Un modèle à composant pour la gestion de contextes pervasifs orientés service

Colin Aygalinc

Communauté
UNIVERSITÉ Grenoble Alpes

THÈSE

Pour obtenir le grade de

DOCTEUR DE LA COMMUNAUTÉ UNIVERSITÉ GRENOBLE ALPES

Spécialité : Informatique

Arrêté ministériel : 25 mai 2016

Présentée par

Colin AYGALINC

Thèse dirigée par **Philippe LALANDA**, PR1, UGA

préparée au sein du **Laboratoire Laboratoire d'Informatique de Grenoble**
dans **l'École Doctorale Mathématiques, Sciences et technologies de l'information, Informatique**

Un modèle à composant pour la gestion de contextes pervasifs orientés service

A component model for pervasive service oriented context management

Thèse soutenue publiquement le **18 décembre 2017**,
devant le jury composé de :

Monsieur PHILIPPE LALANDA
PROFESSEUR, UNIVERSITE GRENOBLE ALPES, Directeur de thèse
Monsieur PHILIPPE ROOSE
MAITRE DE CONFERENCES, UNIVERSITE DE PAU ET DES PAYS DE L'ADOUR, Rapporteur
Monsieur GERMAN EDUARDO VEGA BAEZ
INGENIEUR DE RECHERCHE, CNRS DELEGATION ALPES, Examinateur
Monsieur OLIVIER BARAIS
PROFESSEUR, UNIVERSITE RENNES 1, Rapporteur
Madame CLAUDIA RONCANCIO
PROFESSEUR, GRENOBLE INP, Président
Madame ADA DIACONESCU
MAITRE DE CONFERENCES, TELECOM PARISTECH, Examinateur

UNIVERSITÉ DE GRENOBLE

ÉCOLE DOCTORALE MSTII

Ecole Doctorale Mathématiques, Sciences et Technologies de l'Information, Informatique

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

Colin AYGALINC

Un modèle à composant pour la gestion de contextes pervasifs orientés service

Thèse dirigée par Philippe LALANDA

préparée au Laboratoire d'Informatique de Grenoble (LIG)

soutenue le 18 décembre 2017

Jury :

Rapporteurs :	Philippe ROOSE	-	LIUPPA/UPPA
	Olivier BARAIS	-	INRIA/IRISA Rennes
Directeur :	Philippe LALANDA	-	LIG
Co-Encadrant :	German VEGA	-	CNRS
Présidente :	Claudia RONCANCIO	-	Grenoble INP
Examinaterice :	Ada DIACONESCU	-	Telecom ParisTech

Remerciements

Je remercie les membres du jury; *Olivier Barais, Philippe Roose, Ada Diaconescu & Claudia Roncancio*; qui m'ont fait l'honneur de participer à ma soutenance et d'examiner mes différents travaux.

Je tiens ensuite à remercier les personnes qui m'ont encadré pendant cette thèse car sans eux, je n'aurais pu aboutir à ce résultat tout simplement. Je témoigne donc ma sincère reconnaissance à *Philippe Lalanda & German Vega*.

Je souhaite bien évidemment remercier toutes les personnes qui ont peuplé mon quotidien pendant ces trois années et qui ont pu par leur encouragement et leur présence contribuées à rendre la vie plus agréable. Je remercie donc tout mes amis; ceux que j'ai pu rencontrer grâce au labo; *Paola, Rania, Eva, Jander, Fatemeh, Mario, Cyril, Ornella, Vincent, Stephanie, Ozan & Pierre*; la bande des zyngoles masquées; *Rochedy, Savonet, Bannelier, Francky, Mémé, Lutin, Labio, Elé, Leila, Nico, Génie, Loulou, Redoute, Caillon, Raf, Papa Ours, Lilia, Jannis, Matours, Mathou, Guilhem, Laurie, Polo, Clarinette, The Max, Pauline, Aurel', Gwen & Joanny*; ainsi que mes amis d'enfance; *Guich, Grabit, Tony Boy & Beni*.

Enfin, je souhaite remercier toute ma famille et particulièrement mes parents; *Pac & Soiz*; ainsi que mes frères; *Maël & Gabriel*; pour leur aide et le soutien qu'ils m'ont apporté.

Contents

List of Figures

List of Tables

Table des acronymes

CReAM	*Context Representation And Management*
CPL	*Courant Porteur en Ligne*
DSL	*Domain Specific Language*
iPOJO	*Injected Plain Old Java Object*
JVM	*Java Virtual Machine*
LIG	*Laboratoire d'Informatique de Grenoble*
LWPA	*Low Power Wide Arena*
OSGi	*Open Serivce Gateway Initiative*
POJO	*Plain Old Java Object*
QoC	*Quality of Context*
QoS	*Quality of Service*
SLA	*Service Level Aggreement*
SOA	*Service Oriented Architecture*
SOC	*Service Oriented Computing*
SOCM	*Service Oriented Component Model*

Liste des publications

Les travaux contenus dans ce manuscrit ont été partiellement publiés à travers les articles suivant:

- "***Service-based architecture and frameworks for pervasive health applications***" dans *IEEE Service-Oriented Cyber-Physical Systems in Converging Networked Environments workshop* affilié à *IEEE Emerging Technologies And Factory Automation* [Lal+15]
- "***Service-oriented context for pervasive applications***" dans *IEEE Pervasive Computing* [Ayg+16e]
- "***A Model-Based Approach to Context Management in Pervasive Platforms***" dans *IEEE Context and Activity Modeling and Recognition workshop* affilié à *IEEE Pervasive Computing* [Ayg+16a]
- "***Autonomic management of pervasive context***" dans *IEEE International Conference on Autonomic Computing* [Ayg+16b]
- "***Autonomic service-oriented context for pervasive applications***" dans *IEEE International Conference on Services Computing* [Ayg+16c]
- "***Service-Oriented Autonomic Pervasive Context***" dans *International Conference on Service-Oriented Computing* [Ayg+16d]

Introduction

Nous sommes des nains juchés sur les épaules de géants; nous voyons plus qu'eux, et plus loin; non que notre regard soit perçant, ni élevée notre taille, mais nous sommes élevés, exhaussés, par leur stature gigantesque.

Bernard de Chartres

Contexte et motivation

Un nombre toujours plus important d'objets connectés et intelligents sont intégrés dans nos environnements du quotidien. Ceci est principalement dû aux avancées majeures dans les technologies matérielles et réseaux qui ont permis d'aboutir à la construction de capteurs ou d'actionneurs plus puissants et connectés tandis que leurs coûts et leurs tailles baissaient. Ces environnements de plus en plus digitalisés, qualifiés de pervasifs dans la littérature, sont de plus en plus acceptés par les utilisateurs pendant leurs activités sociales, domestiques ou professionnelles. Cependant, la réelle plus-value de ces environnements réside dans l'émergence d'applications capables d'interagir avec ce patchwork de ressources hétérogènes, volatiles, partagées et distribuées pour fournir des services à forte valeur ajoutée de manière continue et transparente. Ces applications soulèvent de grandes attentes tant économiques que sociales dans des domaines tels que la maison connectée, l'industrie 4.0, le commerce ou la santé.

La notion d'application, centrale dans l'informatique pervasive, n'est cependant pas nouvelle. Depuis les débuts de l'informatique, les systèmes d'exploitation ont fourni des briques logicielles permettant aux applications de s'exécuter. Cependant, les infrastructures d'exécution ne se limitent plus à un unique ordinateur. Les applications pervasives peuvent être déployées et exécutées dans des équipements physiques (capteurs, actionneurs) ou des infrastructures *Cloud Computing* en passant par de multiples plateformes intermédiaires. Ces infrastructures diffèrent en termes de puissance, connectivité, coût ou consommation énergétique. Un consensus semble s'être établi autour de la place de l'application pervasive. Les services nécessitant de grosses ressources de calcul, un nombre important de données provenant de multiples endroits sont situés dans le *Cloud* pour profiter d'un fort potentiel de calcul à un coût moindre. Les services misant davantage sur une forte réactivité se déploient au plus près de l'utilisateur, dans des passerelles qualifiées de *Fog Computing* pour bénéficier de temps de réponse favorable et moins subir les aléas du réseau. Une des caractéristiques communes à ces services résulte de la volonté de s'appuyer sur des informations extraites de l'environnement et d'interagir avec celui-ci pour pouvoir s'adapter et livrer un service de plus en plus personnalisé à l'utilisateur et sa situation. Cette caractéristique a été largement étudiée au cours de l'histoire de l'informatique et prend le nom de sensibilité au contexte, ou *context-awarness* en anglais.

La sensibilité au contexte est la capacité à pouvoir adapter son comportement, ou son exécution dans le cas d'une application, en fonction des informations de l'environnement. Le terme environnement est à comprendre au sens large. Il inclut toutes les informations qui peuvent présenter un intérêt pour une application comme par exemple des grandeurs physiques (température, luminosité, pression, ...), la localisation des équipements ou des utilisateurs, l'activité des utilisateurs, la présence ou non de certaines ressources. Une des approches les plus populaires consiste à déléguer à l'infrastructure d'exécution le soin de collecter, modéliser, enrichir et distribuer ces informations par l'intermédiaire d'un *middleware* spécialisé. Compte tenu de la diversité des infrastructures d'exécution et des besoins et contraintes des applications

s'y exécutant, nous estimons que ce *middleware* doit proposer différentes façons de représenter, interagir, construire et mettre à jour les informations de contexte selon l'infrastructure où il est déployé.

Dans ce travail, nous nous focaliserons sur le niveau *Fog Computing* où le contexte est utilisé dans principalement deux buts. Premièrement, il est utilisé pour implémenter des services déclenchants des actions, en réaction ou en prévention de changement d'informations, avec des contraintes temps réelles. Ensuite, une partie des informations collectées par la plateforme est agrégée, filtrée et synthétisée pour être envoyée au niveau *Cloud* dans le but d'effectuer des analyses sur le long terme. Une solution populaire pour la construction de ce niveau d'infrastructure est d'utiliser l'approche à service. Celle-ci sert de base à la gestion du dynamisme et permet de masquer en partie l'hétérogénéité des ressources aux applications. Cependant, cette complexité ne disparait pas mais doit être intégrée par le module de contexte et ses développeurs.

Nous avons donc pu, à travers nos travaux et la littérature, constater certains éléments sur la construction et l'exécution d'un module de contexte d'une plateforme *Fog Computing*. La construction de celui-ci est particulièrement complexe au regard du dynamisme et de l'hétérogénéité des ressources à intégrer. De plus, l'ajout de nouveaux fournisseurs ou de nouveaux besoins de contexte est rarement facilité par la plateforme dû aux manques d'extensibilité des solutions. A l'exécution, celui-ci nécessite un haut niveau d'autonomie au regard de l'absence physique d'administrateur suffisamment qualifié. Ce travail s'attache donc à fournir une assistance au développement, à l'exécution et à l'administration d'un tel module.

Contribution

Nos travaux ont pour visée de faciliter le développement, l'exécution et l'administration d'un module de contexte dans une infrastructure *Fog Computing*. Dans notre vision, le contexte est modélisé comme un ensemble de service. Ces services sont programmés grâce au support d'un modèle à composant spécialisé. A l'exécution, ils sont incorporés dans un cadre architecturale plus vaste permettant leur introspection et leur administration à l'aide d'un manager autonomique.

Notre solution repose en particulier sur :

- Un cadre de développement permettant de modéliser le contexte comme un ensemble de spécifications de service.
- Un support au développement permettant d'implémenter de façon simple et homogène les spécifications de services de contexte. Ce support prend la forme d'un modèle à composant spécialisé dans le développement d'un module de contexte. Il permet d'exprimer de manière concise la logique de synchronisation des composants, de faciliter la conception de nouveaux fournisseurs ou l'intégration de nouveaux besoins par un mécanisme de composition en plus de reprendre tous les avantages d'un modèle à composant orienté service.
- Un environnement d'exécution assurant l'exécution, l'introspection et la reconfiguration des instances de composant de contexte par l'intermédiaire d'un manager autonomique. Les services implémentés sont exposés dynamiquement selon les besoins des applications et des ressources présentes dans l'environnement.

L'implémentation de référence de notre framework, nommé *CReAM*, est disponible sous Apache Liscense 2 sur le site github. Il est aujourd'hui utilisé dans un projet collaboratif entre Orange Labs et l'équipe Adele. Ce projet prend principalement la forme d'une passerelle domotique nommé iCASA.

Organisation du manuscrit

La suite du document est structurée en deux parties majeures, la première s'articulant autour de l'état de l'art tandis que la seconde présente la contribution de cette thèse.

L'état de l'art inclut deux chapitres :

Le Chapitre 1 présente l'informatique pervasive et les différents concepts qui lui sont rattachés. Il examine entres autres les challenges apportés par cette vision de l'informatique, dont certains impactent grandement ces travaux.

Le Chapitre 2 propose l'étude approfondie d'une des grandes caractéristiques des applications pervasives : la sensibilité au contexte. Il examine plus particulièrement les besoins en contexte des applications s'exécutant sur des plateformes *Fog Computing*.

La contribution inclut trois chapitres :

Le Chapitre 3 définit la problématique de cette thèse et y associe une stratégie de recherche. Il donne ensuite une vision globale de notre approche. Celle-ci propose un modèle à composant spécialisé dans la construction d'un module à incorporer dans un cadre architectural plus vaste permettant la mise en place de comportement autonomique.

Le Chapitre 4 expose une vision détaillée de notre approche en se concentrant sur les différents aspects de celle-ci évoqués lors du chapitre précédent.

Le Chapitre 5 présente l'implémentation de référence de notre approche et lui associe une validation permettant de tester la viabilité de notre approche.

Le **Chapitre 6** sert de conclusion aux présents travaux. Il en résume les concepts et idées principaux et propose des pistes pour des travaux ultérieurs.

Chapter 1

Informatique Pervasive

Sommaire

Dans ce premier chapitre, notre but est de présenter le domaine de l'informatique pervasive aussi nommé informatique ubiquitaire. Ces deux termes seront utilisés de façon interchangeable par la suite.

Dans un premier temps, les définitions et concepts clés de ce domaine seront établis. Nous approfondirons ensuite l'étude de ce domaine par l'analyse des deux éléments qui le composent. Pour ce faire, nous étudierons dans en premier lieu l'environnement pervasif et ses caractéristiques. Dans un second temps, nous étudierons les applications pervasives en axant notre présentation sur leurs besoins. Nous présenterons dans un dernier point les ébauches de solutions mises en place pour faciliter la construction de ces applications.

Ce premier chapitre s'est largement inspiré des précédents travaux de l'équipe ADELE dans le domaine de l'informatique pervasive par l'intermédiaire de [Bou14] et [Gun14].

1.1 Introduction

L'informatique pervasive ou ubiquitaire n'est pas en soi une vision disruptive de l'informatique mais s'inscrit plutôt dans la continuité de l'évolution des environnements informatiques. Un rappel de l'évolution de ces environnements et les premières définitions de l'informatique pervasive serviront d'introduction à ce chapitre.

1.1.1 Evolution des environnements informatiques

Notre rapport à la technologie, et plus précisément aux systèmes informatiques n'a eu de cesse d'évoluer. Actuellement, nos environnements, domestiques ou professionnels, voient proliférer un nombre toujours croissant d'appareils numériques qui se distinguent par leur forme, leur puissance et leur rôle. Ainsi les appareils du quotidien se sont transformés, pour devenir toujours plus petits, puissants, communicants, et pour consommer moins d'énergie. La démocratisation des technologies d'accès au réseau (par exemple : Wifi, 4G, LWPA), la conjecture de Moore [Moo98] sont autant de facteurs ayant pu permettre cette évolution. Les évolutions sont aussi à constater dans nos modes d'interactions avec ces systèmes : le traditionnel « clavier-souris » laissant peu à peu place à des modes d'interaction tactiles, gestuels, vocaux ou basés sur des mesures biométriques. Cette évolution peut paraitre surprenante si l'on considère que cinquante ans à peine nous séparent du remplacement des machines analogiques industrielles par des machines numériques. Une analyse de l'évolution des systèmes, basée sur [WB96] , va ainsi permettre de mieux saisir le contexte global de ces travaux.

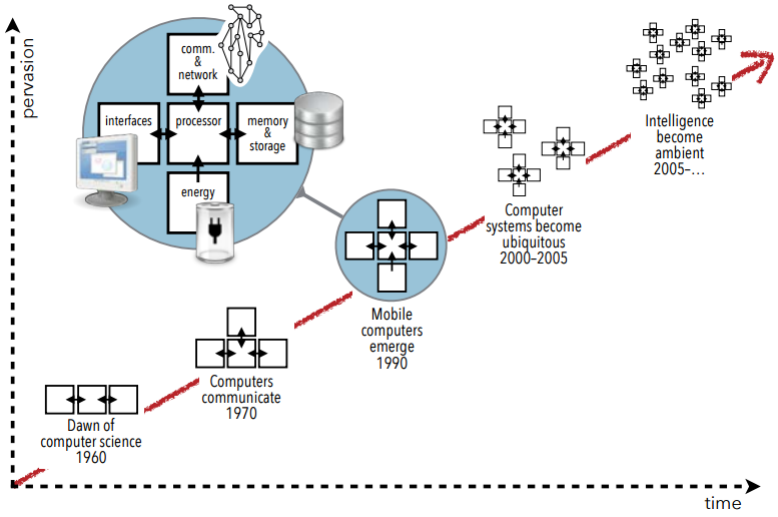

Figure 1.1: Evolution des environnements informatiques [Wal07]

La figure 1.1 détaille les différents paradigmes technologiques liés à l'informatique. Elle met en exergue les différentes notions qui l'ont fait évoluer, les plus notables étant les notions de mobilité, distribution et multiplication des systèmes informatiques.

La décennie 1936-1946 est décisive pour l'histoire de l'informatique par l'apport de la thèse de Church [Chu36] et celle de Turing [Tur38] qui déboucheront sur le modèle de Von Neumann avec la séparation mémoire unité-centrale et celle calcul-séquencement. Les systèmes informatiques sont alors désignés par le terme calculateur et sont l'apanage des universités qui les construisent et les utilisent pour effectuer des calculs scientifiques. Ces efforts donnent naissance en 1951 à l'industrie informatique qui s'ouvre au traitement des données (data processing) et plus seulement au calcul scientifique. A cette époque, la vision d'une informatique centralisée comme seul moyen de construire des systèmes numériques domine. Ces derniers prennent la forme d'ordinateurs isolés, centralisant données et traitement, et étant partagés par plusieurs utilisateurs. Ces personnes sont à la fois administrateurs, développeurs et utilisateurs du matériel et du logiciel. Chacune de ces tâches requiert un haut niveau d'expertise. Un mainframe, nom donné à ces systèmes centralisés, est caractérisé par une utilisation partagée entre plusieurs utilisateurs et des ressources limitées. Le terme mainframe n'a pas disparu mais a évolué. De nos jours on l'attache aux ordinateurs « haut de gamme » dotés d'une forte puissance de calcul et supportant des applications utilisées simultanément par des milliers de personnes. On retrouve cette définition dans [WB96] :

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

La deuxième phase de l'informatique est liée à l'avènement de l'ordinateur personnel ou PC (*personal computer*). Cette deuxième phase coïncide avec l'arrivée de la technologie du microprocesseur en 1970. Ces ordinateurs personnels, bien que, dans un premier temps, réservés à un public de passionnés, vont profiter des avancées logicielles avec l'apparition de « vraies » applications (traitement de texte, tableur, jeux) pour se démocratiser. Cette tendance est renforcée en 1984 avec l'apparition des stations de travail. On notera durant cette période le déclin progressif des mainframes. Marc Weiser [WB96] introduit cette émergence de la manière suivante :

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

L'informatique personnelle se caractérise par le fait que l'utilisateur occupe le rôle central dans sa relation avec le monde numérique. Le système étant personnel, l'utilisateur est aussi amené à jouer le rôle d'administrateur pour s'occuper des tâches d'installation et de configuration des différentes extensions logicielles et matérielles conçues par des tiers. Tout comme les mainframes, les PC disposent de ressources locales limitées. Mais les infrastructures réseaux

telles que l'Internet permettent de palier à cela, en offrant la possibilité de se connecter à des services et données à distance. Cela permet de lier système d'informations personnelles et professionnelles. Pour citer Mark Weiser [WB96] une nouvelle fois :

« 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. »

Cet accès à des services distants permet le développement d'applications plus complexes avec comme finalité une plus forte valeur ajoutée pour l'utilisateur final. Le fait que les ressources ne soient plus seulement locales mais dispersées à travers le réseau a donné le qualificatif distribué à ces systèmes. De nouveaux défis pour les développeurs et administrateurs sont associés à ce paradigme, tel que la prise en compte de la sécurité du système, la communication à distance ou l'intégration d'applications hétérogènes.

Les années 1990 marquent le début des *systèmes dit mobiles*. L'émergence des ordinateurs portables, donc plus transportables, et la mise en avant des technologies de communication sans fil (Bluetooth, Wifi, GSM) ont permis à l'utilisateur de pouvoir se déplacer avec son ordinateur et de toujours profiter des services distants. Ce sentiment de mobilité est de nos jours accentué par la prolifération des smartphones et tablettes tactiles. Ces nouveaux appareils permettent le développement de services encore plus centrés sur l'utilisateur. Par exemple, les applications de cartographie centrent automatiquement la carte affichée sur la position actuelle de l'utilisateur. La position de l'utilisateur peut être relevée grâce à un signal GPS ou par la triangulation des signaux du réseau cellulaire. Au défi des systèmes distribués viennent s'ajouter de nouveaux challenges : l'économie d'énergie, la sensibilité à la position géographique et la gestion efficace et adaptative des ressources [Sat01].

L'informatique ubiquitaire ou pervasive est la dernière évolution abordée par la figure 1.1. Celle-ci donne naissance à des applications qui se doivent d'être invisibles et qui requièrent un minimum d'attention de la part de l'utilisateur. Ces applications délivrent à l'utilisateur des services personnalisés à forte valeur ajoutée en se basant sur les informations issues de l'environnement de celui-ci [Mat01; AIM10] . Pour cela, cette vision nécessite un mode d'interaction radicalement différent entre systèmes informatiques et utilisateurs. Les systèmes informatiques se fondent dans les objets de la vie de tous les jours et les utilisateurs les exploitent de manière transparente pour accéder à des services ou à des données.

1.1.2 Définitions de l'informatique pervasive

Mark Weiser, dans [Wei91], dépeint la vision de l'informatique pervasive, aussi appelée informatique ubiquitaire. Cette vision détaille un monde où l'informatique est omniprésente et

centrée sur l'utilisateur et surtout transparente pour celui-ci. Ainsi, cette vision décrit des environnements remplis de dispositifs informatiques miniaturisés et intégrés dans les objets du quotidien couplés à des infrastructures de communications. Ce couplage est réalisé dans le but de délivrer des services à forte valeur ajoutée, de la manière la plus transparente, naturelle et efficace possible. Ces visions ont inspiré de nombreux chercheurs, ce qui a amené à l'apparition de différents termes tels que « calm computing », « disappearing computer », « everyware » [Gre10] , « Internet of things » [Mat05] , « ambient intelligence » [Han+03] et « things that think » [HPT97]. Malgré la multiplication des qualificatifs pour ces environnements, tous se rattachent à une vision commune où environnement informatique et monde réel fusionnent [Ron09]. Dans ces travaux, le terme informatique pervasive pourra se substituer indistinctement aux différents qualificatifs cités précédemment.

La vision initiale de Weiser se plaçant à un haut niveau d'abstraction et ne se focalisant sur aucun point technologique particulier, la littérature a vu fleurir plusieurs définitions tentant de la raffiner:

« 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]

« 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]

Les définitions précédentes permettent d'extraire un ensemble de propriétés clefs servant à caractériser l'informatique pervasive :

- L'informatique pervasive est **invisible**. Cette propriété met en relief le fait que les interactions homme machine doivent rester transparentes et naturelles.
- L'informatique pervasive est **distribuée** entre les équipements de l'environnement pervasif, qu'ils soient mobiles ou fixes, et les services du réseau.
- L'informatique pervasive est **sensible au contexte**. Cette propriété dénote le fait que les applications sont capables en fonction de l'état de l'environnement, physique, virtuel ou utilisateur, d'adapter leurs fonctionnalités et leur exécution.

