant aux programmeurs d'exprimer des situations conflictuelles tout en gardant le code de l'application simple et vérifiable.

La détection des conflits au moment de la conception est très difficile. Ceci est dû principalement à la nature imprévisible des environnements d'exécution. De mauvaises interactions peuvent se produire à tout moment et sont très dépendantes des conditions d'exécution. Dans notre proposition, nous adoptons une approche optimiste où les conflits sont identifiés à l'exécution au plus tard possible. En effet, le programmeur doit utiliser le modèle de programmation étendu par les mécanismes de verrouillage proposé. Plus précisément, il doit déclarer dans son code l'accès exclusif aux services critiques requis par l'application. A l'exécution, les conflits sont localisés et identifiés par la plateforme au niveau des services du contexte. Concrètement, un conflit est détecté lorsque deux applications demandent un accès exclusif au même service critique pendant le même intervalle

de temps.

4.5.2.2 Résolution des conflits

La stratégie proposée pour la résolution de conflits identifiés à l'exécution est fondée essentiellement sur les priorités d'applications, la visibilité des services et un code d'adaptation. En effet, l'utilisateur peut installer sur sa plateforme des applications appartenant à des domaines tels que la sécurité, le confort, l'énergie et la santé. Il est ainsi clair que dans le cas d'un conflit entre une application de sécurité et une application d'énergie, la ressource doit être accordée à la première application qui est la plus critique. Pour résoudre les conflits, il est ainsi primordial d'attribuer des priorités aux applications afin de ne pas mettre les habitants et leurs biens en danger. Nous proposons donc de classer les applications selon un schéma qui fixe leur ordre de priorités. Ce schéma se base principalement sur les catégories d'applications ordonnées selon des priorités. Un niveau d'application est associé à une seule application à la fois.

Dans le cas d'un conflit, le service sera alloué à l'application ayant la priorité la plus forte, même par préemption. De l'autre côté, ce service devient invisible pour l'application la moins prioritaire. Ceci peut alors provoquer l'interruption de fonctionnement de cette application. Toutefois, nous pensons que l'interruption du service est mal acceptée par l'utilisateur. Ce dernier exige un fonctionnement en continu de ses applications. Dans notre approche, en cas d'un conflit, l'application ayant la priorité la plus faible doit continuer à répondre aux exigences de l'utilisateur et lui fournir le service attendu. Il s'avère ainsi nécessaire que cette application s'adapte dynamiquement aux changements produits dans son environnement. Par conséquence, pour permettre l'adaptation, le développeur peut définir le comportement de son application lorsqu'elle perd un ou plusieurs services critiques.

4.5.2.3 Exemples de détection et de résolution de conflits

Conflit direct au niveau d'un service contextuel. Ce type de conflit se produit lorsque deux applications agissent sur le même service abstrait ou sur le même équipement pendant le même intervalle de temps. Il est présenté par la figure 4.11 où l'application *App2* possède une priorité P2 plus élevée que P1 de l'application *App1* (P1<P2).

- L'application *App1* pose un verrou sur le service S3 initialement libre. La plateforme propage le verrou à tous les services feuilles représentant les dispositifs requis d'une manière explicite ou implicite par S3. En conséquence, les services S1, S2 et S3

passent à l'état verrouillé et l'application *App1* devient alors le propriétaire de ces services.

- L'application *App2* fait une demande de verrou pour utiliser le service abstrait S3. Un conflit est alors identifié entre les applications *App1* et *App2* au niveau de S3. Pour résoudre ce conflit, le verrou est attribué à *App2*, plus prioritaire, qui devient le nouveau propriétaire de S3. Les propriétaires des services S1 et S2 sont aussi mis à jour. *App2* est alors le nouveau propriétaire de ces deux services. La plateforme rend invisible S3 pour l'application *App1* qui peut s'adapter dynamiquement.

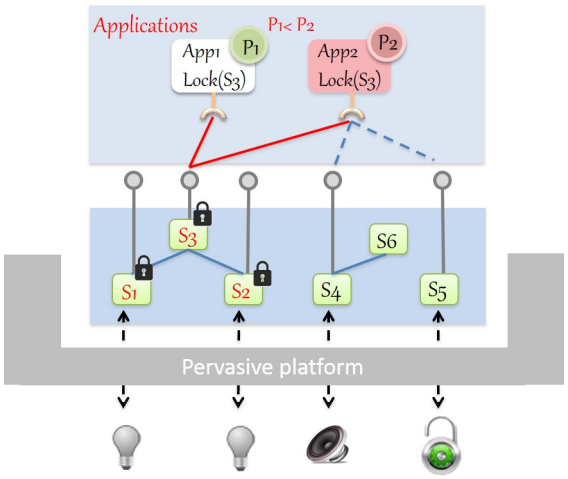

Figure 4.11: Détection et résolution d'un conflit direct.

Conflits indirects au niveau d'un service contextuel. Il existe deux types de conflits indirects. Le premier type de conflit indirect peut se produire lorsqu'une application utilise un service abstrait qui agit sur un service correspondant à un dispositif consommé par une autre application. La figure 4.12 illustre ce conflit entre les applications *App1* et *App2* où *App1* a le niveau de priorité P1 le plus faible (P1<P2).

- L'application *App1* pose un verrou sur le service S3. Le verrou est propagé à tous les services représentant des équipements utilisés d'une façon indirecte par l'application *App1*. Cette application devient alors le propriétaire des services S1, S2 et S3 qui passent à l'état verrouillé.

- Lorsque l'application *App2* envoie une requête de verrouillage au service S2 déjà verrouillé par *App1*, un conflit est identifié. Le verrou est alors accordé à *App2* qui devient le nouveau propriétaire de S2. *App1* moins prioritaire ne voit plus ce service. Concrètement, la plateforme rend invisible le service abstrait S3 utilisant le service

S2 pour l'application *App1* jusqu'à ce que le verrou initialement posé sur S2 soit relâché.

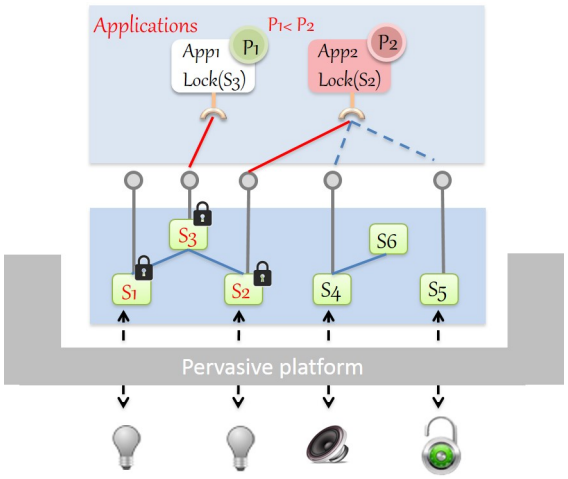

Figure 4.12: Détection et résolution d'un conflit indirect au niveau d'un équipement.

Le deuxième type de conflit indirect peut se produire lorsque deux applications utilisent deux services abstraits différents agissant sur le même service représentant un dispositif. Il s'agit d'une généralisation du cas précédent. La figure 4.13 présente ce conflit entre les applications *App1* et *App2*. L'application *App2* est plus prioritaire que l'application *App1* (P1<P2).

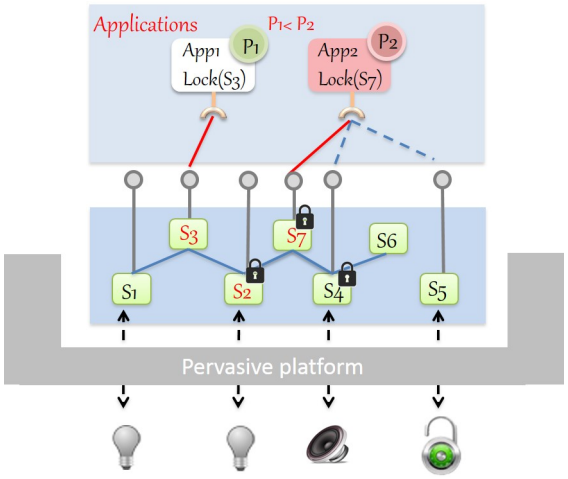

Figure 4.13: Détection et résolution d'un conflit indirect au niveau d'un service abstrait.

- Les services S1, S2 et S3 sont initialement libres. L'application *App1* envoie une demande de verrou pour utiliser le service S3. Cette demande est propagée à tous les services feuilles représentant les dispositifs utilisés d'une façon indirecte par l'application *App1*
- Lorsque l'application *App2* envoie une demande de verrouillage au service S7, la plateforme propage cette demande aux services S4 et S2. Un conflit est identifié au niveau de S2. Pour résoudre ce conflit, le verrou est attribué à l'application *App2* plus prioritaire. *App1* ne voit plus le service S3 jusqu'à ce que le verrou posé sur S2 soit relâché.

4.6 Conclusion

Ce chapitre nous a permis de décrire en détail notre contribution. Pour cela, nous avons commencé par présenter la notion de contexte. Nous avons vu que le contexte repose sur un modèle causal qui représente l'environnement physique à un instant donné sur la plateforme. Nous avons mis en lumière particulièrement le contexte orienté services pour gérer les conflits entre les applications ubiquitaires de la maison. Nous avons définis deux types de conflits : (1) les conflits directs qui se produisent lorsque deux applications invoquent le même service de manière incompatible ; (2) les conflits indirects qui surviennent lorsque deux applications invoquent différents services entraînant des actions incompatibles sur un service critique partagé. Nous avons étendu le modèle du contexte en associant des contrôleurs aux services critiques partagés afin de gérer leurs accès. Nous avons aussi été inspirés des systèmes d'exploitation en nous appuyant sur les mécanismes de verrouillage pour synchroniser l'accès à ces services. Dans notre proposition, l'application doit acquérir un verrou avant d'invoquer un service critique du contexte. Par conséquent, le service au centre d'un conflit est toujours attribué à l'application possédant le verrou. Dans ce chapitre, nous avons alors présenté le modèle de développement d'applications en mettant l'accent sur les API proposées pour acquérir et libérer un verrou.

Dans notre proposition, un niveau de priorité est attribué à chaque application. Ces priorités vont servir pour la gestion des conflits à l'exécution. En effet, lorsqu'une application pose un verrou sur un service, la plateforme rend invisible ce service aux applications moins prioritaires jusqu'à ce que ce dernier soit libre de nouveau. C'est une mesure préventive qui permet d'éviter les conflits qui peuvent se présenter entre l'application détenant le verrou et les applications moins prioritaires. Pour les applications plus prioritaires, nous avons proposé une mesure curative. Pour ce faire, nous avons adopté une approche optimiste où les conflits sont identifiés et résolus à l'exécution avant que leurs effets indésirables se présentent dans l'environnement.

Pour gérer les conflits, nous avons introduit la notion de visibilité des services. Dans notre proposition, en cas de conflit, le service requis devient invisible pour l'application avec la priorité la plus faible. Pour assurer la continuité de fonctionnement de cette application, nous avons considéré la capacité d'adaptation de la plateforme et des applications. L'adaptation est un aspect clé de notre proposition. Le programmeur est guidé pendant la phase de développement pour implanter la stratégie d'adaptation de son application.

Le chapitre suivant présente l'implantation des différents éléments de notre proposition. Les mécanismes proposés et leurs implantations y sont ensuite évalués et validés.

Chapter 5

Implantation et Validation

Sommaire

Ce cinquième chapitre de ce manuscrit est divisé en deux parties principales : la première partie sera consacrée à présenter en détail l'implantation de notre proposition qui se base sur la technologie Apache Felix iPOJO. Nous décrirons alors les différentes entités logicielles développées et leurs interactions. La deuxième partie de ce chapitre servira à la validation de notre proposition. Pour cela, nous présenterons l'environnement de validation iCasa dans un premier volet. Puis, dans un second volet, nous décrirons les applications utilisées pour valider notre solution et nous étudierons son impact sur leur architecture et leur implantation. Ensuite, dans un dernier volet, nous évaluerons notre approche en termes de nombre de lignes de code et de temps d'exécution.

5.1 Implantation de la proposition

Dans le chapitre précèdent, nous avons décrit en détail notre proposition de gestion des conflits qui peuvent se produire entre les applications ubiquitaires de la maison. Durant cette description, nous avons mis l'accent sur deux types des conflits (conflits direct et indirect). Nous avons aussi présenté leur formalisation dans un contexte orienté service. Ces conflits sont traités à l'exécution avec des mécanismes de verrouillage, de changement de visibilité de services et de priorisation ainsi que d'adaptation des applications.

L'approche proposée pour la gestion des conflits est une approche optimiste à trois phases : (1) la prévention pour minimiser les situations de conflits à l'exécution ; (2) leur détection et (3) leur résolution lorsque la prévention est inabordable. Toutefois, il est nécessaire de noter que, dans notre proposition, les conflits sont identifiés et résolus à l'exécution au plus tard possible, avant que leurs effets indésirables apparaissent dans l'environnement.

Dans cette section, nous allons décrire la mise en œuvre de notre proposition. Nous commençons par présenter notre choix technologique (Apache Felix iPOJO) sur lequel nous nous sommes basés pour réaliser ce projet dans le cadre de cette thèse. Ensuite, nous présentons l'architecture globale du système proposé pour la gestion des conflits, les différentes entités qui le construisent ainsi que leurs interactions. Pour finir, nous décrivons en détail ces entités et leur apport par rapport à notre contribution.

5.1.1 Apache Felix iPOJO

Comme nous l'avons déjà présenté dans le chapitre 3, l'un des principaux objectifs d'Apache Felix iPOJO [Esc08] est de rendre le développement d'applications dynamiques le plus simple possible. En effet, le code d'un composant devrait se concentrer uniquement sur la logique métier, et non pas sur les mécanismes liés au dynamisme ou d'autres exigences non fonctionnelles. iPOJO repose sur le principe de conteneur ouvert et fournit un conteneur de composants extensible qui traite tous les problèmes concernant le dynamisme. En particulier, il gère toutes les interactions orientées services : la publication de service, l'instanciation de service, la sélection de service et la découverte de service. Le conteneur peut être étendu afin de supporter d'autres aspects non fonctionnels tels que la configuration, la persistance et la sécurité. L'objectif du conteneur consiste donc à envelopper un objet Java simple et à fournir des propriétés non fonctionnelles.