Ces propriétés fondamentales révèlent en partie ce qu'est la vision de l'informatique pervasive. Cependant, pour avoir un aperçu plus profond des challenges que recèle l'informatique pervasive, nous étudierons en détail ses composantes : l'environnement pervasif (voir section 1.2) et les applications pervasives (voir section 1.3) . Ceci permettra de ressortir des caractéristiques communes aux systèmes pervasifs.

1.2 Environnements pervasifs

L'informatique pervasive est plus une vision de l'évolution de l'informatique qu'un tout nouveau domaine comme vu dans la section précédente 1.1.1. Même si une définition unificatrice de l'informatique pervasive est dure/difficile à trouver, il apparait dans la littérature qu'un ensemble commun de propriétés et de caractéristiques permet de définir ce qu'est un environnement pervasif. Il est donc normal que les caractéristiques des environnements pervasifs résultent de la convergence des précédents paradigmes de l'informatique, informatique distribuée et informatique mobile, comme dénoté par la figure 1.2.

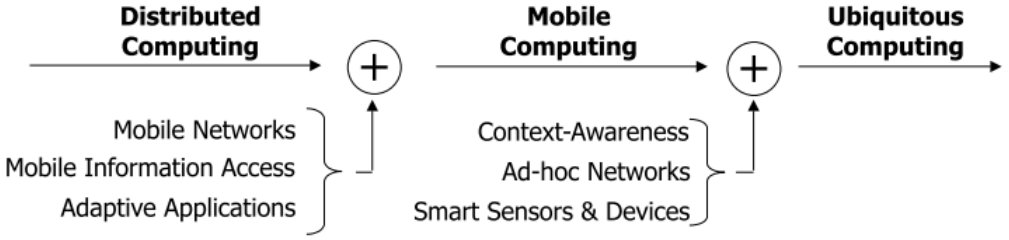

Figure 1.2: Les problématiques de recherche de l'informatique pervasive [SL04]

Cette section détaille les différentes propriétés des environnements pervasifs.

1.2.1 Ouverture

L'environnement pervasif fait cohabiter les différentes ressources informatiques, qui peuvent prendre la forme d'équipements ou de services. Ces ressources sont fabriquées, développées et maintenues par des acteurs différents allant du vendeur d'équipement au fournisseur de service. Pour pouvoir se développer et s'exécuter dans de tels environnements, les applications pervasives doivent pouvoir accéder et interagir avec les différentes fonctionnalités de ces ressources. Cela est possible uniquement dans un environnement qu'on peut qualifier d'ouvert, c'est-à-dire où les ressources sont conçues explicitement pour être accessibles depuis d'autres ressources. On parlera aussi d'interopérabilité des ressources.

Plusieurs raisons font que de nombreux systèmes sont aujourd'hui conçus de manière à limiter l'interopérabilité et restreindre l'ouverture. Une liste non-exhaustive de ces raisons est présentée ci-dessous :

Des raisons économiques un fournisseur d'équipements peut ignorer l'interopérabilité de ces équipements pour préserver sa mainmise sur le développement applicatif se protégeant ainsi du développement de services par des concurrents.

Des raisons liées à la sécurité dans le cas de dispositifs particulièrement sensibles, les constructeurs ne peuvent donner accès à leur matériel par rapport à la législation en vigueur. On peut prendre le cas des systèmes de sécurité liés à l'ouverture des portes où la législation française impose que tout le système fonctionne en vase clos .

Des raisons liées à la vie privée dans le cas où des ressources interagissent avec des données liées à la vie privée de l'utilisateur, des restrictions quant à leur accès peuvent être mises en place pour préserver celles-ci.

Des solutions existent pour garantir aux fournisseurs de ces ressources de garder un niveau de contrôle sur leur produit tout en contribuant à l'environnement ouvert avec des fonctionnalités disponibles publiquement. Les applications peuvent avoir différents niveaux d'accès sur les ressources et leurs fonctionnalités associées. Ainsi une application certifiée par le vendeur d'un dispositif peut avoir un accès complet au dispositif, tandis que les applications non-certifiées n'auront qu'un accès partiel aux fonctionnalités proposées.

Le fait de se trouver dans un environnement où plusieurs applications s'exécutent en même temps et partagent des dispositifs lève aussi d'autres défis. La gestion d'accès avec différentes accréditations, le contrôle d'accès ou la charge de la maintenance d'un tel environnement restent des tâches extrêmement complexes.

1.2.2 Hétérogénéité

Comme vu précédemment (voir section 1.2.1) l'environnement pervasif peut être qualifié d'ouvert, c'est-à-dire que chaque ressource est potentiellement fournie par des acteurs totalement différents. Chaque ressource est conçue par un acteur avec un savoir-faire propre et possédant des besoins différents liés à son utilisation en termes de sécurité, débit, portée de signal, consommation énergétique, chacune de ces ressources étant différentes en termes de fonctionnalité, mémoire, puissance de calcul. Ces différences apportent une forte hétérogénéité dans les domaines suivants :

Technologie et protocole de communication : Les protocoles de communication utilisés sont extrêmement variés dans les environnements pervasifs. Ils peuvent aussi bien s'appuyer sur une couche physique filaire ou sans-fil. Le choix du protocole fourni est déterminé par les besoins liés à l'utilisation de la ressource et du savoir-faire du fournisseur. Par exemple, des équipements nécessitant une faible consommation énergétique et une longue portée de signal se tourneront vers des technologies sans-fil LPWA (Low Power Wide Area) tel Sigfox ou LoRa. Des équipements branchés sur l'alimentation secteur peuvent bénéficier de la technologie CPL G3, comme le compteur Linky de EdF (Electricité de France) : société fondatrice de l'alliance CPL G3 [1]. Cette tendance est aussi vraie pour les services distants accessibles par Internet. Cela requiert d'intégrer différents modèles de services et protocoles de communication.

[1] consortium pour promouvoir le protocole CPL G3 : `http://www.g3-plc.com/home/`

Format des données : Le format des données échangées peut varier d'une ressource à une autre. Cela est particulièrement vrai pour les capteurs de données environnementales qui peuvent remonter des valeurs décrivant le même paramètre mais dans des unités différentes. Ce problème est largement abordé dans les travaux sur la médiation avec les alignements sémantiques [Wie92].

Configuration et administration : Il est nécessaire de pouvoir configurer et administrer les différentes ressources de l'environnement, mais les fournisseurs de ressources sont susceptibles d'utiliser des modèles de management de services ou d'équipements différents.

Des solutions sont mises en place pour tenter de palier à cette hétérogénéité. On retrouve de nombreux groupes de travail qui proposent de standardiser les protocoles qui ont un usage commun avec par exemple CPL G3, Bluetooth, Zigbee, Device Profil For Web User (DPWS). Mais certains industriels préfèrent mettre en place des solutions propriétaires et non-ouvertes à la place des solutions standardisées et souvent plus généralistes pour les raisons économiques évoquées précédemment. Des travaux sont aussi menés sur le format des données à travers l'utilisation de middleware de médiation [Gar12] ou de bibliothèques dédiées [2].

Les concepteurs d'applications ont toujours besoin d'intégrer de nouveaux protocoles de communication, types d'équipement et services distants dans leurs applications. Cela mène inévitablement à augmenter la complexité de telles applications si les solutions mises en place sont ad-hoc. Même si certaines plateformes supportent un nombre limité de protocoles tels que [Hel+05] ou [Kin+06], des solutions sont mises en place pour supporter un nombre extensible de protocoles. Ces protocoles peuvent être déployés selon les besoins des applications [Bar12] .

1.2.3 Distribution

La mise en œuvre des environnements pervasifs repose sur la coopération d'équipements intelligents et communicants disséminés dans l'environnement physique et de services distants au travers d'applications. Les dispositifs présents physiquement dans l'environnement pervasif, tels que les capteurs environnementaux, tablettes, téléphones portables ou actionneurs, permettent un suivi de ce même environnement et des préférences utilisateurs. Ces dispositifs diffèrent en termes de puissance de calcul, capacité de stockage et technologie de communication comme vu précédemment. Les applications qui utilisent et font coopérer ces différentes ressources ne s'exécutent pas obligatoirement directement à l'intérieur de ces mêmes ressources à cause de ces différences. Ces environnements sont donc inévitablement distribués.

En règle générale, plus les ressources sont éloignées de l'environnement physique plus les capacités en termes de puissance et stockage sont importantes et moins coûteuses à augmenter

[2] Par exemple, en java la JSR 363 et son implémentation : `https://github.com/unitsofmeasurement/unit-ri`

selon les besoins. En contrepartie, plus les ressources sont proches de l'environnement physique plus celles-ci disposent d'un pouvoir d'action en temps réel sur celui-ci. Une des principales problématiques des développeurs de logiciels est de savoir à quel niveau déployer quelle partie de leur logiciel, sous condition que le logiciel soit modulaire, pour tirer parti au mieux des contraintes et bénéfices de l'emplacement de la ressource physique sous-jacente. Par exemple dans des applications *M2M* (en anglais : *machine to machine*), des mesures environnementales sont collectées par une passerelle domotique à travers des réseaux de capteurs pour ensuite passer par plusieurs passerelles de médiation servant à filtrer et agréger ces données. Elles sont ensuite envoyées à un serveur distant se chargeant des opérations de stockage et de traitement nécessitant une importante puissance de calcul. Parallèlement à cela, des décisions et des actions peuvent déjà être prises au niveau de la passerelle domotique pour affecter l'environnement en temps réel.

Cette caractéristique apporte son lot de défis : problématiques de déploiement à grande échelle, d'installation et de maintenance des ressources logicielles et matérielles. Les infrastructures logicielles et matérielles doivent pouvoir supporter la montée en charge liée à l'intégration d'un nombre toujours croissant d'équipements, de services distants et de données produites. De leur côté les développeurs de logiciels voient leur tâche se complexifier avec la prise en compte de la nature distribuée des applications, des services et des équipements distants.

1.2.4 Dynamisme

L'environnement informatique est par nature changeant, c'est-à-dire qu'il s'y produit de nombreuses évolutions, souvent imprévues, durant l'exécution. L'hypothèse, pour le développement d'un logiciel, qui consistait à dire que le périmètre logiciel est sous contrôle et qu'il n'est pas impacté par des évolutions externes est fausse. Les ressources de l'environnement évoluent de façon incontrôlée. Elles disparaissent et apparaissent au gré de l'exécution. Les applications peuvent en conséquence ne pas trouver toutes les ressources prévues lors de leur conception. Elles vont ainsi être inévitablement impactées si ce dynamisme n'est pas pris en compte. On peut faire le parallèle avec le passage à l'échelle anarchique de [FT00] , plus axé application distribuée et réseau, qui fait référence à la nécessité de développer des applications capables de fonctionner alors qu'elles sont sujettes à des pertes de communications, possibilité de nouvelles communications, de charges discontinues ou surélevées, etc.

Dans un environnement pervasif, les sources de dynamisme peuvent appartenir à une des catégories suivantes :

Volatilité des ressources : Chaque ressource a un cycle de vie qui lui est propre et celui-ci ne peut pas toujours être contrôlé. Ainsi elle peut être amenée à disparaitre et apparaitre durant l'exécution. Pour les équipements physiques, les disparitions proviennent par exemple d'une panne de batterie, d'une rupture ou interférence de communication ou de défaillance de l'électronique. Pour les services distants, des exemples de cause

de disparition peuvent être la maintenance ou mise à jour d'un serveur ou une rupture de communication. Des ressources ne supportant pas d'accès concurrents peuvent aussi voir leur disponibilité affectée. L'apparition de nouvelles ressources peut faire suite à la réparation d'un des exemples précédemment cités, mais aussi lors du déploiement de nouveaux services distants, l'achat et la mise en fonctionnement de nouveaux équipements sur le réseau. Sur une même ressource matérielle, le déploiement de nouvelles applications peut venir impacter le cycle de vie de celles déjà présentes.

Interaction de l'utilisateur : Une des particularités de la vision de l'informatique pervasive est que celle-ci est centrée sur l'utilisateur. L'environnement pervasif incorpore donc ces utilisateurs, qui à travers leurs actions affectent celui-ci. Une des sources de dynamisme est la mobilité de l'utilisateur, et des équipements qu'il transporte, tel que les smartphones ou les ordinateurs portables. La connectivité aux réseaux et aux équipements va donc dépendre de la localisation de ces équipements. Les interactions de l'utilisateur permettent aussi de récolter des données indiquant ses actions ou ses intentions. Les applications pervasives sont particulièrement sensibles à ces changements pour adapter leur fonctionnement.

Ces sources de dynamisme supplémentaires, en plus de la dérive naturelle de l'environnement, ne permettent pas de produire des applications avec un ensemble prédéfini de ressources.

1.2.5 Autonomie

Une des caractéristiques de la vision de l'informatique pervasive est que l'interaction entre l'utilisateur et l'environnement physique, augmenté de ressources informatiques, s'effectue de manière transparente et naturelle. On précisera que l'utilisateur appartient à un public non-expert et que ses connaissances en informatique sont relativement peu élevées. Il est donc peu probable que celui-ci puisse faire face seul à la complexité engendrée par la superposition du monde physique et du monde numérique. Le voir tenir à la fois le rôle d'administrateur et d'utilisateur semble peu probable. Pour cela, l'environnement pervasif se doit d'être autonome. Il doit ainsi pouvoir décharger l'utilisateur de certaines tâches pour que celui-ci ne soit plus préoccupé par la complexité sous-jacente d'un tel environnement.

1.3 Applications pervasives

Une application est un logiciel qui effectue des tâches spécifiques pour les utilisateurs. Généralement, celles-ci ne sont pas développées et installées en partant de zéro mais bénéficient de plusieurs couches logicielles facilitant la prise en compte de certaines préoccupations transversales. Traditionnellement, les applications sont installées au-dessus d'un système d'exploitation qui permet d'exploiter et contrôler l'accès aux ressources physiques. Des couches logicielles supplémentaires, appelées middleware, peuvent être insérées entre le système d'exploitation et les applications pour offrir des facilités de développement et d'exécution supplémentaires.

Comme évoqué précédemment (voir section 1.2) , l'environnement pervasif impose des contraintes qui doivent être considérées lors du développement et de l'exécution des applications qui y résident. La production de ces applications se retrouve complexifiée par différents besoins qui doivent être pris en compte au plus tôt. Dans un premier temps, nous décrirons plus en détail le processus de production d'une application. Ensuite, nous soulignerons les propriétés clés qui différencient les applications pervasives des applications traditionnelles et qui doivent pouvoir s'intégrer dans leur cycle de production.

1.3.1 Cycle de vie du développement des applications

Les avancées en génie logiciel ont fondamentalement changé la façon dont les logiciels sont produits. Précédemment, le cycle de vie, ou de production d'une application pouvait se résumer en deux phases : analyse des besoins et développement. Mais ce cycle de vie est devenu plus rigoureux, plus méthodologique, avec l'ajout de différentes étapes dotées de caractéristiques propres. Chaque étape nécessite un ensemble de compétences spécifiques, apportées par différents acteurs. Les applications pervasives utilisent ces avancées éprouvées par l'industrie pour augmenter leur qualité, leur robustesse et mieux maitriser les coûts liés à leur production. On utilisera le standard IEEE [IEE08], qui définit une liste exhaustive des tâches élémentaires à effectuer durant tout le cycle de vie de la production d'un logiciel, pour évoquer les différentes étapes de la production d'une application pervasive :

Analyse des besoins : la phase d'analyse des besoins permet de définir quels sont les objectifs du projet. Elle identifie les différentes parties prenantes qui vont intervenir durant le cycle de vie du logiciel. On identifie plus spécifiquement leurs besoins et leurs demandes. Ceux-ci vont être réduits à un ensemble d'exigences, qui a pour but d'exprimer les fonctionnalités du système. Ces exigences vont ensuite être transformées en exigences techniques qui vont servir de ligne de conduite pour les phases suivantes.

Conception : la phase de conception a pour but d'identifier et d'établir l'architecture du logiciel en concordance avec les précédentes exigences. Le logiciel est alors découpé en différents éléments, où chaque élément se voit attribuer des spécifications en termes

de fonctionnalités attendues. Les relations entre les différentes parties sont elles aussi spécifiées. Le choix d'une architecture adaptée aux besoins est déterminant pour la suite car celle-ci conditionne grandement la bonne réalisation du projet (respect des coûts et des délais) que les possibilités de maintenance futures.

Développement : La phase de développement a pour but la réalisation d'un des éléments de l'architecture du logiciel. Cela consiste principalement en une phase de programmation. Chaque élément du logiciel peut être développé en parallèle et indépendamment du reste de l'architecture. Chaque élément doit être conforme aux spécifications établies lors de la phase précédente.

Intégration : La phase d'intégration a pour but d'intégrer les différents éléments du système (éléments logiciels et matériels, systèmes appartenant à une tierce partie, etc...) pour produire un système complet qui satisfait les attentes des utilisateurs et des spécifications du système. A la fin de cette phase d'intégration, le logiciel est prêt à être testé dans son ensemble.

Test : La phase de test est transversale aux phases précédemment évoquées. Celle-ci a pour but de garantir la qualité du logiciel, la conformité du logiciel par rapports aux besoins ou la non-régression du logiciel. Les tests unitaires vérifient que l'implémentation de chaque élément du logiciel est conforme aux spécifications. Les tests d'intégrations valident le fonctionnement de l'assemblage des différents composants et des tierces parties.

Installation : La phase d'installation a pour but de placer le logiciel dans un environnement cible et de venir le configurer pour que celui-ci puisse fonctionner de manière nominale.

Exécution : La phase d'exécution représente le fonctionnement nominal du logiciel. Cette phase est instable car le logiciel peut stopper son exécution ou nécessiter des évolutions. Ces causes d'instabilités doivent être traitées en parallèle de l'exécution du logiciel, dans la phase de maintenance.

Maintenance : La phase de maintenance a pour but de conserver le logiciel dans un état de fonctionnement optimal. Le logiciel est amené à subir des pannes ou à évoluer. Cela peut être dû à des erreurs de développement non détectées par les tests ou des cas d'utilisation non couverts. Une fois la panne détectée, des actions doivent être entreprises par une équipe de maintenance pour proposer un correctif ou une mise à jour du logiciel.

Destruction : La phase de destruction a pour but de mettre fin à l'existence du logiciel. Durant cette phase, l'exécution et la maintenance du logiciel sont stoppées. Les différents éléments du logiciel sont ensuite effacés de l'environnement où celui-ci a été installé. A la fin de cette phase, l'environnement où le logiciel a été installé doit être dans un état jugé acceptable, cela étant défini par les exigences et accords établis dans les phases précédentes.

Un point important est que le poste principal de dépense sur la production d'un logiciel est la phase de maintenance [Leh80] . Cela est principalement lié à la durée de vie des logiciels

qui peut couvrir plusieurs décennies. Cette phase de maintenance est encore alourdie pour les applications s'exécutant dans des environnements pervasifs. Comme vu précédemment (voir section 1.2.4 et 1.2.2), la dynamique et l'hétérogénéité de ces environnements couplés à l'absence d'administrateur expert complexifient lourdement cette tâche. Les besoins utilisateurs, le matériel et les logiciels présents dans l'environnement ne sont pas figés et leur évolution est difficilement prévisible. Cela conduit à réévaluer ces besoins en permanence et adapter le logiciel continuellement à ces nouveaux besoins.

Un cycle de vie réactif, c'est-à-dire permettant des itérations successives rapides, est une des nécessités pour encadrer la production d'une telle application. Mais devant cette complexité grandissante, d'autres outils, plus élaborés, peuvent intégrer ce cycle de vie dans l'optique de faciliter la production d'applications pervasives et d'améliorer leur qualité. La forme de ces outils est détaillée par la suite (voir section 1.4).

1.3.2 Caractéristiques

L'environnement pervasif apporte son lot de contraintes comme vu précédemment. Une étude des fonctionnalités favorisant une production fiable et facilitée des applications pervasives permettra une meilleure compréhension des techniques et méthodes présentées par la suite. On trouve donc dans cette partie une liste de propriétés qui permettent de distinguer les applications pervasives des applications traditionnelles. Des travaux ont déjà été effectué sur le sujet tel que [Ban+00].

1.3.2.1 Gestion des ressources

Les applications traditionnelles sont conçues pour fonctionner avec un ensemble immuable de ressources informatiques définies à la conception. Elles sont exécutées localement sur une machine ou distribuées entre plusieurs machines, mais elles restent limitées par les ressources fournies par ces machines . C'est une vision où l'application se centre et s'attache à un dispositif en particulier. Mais l'avènement des dispositifs mobiles et personnels change ce paradigme en proposant une vision de plus en plus centrée sur l'utilisateur. Les applications pervasives adoptent largement ce nouveau paradigme, se focalisant sur les besoins de l'utilisateur ou sa position géographique par exemple. Pour mener à bien leurs tâches, celles-ci ont donc besoin de découvrir et utiliser opportunément des ressources hétérogènes, telles que des services distants ou des équipements.

Une ressource est définie selon [Kra07] par :

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

L'espace disque, les processeurs, ou la bande passante sont des exemples de ressources physiques. Souvent ces ressources physiques sont abstraites, soit par le système d'exploitation ou/et des couches logicielles supplémentaires, par des ressources virtuelles. Le temps processeur est abstrait par les *threads* ou processus, la mémoire physique par la mémoire virtuelle. Cette abstraction est un des points clés de l'utilisation des ressources. En plus de faciliter leurs utilisations, elle peut fournir des caractéristiques non-fonctionnelles. Par exemple, le système de fichier, qui abstrait la mémoire, incorpore aussi des fonctionnalités de gestion des permissions et de gestion d'accès.

Toujours selon [Kra07] , une ressource est définie par un ensemble de caractéristiques conditionnant son utilisation :

Exclusive ou partagée : une ressource peut être utilisée par seulement une application ou simultanément par plusieurs applications. Le partage simultané de la ressource peut être décliné en plusieurs techniques : partage par un groupe d'application, partage en lecture seulement, etc...

Avec ou sans état : Dans le cas d'une utilisation avec état (*statefull*), la ressource associe un état à l'application qui l'utilise. Cela a pour conséquence que l'état de la ressource doit s'effacer quand l'utilisation cesse ou peut être stocké pour une réutilisation postérieure. Dans le cadre d'une utilisation de ressource sans état (*stateless*), la ressource possède un seul état, partagé par toutes les applications.

Individuelle ou groupée : une ressource peut être une instance unique, ou faire partie d'un groupe de ressources. Par exemple dans le cadre des techniques de *scheduling*, un *thread* est une ressource faisant partie du pool de *threads* du *scheduler*.

Une ressource est décrite par un descripteur de ressource. En plus de contenir les indications sur son utilisation, celui-ci peut contenir d'autres propriétés. La qualité de la ressource, exprimée en termes de performance, de sécurité de son utilisation ou de disponibilité, peut aider lors de sa sélection par un consommateur. Ces descripteurs peuvent aussi contenir les différentes configurations possibles de la ressource, affectant ainsi son comportement. Des propriétés liées au domaine du pervasif telles que la position géographique peuvent aussi être ajoutées.

Une grande partie des ressources de l'environnement pervasif est productrice de données, tels que les capteurs environnementaux. Les applications pervasives réagissent aussi bien aux changements de ces données que des méta-informations de qualité qui les caractérisent. Un schéma de programmation réagissant à ces changements doit être prévu par les solutions mises en place.

Les applications traditionnelles fonctionnent avec un ensemble prédéfini de ressources, souvent abstraites par le système d'exploitation ou par des couches logicielles intermédiaires.

Même si des politiques de d'administration de ces ressources existent [Ber96] , elles deviennent limitées quand l'ensemble des ressources est dynamique. Cet ensemble de ressources hétérogènes et dynamiques requiert de repenser les politiques d'autorisation d'accès, de configuration, de découverte et d'utilisation au sein des différentes couches logicielles.

1.3.2.2 Notion de contexte

Comme mentionné dans l'introduction de ce chapitre (voir section 1.1.2) , la sensibilité au contexte est une des propriétés fondamentales de l'informatique pervasive. De manière générale, on peut définir que les systèmes sensibles au contexte sont ceux qui sont conscients de leur environnement (physique virtuel, utilisateur etc...) et qui sont capables de s'adapter aux changements qui surviennent dans cet environnement [BDR07] . Les notions de contexte et de sensibilité au contexte sont au cœur de ce manuscrit et nous y reviendront de façon plus approfondie (voir chapitre 2).