Les conteneurs d'iPOJO ne sont pas monolithiques mais ils sont constitués d'un ensemble d'éléments appelés *handlers*. Chaque *handler* gère un ou plusieurs aspects non fonctionnels. iPOJO propose un ensemble de *handlers* prédéfinis (par exemple, *Required Service Handler* et *Provided Service Handler*, voir la figure 5.1) qui implantent toutes les capacités du modèle. En plus des *handlers* proposés, d'autres *handlers* peuvent être ajoutés aux conteneurs iPOJO dans le but de répondre à d'autres exigences non fonctionnelles.

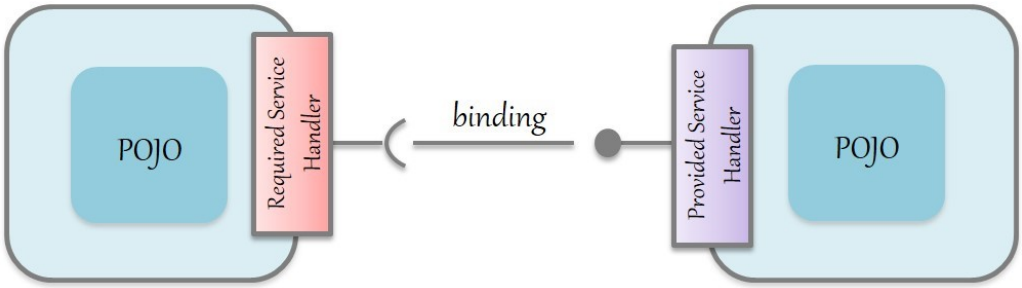

Figure 5.1: Le conteneur extensible d'iPOJO.

iPOJO fournit aussi des mécanismes d'interception qui peuvent changer l'ensemble des services vus par une dépendance de service. Plus précisément, il propose plusieurs types d'intercepteurs :

- le premier type d'intercepteur est le ***Service Tracking Interceptor***. C'est un intercepteur du registre de services qui décide si un service peut être vu par une dépendance de service ou non. Il peut alors accepter ou cacher un service. De plus, cet intercepteur peut ajouter ou supprimer des propriétés de service.
- le deuxième type d'intercepteur est le ***Service Ranking Interceptor***. Il permet le tri des services acceptés par l'intercepteur du registre de services. L'ordonnancement de ces services favorise la consommation de certains services par rapport à d'autres.
- le troisième type est le ***Service Binding Interceptor***. C'est un intercepteur de liaison de services qui permet de modifier les objets de service injectés.

Avant d'être injecté dans le code, un service passe par un ensemble de couches gérées par les différents mécanismes d'interception proposés par iPOJO. Chaque intercepteur peut modifier une de ces couches et refléter ses modifications sur les couches suivantes. Chaque couche présente alors un ensemble de services manipulés par un type d'intercepteur spécifique. La figure 5.2 présente les différents ensembles de services. L'ensemble initial des services est nommé ***Base Set***. Il intègre tous les services qui sont disponibles dans le

registre de services (il contient tous les services publiés sur la plateforme) et qui fournissent l'interface requise. Le deuxième ensemble est le ***Matching Set*** qui présente tous les services acceptés par l'intercepteur du registre de services. L'ensemble des services du *Matching Set* triés par le *Service Ranking Interceptor* constitue le ***Selected Set***. Le dernier ensemble est le ***Bound Set***. C'est l'ensemble des objets qui correspondent aux services du *Selected Set*. Ces objets peuvent être modifiés par le *Service Binding Interceptor* dans le but de changer le comportement d'un service requis.

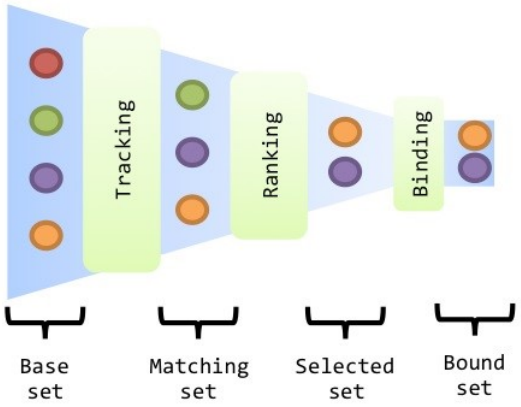

Figure 5.2: Les ensembles de services dans une dépendance[1].

Nous avons choisi Apache Felix iPOJO car c'est une technologie qui satisfait les contraintes imposées par notre contribution. En effet, c'est un modèle à composant orienté service extensible permettant ainsi d'ajouter les entités indispensables pour la gestion des conflits. De plus, elle propose des mécanismes d'interception (précisément l'intercepteur du registre de services) qui permet de gérer la visibilité d'une dépendance de service exigée dans notre proposition.

5.1.2 Architecture générale

Dans cette sous-section, nous décrivons l'architecture globale de notre système de gestion de conflits. La figure 5.3 illustre cette architecture.

[1] http://felix.apache.org/

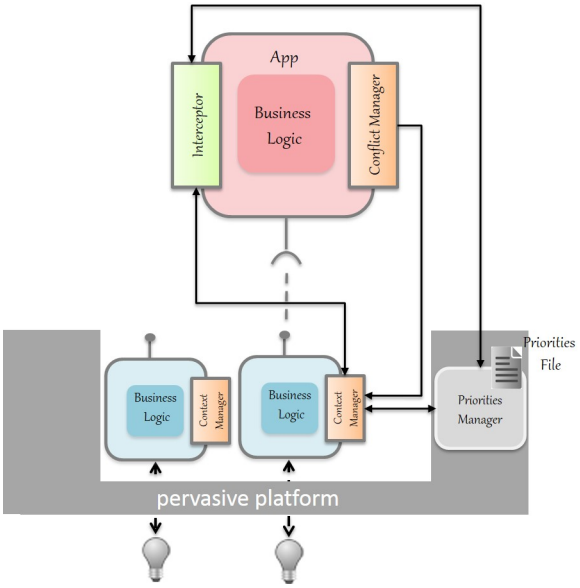

Figure 5.3: Architecture générale de la solution.

Nous décrivons brièvement les différentes entités qui constituent ce système :

- ***Context Manager :*** cette entité implémente le contrôleur défini dans le chapitre proposition de ce manuscrit et gère la machine d'états associée. Elle est liée à tous les services critiques présents dans le contexte fourni par la plateforme.

- ***Conflict Manager :*** cette entité est liée aux applications. Chaque application installée sur la plateforme d'exécution doit interagir avec son gestionnaire de conflits afin de poser ou relâcher un verrou sur un service critique.

- ***Interceptor :*** cette entité est associée aux applications. Elle est responsable de gérer la visibilité des services sélectionnés par une dépendance. Elle décide si une application peut voir ou non un service requis.

- ***Priorities Manager :*** cette entité est unique et intégrée dans la plateforme ubiquitaire. Son rôle principal est de comparer les priorités des applications à partir d'un fichier décrivant leurs niveaux de priorités et de sélectionner l'application avec la priorité la plus élevée.

Dans la suite de cette partie, nous décrivons deux diagrammes de séquence afin de faire apparaître les collaborations entre les différentes entités de notre système de gestion des conflits selon un point de vue temporel et de mettre l'accent sur la chronologie des échanges.

La figure 5.4 décrit le diagramme de séquence d'une demande d'un verrou sur un service simple. Lorsqu'une application demande un verrou sur un service (*lock (serviceID)*), le *Conflict Manager* transmet cette demande et l'identifiant de cette application au *Context Manager* associé à ce service. Deux situations peuvent se présenter : si le service est libre, le *Context Manager* met à jour l'état de service (*updateState()*) qui passe à l'état verrouillé ainsi que son propriétaire (*updateOwner()*). Dans le cas contraire (le service est verrouillé), si l'application demandant le verrou n'est pas le propriétaire actuel du service, le *Context Manager* sollicite le *Priorities Manager* afin de comparer les niveaux de priorités des deux applications (*computeNewOwner (appID, ownerID)*). Par conséquent, si l'application détenant le verrou est moins prioritaire, le *Context Manager* met à jour le propriétaire du service en indiquant l'identifiant de la nouvelle application plus prioritaire.

La figure 5.5 présente le diagramme de séquence relatif à la gestion de la visibilité des services. Ce mécanisme de gestion de visibilité est déclenché lorsqu'un service requis est verrouillé ou déverrouillé par une autre application. Dans le diagramme, l'entité *Interceptor* est liée à chaque application dans la plateforme. Son rôle principal est de gérer la visibilité des services requis par l'application. Pour chaque service, ce dernier sollicite le *Context Manager* afin de récupérer son état (*getState()*). Si le service est dans l'état libre, il est alors accepté par l'intercepteur qui retourne la référence de ce service à l'application. Sinon, il s'adresse, dans un premier lieu, au *Context Manager* pour chercher l'identité de l'application qui possède actuellement le verrou (*getServiceOwner()*). Puis, dans un second temps, il demande au *Priorities Manager* d'identifier l'application la plus prioritaire. Si l'application en question a un niveau de priorité supérieur à celui du propriétaire du service (*AppIDEqualtoNewOwnerID*), alors le service est accepté par l'intercepteur (*return ref*). Sinon, il doit être caché (*return null*). L'application ne voyant plus ce service, elle invoque la méthode *unbind()* permettant l'adaptation à ce changement.

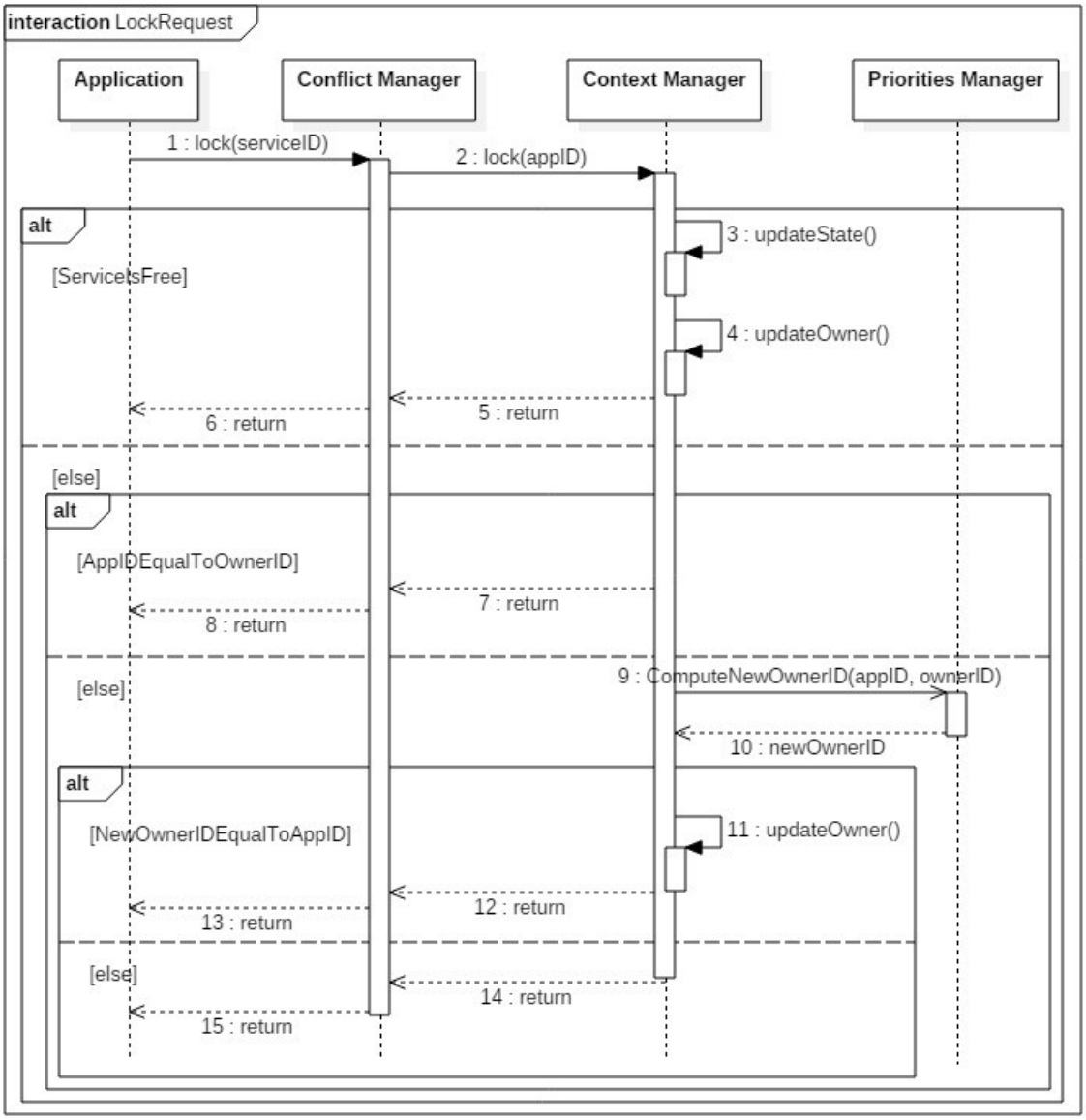

Figure 5.4: Diagramme de séquence de demande d'un verrou sur un service simple.