1.3.2.3 Adaptabilité

Les applications pervasives ont des durées de vie longues et ont pour but d'effectuer une tâche à forte valeur ajoutée. L'exécution de cette tâche dépend du contexte comme décrit précédemment. Ces applications doivent donc adapter leur fonctionnement suivant diverses défaillances des ressources, la localisation, les capacités des dispositifs et services distants, etc... pour fournir un service continu à l'utilisateur.

Pour assurer son exécution sur des durées longues, l'application doit faire preuve de résilience. La résilience est la capacité à résister aux changements imprévisibles et aux conditions qui pourraient causer une perte de service [De 11][Mey09]. De plus, le service doit continuellement s'optimiser pour répondre aux changements de contexte et profiter au mieux de la découverte de nouvelles ressources. Les changements de l'environnement étant imprédictibles, l'application doit être suffisamment flexible pour être capable de se reconfigurer et ainsi changer son comportement. Cette adaptation doit être réalisée de manière autonome et transparente pour l'utilisateur.

Les applications traditionnelles sont développées avec une vue statique des besoins utilisateurs et des ressources nécessaires à leur exécution. Cependant, la dynamique de l'environnement fait que ce mode de développement n'est pas approprié [Hal+12] . La construction d'applications pervasives adaptables implique de laisser une grande part de variabilité dès la conception et que la résolution de celle-ci soit déléguée jusqu'à l'exécution, si possible. Les points de variabilités peuvent être nombreux comme la sélection de services, le déploiement, la configuration, etc... De plus lorsque plusieurs applications s'exécutent sur le même dispositif, un des scénarios est que celles-ci soient en concurrence pour l'utilisation des ressources. Ainsi une adaptation coordonnée entre les différentes applications sera plus

efficace qu'une adaptation locale à une application. D'autres critères dans l'adaptation des applications sont à prendre en compte comme l'impact sur la robustesse, la performance, la scalabilité des solutions, etc...

La conception et la mise en place d'une politique d'adaptation est donc très complexe et implique de nombreux paramètres. De plus, l'absence d'administrateur qualifié dans les environnements pervasifs accentue le besoin de proposer des solutions inspirées de l'informatique autonomique pouvant guider les adaptations dynamiques à l'exécution [KC03].

1.3.2.4 Sécurité et vie privé

Les applications pervasives utilisent des ressources qui sont en constante interaction avec l'utilisateur. Ces ressources sont donc par nature très sensibles au regard de la vie privée de l'utilisateur, car elles peuvent contenir des informations privées sur la vie de ceux-ci ou permettre de les déduire. Il faut donc que les solutions préservent au maximum la confidentialité de l'utilisateur. De plus, un des enjeux des solutions est de proposer des solutions offrant des politiques d'autorisation d'accès aux ressources et de contrôle de leur utilisation. Pour cela, les solutions peuvent utiliser plusieurs mécanismes tels que des protocoles d'authentification, d'autorisation et une gestion de compte utilisateurs.

Un des principaux défis soulevés par l'informatique pervasive est la reconnaissance de l'identité de l'utilisateur. Bien souvent, il n'est pas possible d'avoir cette information explicitement, à l'instar d'une page web où l'on rentrerait son mot de passe et son identifiant. Les applications doivent donc s'adapter avec de nouvelles sources d'authentification et gérer en permanence la sécurité des communications pour ne pas risquer de les corrompre.

1.4 Construire des applications pervasives

Les précédents points ont permis d'identifier les caractéristiques des environnements pervasifs ainsi que leur impact sur le cycle de vie et les besoins des applications qui s'y exécutent. Il en résulte que les applications pervasives sont particulièrement complexes à produire. De nombreux travaux ont été menés par la communauté scientifique pour réduire les difficultés liées à la conception, l'exécution et / ou la maintenance des applications pervasives.

Le besoin de support pour aider à la production de logiciel et réduire les risques liés à l'humain n'est pas nouveau. On en retrouve trace dès la période préindustrielle de l'informatique où l'on peut citer Maurice Vincent Wilkes [Wil85] :

« It was on one of my journeys between the EDSAC room and the punching equipment that "hesitating at the angles of stairs" the realization came over me with full force that a good part of the remainder of my life was going to be spent in finding errors in my own programs. »

Ce n'est que vers la fin des années soixante que nait la discipline du génie logiciel pour répondre à ces besoins. Cette discipline fait son apparition en réponse à la crise du logiciel et l'apparition des grands projets informatiques. A la fin des années soixante, le bilan des grands projets informatiques est catastrophique : les délais de livraisons ne sont pas tenus, les budgets explosent et le logiciel livré répond mal aux attentes des utilisateurs. Un des constats qui est dressé est qu'écrire des programmes corrects est complexe. La production de ceux-ci est effectuée sans assistance. Les conclusions tirées de ce constat sont qu'il faut développer des outils et des méthodes pour accompagner la production de logiciel, mieux former les parties prenantes des projets informatiques et passer la production à un stade industriel. Les méthodes développées par cette nouvelle branche de l'informatique ont permis de mettre fin à cette crise et d'offrir des moyens de produire des logiciels plus sûrs et utilisables de façon plus prédictible.

1.4.1 Support au développement

En génie logiciel, un moyen de simplifier et d'augmenter la qualité de la production d'applications est de fournir des outils pour les activités liées aux phases d'analyse des besoins, de conception, de développement, d'intégration et de test de l'application. Ces outils allègent la tâche des développeurs et réduisent les erreurs humaines.

Une partie de ces outils se focalise sur la simplification de l'activité de programmation liée au développement. Ils proposent en général un modèle de développement permettant d'abstraire certaines ressources telles que l'architecture matérielle, les protocoles de communication ou

le management de la mémoire. Ces outils prennent la forme de langages de programmation, compilateurs ou debuggers.

Une autre partie de ces outils se focalise sur l'aide à apporter autour de l'infrastructure, de la gestion ou de la communication entre les parties prenantes du projet. L'utilisation de ces outils ne produit en soi aucune ligne de programmation mais participe grandement à l'amélioration du développement ou de la maintenance du projet. Ces outils prennent la forme de systèmes de gestion de version, de gestion de projet, d'analyseur de code ou de suivi de *bugs* ou d'incidents.

La dernière partie des outils liés au développement apporte une aide plus globale au développeur et essaye de couvrir le maximum de phase du cycle de vie de l'application, parfois en allant jusqu'à l'exécution. Ces outils intègrent ceux décrits dans les paragraphes précédents au sein d'un même environnement de travail. Ces outils, qualifiés d'outils d'aide au génie logiciel assisté par ordinateur, permettent d'avoir une vue globale sur le projet tout au long des phases de son cycle de vie.

1.4.2 Support jusqu'à l'exécution

Les middlewares sont les solutions les plus répandues de support au développeur jusqu'à la phase d'exécution. Selon [Kra07] , le terme *middleware* est apparu dans les années 1990 et désigne de façon générique une couche logicielle intermédiaire se situant entre des ressources informatiques et des applications les utilisant. Le but de cette couche intermédiaire est de faciliter l'accès à ces ressources, ce qui par effet de bord simplifiera la conception, le développement et l'exécution d'une application.

Un *middleware* fournit un ensemble d'abstractions programmatiques, masquant la complexité du management des ressources informatiques. Les applications profitent donc du modèle de développement proposé par ces abstractions pour se construire. Cela permet de mutualiser des aspects techniques, souvent complexes et propices à l'erreur, pour éviter d'avoir à les intégrer dans chaque application développée. Les applications perçoivent le code fourni par le *middleware* comme non-fonctionnel, c'est-à-dire que celui-ci n'est pas en rapport direct avec la fonctionnalité business de l'application. D'un côté les développeurs profitent d'aspects techniques à moindre coût, seulement celui d'intégration du *middleware*, éliminant ainsi beaucoup de sources potentielles d'erreur. De l'autre cela leur permet de se concentrer sur le développement de la partie business de l'application, là où se trouve la plus forte valeur ajoutée d'un logiciel. Le champ d'action des *middlewares* s'est vu augmenté à mesure que l'informatique évoluait, et ceux-ci peuvent être utilisés pour gérer des préoccupations telles que la distribution, l'hétérogénéité des moyens de communication, la persistance, des opérations transactionnelles, le déploiement, le management ou la supervision des applications.

Les approches middleware sont les plus répandues pour répondre aux challenges de l'informatique pervasives car elles couvrent la majorité du cycle de vie des applications. SOCAM [GPZ05], Aura [Gar+02], Base [Bec+03] and PCOM [Bec+04], DiaSuite[CBC10], WComp [Tig+09], PerLa [Sch+12] sont des exemples de tels *middlewares*. Nous reviendrons par la suite sur une comparaison plus poussée d'une partie de ces différentes approches.

En plus des solutions spécifiques aux domaines de l'informatique, certaines plateformes d'exécution répondant à des préoccupations plus génériques sont utilisées dans le domaine du pervasif. On citera Fractal [Bru+04], Apache Felix iPOJO [Esc08], Kevoree [Fou+12] qui sont des approches orientées composants facilitant la construction de *frameworks* ou de *middlewares*.

1.5 Solution spécifiques pour l'informatique ubiquitaire

Nous allons détailler dans cette section des exemples concrets de solutions appliquées à l'informatique ubiquitaire. Même si la plupart de ces solutions se qualifient de *middleware*, elles entremêlent souvent support au développement et *middleware*, les deux approches se complétant de façon naturelle

1.5.1 Gaia et le langage Olympus

Gaia [Rom+02b; Rom+02a] se définit successivement comme une infrastructure *middleware* ou un meta-operating system facilitant la construction d'applications dans les environnements de type *active space*. Le terme *active space* est présenté comme une extension de l'environnement physique. L'environnement physique est ici une région géographique clairement délimitée (une pièce, un étage ou un bâtiment...) contenant des objets physiques (équipements physiques, connectés et hétérogènes) ainsi que des utilisateurs effectuant des activités. Un *active space* est un espace physique, où une infrastructure logicielle permet à l'utilisateur d'interagir de façon transparente avec son environnement physique ou digital. Gaia se positionne comme une couche d'abstraction au-dessus de l'environnement et les ressources qui le composent servant à faciliter la programmation d'applications pervasives par l'intermédiaire d'un ensemble de service. De plus, la mobilité de l'utilisateur entre différents *active space* est aussi supportée. Les applications utilisateurs et leurs données sont sauvegardées dans une session. Lors d'un changement d'*active space* provoqué par le déplacement de l'utilisateur, ces sessions sont projetées sur les nouvelles ressources de l'environnement.

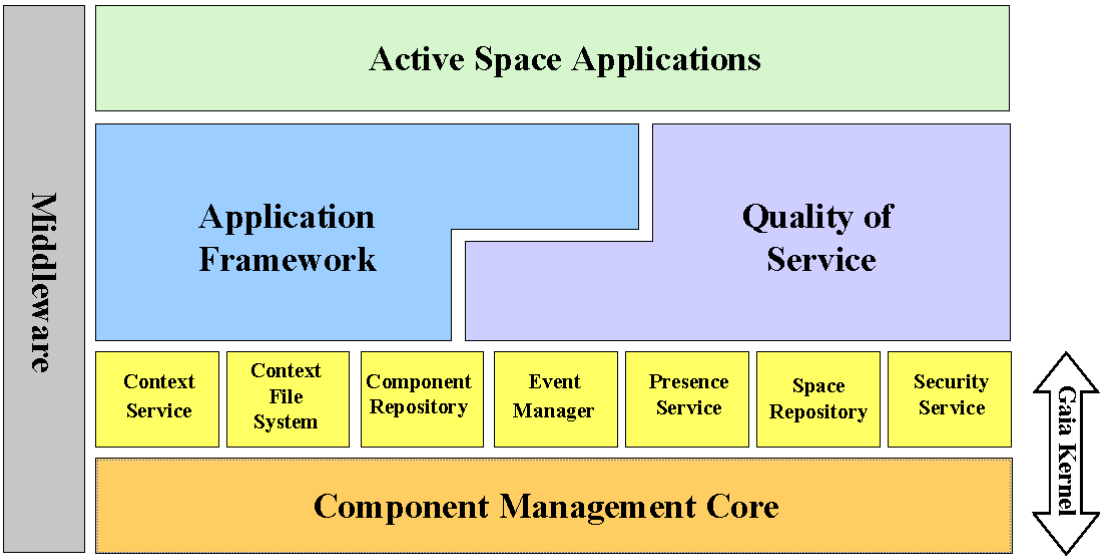

Figure 1.3: Les problématiques de recherche de l'informatique pervasive [Gai]

La figure 1.3 présente l'architecture de la solution Gaia. La brique de base est le *Component Management Core* qui repose elle-même sur une architecture à service CORBA (*Common Object Request Broker*) [Gro08] . CORBA permet de gérer la distribution des composants Gaia sur différentes machines. Le *Component Management Core* s'occupe d'ajouter la gestion de la dynamique du système en prenant en charge les phases de chargement, déchargement, transfert, création et destruction des composants Gaia.

Pour faciliter le développement d'applications s'exécutant dans un *active space*, Gaia fournit un ensemble de services utilisables directement. Les services fournis sont les suivant :

Component Repository : c'est un registre où sont stockés tous les composants Gaia pouvant s'exécuter dans l'Active Space. Des métadonnées supplémentaires associées à chaque composant y sont stockées : par exemple le nom du composant ou l'OS nécessaire à son exécution. Ce service peut être utilisé pour gérer le déploiement (mise à jour, stockage par exemple) des composants sur les différents nœuds de l'Active Space.

Space Repository : C'est une base de données centralisée où sont stockés les différents équipements et services actifs dans l'Active Space. Ces ressources sont décrites à l'aide d'un fichier XML précisant leur nom, leur type, leur localisation et leurs différentes capacités. Les applications peuvent utiliser un langage de requête afin d'y trouver une ressource adaptée à leurs besoins.

Event Manager : C'est un service permettant de transmettre des informations sous formes d'évènements, par l'intermédiaire de canaux, aux différents composants du système. Des évènements sur des canaux prédéfinis sont générés par Gaia, comme l'apparition d'équipement ou de service dans l'Active Space. Pour recevoir les évènements d'un canal, les applications doivent s'y abonner. La possibilité leur est laissée de définir leurs propres évènements et canaux.

Context File System : C'est un système de gestion de fichiers étendu qui peut permettre de stocker les différentes ressources et données sous forme d'arbre. Les applications peuvent l'utiliser pour enregistrer et accéder aux différentes ressources stockées. Elles ont la possibilité d'associer des informations de contexte aux différentes ressources lors des actions citées précédemment. Les données récupérées par une requête peuvent transiter par un mécanisme de médiation pour permettre leur intégration dans l'environnement d'exécution ciblé. La requête suivante permet de trouver la liste des parasols du jardin nord lorsque le temps est ensoleillé :
/**type**:/parasol/**location**:/North-Garden/**weather**:/sunny

Context Service : Ce service permet d'accéder à des informations de contexte relatives à l'Active Space. Les informations y sont stockées sous forme de faits. Les applications peuvent utiliser le service de contexte pour connaitre l'état de l'Active Space mais aussi pour déduire des informations de contexte de plus haut niveau, par l'intermédiaire de règles. Le modèle de règles et le langage qui lui est associé sont fournis par Gaia. La règle suivante permet de déduire que la pièce F018 est occupée ou non, la déduction

s'effectue à partir du nombre de personnes dans la pièce :
Context(number_of_person, RoomF018, >, 0) => **Context**(is_free, RoomF018, =, false)

Presence Service : Le service de présence doit être utilisé par toutes les ressources pour notifier, par l'intermédiaire d'un signal (heartbeat), que celles-ci sont actives. Si un certain délai est dépassé depuis la réception du dernier signal, l'entité est considérée comme inactive et retirée du registre. Gaia fournit le suivi de la présence des utilisateurs par l'intermédiaire de capteurs spécifiques.

Security Service : Un service de sécurité prenant la forme d'un système d'authentification est proposé par Gaia. Il permet entre autre de mettre en place des mécanismes de contrôle d'accès et des modes de coopération pour les différentes ressources ainsi qu'un protocole de communication permettant de préserver l'identité de l'utilisateur.

Le langage Olympus [Ran+05] permet de programmer à un haut niveau d'abstraction les applications s'exécutant dans un *active Space*. Ce langage s'appuie sur les services de Gaia pour l'exécution. Ce modèle de programmation offre deux principales caractéristiques :

La découverte sémantique des entités : par l'intermédiaire d'ontologies, le développeur spécifie les besoins de son application de manière abstraite (services, utilisateurs, équipements physiques, lieux, applications). A l'exécution, les besoins décrits sont liés à des entités présentes dans l'environnement en tenant compte du contexte courant : politique de l'*active space*, préférence de l'utilisateur, etc...

Des opérations de haut niveau relatives au domaine de l'*active space* : les opérations les plus courantes (démarrage ou arrêt d'un équipement, mouvement de l'utilisateur) peuvent être directement intégrées dans le programme du développeur sans se soucier du détail de leur exécution. L'ajout simple de ces nouvelles opérations n'est cependant pas possible.

1.5.2 Aura

Aura [SG02] [Gar+02] se définit comme un framework architectural dont l'objectif est de préserver au maximum l'attention de l'utilisateur des distractions liées aux environnements pervasifs, tels que le management de la volatilité et de l'hétérogénéité des ressources, services ou équipements présents. Ce projet peut se qualifier d'approche centrée utilisateur. Il permet à l'utilisateur d'avoir une instance personnelle de la solution pour l'assister de manière transparente.

Les applications s'exécutant sur Aura sont représentées sous forme de tâches [WG00] . Elles permettent de faire collaborer les ressources disponibles de l'environnement en tenant compte des préférences de l'utilisateur et de ses intentions. Par exemple, si Aura détecte que

l'utilisateur se saisit d'un stylo et interprète une intention d'écrire, cela peut avoir pour effet l'ouverture d'une application de traitement de texte, comme *Microsoft Word* par exemple si celui-ci est présent sur la station de travail. Le choix de l'application à ouvrir étant aussi dépendant des préférences de l'utilisateur et du contexte courant. Dans l'exemple précédent, si l'utilisateur tenait aussi son téléphone dans la main, Aura pourrait choisir d'ouvrir une application servant à rédiger un *SMS*.

Pour permettre aux différentes tâches de s'exécuter en fonction de l'environnement et des préférences utilisateurs, Aura repose sur l'architecture présentée dans la figure 1.4. L'architecture d'Aura se décompose en quatre types de composant :

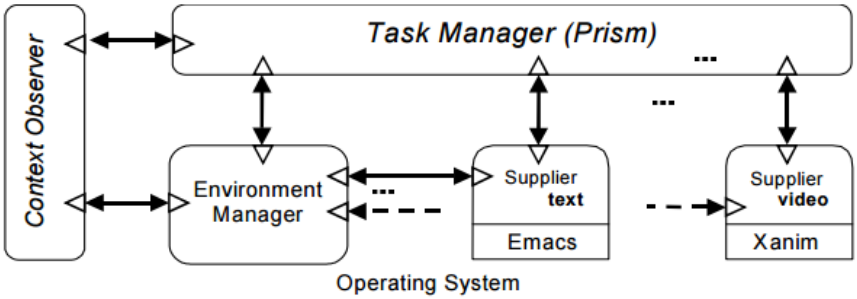

Figure 1.4: Architecture de la solution Aura [SG02]

Task Manager : Ce composant est chargé de l'exécution des différentes tâches. Les tâches sont programmées grâce à un langage dédié et une uniformisation des différents services de l'environnement. L'exécution d'une tâche peut être réévaluée en fonction des changements de contexte, des préférences utilisateur ou l'apparition d'un nouveau fournisseur de service.

Context Observer : Ce composant est chargé de fournir des informations sur le contexte physique de l'environnement et les évènements importants s'y déroulant. Ces informations sont transmises à l'Environment Manager et au Task Manager.

Environment Manager : Ce composant sert de registre aux différents fournisseurs de service. Lorsqu'une nouvelle tâche est instanciée, celle-ci passe par l'Environment Manager pour obtenir les différents services nécessaires à son exécution. La résolution de cette demande est effectuée en fonction des préférences utilisateur et du contexte d'exécution.

Service Suppliers : Ces composants représentent les différents fournisseurs de services. Ceux-ci doivent se conformer à une spécification propriétaire dans Aura, décrit en un format basé sur du XML, pour pouvoir être liés aux différentes tâches.

Les avantages de la solution Aura sont que le développement par tâche permet de masquer l'hétérogénéité de l'environnement en programmant par rapport aux abstractions de service

fournies par Aura. La liaison à ces services est déléguée à l'exécution. Cette liaison tardive permet de prendre en compte le contexte d'exécution et certaines préférences utilisateur. Cependant, cette liaison automatique est rendue possible par l'introduction d'un surcoût de développement des fournisseurs de services. Ceux-ci doivent se conformer à l'interface propriétaire d'Aura et fournir un adaptateur spécifique de leur service. Un manque d'assistance pour étendre les *Service Suppliers* et le *Context Observer* sont à noter ce qui entraine une maintenance et une conception de ces composants particulièrement coûteuses.

1.5.3 MUSIC

MUSIC [Hal+12][Rou+09][GW13] se définit comme une méthodologie de développement dirigée par les modèles pour les applications mobiles auto-adaptatives et sensibles au contexte. Le spectre des outils fournis par l'approche MUSIC est très large, on trouve par exemple un studio de développement (*MUSIC studio*), des outils de tests et de simulations de l'environnement pour évaluer les applications ainsi qu'une couche middleware dédiée à l'exécution du type d'application cité précédemment.

L'architecture à l'exécution de la solution MUSIC est présentée dans la figure 1.5. Celle-ci a pour but de fournir des fonctionnalités de management du contexte, d'adaptation et de reconfiguration d'application. Il est à noter que même si MUSIC se définit comme un middleware indépendant des technologies sous-jacentes, une partie de ses concepts sont liés et hérités de la technologie OSGi [All07] , celle-ci étant utilisée pour l'implémentation de référence. Pour rappel, OSGi fournit les bases pour pouvoir mettre en place une architecture modulaire et une approche à services. OSGi est basé sur le langage Java et s'exécute audessus de la JVM (en anglais : *Java Virtual Machine*), cela offre la possibilité de s'exécuter sur différentes plateformes mobiles.

L'architecture est séparée en 4 couches :

Core : cette couche fournit une abstraction des technologies présentes en- dessous du middleware pour permettre un découplage entre celui-ci et la plateforme d'exécution. Un modèle commun d'information et des structures de données permettant la représentation des applications sont aussi fournis.

System Services : cette couche fournit le support de l'approche à services avec une gestion de la sécurité de l'accès aux services. Plus particulièrement, un support supplémentaire est fourni vis-à-vis de la distribution des hôtes sur le réseau. Elle permet entre autre l'enregistrement et la liaison entre services distants, la communication entre ceux-ci en masquant le protocole de communication sous-jacent.

Context & Adaptation Middleware : cette couche fournit un support vis-à-vis de la sensibilité au contexte et de l'adaption des applications. Elle est en charge de la collecte,

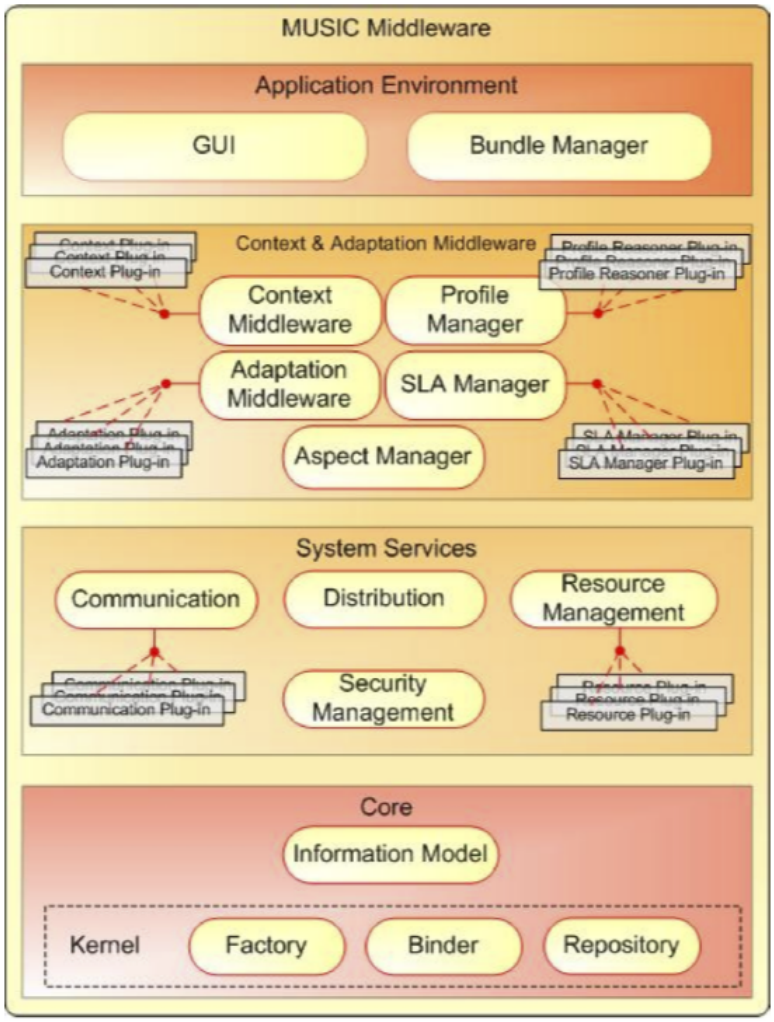

Figure 1.5: Architecture à l'exécution de la solution MUSIC [Hal10]

du stockage et de la distribution des informations de contexte aux applications à travers un langage de requête spécialisé. Celui-ci permet, en plus de sélectionner les différentes informations de contexte, de souscrire aux différents évènements liés à une modification de l'environnement ou de réaliser quelques transformations simples des informations déjà présentes comme par exemple des agrégations. Un entrepôt de l'historique des données est aussi maintenu. Le modèle des différentes informations de contexte est réalisé à l'aide d'une ontologie qui est ensuite convertie dans un langage programmatique. Le composant d'adaptation reçoit les évènements de changements de contexte et réagit en conséquence pour essayer de mettre en place une adaptation fournie par les applications.

Application Environment : cette couche fournit des interfaces graphiques pour interagir avec la solution MUSIC et mettre à jour les applications.

Certaines propriétés comme le dynamisme, l'hétérogénéité et la distribution sont bien couvertes par la solution MUSIC et facilitent grandement le développement des applications pervasives. Mais comme cela est dénoté dans [GW13] , le travail de conception et d'implémentation des différents composants du middleware s'en trouve considérablement alourdi. Par exemple dans [Rei+08] , on notera que le support de modélisation de contexte est réalisé grâce à une ontologie qui exhibe des qualités d'extensibilité et que l'architecture à l'exécution permet d'ajouter à chaud des *plug-ins* capables de collecter de nouvelles informations et donc de conserver cette extensibilité. Or le seul support disponible pour la création de ces nouvelles extensions consiste en la génération de classe représentant les différentes données et leur getter et setter ce qui est peu par rapport aux difficultés liées à la gestion de la synchronisation avec le monde extérieur par exemple.

1.6 Conclusion

Le domaine de l'informatique pervasive n'est plus seulement une vision lointaine qui attend patiemment de voir des solutions émerger. Les technologies nécessaires à son avènement sont de plus en plus présentes dans nos environnements et comme nous avons pu le voir des solutions dédiées ont été proposées par la communauté pour répondre aux différents défis énoncés.

Les solutions apportées offrent des propositions robustes et industrialisables pour les problématiques les plus anciennes, souvent tirées des paradigmes informatiques pré informatique pervasive, tels que la gestion de la distribution, l'hétérogénéité et le dynamisme.

Cependant des problématiques subsistent encore. D'une part, le support pour la gestion de l'autonomie est souvent limité du côté des solutions proposées comme dénoté dans [Bou14]. Les applications ubiquitaires doivent pouvoir changer selon les circonstances d'exécution, s'adapter ou influer l'environnement pour répondre au mieux aux besoins de l'utilisateur. Cela doit être possible de façon autonome sans la gouvernance continue d'un administrateur. D'autre part l'ouverture des environnements pervasifs doit se traduire par une forte extensibilité des différentes solutions pour pouvoir intégrer de façon transparente et efficace l'apparition des nouveaux types d'équipements ou autres services distants. Cette tâche est souvent sans fin car les différents fournisseurs de ressources n'ont pas d'intérêt à coopérer sur une standardisation commune des ressources comme nous l'avons vu. De plus, elle reste hautement technique car elle doit toujours être en phase avec les dernières technologies.

Comme nous avons pu l'entrevoir, le module de contexte est particulièrement impacté par ce manque de support. Une étude bien plus exhaustive des différentes solutions de la gestion de contexte est proposée dans le chapitre suivant.

Chapter 2

Sensibilité au contexte et Fog Computing

Sommaire

Dans ce deuxième chapitre, notre but est d'établir les besoins en termes de contexte d'une plateforme *Fog Computing*.

Pour cela, nous détaillerons les différentes définitions disponibles dans la littérature de l'informatique sensible au contexte et des concepts qui lui sont liées. Dans un second temps, nous nous intéresserons à un cadre de comparaison, lui aussi tiré de la littérature, des différentes solutions informatiques de sensibilité au contexte. Nous aborderons ensuite le domaine du *Fog Computing* et mettront en perspective ses besoins avec le précédent cadre de comparaison. Nous profiterons de la conclusion pour aborder les limites des différentes approches.

2.1 Introduction

Au début des années 1980, on trouve par l'intermédiaire de [Leh80] une classification des différentes applications selon trois catégories. Ces catégories ont été faites pour définir les besoins en évolution et maintenance des différentes applications. Les catégories sont définies selon la taxonomie suivante :

Les S-programs : Ces applications sont définies suivant une spécification formelle et immuable. Elles traitent un problème abstrait et sont souvent peu reliées au monde réel.

Les P-programs : Ces applications traitent un problème, généralement assez abstrait, en se basant sur une représentation partielle du monde. Elles ne sont pas fortement contraintes par une spécification mais doivent intégrer un nombre de règles à respecter.

Les E-programs : Ces applications sont en interactions directes avec le monde réel. Elles ont une relation causale avec celui-ci, chacune de leurs actions l'affectant directement. Leur comportement est totalement conditionné par l'état du monde et donc très faiblement contraint par une spécification formelle.

Les E-programs sont donc exactement le type d'application qui évolue dans les environnements pervasifs. Dans [Leh80], l'auteur les définit comme :

« The program has become a part of the world it models, it is embedded in it. »

La notion de relation à un modèle du monde va être formalisée plus tard, principalement sous la bannière de la notion de contexte au tout début des années 2000. Cette introduction va servir à analyser ce que la communauté comprend par le terme « contexte » et les différents concepts qui lui sont rattachés.

2.1.1 Définitions liées au contexte

2.1.1.1 Contexte

La notion de contexte a été investiguée dans de nombreux travaux. Sa formalisation la plus acceptée par la communauté est celle établie par [Dey01]:

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

Cette définition est l'une des premières à établir le contexte par compréhension (*ie* en mettant des propriétés permettant d'inclure ou non un élément dans un ensemble) plutôt que par extension (*ie* en énumérant tous les éléments de l'ensemble). La propriété mise en avant par [Dey01] est la pertinence de l'information par rapport à l'interaction application-utilisateur. Cette définition est celle de référence dans ce manuscrit. Elle permet entre autres de facilement déterminer si une information fait ou non partie du contexte. Elle accentue le fait que le contexte est construit suivant les besoins des applications.

Les autres définitions que l'on retrouve dans la littérature sont d'intérêt variable. Les définitions données par [FF98] [HNB97] se contentent de définir ce qu'est le contexte par le biais de synonymes tels que la situation de l'utilisateur ou les différents aspects de la situation courante. Les définitions par extension du contexte tel que [BBC97] [SAW94] [Hof+03] [AM00] restent assez spécifiques mais permettent néanmoins d'identifier des ensembles d'informations souvent présents dans le contexte. D'après [SAW94], on peut établir la classification suivante des différents types de contexte, celle-ci étant faite en fonction de la provenance des informations:

L'environnement virtuel inclut les informations permettant de décrire le système informatique, telles que les périphériques, les ressources utilisées, la connexion, la bande passante utilisée, etc...

L'environnement utilisateur inclut les informations sur les personnes en interaction avec l'application telles que leur position géographique, leur activité, les personnes à proximité, etc...

L'environnement physique inclut les informations sur les grandeurs physiques décrivant l'environnement dans lequel est placé l'utilisateur. On retrouve des informations telles que la température, la luminosité, l'humidité, le bruit ambiant etc...

Ces trois environnements sont en interaction constante. Des phénomènes tels qu'un déplacement de l'utilisateur ou son interaction avec des actionneurs peuvent venir à la fois impacter l'environnement virtuel et physique. Par exemple, le fait de déplacer son ordinateur personnel dans une nouvelle pièce et de le brancher à l'alimentation secteur de la maison vient modifier la consommation globale de celle-ci et peut changer les accès aux réseaux et les appareils disponibles à proximité. En étant sensibles aux fluctuations de ces environnements, les systèmes ubiquitaires peuvent délivrer des services plus personnalisés. Cette classification met en lumière l'hétérogénéité des informations de contexte à traiter.

D'autres taxonomies des informations de contexte existent, celles-ci reposent par exemple sur les caractéristiques de la source de l'information [Hen03] ou sur le niveau d'abstraction de l'information [Bet+10] illustré par la figure 2.1. Les avantages et les inconvénients de ces différentes taxonomie sont exposés dans [Per+14a].

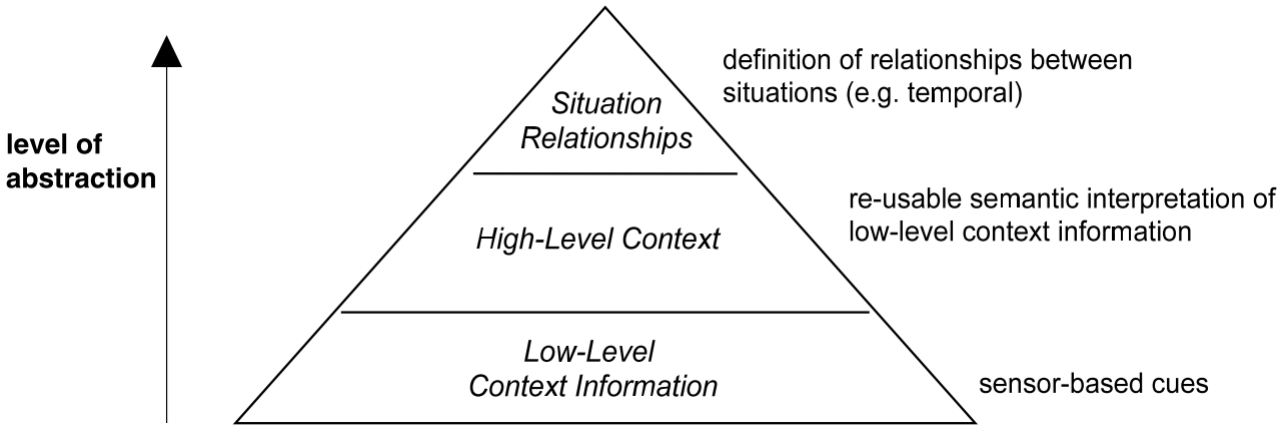

Figure 2.1: Niveau d'abstraction des informations de contexte [Bet+10]

Ces autres classifications mettent en lumière le fait que les informations de contexte sont tantôt extraites et synchronisées directement à partir des capteurs disséminés dans l'environnement ou d'autres sources (services distants, bases de données) ou bien inférées par divers moyens de raisonnement ou de médiation à partir des précédentes informations.

2.1.1.2 Sensibilité au contexte

La sensibilité au contexte ou « context-awarness » est utilisée pour qualifier des systèmes pouvant adapter leur comportement et leurs services fournis à l'utilisateur en fonction des changements dans leur environnement [BDR07]. Il faut prendre la notion d'environnement au sens large comme expliquée précédemment dans la section 2.1.1.1. Initialement, ces systèmes sont apparus avec l'informatique mobile et se concentraient uniquement sur la prise en compte de la localisation des utilisateurs (location-awarness). On retrouve par exemple dans [Wan+92], un système qui permet de localiser des personnes au sein d'une organisation pour plus facilement les contacter. Ces systèmes ont beaucoup évolué pour prendre en compte de plus en plus d'informations à différents niveaux d'abstraction [Bet+10]. De nombreuses descriptions détaillées d'applications sensibles au contexte peuvent être trouvées dans [CK00] .

Pour la suite de ce manuscrit, nous utiliserons la définition de systèmes sensibles au contexte produite par [Abo+99]:

« A system is context-aware if it uses context to provide relevant information and/or service to the user, where relevancy depends on the user's task. »

On retient cette définition car elle établit un critère qui permet de facilement discerner si un système est sensible au contexte ou non. De nombreuses autres définitions peuvent être

trouvées dans la littérature [ST94] [RPM99] , leur critique étant disponible dans [Abo+99]. En résumé, celles-ci sont considérées comme trop spécifiques et ne mettent pas en avant un critère simple permettant de juger si une application est sensible au contexte ou non.

Toujours selon [Abo+99], les applications sensibles au contexte doivent pouvoir fournir trois fonctionnalités générales :

La présentation d'informations ou de services à l'utilisateur : Les applications doivent être capables de proposer des informations ou des services disponibles à l'utilisateur de manière la plus pertinente possible.

L'exécution automatique d'un service : Un service doit pouvoir effectuer des actions, affectant potentiellement l'environnement via des actionneurs, en basant ses décisions sur des informations extraites du contexte.

Marquer des informations avec des éléments de contexte pour un usage ultérieur : Les données extraites des différentes sources de contexte ne fournissent pas seules les informations nécessaires pour appréhender parfaitement une situation. Leur analyse, fusion et interprétation [RFL05] peuvent aider à mieux appréhender la dynamique de l'environnement et les situations qui y prennent place. Cette analyse souvent coûteuse peut être menée hors ligne.

Cette classification met en lumière les besoins des applications en terme de raisonnement et de remontée d'informations. Elle permet aussi de voir que les applications sensibles au contexte ont aussi besoin d'influencer le contexte et donc d'avoir un support pour des interactions bidirectionnelles vis-à-vis du contexte (remontée de données et redescente d'actions) [VSB13].

2.1.1.3 Qualité de contexte

Par essence, les informations de contexte sont imparfaites [CM00] et leur qualité est influencée par la manière dont elles sont produites [BFA05]. La qualité de contexte sert à mesurer ce niveau d'imperfection de manière quantitative pour aider les applications dans plusieurs tâches. On peut citer par exemple la sélection d'une adaptation au détriment d'une autre ou la gestion de conflits entre deux informations de contexte décrivant une même entité de manière contraire [BS03]. Pour la suite du manuscrit, on adoptera la définition suivante pour la qualité de contexte, donnée par [BS03]

« Quality of context is any information that describes the quality of information that is used as context information. »

Plusieurs ensembles de paramètres ont été proposés pour mesurer cette qualité : précision, probabilité que l'information soit correcte, fiabilité, résolution, fraicheur de l'information, ... etc [BS03] [MTD08] . Une étude comparative des différentes approches de la qualité de contexte est présentée dans [Bel+12].

2.1.1.4 Synthèse

D'après les différentes dates des définitions énoncées, on peut avancer que la formalisation de la notion d'informatique sensible au contexte a eu lieu autour de l'année 2000. Mais depuis cette formalisation, sa mise en œuvre n'a cessé d'évoluer. Les travaux sont passés de modèle limité à la localisation des utilisateurs par analyse de simple tag RFID vers des modèles beaucoup plus élaborés reposant par exemple sur la déduction des activités des utilisateurs mettant en jeu de la fusion de capteur ou des techniques de raisonnement plus poussées [Bet+10]. De plus la récente émergence de l'Internet des Objets (« Internet of Things, IoT») continue de faire évoluer ce domaine [Per+14a].

De par la relative ancienneté de la thématique générale abordée, la sensibilité au contexte, un nombre important d'études comparatives a été fourni par la communauté. Notre état de l'art s'appuie en grande partie sur l'énorme somme de connaissances apportées par ces travaux. Voici une liste des principales études comparatives sur lesquelles nous nous sommes appuyés avec leurs caractéristiques :

L'étude [IAC16] se focalise sur les processus d'ingénierie permettant la mise en place des solutions. Elle fournit une taxonomie permettant de classer les processus mis en place plutôt que les approches. Un sondage de la communauté est aussi réalisé pour permettre d'identifier les différentes fonctionnalités attendues et à venir des systèmes de contexte.

L'étude [Per+14a] se concentre sur la mise en perspective des solutions avec les challenges de l'IoT. Elle fournit une taxonomie permettant le classement des solutions et l'applique à un nombre important de projets.

L'étude [Bel+12] se concentre sur la distribution du contexte pour les systèmes ubiquitaires mobiles. Une taxonomie permettant de classer les différentes approches y est présentée. Une section comparant les différentes définitions et approches concernant la qualité de contexte est présente.

L'étude [Bet+10] se concentre sur les différentes techniques permettant la modélisation du contexte et les techniques de raisonnement applicables. Elle établit une petite taxonomie concernant les techniques de modélisation.

Les deux études [Kjæ07] et [Per+14b] fournissent une taxonomie pour différencier les diverses approches middleware liées à la sensibilité au contexte.

Une comparaison plus poussée des différentes études comparatives est disponible dans [Per+14b].

Comme nous l'avons vu les définitions précédentes mettent en lumière certains besoins généraux liés à la construction d'applications sensibles au contexte. Pour les synthétiser, nous présenterons dans la partie suivante le cycle de vie des informations de contexte généralement adopté par la communauté.

2.1.2 Cycle de vie du contexte

Les nombreuses tentatives de construction de systèmes visant à simplifier la mise en place de solutions sensibles au contexte ont permis de dégager un patron concernant le cycle de vie d'une information de contexte [Bel+12] [Per+14b] illustré par la figure 2.2 . Pour rappel, un patron en informatique est la solution à un problème récurrent de conception.

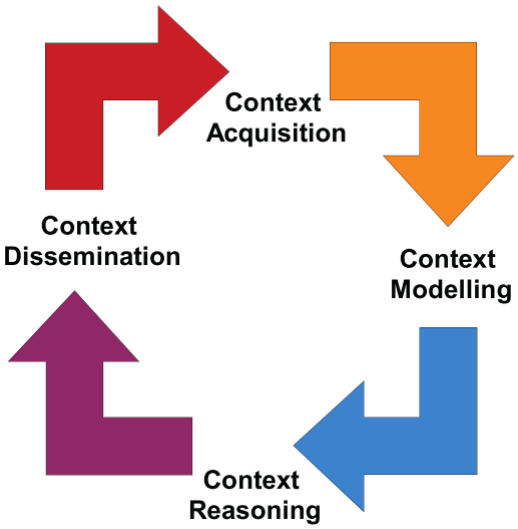

Figure 2.2: Cycle de vie d'une information de contexte [Per+14b]

Le cycle de vie du contexte est constitué de quatre étapes. Premièrement, l'information de contexte est collectée et synchronisée (*context acquisition / context gathering*) depuis plusieurs types de sources possibles : capteurs physiques, virtuels ou logiques [BDR07]. Dans un second temps, les informations collectées viennent s'agréger dans un modèle (*context modelling*) dans le but de leur donner une sémantique plus précise. Dans un troisième temps, les informations modélisées peuvent être utilisées à travers divers moyens de raisonnement (*context reasoning*) pour produire des informations plus abstraites. Pour finir, les consommateurs peuvent interagir avec le modèle (*context dissemination*) pour venir récupérer ces informations ou effectuer des actions.

Il existe plusieurs solutions pour construire des applications sensibles au contexte [Per+14b]. Une des plus populaires est de séparer le contexte et les applications dans deux modules de programmation différents, comme illustré par la figure 2.3. C'est une approche qualifiée de

middleware. Le module de contexte embarque ainsi toutes les tâches liées au cycle de vie du contexte. Nous souscrivons à cette approche car elle permet de clairement séparer le contexte et l'application et permet ainsi d'augmenter la lisibilité du code, son évolution et sa facilité à être testé.

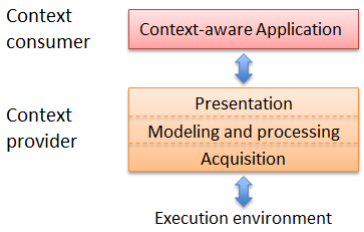

Figure 2.3: Séparation du contexte et de l'application

Nous allons détailler ce cycle de vie et les points à considérer dans le développement d'un middleware de contexte.

2.1.2.1 Modèle du contexte

Dans les systèmes sensibles au contexte, les faits, données, concepts ou évènements utilisés pour décrire le contexte d'une entité sont formalisés à un modèle de contexte. L'étape de modélisation du contexte est le pivot de la gestion de contexte. Les autres étapes ont pour but soit de venir alimenter ce modèle (collecte et raisonnement) soit de le rendre accessible aux applications (dissémination). La notion de modèle est très ancienne et l'on peut trouver trace de beaucoup de définitions dans la littérature [OMG01] [BG01]. On utilisera celle donnée par le standard UML :

« A model is an abstraction of a physical system, with a certain purpose.»

La notion de modèle de contexte a été raffinée dans [Hen03] :

« A context model identifies a concrete subset of the context that is realistically attainable from sensors, applications and users and able to be exploited in the execution of the task. The context model that is employed by a given context-aware is usually explicitly specified by the application developer, but may evolve over time. »

Il faut tout d'abord noter qu'aucun standard ne spécifie explicitement tous les types d'informations à modéliser dans le contexte. Selon [Per+14b] [Jon07], les informations représentées dans les solutions modélisant le contexte sont subjectives et sont elles-mêmes dépendantes du contexte de création du modèle. Comme cité plus haut, on crée un modèle dans un but précis. On peut prendre pour exemple deux applications de domotique. L'une

d'elle propose un suivi lumineux des personnes de pièce en pièce. Elle s'intéresse donc aux types d'équipements lampe et capteur de présence, à leur état (allumé ou éteint) ainsi qu'à leur position géographique traduite sous forme de pièce. Une autre application a pour but de contrôler le réseau d'un protocole spécifique. Les équipements sont cette fois identifiés selon les types contrôleur, répéteur, esclave et l'on s'intéresse plus particulièrement à la puissance du signal que chacun reçoit ainsi qu'à la topologie du réseau (qui est le voisin de qui par exemple). Les deux modèles souhaités ne sont en soit pas incompatibles mais dépeignent des finalités différentes pour chaque application. Comme conclu par [Jon07], nous statuons que le but des solutions de sensibilité n'est pas de produire un modèle où la connaissance est générique et acceptée par tous mais plutôt de fournir des moyens génériques de collecter, disséminer et raisonner sur cette connaissance.

Les techniques de modélisation du contexte les plus répandues ont été reprises dans de nombreuses études comparatives [SL04] [Bet+10] [BDR07] [Per+14b], celles-ci sont présentées brièvement :

Key-value models sont les modèles de contexte les plus simples. Ils associent une valeur à une clé unique. Cette approche a été utilisée dans les premiers travaux sur le contexte tel que ParcTab [SAW94] ou Capeus [SML01].

Markup Schemes structurent le contexte grâce à des tags, qui peuvent être hiérarchisés et dotés d'attributs. Les approches telles que CSCP [HBS02] et PPDL [CF03] utilisent cette méthode à partir de format de données dérivé du SGML ou XML pour représenter le contexte.

Object-oriented models représentent le contexte à l'aide d'un graphe d'objet typé. Les approches comme Hydrogen [Hof+03] ou GUIDES's Active Object Model [CMD99] utilisent cette façon de représenter le contexte.

Graphical models utilisent les avantages de l'UML pour représenter le contexte. On retrouve cette approche dans ORM [HIR03] et ContextUML [SB05].

Logic-based models représentent le contexte à l'aide de faits, d'expressions ou de règles. Des approches telles que [McC93] utilisent ces éléments logiques pour modéliser le contexte.

Ontology-based models spécifient des concepts et leurs relations pour capturer la connaissance du contexte. Les approches comme CoBrA [CFJ03] ou SOCAM [Gu+04] utilisent les ontologies SOUPA [Che+04] et CONON [Wan+04] pour représenter le contexte.

Un comparatif qualitatif des approches ci-dessus est proposé dans la figure 2.4.