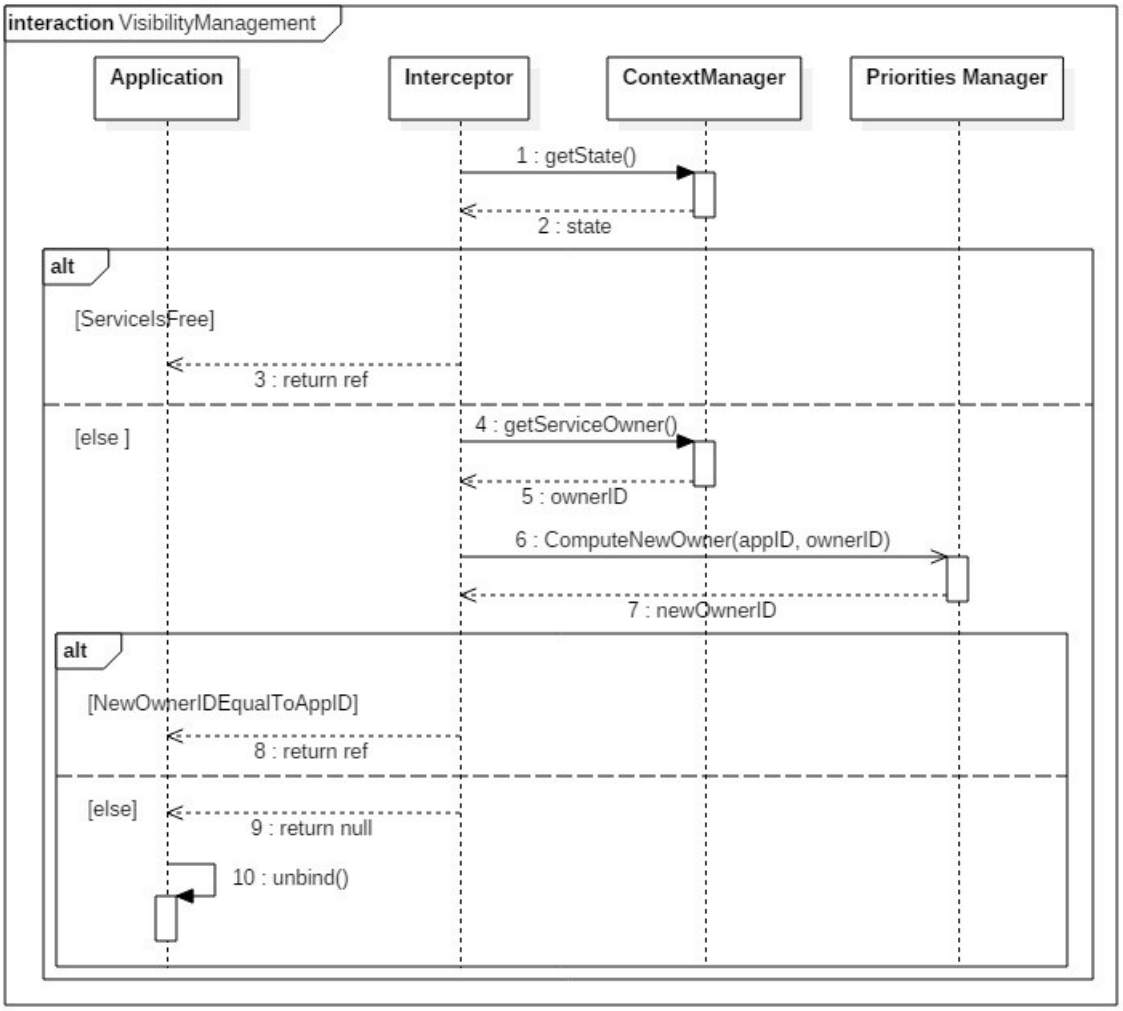

Figure 5.5: Diagramme de séquence de la gestion de la visibilité d'un service simple.

Dans la suite de ce chapitre nous décrivons en détail les différentes entités constituant notre système de gestion des conflits.

5.1.3 Context Manager et Conflict Manager

Notre solution de gestion des conflits a été développée et intégrée dans la plateforme d'exécution en tant que composants iPOJO. En effet, dans notre mise en œuvre, nous avons créé deux types différents de conteneurs, comme les montre la figure 5.6 : (1) le premier type du conteneur est associé aux composants du contexte et (2) le deuxième type du conteneur est lié aux applications. Plus précisément, un composant du contexte est exécuté dans un conteneur intégrant un *handler* spécifique appelé *Context Manager Handler*. Ce *handler* implante le contrôleur décrit en détail dans le chapitre proposition de ce manuscrit et gère la machine d'états associée.

Les composants d'application sont exécutés dans un conteneur comprenant un *handler* de gestion des conflits ou, en anglais, *Conflict Manger Handler*. Le but de ce dernier *handler* est de fournir les API permettant aux composants de poser ou de relâcher un verrou sur un service contextuel (avec des fonctions de verrouillage et de déverrouillage).

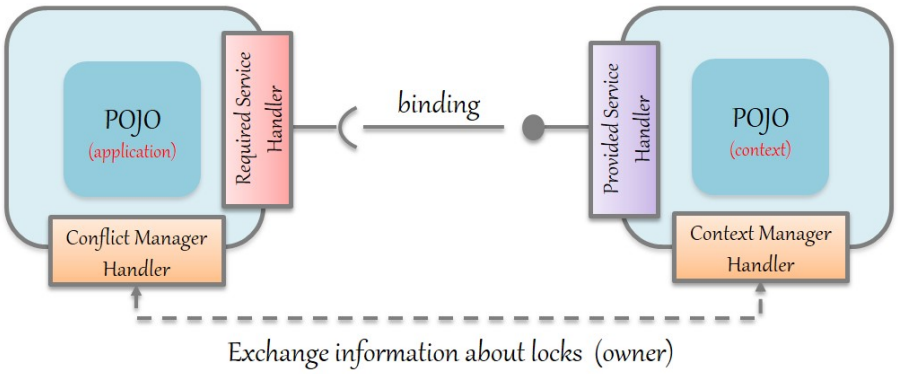

Figure 5.6: Les *handlers* de gestion des conflits.

Les deux *handlers* (*Context Manager Handler et Conflict Manager Handler*) proposés fournissent des annotations Java afin de faciliter leur utilisation. Nous distinguons alors deux types d'annotations (*@ContextManager* et *@ConflictManager*) intégrées dans le langage de programmation proposé. Ces annotations sont utilisées respectivement pour l'implantation des services contextuels et les applications ubiquitaires.

L'annotation*@ContextManager* permet de déclarer un service critique (abstrait ou simple) du contexte et de lui associer un *Context Manager Handler*. A l'exécution, ce *handler* gère l'état du service (libre ou verrouillé), identifie les conflits qui peuvent avoir lieu et implante la stratégie de leur résolution. L'extrait du code 5.1 présente l'implantation d'un service critique. Nous supposons que ce service présente deux méthodes conflictuelles, par exemple *setStatusOn()* et *setStatusOff()*. L'annotation *@ContextManager* est alors utilisée pour déclarer que ce service est critique. A l'exécution, une application ubiquitaire requérant ce service ne peut pas l'utiliser sans poser un verrou. Par conséquent, les méthodes nécessaires pour acquérir et relâcher un verrou doivent être nécessairement appelées dans le code de l'application.

Listing 5.1: Code Java relatif à la déclaration d'un gestionnaire de contexte associé à un service du contexte.

```
1  // Declaration of critical service.
2  @ContextManager
3  public class CriticalServiceImpl implements CriticalService {
4
5     @Override
6     public void setStatusOn() {
7        // Method code
8     }
9
10    @Override
11    public void setStatusOff() {
12       // Method code
13    }
14
15    // Critical Service Code.
16 }
```

Les composants de l'application utilisent l'annotation *@ConflictManager*. Cette annotation est obligatoire pour chaque application requérant des services critiques. Son objectif est de déclarer un gestionnaire de conflits qui fournit les méthodes nécessaires pour détenir et libérer un verrou. Ces méthodes sont respectivement *lockService(serviceID)* et *unlockService(serviceID)*. Elles prennent en argument l'identifiant du service sur lequel l'application souhaite poser le verrou.

L'utilisation de l'annotation *@ConflictManager* est illustrée par l'extrait du code 5.2. Dans cet exemple, le composant de l'application requiert des services critiques dont

l'implantation utilise l'annotation de *@ContextManager* (voir l'extrait du code 5.1). Le consommateur des services critiques doit, tout d'abord, utiliser l'annotation *@Conflict-Manager* pour déclarer un gestionnaire de conflits. Puis, à chaque fois un service critique apparaît, il récupère son identifiant et le stocke. Ensuite, il appelle les méthodes de verrouillage et de déverrouillage selon ses besoins fonctionnels. Plus précisément, avant la première invocation des méthodes conflictuelles d'un service critique, il utilise la méthode *lockService(serviceID)* pour poser un verrou (l'argument de cette méthode correspond à l'identifiant du service critique). Enfin, il fait appel à la méthode *unlockService(serviceID)* pour relâcher le verrou.

Listing 5.2: Code Java relatif à la déclaration d'un gestionnaire de conflits.

```
1  public class ApplicationImpl {
2
3     // Declaration of the Conflict Manager
4     @ConflictManager
5     private ConflictManagerService conflictManager;
6
7     @Requires(id="criticals",optional = true,specification =
          CriticalService.class)
8     private List<CriticalService> criticalServices;
9
10    private Map<CriticalService,String> services = new HashMap<>();
11
12    @Bind(id="criticals")
13    public void bindCriticalService(CriticalService criticalService, Map
          properties) {
14
15       // storage of the id of the critical service
16       services.put(criticalService,(properties.get("service.id")).toString());
17    }
18
19    public void manageCriticalService () {
20
21       if (state == true) {
22          for(CriticalService criticalService : criticalServices) {
23             // locking critical service before invoking a conflicting method
24             conflictManager.lockService(services.get(criticalService));
25             criticalService.setStatusOn();
26          }
27       }
28
```

```
29      else {
30         for(CriticalService criticalService : criticalServices) {
31            // unlocking critical service
32            conflictManager.unlockService(services.get(criticalService));
33
34         }
35      }
36   }
37
38 //Application code
39
40 }
```

L'objectif principal du *Context Manager Handler* est de prendre une décision concernant l'application détenant le verrou. En effet, lorsqu'il reçoit une demande de verrou, il suit une succession d'étapes afin de prendre sa décision. Le processus de prise de décision pour une demande de verrou pour utiliser un service critique représentant un équipement (un service simple) est illustré par l'algorithme 1. A l'exécution, le composant implantant le contrôleur garantit que seule l'application détenant le verrou peut invoquer le service pour exécuter un ensemble d'actions. Dès que ce dernier reçoit une demande de verrou, il vérifie l'état du service. Deux cas peuvent se présenter : libre ou verrouillé. Dans le premier cas, le service bascule à l'état verrouillé et la propriété qui indique son propriétaire retient l'identité de l'application demandant le verrou. Dans le deuxième cas, si l'application demandant le verrou a une priorité plus élevée, le propriétaire du service est mis à jour.

```
begin
    if le service est libre then
        Passer le service à l'état verrouillé;
        Mettre à jour le propriétaire du service;
    else
        if l'application demandant le verrou n'est pas l'application détenant le
         verrou then
            if l'application demandant le verrou est plus prioritaire que l'application
             détenant le verrou then
                Mettre à jour le propriétaire du service;
            end
        end
    end
end
```

Algorithm 1: Verrouiller un service critique simple

Comme nous l'avons déjà mentionné dans le chapitre précédent, la demande de verrouillage d'un service abstrait nécessite la propagation de verrou jusqu'aux services feuilles représentant les dispositifs. Concrètement, une application peut détenir un verrou pour utiliser un service abstrait seulement si elle réussit à verrouiller tous les services critiques simples qu'elle utilise d'une manière explicite ou implicite. Le processus de décision pour une demande de verrouillage d'un service abstrait est présenté par l'algorithme 2. Tout d'abord, l'état du service doit être vérifié. Deux situations peuvent se présenter : libre ou verrouillé. (1) Si le service est libre, la demande de verrou est propagée jusqu'à tous les services simples. Autrement dit, le processus de verrouillage est répété pour chaque dépendance critique directe ou indirecte de ce service. Le service est alors verrouillé si toutes ses dépendances critiques sont verrouillées par l'application demandant le verrou. Toutefois, la demande de verrou échoue si au moins un service requis d'une façon indirecte est verrouillé par une autre application plus prioritaire. (2) Si le service est verrouillé, deux cas peuvent être distingués : le premier cas correspond au cas où l'application demandant le verrou est plus prioritaire que l'application détenant le verrou. Dans ce cas, le verrou est attribué à la première application et les propriétaires des services sont mis à jour. Dans le cas contraire, la demande de verrou échoue.

```
begin
    Initilaliser serviceOwned à true;
    if le service critique est simple then
        Verrouiller le service critique simple;
    else
        if le service est libre ou l'application demandant le verrou est plus
         prioritaire que l'application détenant le verrou then
            for chaque service parmi les services requis do
                if le service requis est critique then
                    Verrouiller le service critique requis;
                    if l'application demandant le verrou est le propriétaire du service
                     then
                        serviceOwned :=serviceOwned && true;
                    else
                        serviceOwned :=serviceOwned && false;
                    end
                end
            end
            if serviceOwned == true then
                Mettre à jour le propriétaire du service;
                if le service est libre then
                    Passer le service à l'état verrouillé;
                end
            else
                Déverrouiller le service que l'application souhaite le verrouiller
            end
        end
    end
end
```

Algorithm 2: Verrouiller un service critique

Le processus de libération de verrou est illustré par l'algorithme 3. En effet, si l'application souhaitant libérer le verrou est celle détenant le verrou, le service passe à l'état libre et le champ qui indique son propriétaire est mis à jour. Sinon, la demande de déverrouillage échoue. Dans la première situation, si le service est abstrait, la demande de libération de verrou est propagée à tous les services critiques utilisés d'une manière explicite ou implicite. Plus précisément, toutes ses dépendances critiques directes et indirectes basculent à l'état libre et leurs propriétaires sont mis à jour.

```
begin
    if le service est verrouillé then
        if l'application souhaitant libérer le verrou est l'application détenant le
          verrou then
            if le service critique est abstrait then
                for chaque service critique parmi les services requis do Déverrouiller
                  le service critique requis;
            end
            Passer le service à l'état libre;
            Mettre à jour le propriétaire du service;
        end
    end
end
```

Algorithm 3: Déverrouiller un service critique

5.1.4 Intercepteur

Dans notre réalisation, nous avons implanté un intercepteur du registre de services. Ce dernier décide si un service peut être vu par une dépendance du service ou non. Comme le montre la figure 5.7, nous avons ajouté un *Interceptor* qui implante le *Service Tracking Interceptor* (voir section 5.1.1 de ce chapitre) qui influence la visibilité des services requis. L'objectif de cet intercepteur est de cacher dynamiquement les services verrouillés aux applications qui possèdent un niveau de priorité inférieur à celui de l'application détenant le verrou.

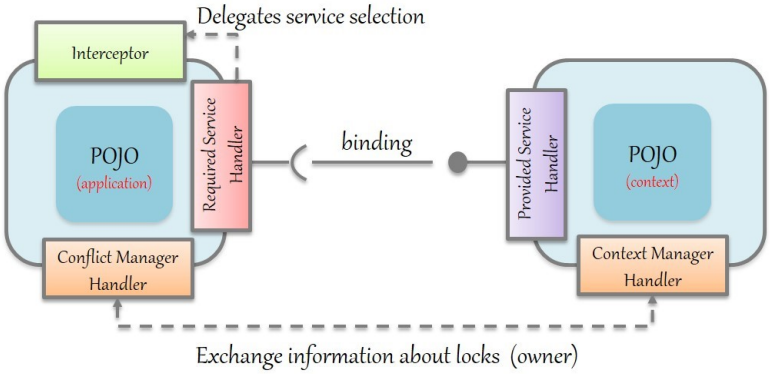

Figure 5.7: Intercepteur de service.

Au début, tous les services du registre fournissant l'interface requise sont pris en compte afin d'établir l'ensemble de base des services (*Base Set*). Ensuite, pour tous les services

sélectionnés, l'intercepteur décide d'accepter le service ou non afin de construire un ensemble final de services (*Matching Set*). Plus précisément, l'intercepteur accepte seulement les services libres ou les services verrouillés dont le propriétaire possède une priorité inférieure à celle de l'application en question. Par conséquent, seuls les services qui peuvent être effectivement invoqués par l'application sont conservés dans l'ensemble final.

5.1.5 Gestionnaire de priorités

Le gestionnaire de priorités (*Priorities Manager*) est intégré dans la plateforme d'exécution. Sa fonction principale est de sélectionner l'application ayant la priorité la plus élevée entre deux applications. Ce composant fournit un service qui permet d'attribuer des niveaux de priorité aux applications. Ce service se base essentiellement sur un fichier XML descriptif de l'ensemble des applications ubiquitaires installées sur la plateforme, leurs catégories et leurs niveaux de priorité. Les priorités d'applications peuvent changer en ajoutant de nouvelles applications à la plateforme. Nous pouvons alors changer ces priorités en changeant le fichier XML. Ce fichier est une description détaillée du schéma qui fixe les priorités des applications consommant des services critiques.

En parcourant le fichier XML, ce composant sauvegarde toutes les informations concernant les applications. Chaque application dans la plateforme est décrite par son identifiant, sa catégorie et son niveau de priorité. Une fois sollicité, le *Priorities Manager* compare les niveaux de priorité des deux applications et identifie l'application ayant la plus forte priorité.

5.2 Validation

L'objectif de la réalisation de notre proposition est de permettre la gestion des conflits qui peuvent se produire entre les applications ubiquitaires de la maison selon une approche orientée services. Pour cela, nous avons utilisé les concepts et les mécanismes que nous avons déjà présentés dans le chapitre 4 pour :

- étendre le modèle de développement par les mécanismes de verrouillage ;
- localiser les conflits au niveau d'un contexte orienté services ;
- gérer la visibilité du service qui peut être au centre d'un conflit ;
- adapter les applications en fonction de leur contexte d'exécution (précisément, selon la disponibilité des services).

Afin de valider notre proposition, il nous a alors été nécessaire de nous appuyer sur une plateforme de développement et d'exécution d'applications ubiquitaires. Cette plateforme devait présenter un ensemble de caractéristiques pour pouvoir implanter et valider notre solution :

- fournir un contexte orienté services ;
- supporter la dynamicité et présenter des capacités d'adaptation ; c'est-à-dire la plateforme doit permettre la reconfiguration dynamique des composants ;
- proposer un modèle de développement d'applications ubiquitaires.

Pour répondre à ces contraintes, nous avons choisi la plateforme ubiquitaire iCasa qui a été développée dans le cadre d'un projet collaboratif (le projet Medical) entre Orange Labs et l'équipe Adèle du Laboratoire d'Informatique de Grenoble. L'objectif d'iCasa est de faciliter le développement, la simulation et l'exécution des applications ubiquitaires dédiées à la maison. iCasa fournit un modèle de développement d'applications basé sur un contexte orienté services. Elle supporte aussi la reconfiguration dynamique des applications afin de les adapter aux changements de leur environnement. De plus, iCasa propose un ensemble de services techniques pour la découverte et la communication avec les dispositifs ainsi que la gestion du contexte. Cette plateforme satisfait alors toutes les exigences que nous avons spécifiées précédemment.

Notre solution de gestion des conflits a été développée et intégrée dans la plateforme ubiquitaire iCasa sous forme des composants iPOJO. Le *Context Manager* et le *Conflict Manager* sont implantés en tant que *handlers* associés respectivement aux services critiques du contexte et les applications consommant ces services. De plus, un *Interceptor* est associé à chaque application permettant ainsi de gérer la visibilité des services critiques requis.

5.2.1 Présentation de l'environnement de validation : iCasa

Afin de montrer la validité de notre contribution, nous nous appuyons sur iCasa [Lal+15; IMA], un environnement d'exécution et de simulation domotique qui répond parfaitement à toutes les contraintes nécessaires pour valider notre solution. En effet, iCasa supporte le développement et l'exécution d'applications fondées sur des capteurs et des effecteurs. Pour ce faire, elle propose :

- un modèle de développement d'applications dynamiques à base de composants orientés service ;
- un contexte modélisé sous la forme d'un graphe de composants fournissant et consommant des services ;
- des services techniques qui permettent la découverte des équipements hétérogènes et leur réification sous forme de services.

En outre, iCasa fournit un environnement de simulation qui permet aux développeurs de tester leurs applications dédiées au contexte domotique. Pour ce faire, iCasa propose :

- un ensemble de dispositifs simulés (lampe, alarme sonore, détecteur de présence, détecteur de mouvements, etc.). Ces dispositifs peuvent changer les propriétés physiques de l'environnement simulé. Par exemple, un radiateur activé va augmenter la température de la pièce simulée.
- une interface graphique de simulation qui permet d'interagir directement avec l'environnement simulé. Cette interface fournit une carte représentant un plan d'une maison. Dans ce plan, des zones peuvent être créées, des personnes et des dispositifs simulés peuvent être ajoutés. Tous ces objets peuvent être déplacés sur la carte. De plus, les surfaces des zones peuvent être modifiées. L'interface graphique proposée permet alors de simuler le cycle de vie des objets indispensables pour la validation des applications (par exemple, l'ajout d'un dispositif, la liaison du dispositif à une application, la suppression de dispositif).

- des scripts : iCasa supporte l'écriture de scripts qui peuvent être utilisés pour créer des scénarios afin de contrôler l'environnement simulé. En effet, l'écriture de scripts est un bonne méthode pour tester les comportements des applications ubiquitaires dans des conditions reproductibles nécessaires pour leur validation.

Un exemple d'environnement simulé est illustré par la figure 5.8. Les carrés en différentes couleurs correspondent aux zones créées (salon, salle de bain, cuisine et chambre à coucher). Les différents dispositifs (détecteurs de présence, lampes) sont localisés dans ces zones. Deux habitants sont ajoutés. C'est dans cet environnement représenté par l'interface graphique d'iCasa que les applications ubiquitaires, proposées par Orange Labs, vont être testées. Ces applications vont servir à la validation de notre solution de gestion des conflits.

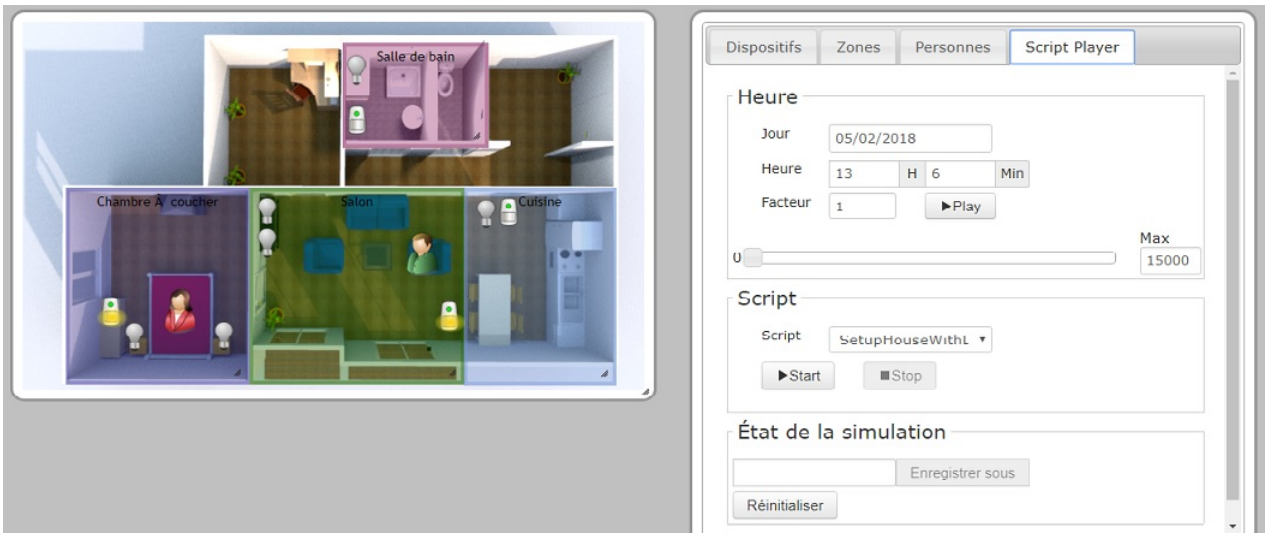

Figure 5.8: Interface Web de l'environnement simulé avec iCASA.

5.2.2 Les applications proposées

Afin de montrer la faisabilité de notre approche dans un scénario réaliste, nous nous sommes appuyés sur un ensemble d'applications proposées par Orange Labs et développées en collaboration avec l'équipe Adèle du Laboratoire d'Informatique de Grenoble. Dans cette section, nous décrivons ces applications.

5.2.2.1 Application de détection d'intrusion

Cette application de détection d'intrusion assure la sécurité de l'habitat en le protégeant contre les intrusions. Elle utilise les détecteurs de présence, les détecteurs d'ouverture de porte/fenêtre, les ampoules communicantes et les verrous de porte installés dans la maison. De plus, elle consomme un service d'alarme (ce service permet de déclencher une alarme et de configurer les équipements qui entrent en jeu lors du déclenchement de l'alarme (camera et/ou son)), un service de notification utilisateur (ce service permet d'envoyer un courrier électronique ou un message téléphonique aux occupants pour les notifier) et un service mode (ce service définit quatre modes (*night*, *holidays*, *away* et *home*) liés à la sécurité et il permet de changer le mode courant).

L'objectif de cette application est de protéger l'habitat et d'avertir l'utilisateur en cas de détection d'intrusion. Pour ce faire, cette application permet de verrouiller la porte d'entrée pendant que la maison est en mode *night*, *holidays* ou *away* pour assurer la sécurité de ses habitants et leurs biens. En outre, lors de la détection d'une intrusion, les lumières clignotent, les alarmes sonores sont déclenchées et une notification est envoyée à l'utilisateur qui peut ainsi consulter les vidéos enregistrées par les caméras installées dans sa maison.

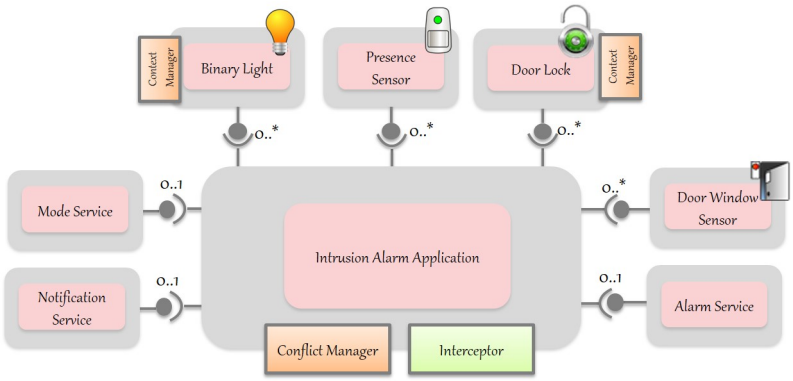

Figure 5.9: Architecture de l'implantation de l'application de détection d'intrusion.

La mise en œuvre de l'application de détection d'intrusion et des services utilisés est réalisée selon le modèle de développement proposé dans cette thèse. L'architecture de cette application est illustrée par la figure 5.9. La logique métier de cette application est prise en charge par un seul composant, qui fournit un service de type *IntrusionAlarm*, et consomme tous les services de type *PresenceSensor*, *DoorWindowSensor*, *BinaryLight*, *DoorLock*, *ModeService*,*AlarmService* et *NotificationService*. Ce composant, comme le montre

la figure 5.9, est exécuté dans un conteneur qui intègre un *handler* de gestion des conflits (*Conflict Manager Handler*) fournissant les API nécessaires pour poser et relâcher les verrous. Le conteneur comprend aussi un intercepteur qui est responsable de gérer la visibilité de l'ensemble des services critiques consommés par l'application.

L'extrait du code 5.3 présente l'implantation de l'application de détection d'intrusion. Lorsque la maison est en mode *night*, *holidays* ou *away*, le composant de cette application doit maintenir les verrous de porte verrouillés (*lock()*). Pour ce faire, il appelle les méthodes (*lockService(ServiceID)* et *unlockService(ServiceID)*) de verrouillage et de déverrouillage afin d'assurer un accès exclusif à tous les services de type *Door Lock* pendant que l'un de ces modes est actif. De plus, cette application détecte une intrusion à partir de l'ensemble des détecteurs de présence (*getSensedPresence()*) et des détecteurs d'ouverture de porte/fenêtre (*isOpened()*). En effet, lorsqu'une intrusion est détectée, le composant se charge de notifier l'utilisateur à travers le service *NotificationService* (*sendNotification()*) et active toutes les alarmes sonores en utilisant le service *AlarmService* (*fireAlarm()*). De plus, il pose un verrou (*lockService(serviceID)*) sur les services de type *BinaryLight*, puis les fait clignoter pour alerter les habitants.