Comme énoncé dans certaines approches [Per+14b] [VSB13] [Ber+14], les interactions de contexte sont bidirectionnelles et leur prise en charge doit être facilitée par le middleware de contexte. Les actions peuvent soit être explicitées dans le modèle [Ber+14] ou laissées à la

Techniques	Pros	Cons	Applicability
Key-Value	• Simple • Flexible • Easy to manage when small in size	• Strongly coupled with applications • Not scalable • No structure or schema • Hard to retrieve information • No way to represent relationships • No validation support • No standard processing tools are available	Can be used to model limited amount of data such as user preferences and application configurations. Mostly independent and non-related pieces of information. This is also suitable for limited data transferring and any other less complex temporary modelling requirements.
Markup Scheme Tagged Encoding (e.g. xml)	• Flexible • More structured • Validation possible through schemas • Processing tools are available	• Application depended as there are no standards for structures • Can be complex when many levels of information are involved • Moderately difficult to retrieve information	Can be used as intermediate data organisation format as well as mode of data transfer over network. Can be used to decouple data structures used by two components in a system. (e.g. SensorML [133] for store sensor descriptions, JSON as a format to data transfer over network)
Graphical (e.g. databases)	• Allows relationships modelling • Information retrieval is moderately easier • Different standards and implementations are available. • Validation possible through constraints	• Querying can be complex • Configuration may be required • Interoperability among different implementation is difficult • No standards but governed by design principles	Can be used for long term and large volume of permanent data archival. Historic context can be store in databases.
Object Based	• Allows relationships modelling • Can be well integrated using programming languages • Processing tools are available	• Hard to retrieve information • No standards but govern by design principles • Lack of validation	Can be used to represent context in programming code level. Allows context runtime manipulation. Very short term, temporary, and mostly stored in computer memory. Also support data transfer over network.
Logic Based	• Allows to generate high-level context using low-level context • Simple to model and use • support logical reasoning • Processing tools are available	• No standards • Lack of validation • Strongly coupled with applications	Can be used to generate high-level context using low-level context (i.e. generate new knowledge), model events and actions (i.e. event detection), and define constrains and restrictions.
Ontology Based	• Support semantic reasoning • Allows more expressive representation of context • Strong validation • Application independent and allows sharing • Strong support by standardisations • Fairly sophisticated tools available	• Representation can be complex • Information retrieval can be complex and resource intensive	Can be used to model domain knowledge and structure context based on the relationships defined by the ontology. Rather than storing data on ontologies, data can be stored in appropriate data sources (i.e. databases) while structure is provided by ontologies.

Figure 2.4: Comparatif des approches de modélisations de contexte proposé par [Per+14b]

charge de la partie dissémination [VSB13] comme nous le verrons par la suite dans la section 2.1.2.4. Le modèle permettant de décrire les actions possibles avec le plus de sémantique et de les intégrer le plus naturellement dans l'application est le modèle orienté objet.

2.1.2.2 Collecte du contexte

L'étape de collecte du contexte est le point d'entrée dans le cycle de vie du contexte. Son but est de maintenir une instance du modèle synchronisée avec les différents environnements (exécution, utilisateur, physique). Pour cela plusieurs aspects sont à prendre en compte lors du développement d'un middleware de contexte.

Dans les environnements pervasifs, les sources de contexte sont volatiles et hétérogènes. Pour gérer la volatilité de ces sources, des approches de découverte dynamique de ressources peuvent être mises en place comme dans [BLE10] [Bec+04]. Par exemple, [BLE10] propose un style architectural permettant de découpler grandement la logique de découverte de nouvelles ressources des préoccupations de synchronisation liées à l'état de la ressource. Ce style architectural apporte des propriétés de flexibilité, c'est-à-dire qu'à l'exécution l'ajout de nouveaux composants ne perturbe pas ceux déjà présents.

Une fois cette volatilité des ressources prise en compte, il reste à gérer la synchronisation de l'état de la ressource de contexte avec son pendant dans le monde réel. Pour cela plusieurs prérogatives ont été énoncées par la communauté. Tout d'abord dans [Pie+08], la responsabilité de l'acquisition est présentée selon deux méthodes : *Push/Pull*. Avec la méthode *pull*, le logiciel est responsable de l'acquisition de la donnée avec par exemple une requête explicite à un capteur. Avec la méthode *push*, les sources de contexte produisent des données sans requête explicite, le logiciel étant en charge de capturer ces données. Avec l'apparition de la demande des interactions bidirectionnelles, on peut ajouter une troisième méthode, nommée *apply*, où le logiciel applique explicitement un état à une ressource pour que celle-ci entreprenne une action. Une autre prérogative concerne la temporalité de ces actions [Per+14b], celles-ci sont soit effectuées ponctuellement soit périodiquement. Par exemple, allumer une lumière, détecter un mouvement ou la fourniture d'une information manuellement par l'utilisateur sont des actions qualifiées de ponctuelles, tandis que les relevés de température ambiante peuvent s'effectuer de manière périodique pour économiser des ressources, compte tenu de la faible dynamique de ce paramètre environnemental. Cette partie du développement est très technique et propice à l'erreur compte tenu de l'hétérogénéité des sources et des protocoles de communication.

2.1.2.3 Raisonnement à partir du contexte

L'étape de raisonnement sur le contexte permet de déduire de nouveaux faits de contexte, à partir de ceux déjà modélisés et instanciés pour ainsi gagner en abstraction [Bet+10].L'établissement de ces nouveaux concepts plus abstraits permet de réduire le décalage

sémantique entre les informations extraites directement de l'environnement par l'intermédiaire de capteurs par exemple et les besoins plus spécifiques des applications.

Les techniques de raisonnement mises en place peuvent être très variées. Les auteurs de [Per+14b] présentent et détaillent un large éventail de ces techniques et les classent en cinq catégories : *Supervised Learning, Unspervised Learning, Rules, Fuzzy Logic, Ontological-Based* et *Probabilistic*. Ces techniques sont très connotées intelligence artificielle. De plus, d'autres catégories de techniques tirées d'autres domaines tels que la médiation [Wie92] ou l'event processing [Luc08] peuvent aussi être applicables. Par exemple, l'approche COPAL [LSD10] introduit la notion de processeur dans sa structure de contexte. Chaque processeur a pour but d'enrichir le contexte avec des concepts plus abstraits grâce à des techniques tirées de l'event processing qui sont regroupées en cinq patrons : *Filter, Abstraction, Enrichment, Differentiation* et *Peeling*.

Le but de ce manuscrit n'est pas d'établir une étude exhaustive des différentes techniques de raisonnement mais de rendre compte au lecteur que celles-ci existent et sont nombreuses. Des études comparatives peuvent être trouvées dans [Bet+10] [PRL09] [Per+14b]. Il est à noter qu'en règle générale les méthodes de raisonnement liées à l'intelligence artificielle ont de grosses demandes en termes de consommation de ressources (temps processeur, espace de stockage) tandis que les méthodes de médiation ou d'*event processing* sont souvent plus souhaitables pour préserver des contraintes temps réel.

2.1.2.4 Dissémination du contexte

L'étape de dissémination du contexte permet de rendre le modèle de contexte disponible à l'exécution aux différents consommateurs de contexte. Plusieurs solutions techniques sont possibles, celle-ci affectant le système en terme de scalabilité, de disponibilité des données ou de découverte dynamique de ressources [Bel+12] [Jon07]. Les auteurs de [Per+14b] présentent les deux méthodes les plus générales :

***Query-based*:** Les consommateurs de contexte peuvent faire des demandes au système de management du contexte à l'aide d'une requête. Le *middleware* utilise le contenu de cette requête pour produire un résultat. Il existe une multitude de langages de requête, certains standardisés tel SPARQL pour les ontologies ou d'autres construits spécialement dans le cadre d'une approche [VSB13] [Sch+12]. Certains langages de requête incluent des moyens d'effectuer des actions sur les différents actionneurs [Sch+12] ou sur des concepts plus abstraits comme la température [VSB13].

***Subscription*:** Les consommateurs de contexte décrivent des exigences en termes de contexte et le système de management du contexte est chargé d'y répondre. Les exigences peuvent aussi être décrites à l'aide de requêtes, par exemple, dans [HIR08] les applications décrivant leurs besoins en termes de fait de contexte. Une grande variété de technologies

et d'architectures peuvent être utilisées pour disséminer le contexte tel que des middlewares orientés messages ou l'architecture SOA. Les auteurs de [Per+14b] recommandent cette méthode pour mettre en place des solutions orientées temps réel.

2.2 Cadre de comparaison des approches

Avant de présenter les différentes approches permettant le développement et l'exécution d'applications sensibles au contexte, nous allons nous intéresser à la taxonomie présentée par [Per+14b]. Elle permet une évaluation des différentes approches selon différentes fonctionnalités en rapport avec le cycle de vie précédemment étudié. Les auteurs se sont servis de cette taxonomie pour évaluer cinquante projets. Dans un premier temps nous étudierons les différents critères composants cette taxonomie. Dans un second temps, nous dresserons une évaluation des critères de comparaison au regard de la structure des applications pervasives actuelles. Nous terminerons cette partie par une synthèse couplant les résultats obtenus par la taxonomie et notre évaluation des critères de comparaison.

2.2.1 Critères de comparaison

Cette première partie sert à présenter les dix-sept critères utilisés pour établir une comparaison des différentes approches liées à la sensibilité au contexte.

2.2.1.1 Type de solution

Le premier critère est le type de la solution mise en place par l'approche (*Project Focus*). Les auteurs distinguent pour cela trois types d'approche :

***System (S)*:** Une approche qualifiée de System propose une solution couvrant la totalité du problème abordé. Elle implique aussi bien des composants matériels que logiciels, applications incluses. C'est la seule approche à être directement utilisée par l'utilisateur final.

***Toolkit (T)*:** Une approche qualifiée de Toolkit propose une brique logicielle se focalisant sur un besoin très spécifique. Cette brique logicielle peut être utilisée dans le développement d'une approche système ou middleware ainsi que d'une application.

***Middleware (M)*:** Une approche qualifiée de Middleware prend en charge un nombre de fonctionnalités communes et non-fonctionnelles au regard du développement de l'activité business d'une application. Ces fonctionnalités sont intégrées dans une couche logicielle qui fournit les abstractions nécessaires aux programmeurs d'application, pour que ceux-ci effectuent leur développement avec plus d'efficience.

2.2.1.2 Modèle Utilisé

Le deuxième critère concerne la technique utilisée pour modéliser le contexte (*Modelling*). Les différentes techniques répertoriées sont : *key-value modelling (K), markup schemes (M),*

graphical modelling (G), object oriented modelling (Ob), logic-based modelling (L) et *ontology-based modelling (On)*. Celles-ci ont été présentées en amont dans la section 2.1.2.1.

2.2.1.3 Techniques de raisonnement

Le troisième critère concerne les techniques de raisonnement supportées par la solution (*Reasoning*). Les techniques de raisonnement sont classifiées selon la liste suivante : *supervised learning (S), un-supervised learning (U), rules (R), fuzzy logic (F), ontology-based (O)* et *probabilistic reasonning (P)*. Par rapport à la discussion entretenue dans la section 2.1.2.3, les auteurs ont décidé de garder dans ce critère les techniques liées à l'intelligence artificielle. Les techniques de raisonnement liées à la médiation sont caractérisées dans le critère *Traitement des données* présenté dans la section 2.2.1.14. Le symbole ($\surd$) est utilisé pour indiquer qu'une ou plusieurs facilités de raisonnement sont supportées mais la technique spécifique employée n'est pas précisée.

2.2.1.4 Technique de dissémination

Le quatrième critère indique la technique de dissémination utilisée (*Dissemination*). La classification établie est : *publish/subscribe (P)* et *query (Q)*. Les détails de cette classification ont été présentés dans la section 2.1.2.4.

2.2.1.5 Architecture

Le cinquième critère est employé pour classifier les architectures des différents projets (*Architecture*). Pour cela, les auteurs décrivent cinq architectures possibles, chaque solution pouvant combiner plusieurs styles architecturaux. Les styles architecturaux décrits sont :

Component based (1): la solution est divisée en plusieurs composants faiblement couplés, chacun s'occupant d'une tâche particulière.

Distributed (2): la solution est conçue de manières distribuée, chacune de ses parties communiquant de manière *peer-to-peer*.

Service based (3): la solution est présentée sous la forme de plusieurs services collaborant ensemble.

Node based (4): la solution est constituée d'un logiciel fragmenté, les différents fragments pouvant avoir des capacités similaires ou différentes. Elle permet les déploiements de ces différents fragments, qui vont communiquer et traiter les données de manière collective.

Centralised (5): la solution est centralisée, toutes les couches logicielles se trouvent physiquement au même endroit. Les applications sont développées directement au-dessus de ces couches logicielles. Aucun moyen de communication n'est prévu pour communiquer entre les différentes instances de la solution.

Client-server (6): la solution sépare les fonctions de collecte du contexte et la partie traitement et raisonnement.

2.2.1.6 Historisation des données et stockage

Le sixième critère permet de classifier si l'approche dispose ou non de facilités de stockage et d'historisation des données (*History and Storage*). Le fait de pouvoir stocker les données et garder une trace de leurs changements successifs permet à certaines applications et techniques de raisonnement de mieux comprendre les comportements des utilisateurs, de détecter des patterns dans les données et de pouvoir effectuer des prédictions sur les changements à venir. Ces nouvelles perspectives s'accompagnent d'un coût en termes de performance et de maintenance lié aux évolutions du modèle de contexte. Le symbole ($\surd$) marque le fait que la solution met en place des facilités d'historisation des données.

2.2.1.7 Gestion de la connaissance

Le septième critère permet de classifier si l'approche incorpore une fonctionnalité de gestion de la connaissance (*Knowledge Management*). Le symbole ($\surd$) est utilisé si la solution incorpore des mécanismes spécifiques à la représentation et la gestion de la connaissance tels que des ontologies ou des règles.

2.2.1.8 Détection d'évènements

Le huitième critère met en lumière la capacité de la solution à alerter une application d'un changement sur une information d'intérêt vient de se produire (*Event detection*). Une certaine catégorie d'applications sensibles au contexte peut être considérée comme réactive voir proactive [VSB13], c'est-à-dire qu'un changement dans l'environnement entrainera une série d'actions de la part de l'application. Les applications proactives essayent même d'anticiper ces changements pour effectuer des actions moins coûteuses. Ces changements doivent si possible être rapportés en temps réel aux différents consommateurs de contexte. Une solution permettant de facilement capturer ces changements par le programmeur facilite ainsi le développement. Le symbole ($\surd$) est utilisé si la solution fournit des mécanismes de détection d'évènements.

2.2.1.9 Découverte du contexte et annotations

Le neuvième critère précise si la solution emploie des mécanismes facilitant la découverte du contexte et son annotation (*Context discovery and annotation*). Comme décrit dans le chapitre précédent, l'environnement pervasif est hétérogène et dynamique. Cela implique que de nouvelles sources de données ou actionneurs peuvent venir s'ajouter à l'environnement. Les solutions capables de venir découvrir ces nouvelles sources permettent entre autres d'enrichir

le contexte avec plus d'informations et d'offrir une meilleure vue d'ensemble de la situation aux applications. De plus, l'enrichissement des informations nouvellement acquises avec des métadonnées peut permettre d'effectuer des raisonnements plus précis. Cela est possible car les données possèdent plus de sémantique. Les métadonnées couramment produites sont : un *timestamp*, l'identité de la source, type de contexte ou des informations concernant la qualité. Le processus d'annotation peut s'effectuer de manière automatique par la solution. Les solutions supportant la mise en place de mécanismes de découverte et d'annotation du contexte sont respectivement étiquetées *(D)* et *(A)*.

2.2.1.10 Niveau de sensibilité au contexte

Le dixième critère précise à quel niveau la sensibilité au contexte est utilisé (*Level of context awareness*). Le premier niveau considéré est le niveau matériel où l'information de contexte est principalement utilisée pour effectuer un pré-filtrage des données dans le but de sauvegarder des caractéristiques matérielles (principalement la durée de vie de la batterie l'équipement, la bande passante,...). Le second niveau de sensibilité au contexte envisagé est le niveau logiciel. Celui-ci permet d'utiliser l'information de contexte pour offrir un service à forte valeur ajoutée à un utilisateur. Le niveau de sensibilité de contexte visé et offert par une solution est respectivement noté *(L)* pour le niveau matériel et *(H)* pour le niveau logiciel.

2.2.1.11 Sécurité et vie privée

Le onzième critère permet de valider si un support est fourni par rapport à la gestion de la sécurité et de la confidentialité des données transitant dans la solution (*Security and privacy*). Les données collectées par les solutions de contexte sont sensibles et personnelles car elles révèlent des aspects privés de la vie de l'utilisateur. L'utilisateur veut éventuellement préserver des données telles que sa position ou ses informations de santé. Des mécanismes relatifs à cette préoccupation peuvent être implémentés à différents niveaux : des protocoles de communication sécurisés, un système de stockage de données privées, des politiques et des règles de droits d'accès pour les applications. Le symbole ($\surd$) marque le fait que la solution propose un support, sous une forme quelconque, relatif à la gestion de la sécurité et de la vie privée.

2.2.1.12 Source de données supportées

Le douzième critère permet d'évaluer les différentes sources de données supportées par la solution (*Data source support*). Comme énoncé précédemment, les sources de données sont mobiles et hétérogènes. Un support exhaustif des types de source de données permettra de construire un modèle plus complet. Pour cela les auteurs utilisent la classification des sources de données fournies par [IS03]

***Physical Sensor (P)*:** les capteurs physiques sont ceux présents dans l'environnement. Ils

sont capables de capturer des données environnementales brutes mais peuvent aussi servir à fournir des informations plus abstraites sur la situation qui se produit dans l'environnement, un incendie par exemple. On citera par exemple les photomètres, les capteurs de mouvement ou les détecteurs d'incendie comme capteurs physiques.

***Virtual or Logical Sensor (S)*:** Les capteurs virtuels extraient leurs informations de l'environnement virtuel à travers des services ou des applications. Par exemple, la localisation ou l'activité d'une personne peut être extraite à partir de son agenda ou de ses emails. Les capteurs logiques quant à eux combinent informations physiques et virtuelles pour fournir de nouvelles informations. Par exemple, la localisation d'un employé peut être déduite en analysant si celui-ci est connecté à travers son compte utilisateur à une machine du réseau et en croisant cette information avec une base de données répertoriant la position des différents équipements du réseau.

Si la solution supporte tous les types de sources, on la classifiera à l'aide de l'étiquette *(A)*. Les auteurs ajoutent une classification supplémentaire liée à la mobilité des sources de données : la classification *(M)* est employée si la solution supporte l'intégration d'équipements mobiles.

2.2.1.13 Support de la qualité de contexte

Le treizième critère est établit dans le but d'évaluer la qualité des informations de contexte produites par la solution. Comme énoncé dans la section 2.1.1.3, les informations produites par les différentes sources de contexte ne produisent pas des données précises à cent pour-cent pour diverses raisons : imprécision/perturbation/avarie des capteurs physiques, imprécision des modèles de raisonnement...etc. Pour cela, il est intéressant de discriminer les solutions proposant des mécanismes de résolutions des conflits d'informations et de validation par rapport à celles n'en possédant pas. Les mécanismes de validation servent à s'assurer qu'une donnée soit correct. Pour cela plusieurs techniques peuvent être utilisées: limite, cardinalité, consistance, unicité, échelle,.... Les mécanismes de gestions de conflits servent à s'assurer qu'il y a une consistance entre de multiples données. Un conflit sur une information de contexte se produit lorsque deux sources de contexte fournissent des informations contraires sur une situation ou une entité. Par exemple, si un capteur de présence indique que l'utilisateur se trouve dans sa cuisine tandis que son calendrier signale qu'il se trouve dans sa salle de sport, un conflit d'information apparait. Les solutions supportant des mécanismes de validation des données sont annotées *(V)* et celles supportant des mécanismes de résolution de conflit sont annotées *(C)*.

2.2.1.14 Traitement des données

Le quatorzième critère permet de savoir quelles sont les techniques de traitement de données mises en place dans la solution (*Data Processing*). Comme décrit par [Eva11], dans les environnements ubiquitaires l'énorme quantité de données est liée à la présence d'un grand nombre

de source. Ces capteurs mesurent souvent le même paramètre environnemental. Ainsi des opérations simples telles que l'agrégation ou le filtrage de ces données permettent de réduire la quantité de donnée. Cela permet de rendre plus simple et efficace la sélection des données correspondant aux besoins d'une application. Les solutions usant de mécanismes d'agrégation et de filtrage sont respectivement annotées avec *(A)* et *(F)*.

2.2.1.15 Composition dynamique

Le quinzième critère permet de discriminer les solutions autorisant la composition dynamique des différentes parties prenantes de la solution (*Dynamic Composition*). Comme identifié dans la section 1.2, le développeur, à cause des différentes caractéristiques de l'environnement pervasif, ne peut connaitre au moment du développement les ressources qui vont être disponibles à l'exécution [CLK08]. Le modèle de programmation doit pouvoir exprimer cette variabilité à la conception pour ensuite être capable de la résoudre à l'exécution. Cela permet à la solution d'adapter sa configuration selon les besoins exprimés par les applications. Ainsi de nouveaux collecteurs de contexte, opérateurs de traitement de données ou modèle de raisonnement peuvent être déployés ou reconfigurés si un nouveau besoin apparaît. Ces opérations ne doivent pas affecter la stabilité globale du système. Le symbole ($\surd$) indique que la solution supporte une composition dynamique de ces différentes parties logicielles.

2.2.1.16 Traitement temps réel

Le seizième critère permet d'identifier les solutions proposant une interaction avec le contexte temps réel (*Real time processing*). Certaines applications sensibles au contexte doivent respecter des contraintes temporelles pour fournir un service à l'utilisateur. Comme évoqué dans la section précédente 2.2.1.9 , des facilités liées à la détection d'événements sont un premier pas dans cette direction. Les mécanismes permettant d'agir sur le contexte au travers d'actionneurs doivent aussi prendre en compte ces contraintes pour permettre le bon fonctionnement de cette classe d'application. Le symbole ($\surd$) indique que la solution présente un fonctionnement temps réel.

2.2.1.17 Maintenance d'un registre et service de recherche

Le dix-septième critère permet d'évaluer la présence d'un registre où peuvent s'enregistrer les différentes parties prenantes liées à la solution de contexte (consommateurs de contexte, producteurs de contexte), sa maintenance et des services de recherches associés (*Registry Maintenance and Lookup Services*). Les différents composants pouvant apparaître ou disparaître au cours de l'exécution, la maintenance du registre doit être assurée pour refléter cette dynamique. La présence d'un registre facilite grandement la composition dynamique à l'exécution des applications et la reconfiguration de celle-ci au besoin. Le symbole ($\surd$) indique la présence d'un registre et des fonctionnalités de maintenance et recherche associées.

2.2.2 Analyse du cadre de comparaison

Nous avons présenté un possible cadre de comparaison entre les différentes solutions facilitant la sensibilité au contexte. Cependant, les auteurs ont définis ce cadre indépendamment des paradigmes d'exécutions qui peuvent varier d'une application à l'autre. Pour cela, nous allons proposer une pondération de ces critères basés sur l'environnement d'exécution de la solution. Dans la partie suivante, nous allons présenter brièvement les différentes possibilités d'environnement d'exécution envisagées et nous nous concentrerons ensuite sur la présentation du niveau *Fog Computing* et ses défis particuliers. Nous mettrons ensuite ces défis en perspective avec les critères de comparaison pour ensuite analyser les résultats produits par [Per+14b].

2.3 Fog computing

Cette partie va en premier lieu permettre d'appréhender ce qu'est une plateforme de *Fog Computing*. Pour cela, nous présenterons tout d'abord notre vision des possibilités d'exécution d'une application pervasive en mettant en lumière les différentes infrastructures d'exécution que l'on a pu recenser dans la littérature. Dans un second temps, nous détaillerons les besoins d'une infrastructure *Fog Computing* en termes de contexte. Nous finirons par mettre en relation le cadre de comparaison présenté dans la précédente partie 2.2 et les besoins précédemment déterminés.

2.3.1 Architecture d'une application pervasive

Les architectures des applications pervasives sont largement distribuées, s'exécutant sur des capteurs jusqu'à des facilités de type *Cloud Computing* [SW14] [Bon+12] [AH14]. Une partie du code d'une application peut s'exécuter directement à l'intérieur des capteurs, d'autres sur des passerelles Internet ou *set-top-box* communicant avec toute une infrastructure de capteurs, ou d'autres sur des fermes de serveurs situées dans le *Cloud*. On trouve aussi de plus en plus de machines intermédiaires servant à réduire l'énorme quantité de données transmises aux infrastructures *Cloud*, par l'intermédiaire d'opérations de médiation. Ces architectures et les différents paradigmes mis en jeu sont illustrés par la figure 2.5 .