Listing 5.3: Extrait de code de l'application de détection d'intrusion.

```
1  public class IntrusionAlarmApplication implements PeriodicRunnable {
2     @ConflictManager
3     private ConflictManagerService conflictManager;
4
5     public boolean detectIntrusion() {
6        // Detecting intrusion
7        for(DoorWindowSensor doorWindowSensor : doorWindowSensors) {
8           if(doorWindowSensor.isOpened()) {
9              return true;
10          }
11       }
12       for(PresenceSensor presenceSensor: presenceSensors) {
13          if(presenceSensor.getSensedPresence()){
14             return true;
15          }
16       }
17    }
18
19    @Override
20    public void run() {
21       String mode = modeService.getCurrentMode();
```

```
22      if (mode.equals(ModeUtils.NIGHT) || mode.equals(ModeUtils.HOLIDAYS)||
            mode.equals(ModeUtils.AWAY)) {
23         for(DoorLock doorLock : doorLocks) {
24            // Lock the DoorLock
25             conflictManager.lockService(services.get(doorLock));
26             doorLock.lock();
27         }
28      }
29      else if (mode.equals(ModeUtils.HOME)) {
30             for(DoorLock doorLock : doorLocks) {
31                 conflictManager.unlockService(services.get(doorLock));
32             }
33      }
34
35      if (detectIntrusion()) {
36          // Send a notification
37          notificationService.sendNotification("[ICASA] Intrusion Alarm", "
                Intrusion is detected.");
38          // Turn on alarms
39            alarmService.fireAlarm();
40         if (state == true) {
41             for(BinaryLight binaryLight : binaryLights) {
42               // Lock BinaryLights
43               conflictManager.lockService(services.get(binaryLight));
44               binaryLight.turnOn();
45            }
46             state = false;
47         }
48         else {
49             for(BinaryLight binaryLight : binaryLights) {
50                 binaryLight.turnOff();
51            }
52            state = true;
53         }
54      }
55      else {
56         state = true;
57         for(BinaryLight binaryLight : binaryLights) {
58            // Unlock BinaryLights
59             conflictManager.unlockService(services.get(binaryLight));
60         }
61      }
62   }
```

```
63
64 //Application Code
65 }
```

5.2.2.2 Light Follow Me

L'application *Light Follow Me* est une application d'éclairage proposée dans le cadre du projet collaboratif MEDICAL. Son objectif principal est de permettre aux lumières de suivre les déplacements des habitants d'une maison. En effet, cette application ubiquitaire détecte leur présence dans le but d'allumer et d'éteindre les lumières tout au long de leurs parcours. Pour cela, elle utilise des ampoules communicantes et des services de présence (un service de présence permet de détecter la présence d'une personne dans une zone).

L'architecture utilisée pour l'implantation de cette application est illustrée par la figure 5.10. Elle présente un seul composant qui implante la logique métier de l'application. Son conteneur intègre un *Conflict Manager* et un *Interceptor* indispensables pour la gestion des conflits qui peuvent se produire à l'exécution. Le composant de l'application consomme tous les services de type *PresenceService* et *BinaryLight*. Les services *BinaryLight* étant critiques, leurs conteneurs présentent alors un *Context Manager*.

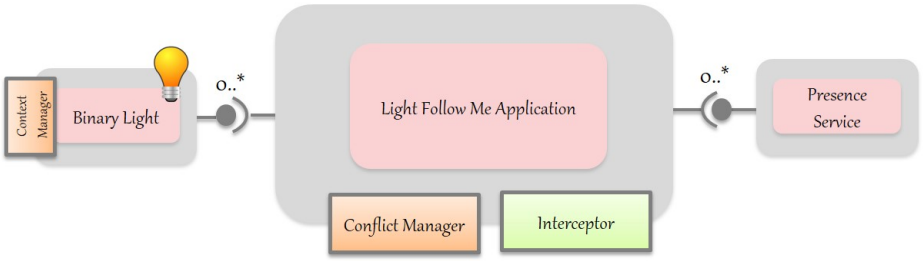

Figure 5.10: Architecture de l'application *Light Follow Me*.

L'extrait du code 5.4 présente un extrait de code de l'application *Light Follow Me*. L'application détermine la zone où une présence est détectée à partir de *PresenceService* (*sensePresenceIn()*). Puis, elle allume (*turnOn()*) les lumières installées dans cette zone. Dans le code de l'application, les méthodes (*lockService(ServiceID)* et *unlockService(ServiceID)*) de verrouillage et de déverrouillage ont été appelées pour assurer un accès exclusif aux services de type *Binary Light* pendant la présence de l'utilisateur dans une zone.

Listing 5.4: Extrait du code application de *Light Follow Me*.

```
public class LightFollowMeApplication {

   // Declaration of Conflict Manager
   @ConflictManager
   private ConflictManagerService conflictManager;

   private void manageLight(PresenceService presenceService) {
      String zoneName = presenceService.sensePresenceIn();
      Set<BinaryLight> lightInZone = getLightInZone(zoneName);
      if
          (presenceService.havePresenceInZone().equals(PresenceService.PresenceSensing.YES))
          {
         // lock all Binary Lights in the zone
         lightInZone.stream().forEach((light)
             ->conflictManager.lockService(services.get(light) ));
         lightInZone.stream().forEach((light) ->light.turnOn());
      }
      else {
         lightInZone.stream().forEach((light) ->light.turnOff());
         // unlock all Binary Lights in the zone
         lightInZone.stream().forEach((light)
             ->conflictManager.unlockService(services.get(light) ));
      }
   }
   //  Application code
}
```

5.2.2.3 Application de détection d'incendie

Cette application assure la sécurité de la maison contre les incendies. Elle interagit avec des détecteurs de fumée, des ampoules communicantes, des verrous de porte, un service d'alarme (permet de déclencher une alarme et de configurer les équipements qui entrent en jeu lors du déclenchement de l'alarme (camera et/ou son)) et un service de notification utilisateur (ce service permet d'envoyer un courrier électronique ou un message téléphonique aux occupants pour les notifier).

Pour détecter un incendie, l'application utilise les détecteurs de fumée installés dans une maison. En effet, un incendie est détecté si le taux de CO_2 détecté dans une pièce dépasse une valeur seuil spécifiée par un expert du domaine.

Dès qu'un incendie est détecté, les portes sont déverrouillées afin d'assurer l'évacuation de tous les occupants de la maison et les alarmes sonores sont activées. L'utilisateur notifié par un message téléphonique ou un courrier électronique, peut consulter à distance les vidéos enregistrées par la caméra.

Nous avons implanté cette application ubiquitaire en deux versions pour évaluer notre proposition de gestion des conflits en terme de nombre de lignes de code et de temps d'exécution. La première mise en œuvre de l'application de détection d'incendie, qui va servir de référence, est réalisée sans l'utilisation de notre solution de gestion de conflits. L'architecture de cette application est présentée par la figure 5.11. Cette version est constituée d'un seul composant métier dont le conteneur n'intégre pas un *handler* de gestion de conflits. A l'exécution, cette application utilise directement les services représentant les lampes sans acquisition des verrous.

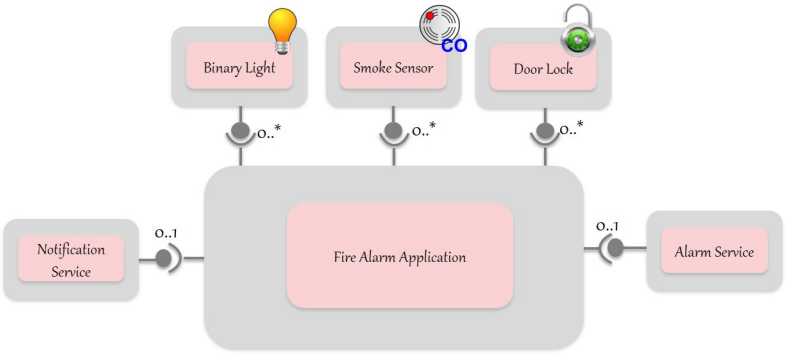

Figure 5.11: Architecture de l'implantation de référence de l'application de détection d'incendie.

La deuxième mise en œuvre de cette application est réalisée en respectant le modèle de développement proposé dans cette thèse. Plus particulièrement, cette version exige un accès exclusif aux verrous de porte ainsi qu'aux lampes connectées installées dans une maison. Par conséquent, les méthodes nécessaires pour poser et relâcher les verrous (*lockService (ServiceID)* et *unlockService(serviceID)*) sont appelées dans le code de l'application (voir l'extrait du code 5.5). L'architecture de cette version est illustrée par la figure 5.12. Contrairement à la première version, le conteneur du composant métier qui correspond à cette application intègre le *Handler* de gestion des conflits et l'*Interceptor* permettant la gestion de la visibilité des services critiques requis. Les méthodes de *callback* ont été implantées. Ces méthodes permettent l'adaptation de l'application lorsque les services requis deviennent indisponibles.

Listing 5.5: Extrait du code application de détection d'incendie.

```
public class FireAlarmApplication implements PeriodicRunnable {

   // Declaration of Conflict Manager
   @ConflictManager
   private ConflictManagerService conflictManager;

    @Override
    public void run() {
      if (checkCo2()) {
         m_logger.info("CO2 is too high ! ");
         // send a notification
         notificationService.sendNotification("[ICASA] CO2 Alert", " CO2 is too
             high in the house.");
         // Turn on alarms
         alarmService.fireAlarm();
         for(DoorLock doorLock : doorLocks) {
            // Lock DoorLock
            conflictManager.lockService(services.get(doorLock));
            doorLock.unlock();
         }
         if (state == true) {
            for(BinaryLight binaryLight : binaryLights) {
               conflictManager.lockService(services.get(binaryLight));
               binaryLight.turnOn();
            }
            state = false;
         }
         else {
            for(BinaryLight binaryLight : binaryLights) {
               binaryLight.turnOff();
            }
         state = true;
         }
       }
      else {
          state = true;
         for(BinaryLight binaryLight : binaryLights) {
            // Unlock binaryLight
            conflictManager.unlockService(services.get(binaryLight));
         }
         for(DoorLock doorLock : doorLocks) {
            // Unlock doorLock
```

```
42              conflictManager.unlockService(services.get(doorLock));
43          }
44      }
45  }
46  }
```

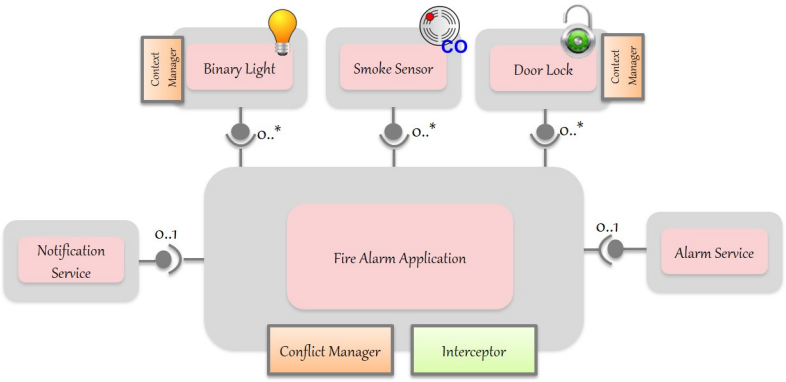

Figure 5.12: Architecture de l'implantation étendue de l'application de détection d'incendie.

Les deux versions de l'application de détection d'incendie seront comparées en termes de nombre de lignes de code et de temps d'exécution dans le but d'évaluer notre approche de gestion des conflits. Cette évaluation est présentée dans la suite de ce chapitre.

5.2.3 Situations de conflits

Les trois applications présentées dans les sous-sections précédentes ont été mises en œuvre et ont été testées avec le simulateur iCasa. Plusieurs situations de conflits ont été identifiées et résolues efficacement par l'approche proposée dans cette thèse.

Dans cette section, nous examinons deux exemples de conflit. La première situation de conflit se produit entre les applications de détection d'incendie et *Light Follow Me*. Ce conflit survient lorsqu'une présence est détectée, en cas d'incendie. Initialement, le service représentant la lampe est libre, c'est-à-dire qu'il n'est verrouillé par aucune application. Ainsi, lorsqu'une présence est détectée, l'application *Light Follow Me* peut acquérir le verrou. Toutefois, en cas d'incendie, l'application de détection d'incendie appelle la méthode nécessaire pour le verrouillage. Le contrôleur associé au service représentant la lampe interagit avec le gestionnaire des priorités afin de prendre une décision concernant le nouveau propriétaire du service. Cette décision est prise en fonction des priorités des applications. L'application de détection d'incendie devient le nouveau propriétaire de ce service grâce à son niveau de priorité plus élevé. De l'autre côté, le service devient invisible pour l'application *Light Follow Me* qui s'adapte automatiquement.

La deuxième situation de conflit se présente entre les applications de détection d'intrusion et de détection d'incendie. Plus précisément, un conflit se produit, en cas d'incendie, lorsque la maison est en mode *night*, *holidays* ou *away*. En effet, l'application de détection d'intrusion pose un verrou sur le service représentant le verrou de la porte. Le contrôleur associé vérifie que le service est libre et met à jour son propriétaire. Lorsqu'un incendie est détecté, l'application de détection d'incendie, étant plus prioritaire, préempte le verrou et le service devient indivisible pour l'application de détection d'intrusion. Cette dernière application s'adapte en invoquant la méthode de callback.

Nous avons exécuté plusieurs scénarios sur la plateforme ubiquitaire iCasa. En effet, iCasa propose un langage de script permettant de créer dynamiquement des dispositifs simulés, des zones dans le plan d'une maison et des utilisateurs. Ce script permet de définir des scénarios très précis pour tester notre approche. Les situations de conflits ont toujours été identifiées et traitées. Ce qui est très intéressant, c'est la façon dont les applications qui n'ont pas pu accéder aux services ont été adaptées. Il est clair que les solutions mises en œuvre dans les méthodes de callback n'étaient pas toujours satisfaisantes, selon les situations d'exécution. Toutefois, l'exécution des scénarios nous a permis d'affiner la stratégie d'adaptation, qui est un point clé de notre proposition.

5.2.4 Évaluation

Dans la section précédente, nous avons montré la faisabilité de l'approche proposée pour la gestion des conflits entre les applications de la maison dans des scénarios réalistes. Cette approche introduit des couches pour la gestion de verrous et d'interception qui présentent certainement des impacts pendant la phase de développement et d'exécution d'applications concernant respectivement le nombre de lignes de code et le temps d'exécution. Dans cette section, nous mesurons ces impacts pendant ces deux phases de cycle de vie des applications.

Afin de mesurer les principaux impacts de notre approche, nous comparons les deux versions de l'application de détection d'incendie qui ont été mises en œuvre avec et sans notre solution de gestion des conflits. La première mise en œuvre correspond à la version de référence implantée sans le code nécessaire pour la gestion des conflits. Cette application invoque directement les services critiques requis (les services représentant les ampoules communicantes). La deuxième version de l'application est implantée en utilisant les méthodes de gestion des conflits. Ces méthodes sont appelées dans le code de l'application en fonction de ses besoins. L'accès au service de verrouillage est géré par le contrôleur au moment de l'exécution. Ce processus est plus coûteux en termes de temps d'exécution et de nombre de lignes de code.

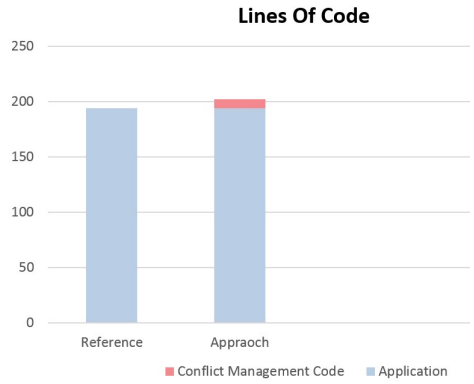

Figure 5.13: Évaluation en termes de nombre de lignes de code.

Nous avons comparé le nombre de lignes de code des deux versions de l'applications développées avec et sans notre solution de gestion des conflits, comme illustré par la figure 5.13. L'histogramme en bleu correspond à la version de référence développée sans utiliser les API necessaires pour la gestion des verrous. Le nombre de lignes de code de l'application de référence est 194 lignes alors qu'il s'agit de 202 lignes de code pour la version avec notre

code de gestion de conflits. Ainsi, le nombre de lignes ajoutées dans le code de l'application pour la gestion des conflits est d'environ 4%. Cette différence en terme de lignes de code n'est pas significative et ne complique pas la tâche des développeurs surtout que le code complexe permettant de traiter les aspects relatifs à la gestion des conflits est integré dans le conteneur des composants. Ceci permet alors aux développeurs d'éviter de produire ce code grâce à l'utilisation des annotations et des API proposées. Néanmoins, ce code ajouté garantit une exécution sans conflit au moment de l'exécution.

Comme le montre la figure 5.14, nous avons également évalué l'*overhead* en terme de temps d'exécution pour une application utilsant un service critique simple. En fait, le temps d'exécution de l'application avec le code de gestion des conflits est supérieur au temps d'exécution de l'application de référence. Cet *overhead* est principalement lié au temps nécessaire à l'acquisition du verrou dans notre scénario. En effet, il n'est pas vraiment significatif dans l'échelle de temps du scénario.

Figure 5.14: Évaluation en termes de temps d'exécution.

Le nombre de services utilisés d'une manière indirecte par une application est un autre critère permettant d'influencer le temps d'exécution. Nous avons alors évalué l'*overhead* en terme de temps d'exécution pour la même application quand elle consomme un service critique abstrait. Pour ce faire, nous avons développé trois composants : un composant qui fournit un service critique abstrait S1 et requiert deux services critiques S2 et S3 exposés par deux autres composants différents. Nous nous sommes appuyés alors sur deux versions de cette implantation : une implantation de référence et une implantation basée sur notre solution de gestion des conflits. La figure 5.15 montre que le temps d'exécution de la version avec notre approche est plus élevé que celui de la version de référence. Le temps supplémentaire est particulièrement dû à la propagation du verrou à travers le service

abstrait jusqu'aux deux services simples. Nous pouvons en effet considérer que celui-ci n'est pas significatif par rapport au temps d'exécution de la version de référence. Toutefois, il est important de constater que ce temps évolue avec le nombre de services critiques utilisés d'une manière implicite par une application. Nous estimons cependant que l'*overhead* en terme de temps d'exécution peut être considéré comme négligeable surtout que, dans la pratique, le nombre de services dans un contexte d'une plateforme ubiquitaire domotique est souvent d'environ quelques dizaines, ce qui n'impacte pas la performance de notre système.

Figure 5.15: Évaluation en termes de temps d'exécution.

En conclusion, nous considérons que, en raison du faible surcoût en termes de temps d'exécution et de lignes de code, notre solution ne complique pas la tâche des développeurs par rapport aux approches de gestion des conflits présentées dans l'état de l'art. De plus, notre solution assure l'exécution des applications ubiquitaires sans conflit.

5.3 Conclusion

Dans la première partie de ce chapitre, nous avons exposé, tout d'abord, l'architecture générale de notre proposition, les entités qui la construisent et leurs interactions. Puis, nous avons présenté en détail l'implantation fournie dans le cadre de cette thèse. Cette implantation s'est appuyée sur le modèle de composants orientés service Apache Felix iPOJO et ses mécanismes d'interception.

La deuxième partie de ce chapitre a été consacrée à la validation de notre proposition. Nous nous sommes alors basés sur les développements effectués pour tester et évaluer l'approche proposée dans cette thèse. Cette approche a été développée en tant que composants iPOJO, intégrés à la plateforme ubiquitaire iCasa et testés en s'appuyant sur des applications domotiques proposées par Orange Labs. Les différents scénarios exécutés nous ont permis de montrer que cette approche est fonctionnelle. Le système de gestion de conflits proposé étant déterministe, nous avons pu constater que les situations de conflits ont été en effet toujours identifiées et traitées.

Nous avons aussi évalué l'*overhead* en termes de lignes de code. Nous avons estimé que cet *overhead* n'est pas significatif et ne complique pas la tâche des développeurs. Nous avons considéré alors que cette tâche est minime dans notre approche. Cela est dû notamment aux API fournies pour poser et relâcher les verrous et aux annotations proposées permettant de traiter un ensemble d'aspects liés à la gestion des conflits de façon déclarative. Le code ajouté est considéré en effet négligeable par rapport au code métier. Cependant, le code complexe lié à la gestion des conflits est réparti et intégré dans les conteneurs des composants de contexte et d'applications sous la forme de *handlers*.

De plus, nous avons évalué l'*overhead* en terme de temps d'exécution. Nous avons considéré que celui-ci est négligeable par rapport à l'échelle de temps du scénario exécuté. Toutefois, il est compensé par la capacité de gestion (prévention, identification et résolution) des conflits entre les applications au moment de l'exécution.

Conclusion et perspectives

5.4 Résumé des travaux de thèse

Les solutions *smart home* sont en plein essor et s'étendent à un large spectre d'applications ubiquitaires qui exploitent les dispositifs, capteurs et actionneurs, dispersés dans nos habitats afin de nous faire profiter de meilleures conditions de vie. L'approche orientée services est aujourd'hui largement adoptée pour construire des plateformes sur lesquelles ces applications sont généralement exécutées. Certaines plateformes fournissent des services pouvant être consommés par les applications ubiquitaires pour satisfaire les besoins de leurs utilisateurs. Ces services peuvent être simples représentant des dispositifs (par exemple, une lampe) ou des services abstraits fournissant des fonctions de plus haut niveau (par exemple, la luminosité dans une pièce). Cependant, le problème de conflit entre les applications partageant ces services s'avère inévitable, surtout qu'elles peuvent y agir d'une manière incompatible pouvant parfois mettre nos maisons dans un état dangereux. Certes, la gestion de ces conflits dans ce type de plateforme s'avère un défi difficile à relever. Ceci est dû principalement au fait que les applications peuvent agir sur les dispositifs qui les entourent d'une manière implicite ou explicite pouvant compliquer et multiplier les situations de conflits.

Au cours de cette thèse, nous avons opté pour traiter les conflits pouvant se présenter entre les applications domotiques s'exécutant sur une plateforme ubiquitaire orientée services. Certes, il est difficile de gérer les conflits en phase de conception, ceci est dû à la nature dynamique de l'environnement *smart home* permettant à leur évolution d'être imprédictible (ajout ou retrait des dispositifs, installation des nouvelles applications, etc.). En revanche, la gestion des conflits entre les applications de la maison exige forcément de connaître des nombreuses informations sur leur environnement (dispositifs disponibles, etc.). Pour cette raison, nous avons adopté une approche où les conflits sont traités à l'exécution au niveau d'une représentation synchronisée de l'environnement, nommée contexte et modélisée sous forme d'un graphe des composants orientés service. A l'exécution, le contexte fournit dynamiquement tous les services simples ou abstraits requis par les applications s'exécutant sur la plateforme. Ces applications doivent passer absolument par les services contextuels pour interagir avec les dispositifs assimilés dans leur environnement. Nous avons défini deux types de conflits dans un contexte orienté services : (1) conflit direct : ce type de conflit peut se produit lorsque plusieurs applications utilisent le même service d'une manière incompatible ; (2) conflit indirect : ce type de conflit se présente entre plusieurs applications utilisant des services différents mais entraînant des actions incompatibles sur un service partagé.

Inspiré par les systèmes d'exploitation, nous avons étendu le modèle de développement d'application par des mécanismes de verrouillage/déverrouillage pour contrôler l'accès aux services contextuels critiques. Ces mécanismes permettent aux développeurs de déclarer les accès exclusifs aux services requis, apportant ainsi quelques changements mineurs sur leur code qui reste néanmoins simple et perceptible. A l'exécution, ils assurent qu'uniquement l'application détenant le verrou peut utiliser le service et effectuer les actions envisagées dans l'environnement. Un verrou présente une durée de vie qui se réfère à l'intervalle de temps au cours duquel l'application possédant le verrou est habilitée à utiliser le service à l'exécution. Cette durée dépend fortement de la logique de l'application et elle est souvent contextuelle (par exemple, maintenir les lumières allumées pendant la présence d'une personne dans une pièce). Pour cette raison, il est de la responsabilité du programmeur de la décrire dans son code selon la sémantique applicative. Il peut alors fixer explicitement la durée, l'associer à une condition basée sur des services contextuels ou définir ses limites marquées par la pose et la libération du verrou représentant le début et la fin de sa durée de vie. Le verrou présente aussi une autre propriété clé : lorsqu'il est posé sur un service abstrait, il doit nécessairement se propager jusqu'à tous les services simples requis d'une manière implicite ou explicite. En outre, à chaque service critique du contexte, nous avons associé un contrôleur responsable de gérer sa machine d'états, identifier les conflits et définir la stratégie de résolution.

Pour gérer ces conflits, nous avons adopté une approche optimiste à trois phases :

- **la prévention des conflits :** lorsqu'une application pose un verrou sur un service initialement libre, la plateforme le rend invisible (ou tous les services verrouillés si le service est abstrait) pour toutes les applications moins prioritaires. Ceci permet de minimiser les situations de conflits pouvant se présenter à l'exécution. Toutefois, ces applications doivent adapter leur comportement à ce changement afin de continuer à satisfaire les besoins de son utilisateur. Pour ce faire, le développeur est sollicité à ajouter un code d'adaptation décrivant le comportement de son application lorsqu'un service requis devient invisible.

- **la détection des conflits :** lorsque la prévention des conflits est hors de portée, leur détection s'avère indispensable. Dans notre approche, un conflit est détecté à l'exécution lorsqu'une application demande un verrou sur un service verrouillé par une autre application.

- **la résolution des conflits détectés :** pour résoudre un conflit détecté, le verrou est accordé à l'application ayant le niveau de priorité le plus élevé, même par préemption. En conséquence, seule l'application détenant le verrou peut utiliser le service au centre du conflit et effectuer toutes les actions envisagées. En revanche, l'application moins

prioritaire ne voit plus ce service et le code décrivant ses capacités d'adaptation est invoqué.

Notre approche de gestion des conflits entre les applications de la maison a été implantée selon le modèle des composants orientés service iPOJO. Cette solution a été aussi intégrée à la plateforme ubiquitaire domotique iCASA. Puis, elle a été testée et évaluée à l'aide des applications proposées par Orange Labs.

5.5 Perspectives

Au cours de cette thèse, nous avons montré qu'il est possible d'assurer la cohabitation des applications ubiquitaires dans un environnement Smart Home ouvert et dynamique. Notre travail de recherche propose une approche pour gérer les conflits pouvant se présenter entre ces applications dans une plateforme ubiquitaire orientée services. Toutefois, ce travail reste ouvert à plusieurs améliorations et nous considérons que différentes perspectives peuvent alors être investiguées.

5.5.1 Priorités dynamiques d'applications

Une des perspectives importantes qui peut être considérée afin d'étendre notre approche concerne les priorités dynamiques des applications ubiquitaires s'exécutant sur la plateforme. Actuellement, nous avons adopté une politique de contrôle d'accès simple basée sur des priorités statiques d'applications. Les applications sont classées selon un schéma qui fixe leur ordre de priorité. Ces priorités sont définies selon les catégories d'applications et à chaque application déployée sur la plateforme est attribué un seul niveau de priorité.

Certes, l'aspect dynamique des priorités d'applications peut être intéressant. Ces priorités peuvent être calculées en fonction de leur contexte d'exécution, ce qui impacte certainement le contrôle d'accès aux ressources. Par exemple, une application dédiée pour la gestion de la consommation énergétique peut avoir une priorité plus élevée pendant la journée mais pas le soir. Pour faire évoluer notre approche dans cette voie, il peut être intéressant d'adopter une politique de contrôle d'accès plus sophistiquée basée sur des priorités dynamiques. Cette politique peut être décrite par un modèle de contrôle d'accès où nous pouvons définir des contraintes dynamiques liées à l'environnement (par exemple, l'état de la maison, etc.).