Cette distribution verticale est due au fait que les applications ont différents besoins en termes de temps de réponse, ressources matérielles et proximité de l'utilisateur [Bon+12]. Nous allons illustrer ces besoins avec deux exemples.

Le premier exemple est tiré de notre partenariat avec *Orange Labs* et traite d'une application de santé rattachée à l'activité d'actimétrie, c'est-à-dire du suivi des personnes âgées et de leurs habitudes [Lal+15]. Deux fonctionnalités majeures peuvent être distinguées dans cette application :

Le diagnostic de maladies neurodégénératives : un diagnostic précoce des maladies neurodégénératives peut être effectué grâce à une étude de la déviation des habitudes d'une personne sur plusieurs mois. Cette première fonctionnalité requiert des analyses complexes et lourdes en en termes de charge processeur, des corrélations d'évènements en utilisant des données issues d'un contexte évoluant lentement sur plusieurs mois.

La supervision temps réel de la personne âgée : un des besoins identifiés est de pouvoir rapidement alerter un proche dans le cas d'activité suspecte : sommeil irrégulier, inactivité prolongée ou détection de chute. Ces données doivent être rapidement mises à disposition de l'application pour que celle-ci puisse réagir et avertir les aidants présents dans la maison par l'intermédiaire d'actionneurs locaux ou par un autre moyen de communication.

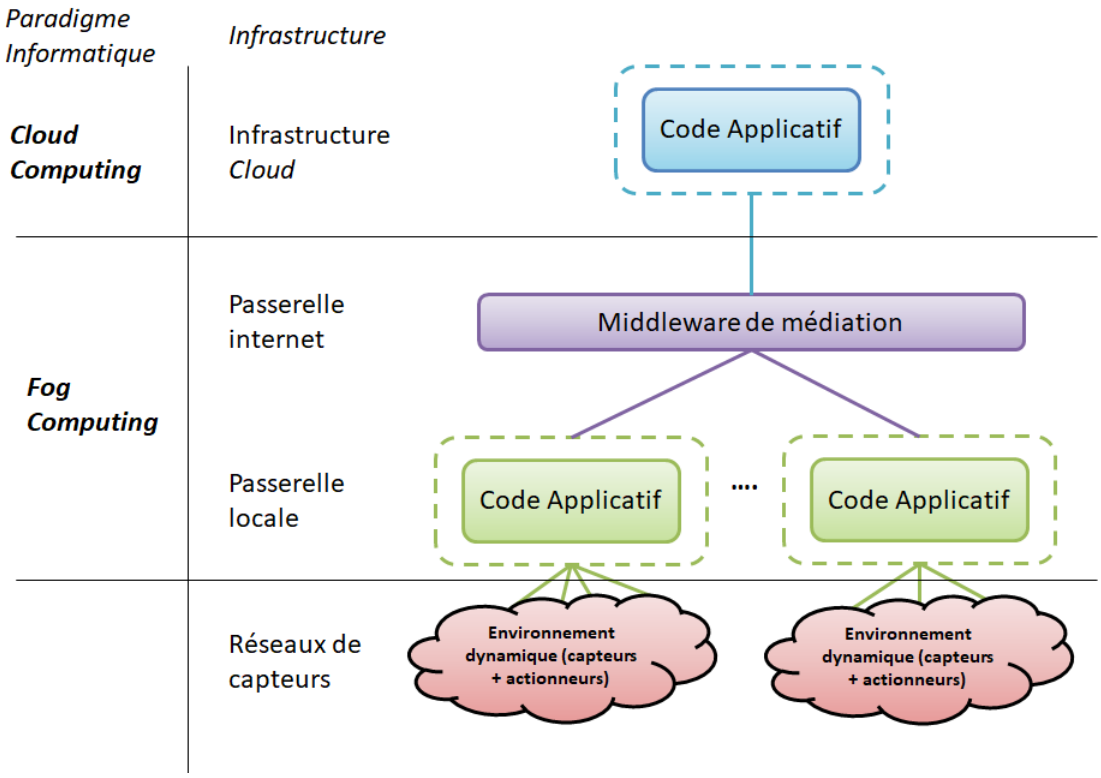

Figure 2.5: Architecture d'une application pervasive

Ces deux fonctionnalités utilisent les mêmes données environnementales mais avec des contraintes temps réel pour la seconde tandis que la première nécessite des ressources matérielles plus conséquentes et une historisation des données.

Le second exemple qu'on a tiré de la littérature [SW14] [Bon+12] concerne les *Smart Grids*. Les applications de répartition de la charge électrique prennent en charge plusieurs fonctionnalités réparties sur diverses plateformes d'exécution. Les différentes fonctionnalités sont :

Des boucles de contrôles locales qui ont pour but de protéger le réseau d'une surcharge en venant agir sur les différents actionneurs ou de changer de source d'énergie si certaines, plus avantageuses économiquement, apparaissent. Les actions doivent être effectuées en temps réel pour être efficaces et requièrent un accès aux actionneurs du réseau. Les données traitées sont quasiment brutes.

Des outils de visualisation temps réel qui permettent un suivi soit instantané ou sur quelques jours des variations de consommation. Les données utilisées ici ont été agrégées et filtrées puis sont stockées de manière semi-permanente. Le délai des actions est de l'ordre de la seconde à la minute.

Des fonctionnalités de *business intelligence* qui permettent une analyse profonde des données et de leur historique pour en dégager des prévisions. Celles-ci nécessitent le stockage des données de manière permanente pour établir des tendances sur plusieurs années.

La première fonctionnalité est exécutée sur des passerelles locales qui sont en charge de collecter les données produites par les capteurs mesurant la consommation électrique. Ces passerelles doivent donc faire face à toute la dynamique et l'hétérogénéité des solutions de collecte et des différents actionneurs. Celles-ci traitent et filtrent certaines données localement puis envoient le reste à des passerelles de médiation chargées de leur agrégation et filtrage. C'est sur ce second groupe de passerelles que s'exécute la seconde fonctionnalité. Une analyse de l'ensemble du réseau est menée dans un dernier temps dans les infrastructures *Cloud* à l'aide d'outils de business intelligence [Bon+12].

A travers l'introduction de ces deux exemples, on se rend compte que les applications pervasives sont complexes, verticalement distribuées et doivent faire face à différents challenges selon la localisation de leur exécution. Les caractéristiques des différents paradigmes d'exécution sont résumés dans [Lua+15] [GR14] [Arm+10] :

***Cloud Computing* :** Les infrastructures *Cloud Computing* sont éloignées physiquement de l'utilisateur et possèdent des capacités de stockage et de traitement élevées et peuvent facilement monter en charge. Cette facilité de montée en charge s'accompagne en contrepartie d'un surcoût économique. Ces infrastructures sont particulièrement appréciées pour des applications nécessitant un stockage permanent, le traitement de don-

nées privées ou des applications effectuant des traitements par lots (*batch processing*) [Arm+10].

Fog Computing **:** Les infrastructures *Fog Computing* sont en interaction directe avec l'environnement utilisateur et possèdent des capacités de stockage et de traitement limitées. Ces infrastructures hébergent principalement des fonctionnalités ayant des contraintes temps réel facilitées par la proximité de l'environnement mais aussi des opérations de médiation coûteuses [Bon+12].

De plus, comme nous avons pu le constater des informations de contexte sont nécessaires au fonctionnement des applications indépendamment de l'endroit où celles-ci s'exécutent comme illustré par la figure 2.6 . Les solutions infusant la sensibilité au contexte dans la logique fonctionnelle de l'application sont différentes dans leur forme pour répondre au mieux aux caractéristiques et besoins du paradigme d'exécution. Elles peuvent varier en termes de patron d'interaction entre producteur et consommateur de contexte, de formalisme, de capacité de raisonnement, contraintes temps réel, autonomie,...etc. Pour bien cerner les attentes spécifiques du *Fog Computing* nous nous attarderons sur sa présentation dans la partie suivante.

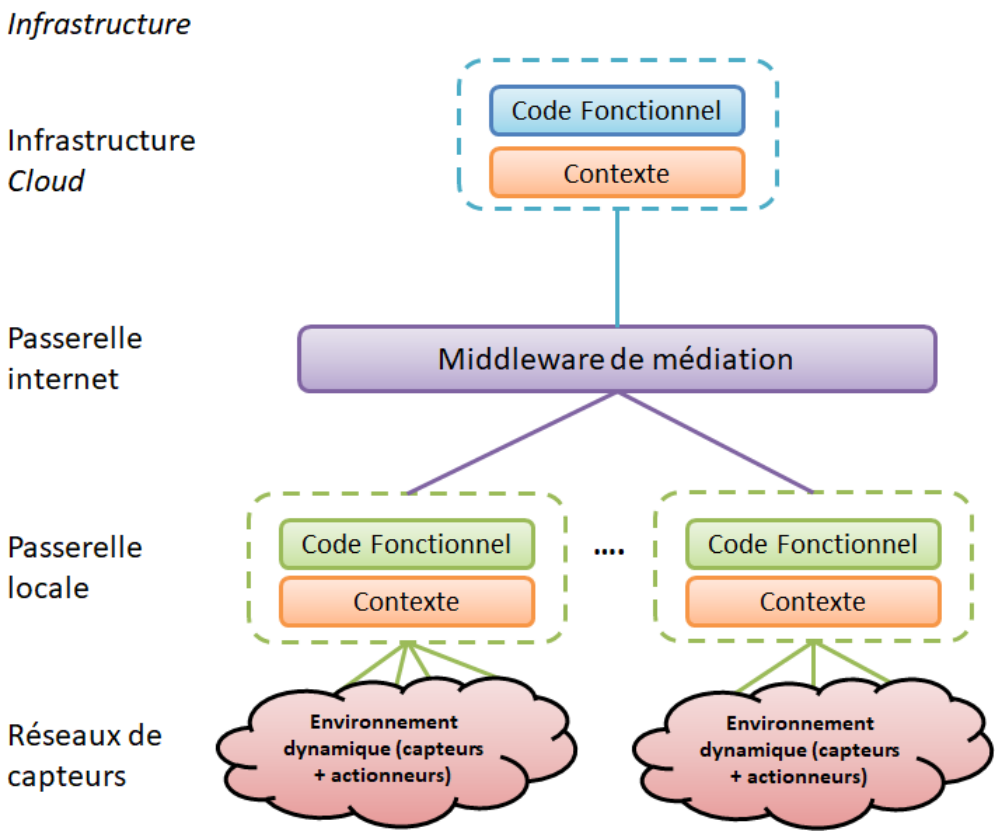

Figure 2.6: Architecture d'une application pervasive avec module de contexte

2.3.2 Fog computing et contexte

Le terme *Fog Computing* a récemment émergé par l'intermédiaire de Cisco en 2012 [Bon+12]. Des propositions similaires peuvent aussi être trouvées sous différents noms comme *Cyber Foraging* [Bal+02], *Cloudlets* [Sat+09], *mobile edge computing* [Pat+14]. L'idée principale du *Fog Computing* est d'avoir des facilités semblables à celles du *Cloud Computing* mais plus légères, placées à proximité de l'utilisateur pour délivrer des actions temps réel à l'utilisateur [Lua+15] comme cela est illustré dans la figure 2.5. Une définition formelle du Fog Computing a été proposée dans [Bon+12]:

« Fog computing is a highly virtualized platform that provides compute, storage, and networking services between end devices and traditional Cloud Computing Data Centers, typically, but not exclusively located at the edge of the network.»

Le *Cloud Computing* a permis aux entreprises de libérer les entreprises et les utilisateurs finaux des préoccupations telles que les ressources de stockage, les limites de puissance de calcul ou du coût des communications [Arm+10]. En adoptant un modèle qualifié de *«pay-as-you-go»* ou *«pay-as-you-use»*, les préoccupations citées précédemment sont seulement d'ordre économique : si un nouveau besoin apparait, il suffit de payer plus pour le combler. Mais, dans le cadre des applications ayant des contraintes temporelles et interagissant directement avec l'environnement utilisateur, ce modèle n'est pas adapté. En effet il peine à satisfaire les contraintes temps réel et supporte mal la mobilité des utilisateurs et des équipements [Bon+12] [GR14].

Le *Fog Computing* est apparu pour satisfaire cette demande et venir apporter une approche complémentaire aux solutions exclusivement basées sur une infrastructure *Cloud Computing*. Il introduit ainsi des nœuds à l'extrémité du réseau permettant de satisfaire les contraintes temporelles de cette classe d'applications. De plus, ces nœuds apportent d'autres avantages, ils permettent d'éviter un gaspillage des différentes ressources : bande passante de l'infrastructure *Cloud*, énergie du réseau de capteur etc... . Le nombre de données transitant vers le *Cloud* peut être limité par exemple en les filtrant et en les prétraitant localement [AH14]. Si les informations produites par les différents capteurs ne sont pas utilisées temporairement, les fréquences de synchronisation peuvent être ajustées pour préserver la durée de vie de ces équipements. De plus, l'interface réseau de ces nœuds n'est plus limitée au support des protocoles *IP* et permet d'intégrer un nombre supérieur d'équipements communiquant par *Zigbee*, *Zwave*, *Bluetooth*, *ANT+* etc....

Une plateforme de type *Fog Computing* est donc faite pour héberger plusieurs applications, ayant des contraintes temps réel. Ces applications sont par essence sensibles au contexte car elles utilisent des informations extraites de l'environnement, principalement à l'aide de

capteurs. D'après [Bon+12] [GR14], nous avons établi la liste d'exigences à mettre en avant pour une solution de contexte appliquée au *Fog Computing*:

Les applications sont réactives. Elles ont besoins d'être alertées en cas de changement d'une information de contexte et ce de manière temps réel. Les changements opérés par l'intermédiaire d'actionneurs doivent aussi s'effectuer avec le minimum de délai.

Les applications et la solution de sensibilité au contexte doivent être couplées au minimum. Les applications ne doivent pas être conscientes des diverses sources d'information. Elles doivent seulement être conscientes de la disponibilité de l'information recherchée. Cette caractéristique facilite grandement la maintenance des deux parties.

Réduction au minimum de l'intervention humaine. Les opérations de maintenance et d'optimisation sont extrêmement complexes et mettent en jeu des connaissances très diverses. L'absence d'administrateurs assez qualifiés reliée au fait qu'une partie des ressources changent continuellement fait que la solution doit présenter une autonomie accrue. Cette autonomie prend de multiples formes. La solution doit pouvoir répondre aux besoins de nouvelles applications si celles-ci apparaissent, être capables d'ajuster divers paramètres liés à la synchronisation si les données ne sont plus utilisées ou aller même jusqu'à la désactivation (ou activation) de certains capteurs pour diminuer la consommation d'énergie des équipements.

Haute volatilité et forte hétérogénéité de l'environnement. L'infrastructure de *Fog Computing* doit en permanence superviser un environnement toujours changeant et très hétérogène. Pour cela, la solution doit être capable d'intégrer un maximum de sources de contexte et être en mesure de les découvrir à l'exécution.

Solution de médiation. Comme présenté précédemment, un des rôles de la solution de sensibilité au contexte est de pouvoir réduire la quantité d'informations produites par l'environnement. Cela permet de diminuer considérablement le trafic entre les différentes plateformes et aussi de décharger de cette responsabilité les applications s'exécutant localement.

2.3.3 Mise en perspective avec le cadre de comparaison

Nous allons maintenant établir la correspondance entre les exigences du *Fog Computing* et les critères utilisés dans le cadre de comparaison, fournis par [Per+14b], ceux-ci étant rappelés dans la figure 2.7 :

Réactivité : La solution doit impérativement fournir des mécanismes de détection d'évènements (*Event detection*) et un mode de fonctionnement orienté temps réel (*Real time processing*).

Couplage minimum : un moyen d'obtenir un couplage minimum entre la solution et les applications est l'utilisation d'un registre extérieur où les différentes parties prenantes

viennent enregistrer des définitions abstraites des fonctionnalités fournies. Le critère mis en avant pour cette partie est la présence d'un registre externe et sa maintenance (*Registry Maintenance*).

Autonomie : Les critères reflétant le degré d'autonomie de la solution sont d'une part la composition dynamique pour répondre aux besoins des applications (*Dynamic composition*) et l'utilisation du contexte à tous les niveaux (*Level of context awareness*).

Haute volatilité et forte hétérogénéité : La solution pour prendre en charge l'hétérogénéité de l'environnement doit supporter le plus de types de sources possibles (*Data Source support*) et venir les découvrir à l'exécution (*Context Discovery*).

Solution de médiation : Le critère reflétant les possibilités de médiation de la solution est celui concernant le traitement des données (*Data processing*). La solution doit être au minimum capable de supporter le filtrage et l'agrégation des données.

Taxonomy	Description
Modelling	Key-value modelling (K), Markup schemes (M), Graphical modelling (G), Object oriented modelling (Ob), Logic-based modelling (L), and Ontology-based modelling (On)
Reasoning	Supervised learning (S), Un-supervised learning (U), rules (R), Fuzzy logic (F), Ontology-based (O), and Probabilistic reasoning (P)
Distribution	Publish/subscribe (P) and Query (Q)
Architecture	Component based architecture (1) , Distributed architecture (2), Service based architecture (3), Node based architecture (4) , Centralised architecture (5), Client-server architecture (6)
History and Storage	Available (✓)
Knowledge Management	Available (✓)
Event Detection	Available (✓)
Context Discovery and Annotation	context Discovery (D) and context Annotation (A)
Level of Context Awareness	High level (H) and Low level (L).
Security and Privacy	Available (✓)
Data Source Support	Physical sensors (P), Software sensors (S), Mobile devices (M), Any type of sensor (A)
Quality of Context	Conflict resolution (C), context Validation (V)
Data Processing	Aggregate (A), Filter (F)
Dynamic Composition	Available (✓)
Real Time Processing	Available (✓)
Registry Maintenance	Available (✓)

Figure 2.7: Rappel des différents critères de comparaison des solutions [Per+14b]

2.3.4 Analyse des résultats du cadre de comparaison

Les résultats de l'étude menée par [Per+14b] sur 50 projets sont disponibles dans la figure 2.8.

Comme on peut le constater, aucune des solutions proposées ne remplit totalement les critères mis en avant par le *Fog Computing*. Il est à noter que le support pour l'autonomie est particulièrement peu présent alors que c'est une des caractéristiques centrales du *Fog*

TABLE XIII
EVALUATION OF SURVEYED RESEARCH PROTOTYPES, SYSTEMS, AND APPROACHES

Project Name	Citations	Year	Project Focus	Modelling	Reasoning	Distribution	Architecture	History and Storage	Knowledge Management	Event Detection	Context Discovery and Annotation	Level of Context Awareness	Security and Privacy	Data Source Support	Quality of Context	Data Processing	Dynamic Composition	Real Time Processing	Registry Maintenance
(1)	(2)	(3)	(4)	(5)	(6)	(7)	(8)	(9)	(10)	(11)	(12)	(13)	(14)	(15)	(16)	(17)	(18)	(19)	(20)
Context Toolkit	[72]	2001	T	K	✓	Q	1,5	✓	–	–	–	H	–	A	–	A	–	–	–
Solar	[184]	2002	M	K,M,Ob	R	P	2	–	–	✓	D	H	✓	P	✓	A	✓	–	–
Aura	[191]	2002	M	M	R	P	2	–	–	✓	D	H	–	A	–	–	–	–	✓
CoOL	[192]	2003	T	On	R,O	Q	1	–	✓	✓	D	H	–	S	–	–	–	–	✓
CARISMA	[193]	2003	M	M	R	Q	2	–	–	–	–	H	–	M	C	–	–	–	–
CoBrA	[119]	2004	M	On	R,O	Q	1	✓	✓	✓	–	H	✓	A	–	–	–	–	–
Gaia	[168]	2004	M	F,On	S,P, F	Q	2,3	✓	✓	✓	D	H	✓	A	–	–	✓	–	✓
SOCAM	[194]	2004	M	On	R,O	Q,P	3	✓	✓	✓	D	H	–	A	–	A	–	–	✓
CARS	[195]	2005	S	K	U	–	–	–	–	✓	A	H	–	P	–	–	–	–	–
CASN	[188]	2005	M	F,On	F,O	P	2	–	✓	–	D	L	–	P	–	–	–	–	–
SCK	[142]	2005	M	M,On	R,O	Q	1	✓	✓	✓	A,D	H	–	A	V	–	–	–	✓
TRAILBLAZER	[196]	2005	S	K	R	Q	2	–	–	–	D	L	–	P	–	–	–	–	–
BIONETS	[197]	2006	M	On	R,O	Q	1	–	✓	–	A	H	–	A	–	–	–	–	–
PROCON	[86]	2006	S	K	R	Q	2	–	–	✓	D	L	–	P	–	A,F	–	–	–
CMF (MAGNET)	[85]	2006	M	M	R	P,Q	2,4	✓	–	–	D	H	–	A	C	–	✓	–	–
e-SENSE	[44]	2006	M	–	R	Q	2,4	–	✓	–	D	H	✓	P	–	F	–	–	–
HCoM	[198]	2007	M	G,On	R,O	Q	5	✓	✓	–	D	H	–	S	V	F	–	–	✓
CMS	[106]	2007	M	On	O	P,Q	1,2	✓	–	✓	S	H	–	A	–	A	–	–	✓
MoCA	[199]	2007	M	M,Ob	O	P,Q	4,5	–	–	✓	D	H	✓	A	V	–	–	✓	✓
CaSP	[185]	2007	M	M,On	O	P,Q	6	✓	–	–	D	H	–	A	–	–	–	–	✓
SIM	[200]	2007	M	K,G	R	–	2	✓	–	–	–	H	–	P	C	A	–	–	–
—	[124]	2007	M	On	O	Q		–	–	✓	D	H	–	P	V	A	–	–	–
COSMOS	[201]	2008	M	Ob	R	Q	2,4	–	–	✓	–	H	–	P	–	A	✓	–	✓
DMS-CA	[202]	2008	S	M	R	Q	5	–	–	✓	–	H	–	A	–	–	–	–	–
CDMS	[203]	2008	M	K,M	R	Q	2	✓	–	✓	D	H	–	A	–	A,F	–	–	✓
—	[141]	2008	M	On	O,P	Q	5	–	✓	–	D	H	–	–	V	–	–	–	–
—	[204]	2008	M	On	R,O	P,Q	5	–	–	✓	D	H	–	P	–	A	–	–	–
AcoMS	[88]	2008	M	M,G,On	R,O	P	5	–	✓	✓	A	H	–	P	–	–	–	–	✓
CROCO	[118]	2008	M	On	R,O	Q		✓	✓	–	D	H	✓	A	C,V	–	–	–	✓
EmoCASN	[205]	2008	S	K	R	Q	2,4	–	–	–	D	L	–	P	–	–	–	–	–
Hydra	[61]	2009	M	K,On,Ob	R,O	Q	3	✓	✓	✓	–	H	✓	P	V	–	–	–	–
UPnP	[206]	2009	M	K,M	R	Q	4	✓	–	✓	D	H	✓	A	–	A	✓	–	✓
COSAR	[151]	2009	M	On	S,O	Q	5	–	✓	✓	A	H	–	P	–	–	–	–	–
SPBCA	[161]	2009	M	On	R,O	Q	2	–	–	✓	A	H	✓	A	–	–	–	–	–
C-CAST	[207]	2009	M	M	R	P,Q	5	✓	–	✓	D	H	–	A	–	–	–	–	✓
—	[208]	2009	M	On	O	P	5	✓	–	✓	D	H	–	A	–	A	–	–	–
CDA	[209]	2009	M	Ob	–	Q	4,6	–	–	–	–	H	–	V	–	–	–	–	✓
SALES	[210]	2009	M	M	R	Q	2,4	–	–	✓	D	L	–	P	–	F	–	–	✓
MidSen	[52]	2009	M	K	R	P,Q	5	–	✓	✓	D	H	–	P	–	–	–	–	✓
SCONSTREAM	[211]	2010	S	G	R	Q	5	✓	–	✓	–	H	–	P	–	–	–	✓	–
—	[101]	2010	M	M	P	Q	2,4	✓	–	✓	–	H	–	A	–	F	✓	–	–
Feel@Home	[212]	2010	M	G,On	O	P,Q	2,4	–	✓	✓	–	H	✓	A	–	–	–	–	✓
CoMiHoC	[213]	2010	M	Ob	R,P	Q	5	–	✓	✓	D	H	–	A	V	–	–	–	–
Intelligibility	[108]	2010	T	–	R,S,P	Q	1,5	–	–	✓	D	H	–	A	V	–	–	–	–
ezContext	[105]	2010	M	K,Ob	R	Q	5	✓	✓	✓	–	H	–	A	–	A	–	–	✓
UbiQuSE	[214]	2010	M	M	R	Q	5	✓	–	✓	D,A	H	–	A	–	–	–	✓	–
COPAL	[215]	2010	M	M	R	P,Q	1,5	–	–	✓	D	H	✓		V	A,F	–	✓	✓
Octopus	[50]	2011	S	✓	✓	P	2,4	–	–	✓	D	H	–	A	–	A	✓	–	–
—	[216]	2011	M	–	✓	P	2	–	–	–	D	H	–	P	–	A	–	–	✓
—	[153]	2011	S	K,Ob	S,P		2,4	✓	✓	✓	D,A	H	–	M	V	A,F	–	–	✓

Figure 2.8: Résultats de la comparaison proposé par [Per+14b] avec mise en avant des critères prépondérant du *Fog Computing*

Computing. De plus, on peut remarquer que la grande majorité des approches est conçue uniquement pour fournir des informations de contexte aux consommateurs mais que très peu offre des moyens pour d'influencer le contexte par l'intermédiaire d'actionneurs [Van15].