5.5.2 Vers des contrôleurs plus intelligents et une adaptation plus spécifique

Une autre perspective qui peut étendre notre travail de recherche porte sur les contrôleurs proposés. En effet, dans notre approche, nous avons étendu le modèle du contexte par des contrôleurs associés aux services critiques. L'une des responsabilités majeures de ces contrôleurs est de gérer la machine d'états de ces services. En effet, lorsqu'une application pose un verrou sur un service critique, ce dernier passe à l'état verrouillé et la plateforme le rend indisponible pour toutes les applications moins prioritaires. Toutefois, la dynamicité

étant une caractéristique clé de l'environnement *smart home*, les services peuvent disparaitre (devenir indisponible) à tout moment pour diverses raisons, par exemple à cause d'un dysfonctionnement, pour une durée de vie de batterie limitée, ou tout simplement le déplacement ou le retrait du dispositif par son utilisateur. Il peut absolument être intéressant de rendre les contrôleurs plus intelligents et plus conscients de l'évolution de leur l'environnement. Ces contrôleurs peuvent alors avoir la capacité d'examiner et d'identifier les causes exactes de la disparition des services proposés par la plateforme. Ceci va permettre de mettre en place différentes sortes d'adaptation de l'application qui peuvent se changer selon les causes de l'indisponibilité de service. Dans l'approche actuelle, le code décrit par le développeur pour adapter le comportement de son application lorsqu'elle est notifiée de la disparition d'un service requis à l'exécution est indépendant de toute cause. Certes, déterminer la cause permettra aux développeurs d'adapter leurs applications aux changements dans leur contexte d'exécution d'une façon plus pertinente. Les applications pourront alors avoir des capacités d'adaptation plus spécifiques.

5.5.3 Interopérabilité des plateformes ubiquitaires

Aujourd'hui, certains projets de recherche promeuvent l'interopérabilité des plateformes ubiquitaires dans les environnements Smart Home. Cette interopérabilité étant un aspect crucial, elle permet aux systèmes ubiquitaires domotiques de collaborer afin d'améliorer la qualité de vie des habitants. Ces projets envisagent un réseau domestique où les applications ubiquitaires exécutées sur une plateforme sont habilitées à consommer des services distants proposés par d'autres plateformes. Il s'avère alors indispensable d'assurer la cohabitation de ces plateformes et leur collaboration dans un cadre bien contrôlé. Dans ce cadre, un aspect important à étudier est de gérer les conflits des applications à l'échelle globale et non seulement les conflits au niveau de chaque plateforme d'une manière isolée sans tenir compte des conflits pouvant se produire entre les applications s'exécutant sur des plateformes différentes.

Certes, notre approche de gestion de conflits est applicable pour chaque plateforme du réseau présentant son propre contexte. Cependant, le nouveau défi soulevé est la gestion des conflits qui peuvent se présenter entre des applications déployées sur des plateformes différentes. Notre approche peut en effet être adoptée pour relever ce défi. La topologie de réseau domestique actuellement proposée étant une topologie en étoile où les plateformes ubiquitaires sont reliées à un nœud central, il peut alors être intéressant que ce nœud ait une représentation globale de tout l'environnement (autrement dit un contexte global représentant les contextes des différentes plateformes présentes dans l'environnement) ainsi que tous mécanismes nécessaires pour la gestion de conflits.

Bibliography

[Abo+99] Gregory D. Abowd et al. "Towards a Better Understanding of Context and Context-Awareness." In: *Handheld and Ubiquitous Computing, First International Symposium, HUC'99, Karlsruhe, Germany, September 27-29, 1999, Proceedings*. 1999, pp. 304–307. DOI: 10.1007/3-540-48157-5_29. URL: https://doi.org/10.1007/3-540-48157-5_29.

[AIM10] Luigi Atzori, Antonio Iera, and Giacomo Morabito. "The Internet of Things: A survey." In: *Computer Networks* 54.15 (2010), pp. 2787–2805. DOI: 10.1016/j.comnet.2010.05.010. URL: https://doi.org/10.1016/j.comnet.2010.05.010.

[Ald03] Frances K. Aldrich. "Smart Homes: Past, Present and Future." In: *Inside the Smart Home*. Ed. by Richard Harper. London: Springer London, 2003, pp. 17–39. ISBN: 978-1-85233-854-1. DOI: 10.1007/1-85233-854-7_2. URL: https://doi.org/10.1007/1-85233-854-7_2.

[All] OSGi Alliance. *OSGi website*. URL: https://www.osgi.org/.

[Ayg+16] Colin Aygalinc et al. "A model-based approach to context management in pervasive platforms." In: *Pervasive Computing and Communication Workshops (PerCom Workshops), 2016 IEEE International Conference on*. IEEE. 2016, pp. 1–6.

[Ban+00] Guruduth Banavar et al. "Challenges: an application model for pervasive computing." In: *MOBICOM 2000, Proceedings of the sixth annual international conference on Mobile computing and networking, Boston, MA, USA, August 6-11, 2000*. 2000, pp. 266–274. DOI: 10.1145/345910.345957. URL: http://doi.acm.org/10.1145/345910.345957.

[Bar12] Jonathan Bardin. "RoSe : un framework pour la conception et l'exécution d'applications distribuées dynamiques et hétérogènes. (RoSe : A framework for the design and execution of dynamic and heterogeneous distributed applications)." PhD thesis. Grenoble Alpes University, France, 2012. URL: https://tel.archives-ouvertes.fr/tel-00750739.

[BBC97] Peter J. Brown, J. D. Bovey, and Xian Chen. "Context-aware applications: from the laboratory to the marketplace." In: *IEEE Personal Commun.* 4.5 (1997), pp. 58–64. DOI: 10.1109/98.626984. URL: https://doi.org/10.1109/98.626984.

[BBL] MA Benatia, D Baudry, and A Louis. "Livrable CREST." In: ().

[BC10] Emilie Balland and Charles Consel. "Open platforms: new challenges for software engineering." In: *Programming Support Innovations for Emerging Distributed Applications*. ACM. 2010, p. 3.

[BDR07] Matthias Baldauf, Schahram Dustdar, and Florian Rosenberg. "A survey on context-aware systems." In: *IJAHUC* 2.4 (2007), pp. 263–277. DOI: 10.1504/IJAHUC.2007.014070. URL: https://doi.org/10.1504/IJAHUC.2007.014070.

[Bec+03] Christian Becker et al. "BASE - A Micro-Broker-Based Middleware for Pervasive Computing." In: *Proceedings of the First IEEE International Conference on Pervasive Computing and Communications (PerCom'03), March 23-26, 2003, Fort Worth, Texas, USA*. 2003, pp. 443–451. DOI: 10.1109/PERCOM.2003.1192769. URL: https://doi.org/10.1109/PERCOM.2003.1192769.

[Bec+04] Christian Becker et al. "PCOM - A Component System for Pervasive Computing." In: *Proceedings of the Second IEEE International Conference on Pervasive Computing and Communications (PerCom 2004), 14-17 March 2004, Orlando, FL, USA*. 2004, pp. 67–76. DOI: 10.1109/PERCOM.2004.1276846. URL: https://doi.org/10.1109/PERCOM.2004.1276846.

[Ber+14] Benjamin Bertran et al. "DiaSuite: A tool suite to develop Sense/Compute/Control applications." In: *Sci. Comput. Program.* 79 (2014), pp. 39–51. DOI: 10.1016/j.scico.2012.04.001. URL: https://doi.org/10.1016/j.scico.2012.04.001.

[Bet+10] Claudio Bettini et al. "A survey of context modelling and reasoning techniques." In: *Pervasive and Mobile Computing* 6.2 (2010), pp. 161–180. DOI: 10.1016/j.pmcj.2009.06.002. URL: https://doi.org/10.1016/j.pmcj.2009.06.002.

[Bib77] Kenneth J Biba. *Integrity considerations for secure computer systems*. Tech. rep. MITRE CORP BEDFORD MA, 1977.

[BJC09] Julien Bruneau, Wilfried Jouve, and Charles Consel. "DiaSim: A parameterized simulator for pervasive computing applications." In: *6th Annual International Conference on Mobile and Ubiquitous Systems: Computing, Networking and Services, MOBIQUITOUS 2009, Toronto, Canada, July 13-16, 2009*. 2009, pp. 1–10. DOI: 10.4108/ICST.MOBIQUITOUS2009.6851. URL: https://doi.org/10.4108/ICST.MOBIQUITOUS2009.6851.

[BL73] D Elliott Bell and Leonard J LaPadula. *Secure computer systems: Mathematical foundations*. Tech. rep. MITRE CORP BEDFORD MA, 1973.

[BV94] LG Bouma and Hugo Velthuijsen. *Feature interactions in telecommunications systems*. IOS press, 1994.

[Can92] Anne Cancellieri. *L'habitat du futur: défis et prospective pour le prochain quart de siècle*. Documentation française, 1992.

[Cas+09] Damien Cassou et al. "A generative programming approach to developing pervasive computing systems." In: *ACM Sigplan Notices*. Vol. 45. 2. ACM. 2009, pp. 137–146.

[Cas11] Damien Cassou. "Développement logiciel orienté paradigme de conception: la programmation dirigée par la spécification." PhD thesis. Université Sciences et Technologies-Bordeaux I, 2011.

[Cer04] Humberto Cervantes. "Vers un modèle à composants orienté services pour supporter la disponibilité dynamique." PhD thesis. Université Joseph-Fourier-Grenoble I, 2004.

[CH01] Bill Councill and George T Heineman. "Definition of a software component and its elements." In: *Component-based software engineering: putting the pieces together* (2001), pp. 5–19.

[CH04] Humberto Cervantes and Richard S Hall. "Autonomous adaptation to dynamic availability using a service-oriented component model." In: *Proceedings of the 26th International Conference on Software Engineering*. IEEE Computer Society. 2004, pp. 614–623.

[Cha+08] Marie Chan et al. "A review of smart homes—Present state and future challenges." In: *Computer methods and programs in biomedicine* 91.1 (2008), pp. 55–81.

[CK00] Guanling Chen and David Kotz. *A Survey of Context-Aware Mobile Computing Research*. Tech. rep. Hanover, NH, USA, 2000.

[CM00] Muffy Calder and Evan Magill. *Feature Interactions in Telecommunications and software systems VI*. IOS Press, 2000.

[Col+16] Aygalinc Colin et al. "Autonomic service-oriented context for pervasive applications." In: *Services Computing (SCC), 2016 IEEE International Conference on*. IEEE. 2016, pp. 491–498.

[CV93] E Jane Cameron and Hugo Velthuijsen. "Feature interactions in telecommunications systems." In: *IEEE Communications Magazine* 31.8 (1993), pp. 18–23.

[Den76] Dorothy E Denning. "A lattice model of secure information flow." In: *Communications of the ACM* 19.5 (1976), pp. 236–243.

[ECL14] Clément Escoffier, Stéphanie Chollet, and Philippe Lalanda. "Lessons learned in building pervasive platforms." In: *11th IEEE Consumer Communications and Networking Conference, CCNC 2014, Las Vegas, NV, USA, January 10-13, 2014*. 2014, pp. 7–12. DOI: 10.1109/CCNC.2014.6866540. URL: https://doi.org/10.1109/CCNC.2014.6866540.

[EG01] W Edwards and Rebecca Grinter. "At home with ubiquitous computing: Seven challenges." In: *Ubicomp 2001: Ubiquitous Computing*. Springer. 2001, pp. 256–272.

[EOM09] William Enck, Machigar Ongtang, and Patrick McDaniel. "Understanding android security." In: *IEEE security & privacy* 7.1 (2009), pp. 50–57.

[Esc08] Clément Escoffier. "iPOJO : Un modèle à composant à service flexible pour les systèmes dynamiques. (iPOJO : A flexible service-oriented component model for dynamic systems)." PhD thesis. Joseph Fourier University, Grenoble, France, 2008. URL: https://tel.archives-ouvertes.fr/tel-00347935.

[Esw+76] Kapali P. Eswaran et al. "The notions of consistency and predicate locks in a database system." In: *Communications of the ACM* 19.11 (1976), pp. 624–633.

[Far13] Aurélien Faravelon. "Une démarche de conception et d'implémentation de la protection de la vie privée basée sur le contrôle d'accès appliquée aux compositions de services." PhD thesis. Université de Grenoble, 2013.

[Fer+01] David F Ferraiolo et al. "Proposed NIST standard for role-based access control." In: *ACM Transactions on Information and System Security (TISSEC)* 4.3 (2001), pp. 224–274.

[Fer+03] David F Ferraiolo et al. "The role control center: features and case studies." In: *Proceedings of the eighth ACM symposium on Access control models and technologies*. ACM. 2003, pp. 12–20.

[Fow00] Martin Fowler. *POJO : An acronym for: Plain Old Java Object*. 2000. URL: http://www.martinfowler.com/bliki/POJO.html.

[Gai] Gaia. *Gaia web site*. URL: http://gaia.cs.illinois.edu/.

[Gal12] Mathieu Gallissot. "Modéliser le concept de confort dans un habitat intelligent: du multisensoriel au comportement." PhD thesis. Grenoble, 2012.

[Gar03] Georges Gardarin. *Bases de données*. Editions Eyrolles, 2003.

[Gar12] Issac Noe Garcia. "Modèles de conception et d'exécution pour la médiation et l'intégration de services. (Conception and execution models to mediate and integrate service)." PhD thesis. Grenoble Alpes University, France, 2012. URL: https://tel.archives-ouvertes.fr/tel-00767953.

[GB03] Robert Grimm and Brian N. Bershad. "System Support for Pervasive Applications." In: *Future Directions in Distributed Computing, Research and Position Papers*. 2003, pp. 212–217. DOI: 10.1007/3-540-37795-6_42. URL: https://doi.org/10.1007/3-540-37795-6_42.

[Gre10] Adam Greenfield. *Everyware: The dawning age of ubiquitous computing*. New Riders, 2010.

[Gro08] Object Management Group. *CORBA 3.1*. 2008. URL: http://www.omg.org/spec/CORBA/3.1/.

[GW12] Bernhard Ganter and Rudolf Wille. *Formal concept analysis: mathematical foundations*. Springer Science & Business Media, 2012.

[Hal+11] Richard Hall et al. *OSGi in action: Creating modular applications in Java*. Manning Publications Co., 2011.

[Han+03] Uwe Hansmann et al. *Pervasive computing: The mobile world*. Springer Science & Business Media, 2003.

[Hen03] Karen Henricksen. *A framework for context-aware pervasive computing applications*. University of Queensland Queensland, 2003.

[HPT97] Michael Hawley, R. Dunbar Poor, and Manish Tuteja. "Things that think." In: *Personal Technologies* 1.1 (1997), pp. 13–20. ISSN: 1617-4917. DOI: 10.1007/BF01317884. URL: http://dx.doi.org/10.1007/BF01317884.

[HRU76] Michael A Harrison, Walter L Ruzzo, and Jeffrey D Ullman. "Protection in operating systems." In: *Communications of the ACM* 19.8 (1976), pp. 461–471.

[IMA] IMAG. *Self-star website*. URL: https://self-star.imag.fr/.

[Jak11] Henner Jakob. "Towards securing pervasive computing systems by design: a language approach." PhD thesis. Université Sciences et Technologies-Bordeaux I, 2011.

[JCL11] Henner Jakob, Charles Consel, and Nicolas Loriant. "Architecturing Conflict Handling of Pervasive Computing Resources." In: *DAIS*. Vol. 11. Springer. 2011, pp. 92–105.

[JD12] Alma L Juarez Dominguez. "Detection of feature interactions in automotive active safety features." In: (2012).

[Jeu05] François-Xavier Jeuland. *La maison communicante*. Eyrolles, 2005.

[KA00] Mari Korkea-Aho. "Context-aware applications survey." In: *Department of Computer Science, Helsinki University of Technology* (2000).

[KLS07] Mathias Kohler, Christian Liesegang, and Andreas Schaad. "Classification model for access control constraints." In: *Performance, Computing, and Communications Conference, 2007. IPCCC 2007. IEEE Internationa*. IEEE. 2007, pp. 410–417.

[KMW03] Mario Kolberg, Evan H Magill, and Michael Wilson. "Compatibility issues between services supporting networked appliances." In: *IEEE Communications Magazine* 41.11 (2003), pp. 136–147.

[Kra07] Sacha Krakowiak. *Middleware Architecture with Patterns and Frameworks*. 2007.

[KV95] K Kimbler and H Velthuijsen. "Feature interaction benchmark." In: *Third Feature Interaction Workshop (FIW'95)*. 1995.

[Lal+10] P Lalanda et al. "Smart home systems." In: *Smart Home Systems*. InTech, 2010.

[Lal+15] Philippe Lalanda et al. "Service-based architecture and frameworks for pervasive health applications." In: *20th IEEE Conference on Emerging Technologies & Factory Automation, ETFA 2015, Luxembourg, September 8-11, 2015*. 2015, pp. 1–8. DOI: 10.1109/ETFA.2015.7301659. URL: https://doi.org/10.1109/ETFA.2015.7301659.

[Lam74] Butler W Lampson. "Protection." In: *ACM SIGOPS Operating Systems Review* 8.1 (1974), pp. 18–24.

[Li11] Ninghui Li. "Discretionary access control." In: *Encyclopedia of Cryptography and Security*. Springer, 2011, pp. 353–356.

[Mat01] Friedemann Mattern. "The Vision and Technical Foundations of Ubiquitous Computing." In: *Upgrade* 2.5 (Oct. 2001). Journal article (.pdf), pp. 2–6.

[Mat05] Friedemann Mattern. "Ubiquitous Computing: Scenarios from an informatised world." In: *E-Merging Media - Communication and the Media Economy of the Future*. Ed. by Axel Zerdick et al. Springer-Verlag, 2005, pp. 145–163.

[MBT08] Silvestro Micera, Paolo Bonato, and Toshiyo Tamura. "Gerontechnology." In: *IEEE Engineering in Medicine and Biology Magazine* 27.4 (2008), pp. 10–14.

[MG13] Diana Moreno-Garcia. "Modèles, outils et plate-forme d'exécution pour les applications à service dynamiques." PhD thesis. Université de Grenoble, 2013.

[MS14] Sirajum Munir and John A Stankovic. "DepSys: Dependency aware integration of cyber-physical systems for smart homes." In: *Cyber-Physical Systems (ICCPS), 2014 ACM/IEEE International Conference on*. IEEE. 2014, pp. 127–138.

[MT10] Nenad Medvidovic and Richard N. Taylor. "Software architecture: foundations, theory, and practice." In: *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering - Volume 2, ICSE 2010, Cape Town, South Africa, 1-8 May 2010*. 2010, pp. 471–472. DOI: 10.1145/1810295.1810435. URL: http://doi.acm.org/10.1145/1810295.1810435.

[NIM05] Masahide Nakamura, Hiroshi Igaki, and Ken-ichi Matsumoto. "Feature interactions in integrated services of networked home appliances." In: *Proc. of Int'l. Conf. on Feature Interactions in Telecommunication Networks and Distributed Systems (ICFI'05)*. 2005, pp. 236–251.

[Pap03] Mike P. Papazoglou. "Service-Oriented Computing: Concepts, Characteristics and Directions." In: *4th International Conference on Web Information Systems Engineering, WISE 2003, Rome, Italy, December 10-12, 2003*. 2003, pp. 3–12. DOI: 10.1109/WISE.2003.1254461. URL: https://doi.org/10.1109/WISE.2003.1254461.

[Par+05] Insuk Park et al. "A dynamic context conflict resolution scheme for group-aware ubiquitous computing environments." In: *Proceedings of the 1st International Workshop on Personalized Context Modeling and Management for UbiComp Applications (ubiPCMM 2005)*. 2005, pp. 42–47.

[PR98] Malte Plath and Mark Dermot Ryan. "Plug-and-play Features." In: *FIW*. 1998, pp. 150–164.

[Ran+05] Anand Ranganathan et al. "Olympus: A High-Level Programming Model for Pervasive Computing Environments." In: *3rd IEEE International Conference on Pervasive Computing and Communications (PerCom 2005), 8-12 March 2005, Kauai Island, HI, USA*. 2005, pp. 7–16. DOI: 10.1109/PERCOM.2005.26. URL: https://doi.org/10.1109/PERCOM.2005.26.

[RC03] Anand Ranganathan and Roy H Campbell. "An infrastructure for context-awareness based on first order logic." In: *Personal and Ubiquitous Computing* 7.6 (2003), pp. 353–364.

[Rom+02a] Manuel Román et al. "A Middleware Infrastructure for Active Spaces." In: *IEEE Pervasive Computing* 1.4 (2002), pp. 74–83. DOI: 10.1109/MPRV.2002.1158281. URL: https://doi.org/10.1109/MPRV.2002.1158281.

[Rom+02b] Manuel Román et al. "Gaia: a middleware platform for active spaces." In: *Mobile Computing and Communications Review* 6.4 (2002), pp. 65–67. DOI: 10.1145/643550.643558. URL: http://doi.acm.org/10.1145/643550.643558.

[Rom+12] Rim Romdhane et al. "Automatic video monitoring system for assessment of Alzheimer's disease symptoms." In: *The journal of nutrition, health & aging* 16.3 (2012), pp. 213–218.

[Ron09] Daniel Ronzani. "The battle of concepts: Ubiquitous Computing, pervasive computing and ambient intelligence in Mass Media." In: *Ubiquitous Computing and Communication Journal* 4.2 (2009), pp. 9–19.

[Sat01] Mahadev Satyanarayanan. "Pervasive computing: vision and challenges." In: *IEEE Personal Commun.* 8.4 (2001), pp. 10–17. DOI: 10.1109/98.943998. URL: https://doi.org/10.1109/98.943998.

[SAW94] Bill N. Schilit, Norman Adams, and Roy Want. "Context-Aware Computing Applications." In: *First Workshop on Mobile Computing Systems and Applications, WMCSA 1994, Santa Cruz, CA, USA, December 8-9, 1994*. 1994, pp. 85–90. DOI: 10.1109/WMCSA.1994.16. URL: https://doi.org/10.1109/WMCSA.1994.16.

[Sch11] Stephen R Schach. *Object-oriented and classical software engineering*. Boston: McGraw-Hill Higher Education, 2011.

[SFK04] Ravi Sandhu, David Ferraiolo, and Richard Kuhn. "American National Standard for Information Technology–Role based Access Control." In: *ANSI INCITS* 359 (2004), pp. 1–49.

[SGM02] Clemens A. Szyperski, Dominik Gruntz, and Stephan Murer. *Component software - beyond object-oriented programming, 2nd Edition*. Addison-Wesley component software series. Addison-Wesley, 2002. ISBN: 0201745720. URL: http://www.worldcat.org/oclc/248041840.

[SS94] Ravi S Sandhu and Pierangela Samarati. "Access control: principle and practice." In: *IEEE communications magazine* 32.9 (1994), pp. 40–48.

[ST94] Bill N Schilit and Marvin M Theimer. "Disseminating active map information to mobile hosts." In: *IEEE network* 8.5 (1994), pp. 22–32.

[SV00] Pierangela Samarati and Sabrina Capitani de Vimercati. "Access control: Policies, models, and mechanisms." In: *International School on Foundations of Security Analysis and Design*. Springer. 2000, pp. 137–196.

[Tay97] David A Taylor. *Object technology: a manager's guide*. Addison-Wesley Longman Publishing Co., Inc., 1997.

[TSB07] Verena Tuttlies, Gregor Schiele, and Christian Becker. "Comity-conflict avoidance in pervasive computing environments." In: *On the move to meaningful internet systems 2007: OTM 2007 workshops*. Springer. 2007, pp. 763–772.

[VSB13] Sebastian VanSyckel, Gregor Schiele, and Christian Becker. "Extending context management for proactive adaptation in pervasive environments." In: *Ubiquitous Information Technologies and Applications*. Springer, 2013, pp. 823–831.

[Wal07] Jean-Baptiste Waldner. *Nano-informatique et intelligence ambiante: inventer l'ordinateur du XXIe siècle*. Hermès Science, 2007.

[Wan+92] Roy Want et al. "The Active Badge Location System." In: *ACM Trans. Inf. Syst.* 10.1 (1992), pp. 91–102. DOI: 10.1145/128756.128759. URL: http://doi.acm.org/10.1145/128756.128759.

[WB96] Mark Weiser and John Seely Brown. "Designing calm technology." In: *PowerGrid Journal* 1.1 (1996), pp. 75–85.

[Wei91] Mark Weiser. "The computer for the 21st century." In: *Scientific american* 265.3 (1991), pp. 94–104.

[YIH15] Miki Yagita, Fuyuki Ishikawa, and Shinichi Honiden. "An application conflict detection and resolution system for smart homes." In: *Proceedings of the First International Workshop on Software Engineering for Smart Cyber-Physical Systems*. IEEE Press. 2015, pp. 33–39.

[YT05] Eric Yuan and Jin Tong. "Attributed based access control (ABAC) for web services." In: *Web Services, 2005. ICWS 2005. Proceedings. 2005 IEEE International Conference on*. IEEE. 2005.

Résumé — L'émergence avérée des dispositifs dynamiques et hétérogènes ouvre la voie à l'apparition d'un nouveau type d'applications, qualifié d'ubiquitaire, qui présente la capacité d'interagir avec les capteurs et les actionneurs pénétrant nos environnements d'une manière transparente. Dans ce travail de recherche, nous nous intéressons en particulier aux applications ubiquitaires déployées dans les maisons intelligentes. Les plateformes ubiquitaires orientées services sont largement utilisées pour exécuter ces applications. Ces plateformes présentent la capacité de fournir, d'une manière dynamique, des services en fonction des besoins des applications s'y exécutant. Ces services peuvent être simples correspondant à des dispositifs ou plus abstraits fournissant des fonctions de plus haut niveau. Les applications ubiquitaires partagent ces services pour réaliser des objectifs différents, parfois conflictuels. Ces conflits doivent être traités afin de maintenir les maisons dans des états cohérents. Cette thèse définit une approche pour la gestion des conflits entre les applications de la maison dans une plateforme orientée services. Cette approche est optimiste et gère les conflits à l'exécution via un modèle causal de l'environnement, nommé contexte. Cette approche s'articule principalement autour de trois axes : le premier axe se concentre sur la description des conflits dans un contexte modélisé sous la forme de composants orientés services ; le deuxième axe consiste en l'extension du modèle de programmation d'applications ubiquitaires par des mécanismes de verrouillage/déverrouillage ; le troisième axe se penche sur la gestion des conflits en adoptant une approche à trois phases (prévention, détection et résolution). La solution proposée a été développée sous la forme de composants iPOJO et intégrée dans la plateforme ubiquitaire domotique iCasa.

Mots clés : gestion des conflits, contexte, plateforme orientée services, application ubiquitaire, maison intelligente.

Université Grenoble Alpes, LIG

Abstract — The important emergence of dynamic and heterogeneous devices paves the way for the emergence of a new type of ubiquitous applications that has the ability to interact with sensors and actuators that penetrate our environments in a transparent way. In this research, we are particularly interested in the ubiquitous applications deployed in smart homes. Service-oriented platforms are widely used to run these applications. These platforms present the ability to dynamically provide services, according to the applications needs. These services can be simple, representing a device, or more abstract, providing higher level functions. Ubiquitous applications share these services to achieve different and sometimes conflicting goals. These conflicts need to be managed in order to keep the houses in consistent states. This thesis defines an approach for managing conflicts between home applications in a service-oriented platform. This approach is optimistic and addresses conflicts at runtime via a causal model of the environment, called context. This approach focuses mainly on three axes: the first axis focuses on the description of conflicts in a context modeled as service-oriented components; the second axis consist in the extension of the programming model of ubiquitous applications by locking/unlocking mechanisms; the third axis focuses on conflict management using a three-phase approach (prevention, detection and resolution). The proposed solution was developed as iPOJO components and integrated into the ubiquitous platform iCasa.

Keywords: conflict management, context, service-oriented platform, ubiquitous application, smart home.

Université Grenoble Alpes, LIG