Dans la partie suivante, nous allons enrichir cette étude comparative avec de nouvelles solutions.

2.4 Comparaison de différentes approches

En addition des résultats présentés et disponibles dans la section 2.3.4, nous allons présenter deux approches mettant en place la sensibilité au contexte. La sélection de ces deux approches a été établie sur le fait que celles-ci ont eu des domaines d'application proches du Fog Computing et qu'elles permettent d'effectuer des actions sur le contexte.

2.4.1 DiaSuite

DiaSuite [Ber+14] [Cas+11] [Ena+13] se définit comme une méthodologie servant au développement d'applications ubiquitaires suivant le patron *Sense-Compute-Control* [MT10]. Pour cela, cette solution prend plusieurs formes : un langage de description nommé DiaSpec [Cas+11], son compilateur associé DiaGen, un *framework* d'exécution et un outil de simulation aidant au développement d'applications pervasives DiaSim [BJC09].

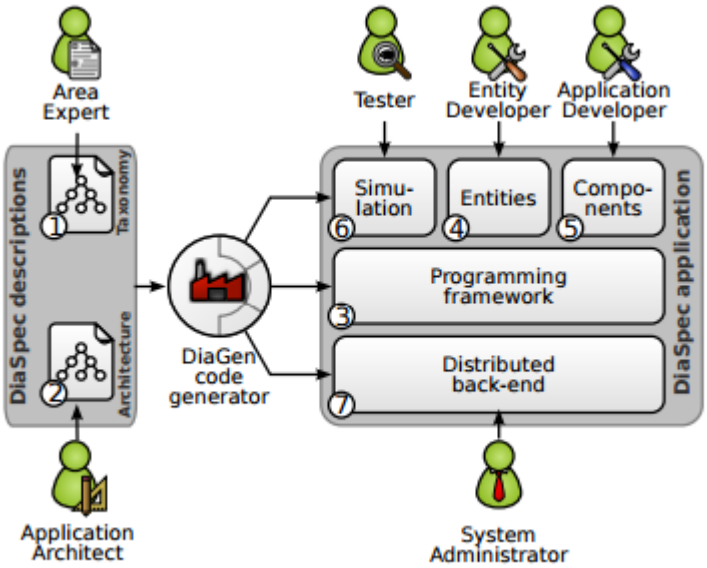

Figure 2.9: Cycle de vie d'un développement DiaSuite [Cas+11]

Le cycle de vie d'un développement DiaSuite est présenté dans la figure 2.9 . Tout d'abord le langage DiaSpec est utilisé par deux acteurs :

L'expert du domaine spécifie quelles sont les entités dont la présence est requise dans l'environnement avec par exemple les équipements, les informations qu'ils produisent, les actions qu'ils peuvent exécuter.

L'architecte logiciel spécifie quels composants vont être utilisés par l'application en plus des composants décrits par l'expert du domaine. Les composants à décrire sont les

composants de contexte et les contrôleurs. Ces composants à la différence des précédents contiennent une spécification des différents liens les unissant.

A partir de ces descriptions, le compilateur DiaSuite va générer un squelette de programmation servant à guider l'implémentation. Le code complété ainsi que la spécification DiaSpec sont ensuite utilisés par le *framework* qui va exécuter et lier les différentes instances de composants. L'application peut être testée grâce au simulateur DiaSim par la suite.

Pour créer une application sensible au contexte, les développeurs utilisent les 4 types d'entités suivantes, illustré par la figure 2.10 :

Les entités en charge de la collecte : Elles ont pour rôle de venir capturer les différentes informations brutes liées au contexte. Comme elles sont entièrement programmées, le support de tous les types de sources est possible. Elles supportent des mécanismes de synchronisation de types push et pull. Les informations collectées sont exclusivement transmises aux opérateurs de contexte.

Les opérateurs de contexte : Ces composants sont en charge de donner de la sémantique aux informations fournies par les collecteurs de contexte ou de raffiner celle d'autres opérateurs de contexte. Leurs différentes dépendances envers les autres composants sont décrites explicitement ce qui n'autorise pas la substitution efficace des implémentations. Cela ne permet pas de mettre en place la propriété de composition dynamique. De plus, ces composants sont utilisés pour programmer des mécanismes de filtrage ou d'agrégation des données dans différents Callbacks. Ces Callbacks sont appelés à chaque changement d'une valeur liée à une dépendance. Ces entités sont uniquement réactives et ne peuvent opérer des synchronisations de type Pull, elles attendent d'être notifiées par l'intermédiaire des Callbacks pour exécuter leur logique de médiation.

Les contrôleurs : ils représentent la logique applicative et prennent des décisions en fonction des informations fournies par les opérateurs de contexte. Ils décrivent explicitement leurs dépendances en termes de contexte et sont notifiés des changements par l'intermédiaire de Callbacks.

Les entités en charge des actions : elles sont invoquées par les contrôleurs pour déclencher des actions sur l'environnement.

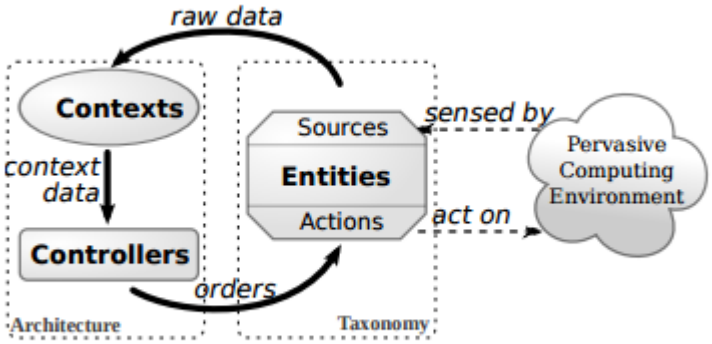

Figure 2.10: Modèle Architectural de DiaSuite et exemple [CBC10]

On notera que dans [Bru+11], où l'approche DiaSuite est appliquée à une simulation aéronautique, des contraintes pouvant être qualifiées de temps réel sont respectées. Un résumé de l'approche DiaSuite appliqué à la gestion de contexte est disponible ci-dessous :

Project Focus	M
Modelling	Object
Reasonning	√
Distribution	(P)
Architecture	(1)(5)
History and Storage	-
Knowledge Management	-
Event Detection	√
Discovery and Annotation	-
Level of Context Awareness	H
Security and Privacy	-
Data Source Support	A
Quality of Context	-
Data Processing	(A)(F)
Dynamic Composition	-
Real Time Processing	√
Registry maintenance	-

Table 2.1: Cadre de comparaison appliqué à l'approche *DiaSuite*

2.4.2 System support for proactive adaptation

L'université de Mannheim propose à travers plusieurs contributions [Van15] [Van13] [VSB13] [Van+13] un support pour effectuer des adaptations proactives. Leur but est d'anticiper les changements liés au contexte pour pouvoir effectuer une adaptation en amont avec possiblement une opération sur un actionneur. Ces adaptations anticipées contrairement aux adaptations réactives permettent de ne pas se préoccuper des contraintes temps réel, de les coordonner plus facilement et ainsi éviter les conflits [Van15]. L'approche est construite au-dessus du *middleware* BASE [Bec+03] qui repose sur un *micro-broker* en charge de répartir les invocations des différents objets entre services locaux et services distants par l'intermédiaire de *plug-in* de transport.

L'architecture de la solution est présentée dans la figure 2.11. Elle propose de modéliser le contexte à l'aide de variables d'environnement, ce qui se rapproche d'un modèle clé-valeur. Ce modèle est interrogé par les différents consommateurs de contexte à l'aide d'un langage de requête spécifique. Il existe plusieurs types de requête permettant entre autre d'avoir des informations sur le contexte prédites ou instantanées, d'établir les possibilités d'influencer le contexte à travers des actionneurs et d'effectuer ces actions. L'exécution de ces requêtes est effectuée en fonction des différents composants de contexte présents au moment de l'exécution, ce qui permet d'obtenir la propriété de composition dynamique.

L'architecture est divisée en trois tiers : le tiers applicatif qui contient les composants applicatifs, qui est un des types de composants consommateurs de contexte. Le tiers contexte qui contient un ensemble de composants chargé du management du contexte. Le dernier tiers étant le *middleware* BASE présenté au-dessus. Nous allons détailler plus particulièrement les composants faisant partie du tiers contexte pour en ressortir les principales caractéristiques :

Context Broker **:** C'est le composant qui sert de point d'entrée aux consommateurs de contexte : les applications et les composants de prédiction de contexte. Il interprète leurs différentes requêtes qui peuvent prendre la forme de demande d'information, prédiction, souscription à des évènements et les transmet au composant concerné.

Context Information Component **:** Ces composants servent à représenter et acquérir le contexte au travers des composants représentant les capteurs, les composants de prédiction présents dans le registre de service de contexte ou une base de données. Cette dernière permet de sauvegarder l'historique des évolutions de contexte.

Context Adaptation Component **:** Ces composants ont la responsabilité de déterminer quelle variable de contexte peut être adaptée, soit immédiatement, soit dans le futur en se basant sur une prédiction. Ils peuvent donc s'appuyer sur les composants de prédiction ou les actionneurs présents dans le registre de service de contexte pour déterminer et effectuer les possibles adaptations de contexte.

Context Subscription Component **:** Les composants de souscription servent à alerter les différents consommateurs de contexte qui ont souscrit à au changement d'une variable de contexte.

Context Services **:** Les services de contexte partagent une interface commune permettant de connaitre leur localisation, la variable de contexte qu'ils supervisent et leurs types spécifiques. Les trois types de services sont capteurs, actionneurs et services de prédiction. Ils sont stockés dans un registre de service qui permet de refléter la dynamique du système.

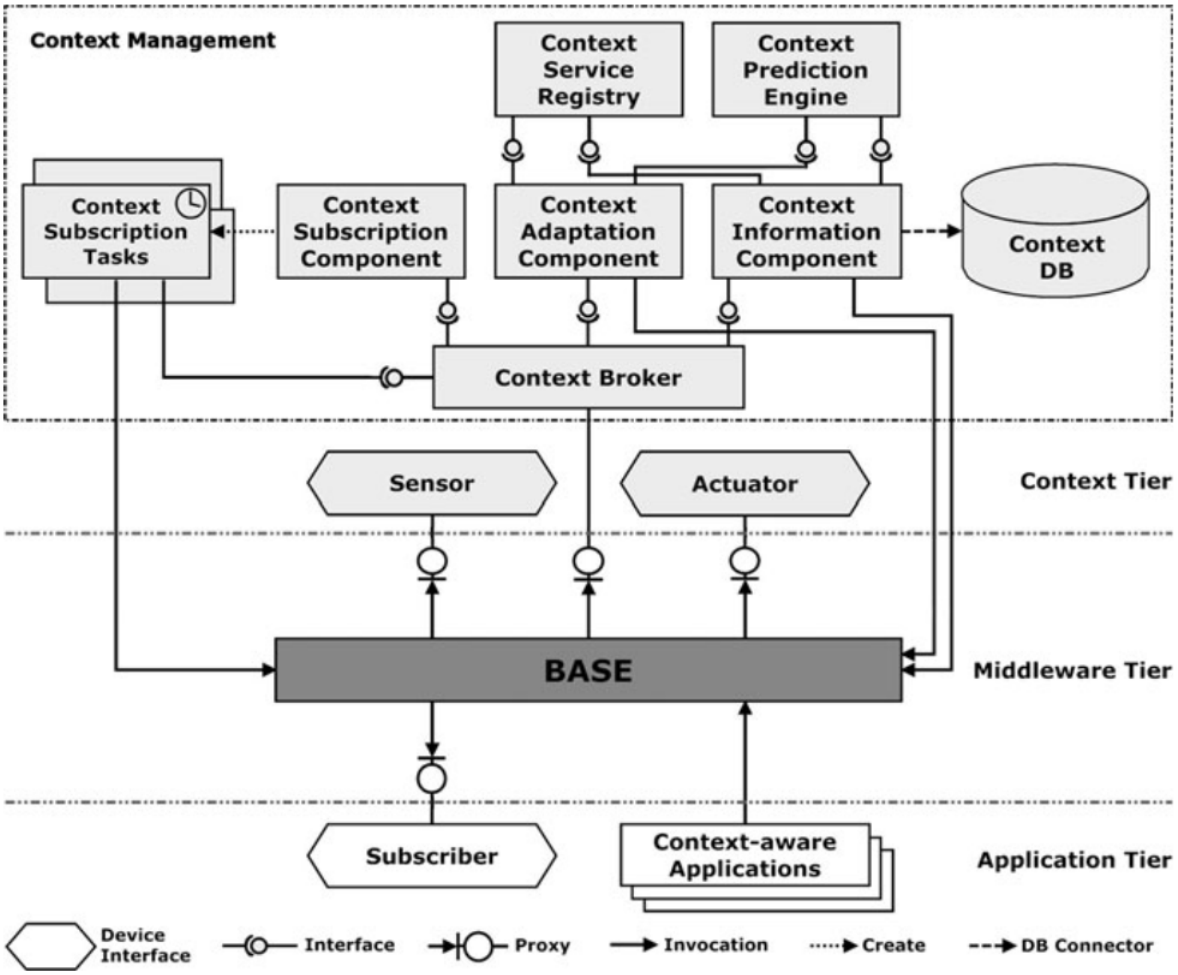

Figure 2.11: Architecture du système d'adaptation proactive [VSB13]

L'application du cadre de comparaison à cette approche est résumée ci-dessous :

Project Focus	M
Modelling	K
Reasonning	√
Distribution	Q
Architecture	(1)(2)(5)
History and Storage	√
Knowledge Management	-
Event Detection	√
Discovery and Annotation	D
Level of Context Awareness	H
Security and Privacy	-
Data Source Support	P
Quality of Context	-
Data Processing	-
Dynamic Composition	√
Real Time Processing	-
Registry maintenance	√

Table 2.2: Cadre de comparaison appliqué à l'approche *System support for proactive adaptation*

2.5 Conclusion

Ce chapitre a permis de présenter les grandes idées derrière la notion de contexte et de sensibilité au contexte, c'est-à-dire la capacité d'une application à adapter son comportement et les services qu'elle fournit en fonction de l'environnement. Dû à la longévité du sujet de recherche, nous avons pu voir que plusieurs patrons sont largement acceptés et considérés comme acquis par la communauté :

- L'information de contexte possède un cycle de vie qui lui est propre, distinct de celui de l'application.
- La solution la plus acceptée est de déplacer en dehors des applications la complexité liée aux différentes tâches de ce cycle de vie dans un *middleware*.

Nous avons pu voir que dans le cadre de nos travaux, en raison de la distribution des applications au sein de plusieurs infrastructures d'exécution, *Cloud Computing* et *Fog Computing* , plusieurs modules de contexte souvent différents en termes de fonctionnalités requises et d'utilisations sont disséminés au sein de ces infrastructures. Nous nous sommes particulièrement intéressés au paradigme du *Fog Computing*, qui est l'idée d'avoir une plateforme légère à proximité de l'utilisateur permettant entre autres de délivrer des actions temps réel à l'utilisateur et de réduire l'énorme quantité de données produites par les capteurs de l'environnement au travers d'un processus de médiation. Notre état de l'art s'est basé sur un cadre de comparaison fourni par la littérature et l'on peut établir plusieurs limites aux solutions proposées :

- Les solutions actuelles n'offrent pas toutes les caractéristiques requises pour faciliter un développement rapide, efficace et à moindre coût des applications *Fog Computing*. On pointera particulièrement le manque d'autonomie de la majorité des solutions et la faible tolérance de celles-ci aux contraintes temps réel. De plus comme nous avons pu le noter peu de solutions offrent la possibilité de venir effectuer des actions de façon simple sur l'environnement, ce qui est caractéristique de ce type d'application.
- Les solutions les plus autonomes sont peu flexibles et leur élargissement à de nouveaux besoins n'est souvent peu ou pas guidé. Comme noté dans le chapitre précédent, l'ajout de nouveaux besoins est une tâche infinie due aux caractéristiques de l'environnement pervasif. Or, même si la complexité liée au cycle de vie du contexte a été déplacée, elle reste présente. Un support est donc indispensable pour accompagner cette flexibilité et permettre au module de contexte de rester maintenable.

Ces observations ont motivé le présent travail de recherche. Le prochain chapitre va permettre d'établir notre proposition, à partir d'objectifs clairement identifiés, d'un support à la définition et l'implémentation d'un module de contexte autonome, maintenable et flexible.

Chapter 3

Proposition

Sommaire

Dans ce troisième chapitre, notre but est d'établir la proposition de recherche du présent manuscrit.

Pour cela, nous reformulerons le problème de recherche de manière concise et lui associerons une stratégie de recherche. La partie suivante servira de base à notre proposition et en reprendra les caractéristiques essentielles. On y retrouvera notre volonté que le modèle du contexte soit exprimé en tant qu'un ensemble de services rendus disponibles à l'exécution selon les besoins des applications et des contraintes de la plateforme de *Fog Computing*. Un modèle à composant spécialisé est fourni pour permettre un développement simple et homogène de ces services. Une fois instancié, celui-ci garantit une exécution administrable. A l'exécution, ce modèle est intégré dans une infrastructure plus complexe, permettant la mise en place de comportement autonomique.

3.1 Introduction

3.1.1 Enoncé du problème

La sensibilité au contexte a pour objet la production d'applications plus sensibles à leur environnement, au sens large, et capables de s'adapter pour réagir aux changements qui y apparaissent au fil de l'exécution. Comme nous avons pu le voir au long du chapitre 2 , la gestion du contexte est une tâche très technique qui se doit de satisfaire un certain nombre d'exigences fonctionnelles. De plus comme nous l'avons avancé, certaines de ces exigences sont exacerbées selon l'environnement d'exécution et la topologie de toute l'infrastructure pervasive. En effet, les applications pervasives s'exécutant sur une plateforme de type Fog Computing requièrent une solution de contexte exhibant de fortes qualités d'autonomie, des possibilités d'effectuer des interactions bidirectionnelles en respectant certaines contraintes temps réel.

L'architecture globale d'une solution facilitant le développement de contexte proposé et acceptée par la communauté est de déplacer la complexité de la gestion de contexte, soit les phases de collecte, modélisation et dissémination des informations de contexte ainsi que des facilités de raisonnement, dans une couche logicielle dédiée : un *middleware*.

Cependant comme nous avons pu le voir, les solutions existantes ont des lacunes vis-à-vis des fonctionnalités exigées par les environnements de type *Fog Computing*. Nous résumerons ces lacunes dans les points suivants :

Tâche complexe: Les systèmes sensibles au contexte sont par nature plus complexes que les systèmes traditionnels. La majeure partie de la complexité technique est absorbée par le middleware de contexte délivrant ainsi les développeurs applicatifs de plusieurs fardeaux. Mais cette complexité ne disparait pas, elle est laissée au développeur du module de contexte. Or, la gestion de contexte est une tâche complexe et demande un niveau de compétence rarement disponible de par le large spectre des technologies mises en jeu. Cependant nous avons pu noter que contrairement à la dissémination et au raisonnement, il y a un manque de support pour la tâche collecte de contexte. Bien que certains patrons de synchronisation avec des entités de l'environnement aient été formalisés, cette tâche reste souvent totalement à la charge du développeur et effectuée de manière ad-hoc. Il faut donc reconnaitre que la construction d'un module de contexte reste une tâche difficile et coûteuse.

Evolution difficile: Le fait que la totalité des applications s'exécutant dans les environnements *Fog Computing* n'est pas connue à l'avance, les besoins en contexte vont naturellement évoluer en fonction de la présence ou non d'une application. Cela signifie par exemple que aussi bien les propriétés fonctionnelles que les propriétés non fonctionnelles associées à une plateforme *Fog Computing* ne sont pas figées et demandent des adapta-

tions et des évolutions au cours du temps. Or, la majorité des solutions n'offrent pas ou peu de support à l'extensibilité. Il est particulièrement difficile de rajouter des nouveaux éléments, propriétés fonctionnelles ou non fonctionnelles, sans devoir faire une refonte totale de l'architecture. De plus certaines propriétés, comme par exemple la gestion des conflits ou la sécurité, doivent être infusées à une granularité très fine dans la solution, souvent au niveau de la ressource de contexte, et une approche où l'on empilerait des couches de nouveau middleware ne convient pas forcément.

Maintenance complexe: Les activités de maintenance sont d'une haute technicité. Elles demandent souvent l'arrêt des opérations et des techniciens qualifiés doivent se déplacer. Cette complexité accrue, conjuguée aux lacunes précédentes, contribue à diminuer la productivité et à augmenter les coûts de maintenance. Or, trop peu de solutions disposent d'un réel niveau d'autonomie permettant d'alléger cette tâche.

Cependant, il ne faut pas perdre de vue que les systèmes sensibles au contexte sont soumis aux mêmes exigences que les autres systèmes informatiques. Ceux-ci doivent pouvoir être produits pour pouvoir exhiber un certain nombre de qualités comme la maintenabilité, des performances adaptées à l'environnement d'exécution, une haute disponibilité tout en subissant les contraintes des développements logiciels actuelles, c'est-à-dire un *time-to-market* court, l'intégration de technologies toujours plus nombreuses, etc... Il est donc intéressant de fournir au développeur de systèmes sensibles au contexte une solution facile d'utilisation, extensible et autonome pour pallier à ces lacunes.

3.1.2 Objectifs de recherche

Les solutions actuelles de sensibilité au contexte peinent à répondre complétement aux exigences d'un environnement d'exécution de type *Fog Computing*. Au regard de cela, l'objectif des présents travaux est de définir et d'implémenter un *framework* facilitant le développement d'un module de contexte permettant le développement et l'exécution d'applications aux extrémités du réseau. Les présents travaux devront présenter des qualités d'extensibilité, de facilité d'utilisation et d'autonomie. Ces qualités sont tirées des principes du Génie Logiciel et coïncident avec les propriétés manquantes extraites de l'état de l'art. Ces critères, qui restent très généraux, ont été associés avec un ensemble d'objectifs plus précis comme montré dans le tableau 3.1. Un détail des différents critères et objectifs est présenté par la suite.

Facilité d'utilisation	**Développeur d'application**	Faible Couplage
		Support à la composition dynamique
	Développeur de contexte	Adoptions des patterns existants
		Séparation des préoccupations
Support à l'extensibilité		Modularité
		Souplesse
		Homogénéité
Autonomie		Modularité
		Administrable
		Management autonomique externe

Table 3.1: Tableau des différentes stratégies adoptées

3.1.2.1 Facilité d'utilisation

Le premier critère est la facilité d'utilisation. Ce critère se traduit par le support offert au développeur à travers divers mécanismes pour diminuer la complexité de ses tâches. Comme nous l'a montré [Dav89], l'adoption d'une solution logicielle dépend en grande partie de sa facilité d'utilisation. Cette adoption est d'autant plus accélérée lorsque le domaine est complexe et technique. Comme dans [Ber+14], nous distinguons deux groupes d'utilisateurs des solutions de contexte : les développeurs d'applications qui interagissent avec le contexte uniquement avec la partie dissémination du middleware et les développeurs du module de contexte chargés de l'implémentation de toutes les phases du cycles de vie du contexte. Pour cela, nous avons découpé les objectifs de facilité d'utilisation en deux sous-catégories, une pour chaque groupe d'utilisateur de par leur différence d'interaction avec le module de contexte.

Les deux objectifs liés à la facilité d'utilisation pour les développeurs d'application sont les suivant :

- Obtention d'un faible couplage entre le module de contexte et le code applicatif.
- Obtention d'un support à la composition dynamique pour les consommateurs de contexte.

Faible couplage

Le couplage entre les applications et le middleware de contexte doit être le plus faible possible. Les applications doivent seulement se programmer au-dessus d'interfaces de programmation cohérentes rendant compte explicitement des différentes possibilités d'interactions : actions, données ou évènements disponibles. Les applications n'ont en aucun cas à se soucier

de comment les données sont collectées ou les actions synchronisées, ce qui renforce la séparation des préoccupations [Dij82] et facilite ainsi leur développement [SDA99] [Pas09]. Les avantages d'un couplage qualifié de faible ou lâche sont que les deux parties peuvent évoluer indépendamment de manière plus lisible, l'hétérogénéité des différentes sources est masquée derrière les abstractions communes. De plus cela permet de substituer les implémentations des différentes abstractions, ce qui aide à la réalisation de l'objectif suivant.

Support pour la composition dynamique

La définition et les avantages de la composition dynamique ont été présentés précédemment dans la section 2.2.1.15. La solution doit offrir un support permettant d'intégrer naturellement cette caractéristique dans le modèle de programmation des applications ou autres consommateurs de contexte. Les deux objectifs liés à la facilité d'utilisation pour les développeurs du module de contexte sont les suivants :

- Obtention d'un support pour la mise en place des différents patterns présents dans le cycle de vie du contexte.
- Obtention d'un support pour la séparation des préoccupations.

Adoption des patterns existants

L'adoption des *patterns* existants permet de faciliter le travail du développeur en lissant la courbe d'apprentissage de celui-ci et parfois de bénéficier de code déjà existant. Dans le cadre du développement d'un module de contexte, nous avons pu voir que plusieurs *patterns* ont pu être formalisés par la communauté: pour la phase de collecte du contexte, les différentes techniques de synchronisation *push/pull* avec leur variante périodique et ponctuelle (voir section 2.1.2.2), pour la partie raisonnement, les différentes techniques de médiation (voir section 2.2.1.14) ou de raisonnement (voir section 2.1.2.3). La solution doit offrir un support pour rapidement implémenter ou configurer ces différentes techniques et ainsi alléger le travail du développeur tout en le rendant plus maintenable.

Séparation des préoccupations

La notion de séparation des préoccupations met un avant le fait de diviser le développement d'un logiciel en plusieurs fonctionnalités, celles-ci se recoupant le moins possible [Dij82] [Rea89]. Certaines préoccupations communes à tous les développements liés au module de contexte doivent pouvoir être gérées par le *middleware* pour soulager le travail du développeur. Dans notre cas, le *middleware* doit pouvoir être capable de gérer la dissémination des informations de contexte, la propagation des évènements ou encore le cycle de vie des fonctionnalités de synchronisation liées à chaque entité du contexte. Cela permettra de favoriser la lisibilité du code spécifique à chaque entité, sa maintenance et son évolution.

3.1.2.2 Support à l'extensibilité

Le second critère concerne la possibilité d'avoir un support facilitant l'ajout de nouvelles fonctionnalités sans perturber les consommateurs des anciennes : une propriété souvent référencée comme extensibilité, flexibilité ou élasticité [Bar12]. Comme nous l'avons vu, les nombreuses sources de dynamisme (ensemble d'applications non borné, évolution technologique, ... etc.) impactent fortement la manière dont la solution de contexte est développée. Celle-ci doit en effet être capable de facilement supporter l'intégration de ses nouveaux besoins pour rester en phase avec l'environnement qu'elle essaye de modéliser. Ce besoin s'est fait sentir très tôt avec seulement des essais au design time [SDA99] puis s'est déplacé jusqu'à l'exécution sur certains points [JBB07] [Pas09] pour tenter d'offrir une continuité de service.

Les trois objectifs liés à l'extensibilité de la solution pour les développeurs du module de contexte sont les suivant :

- Obtention d'une solution modulaire.
- Obtention d'une solution souple.
- Obtention d'une solution homogène.

Modularité

La modularité [Par72] est un des principes les plus éprouvés du génie logiciel. Ce mécanisme est la pierre angulaire pour obtenir une solution flexible et réduire la complexité globale de la solution. Dans le cadre de la programmation d'un module de contexte, qui est extrêmement complexe et technique s'il est considéré comme un module unique, ce premier mécanisme semble indiqué pour soulager le travail du développeur. Il est donc impératif de proposer des mécanismes pour la définition séparée des différentes fonctionnalités à fournir par la solution de contexte. La définition séparée des différentes fonctionnalités permet de venir en ajouter de nouvelles sans perturber les anciennes et leurs consommateurs, offre un premier pas vers la réutilisation et réduit le temps de développement du système [Par72] .

Souplesse

Les environnements d'exécutions et les besoins des applications s'y exécutant variant fortement, enfermer le développeur dans un cadre trop strict réduirait sa capacité à répondre à la totalité des situations et fonctionnalités souhaitées comme par exemple s'adapter à toutes les sources de données possibles (voir section 2.2.1.12) . Il est important que le développeur puisse disposer d'une certaine liberté quant au découpage architectural des fonctionnalités et à l'implémentation de celles-ci. La solution doit promouvoir une manière de développer indépendamment les différentes fonctionnalités, que celles-ci soient d'ordre fonctionnel ou non-fonctionnel, et de venir les composer de manière naturelle.

Homogénéité

Pour faciliter le développement de nouvelles fonctionnalités ou extensions, il est important de disposer d'un cadre architectural homogène. Il faut ainsi ne pas multiplier inutilement les concepts pour faciliter le travail du développeur. Cela permet une utilisation naturelle et efficace de la solution et facilite la réutilisation de code.

3.1.2.3 Autonomie

Le troisième et dernier critère concerne l'autonomie de la solution. Les solutions évoluant dans les environnements *Fog Computing* sont extrêmement difficiles à maintenir et à adapter aux besoins des applications qui y sont déployées dynamiquement. D'une part, aucun administrateur n'est disponible ou assez compétent pour faire face à l'énorme complexité technique d'une solution de contexte, d'autre part tenter de réaliser cette tâche manuellement est souvent lent, inefficace, lourd et propice à l'erreur. Nous avançons que cette adaptation doit être réalisée dans la mesure du possible de façon autonome.

Les trois objectifs liés à l'autonomie de la solution pour les développeurs du module de contexte sont les suivants :

- Obtention d'une solution modulaire.
- Obtention d'une solution administrable.
- Obtention d'une solution dotée d'un manager autonomique.

Modularité

Le large spectre d'environnements d'exécution et d'applications entraine une infinie combinaison de besoins et de contraintes à l'exécution. Pour cela, la solution de sensibilité se doit de pouvoir être spécifique et personnalisable à chaque situation pour s'assurer que les besoins et les exigences de chaque partie prenante sont satisfaits. Une architecture modulaire (voir paragraphe 3.1.2.2) est donc un passage obligé pour produire une solution personnalisable et adaptée à chaque situation. Le principal avantage d'une solution personnalisée est la diminution de la consommation de diverses ressources (mémoire, cpu, etc...).

Administrable

Pour pouvoir établir un comportement autonome, il est impératif que la solution soit administrable, c'est-à-dire qu'elle possède des points de configuration et qu'il soit possible de les modifier à l'exécution tout en en maîtrisant les effets. Dans le cadre de la sensibilité au contexte, on pense particulièrement à la possibilité d'activer ou désactiver certains modules récoltant les informations des capteurs de l'environnement ou la possibilité de reconfigurer les fréquences de synchronisation pour atteindre des niveaux de qualité suffisants ou économiser l'énergie des équipement disséminés dans l'environnement.

Management autonomique externe

Le management autonomique est lui aussi sujet à l'extension. Pour permettre une évolution maintenable et moins complexe de celui-ci, nous pensons que la meilleure solution est de se conformer à l'architecture prônée par IBM [KC03]. Les éléments managés, dans notre cas les modules responsables de l'exécution des différentes phases du cycle de vie du contexte, sont ainsi découplés des éléments qui les managent, appelés managers autonomiques. La tâche principale du manager autonomique est de pouvoir fournir une continuité de service à un niveau de qualité accepté par les applications lorsque cela est possible. Il doit donc être capable d'après les besoins des applications de réaliser les adaptations nécessaires du module de contexte au travers des points évoqués dans l'objectif précédent.

3.2 Approche

3.2.1 Vision globale

Cette thèse propose un modèle à composant spécifique ayant pour but de faciliter le développement et l'exécution autonomique d'un module de contexte. Plus particulièrement, nous proposons de combiner des concepts issus de l'approche à service[1] [Pap03] pour faciliter la consommation d'un module de contexte par les applications, de la programmation orientée composant[2] [SGM02] et des bonnes pratiques du génie logiciel [Gam+95] [Dij82] pour en accélérer le développement et l'extension et de l'informatique autonomique [KC03] pour alléger la maintenance et augmenter l'efficacité de son exécution

Notre approche expose les concepts suivants :

Un cadre architectural (voir figure 3.1), où le contexte est rendu accessible sous la forme de services de contexte aux applications. Ces services sont implémentés dans un module de contexte par un modèle à composant spécifique, qui est administré par un manager autonomique externe.

Un modèle à composant spécifique pour la programmation des services de contexte reposant sur deux principes abstraits : la séparation des fonctionnalités de synchronisation et le développement d'un composant de contexte par composition, en plus de reprendre toute les bonnes pratiques des SOCM.

Un DSL directement intégré dans un langage de programmation générale pour faciliter le développement des composants du modèle et rendre accessible de façon concrète les principes précédemment évoqués.

Un framework d'exécution du modèle à composant rendant introspectable et administrable les concepts décrits grâce au modèle à composant et au DSL.

Un manager autonomique dont le rôle est d'exposer uniquement les services de contexte correspondant aux besoins des applications tout en tenant compte des ressources présentes dans l'environnement pour éviter tout gaspillage de ressources.

[1] Un rappel des concepts de l'approche à service est disponible dans la section 4.1.2
[2] Un rappel des concepts de la programmation orientée composant est disponible dans la section 4.1.1

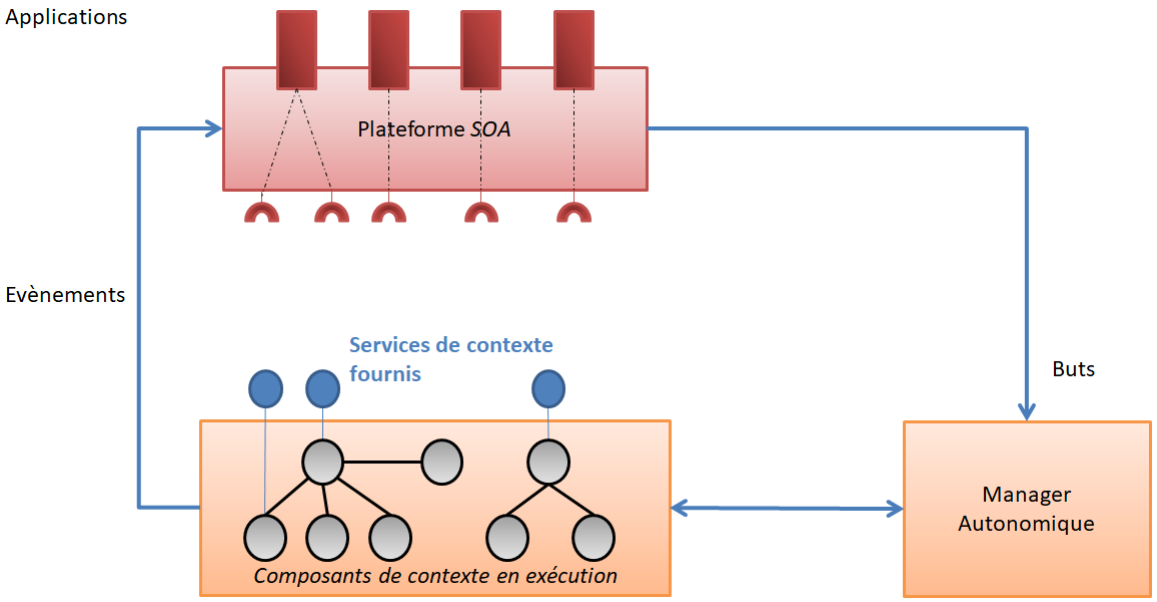

Figure 3.1: Approche globale

3.2.2 Cycle de vie

Notre approche promeut naturellement un processus de développement du contexte en deux phases. Premièrement, à l'instar de [Ber+14] , des experts du domaine définissent un ensemble de services qui peuvent être utilisés par des applications s'exécutant sur la même plateforme *Fog Computing*. Dans un second temps, les développeurs d'applications peuvent construire leur application en se basant uniquement sur les contrats de services définis précédemment. Cela permet de simplifier leur phase de développement et de se concentrer sur la logique business de leurs applications. La première étape, habituellement appelée Ingénierie du domaine, implique plusieurs acteurs avec un large spectre de compétences, comme illustré par la figure 3.2. Les différentes tâches effectuées dans cette étape sont :

- Les développeurs d'applications et les fournisseurs de contexte déterminent les différents services à inclure dans le contexte, en accord avec les capacités de la plateforme. Notre approche propose un support pour la description des services de contexte.
- Les fournisseurs de contexte implémentent les services définis précédemment et peuvent en implémenter des supplémentaires pour servir de support à la médiation ou à des traitements particuliers. Le développement de ces services est d'une part simplifié par la présence d'un support pour décrire la synchronisation avec les sources de données de l'environnement. D'autre part, la plateforme met à disposition certaines implémentations de services de contexte, avec un support pour les réutiliser simplement.
- La description des composants de contexte est passée par une mécanique d'intégration et est packagée sous forme d'unité de déploiement. L'administrateur de la plateforme a la charge de déployer ces artéfacts dans la plateforme d'exécution. Aucun support n'est

fourni pour cette activité, mais des approches telles que [Gun14] peuvent s'intégrer dans notre processus.

- L'environnement d'exécution prend ensuite en charge l'exécution des entités de contexte.

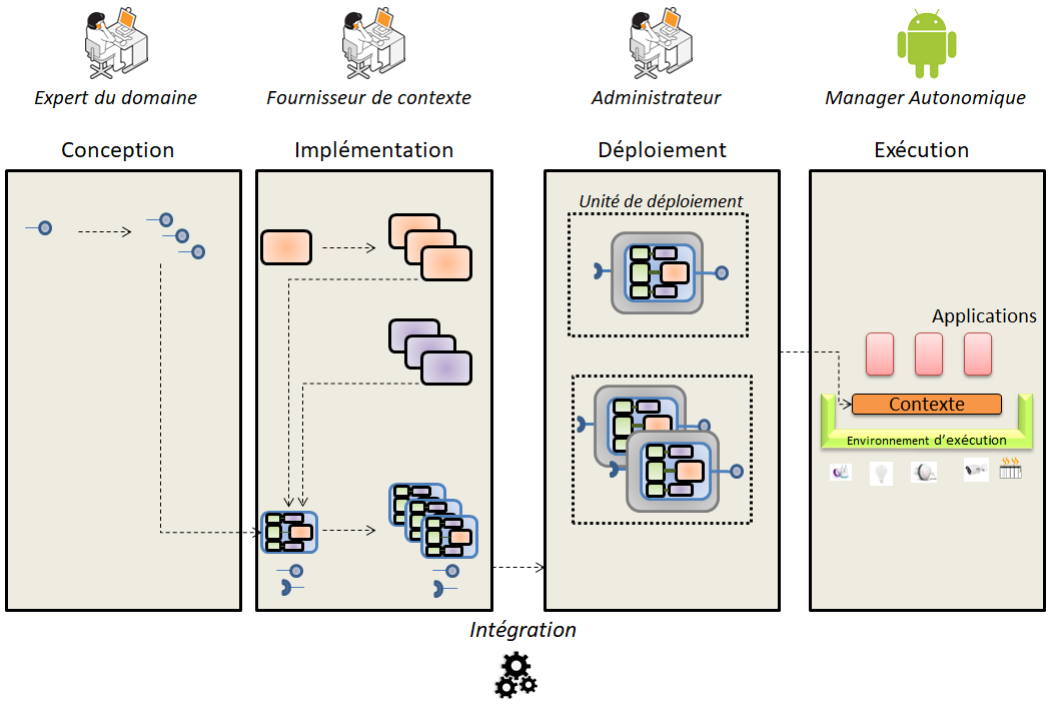

Figure 3.2: Développement et déploiement du contexte

La seconde étape voit les développeurs d'applications implémenter et déployer leurs applications. Il est à rappeler qu'à l'exécution, les services de contexte exposés par le module de contexte dépendront simultanément des applications déployées et des ressources présentes dans l'environnement. Cette gestion est assurée par le manager autonomique.

3.2.3 Composants de contexte

Le rôle d'un modèle à composant est de fournir un ensemble de règles, contraintes et formalismes servant à guider le développeur. Dans le cas de notre modèle à composant de contexte, un composant se définit de la sorte et est illustré dans la figure 3.3 :

« Un composant de contexte, ou entité de contexte, est utilisé pour décrire un type de fournisseur de contexte. Celui-ci décrit un ensemble de services de contexte à fournir et une composition d'éléments fournissant la logique d'implémentation de ces services. La composition est effectuée à partir de plusieurs éléments : le Core et ses extensions fonctionnelles. Le Core permet l'implémentation des services subissant les contraintes spécifiques du fournisseur tandis que les extensions permettent de réutiliser des implémentations fournies par un

tiers. Chacun de ces éléments peut décrire sa logique de synchronisation avec le monde réel et dépendre de code externe pour la réaliser. A l'exécution, la composition de ses différentes parties est réalisée dynamiquement sous la forme d'une enveloppe fonctionnelle.»

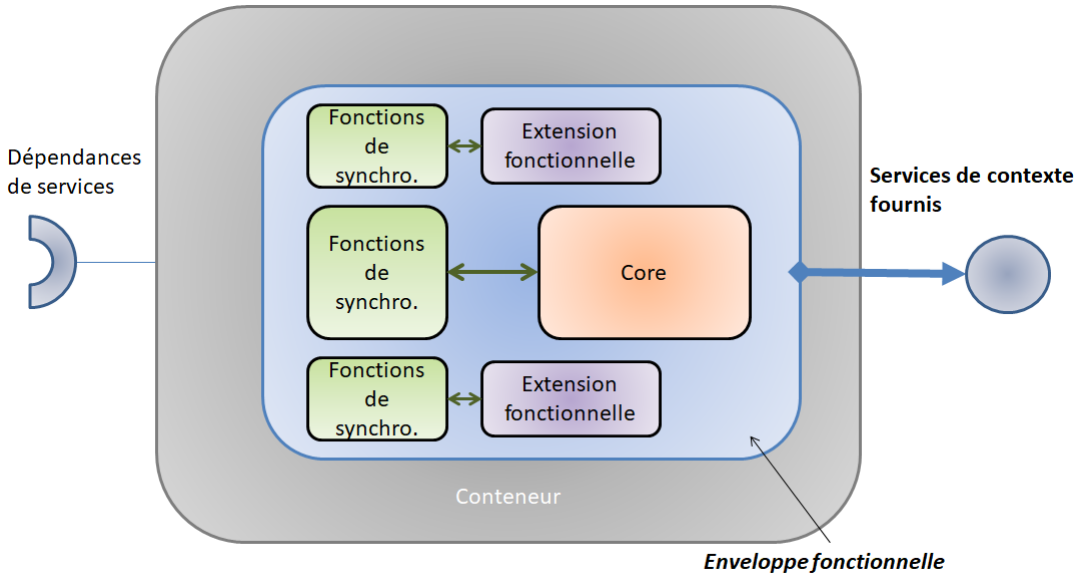

Figure 3.3: Structure des composants de contexte à l'exécution

D'une manière plus précise, notre modèle à composant se base sur un SOCM pour automatiser la fourniture des services de contexte et la prise en compte de code externe sous la forme de dépendances de service. De plus, Il cherche à exprimer deux aspects essentiels liés à la programmation d'un fournisseur de contexte, le tout en essayant de maintenir l'implémentation d'un service de contexte la plus simple et homogène possible (héritée de l'idée de POJO [Fow00] mise en œuvre dans [Esc08]).

3.2.3.1 Séparation de la gestion des préoccupations de synchronisation

Comme nous l'avons vu, l'implémentation d'un service de contexte fait intervenir un ensemble de patrons de synchronisation pour rester à jour vis-à-vis de l'environnement. Pour cela nous proposons, comme illustré dans la figure 3.4, que le développeur décrive séparément la logique de synchronisation à l'aide d'un DSL supportant les trois différents type de synchronisation: *pull*, *push* et *apply* (décrit dans la section 2.1.2.2) et leur orchestration: périodique ou ponctuelle. A l'exécution, la gestion du cycle de vie de ces fonctions est déléguée au conteneur. A la manière de la gestion du dynamisme dans les approches SOCM, c'est maintenant aux *framework* d'exécution de masquer cette synchronisation et les problèmes qui en découlent: gestion d'erreur, méthode de cache des données, concurrence.

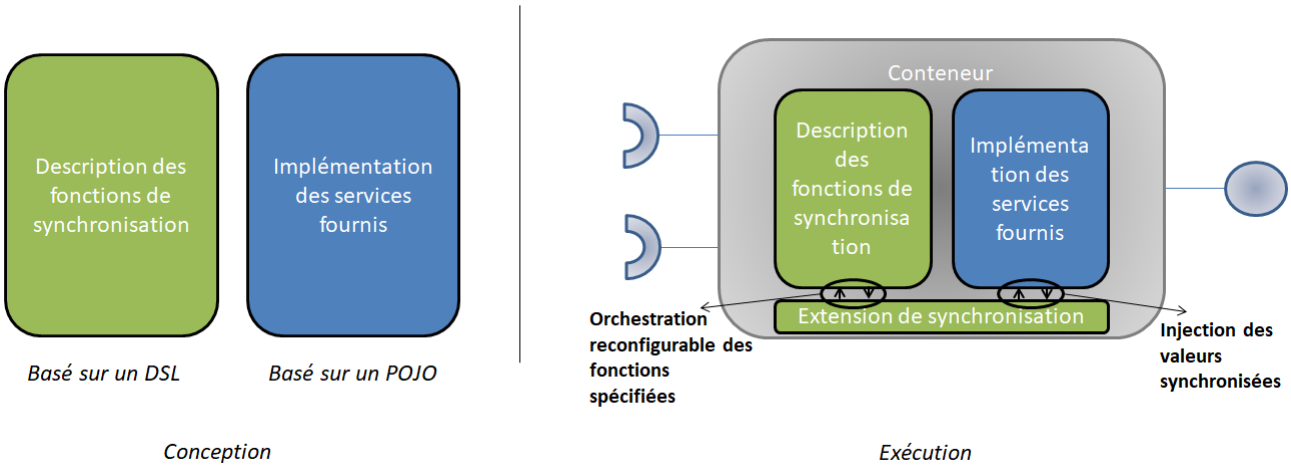

Figure 3.4: Séparations des préoccupations de synchronisation

3.2.3.2 Faciliter l'extensibilité

Une des exigences de notre modèle à composant spécialisé est de favoriser l'extension du module de contexte, que l'on traduira par la conception et l'exécution de nouveaux fournisseurs de services de contexte. Un des moyens de faciliter la conception de nouveaux fournisseurs est de favoriser la réutilisation [Gam+95] de certaines implémentations de services de contexte.

Comme suggéré dans [Gam+95], dans un monde idéal on pourrait construire un fournisseur en composant seulement les différentes implémentations fournies par les divers *toolkits* ou *frameworks* disponibles. Mais, dans la réalité cet ensemble d'implémentations réutilisables n'est pas assez riche pour satisfaire tous les cas d'usages et une partie spécifique de l'implémentation doit être développée par chaque fournisseur.

Notre modèle à composant reflète ce fait: un nouveau fournisseur peut faciliter son implémentation en venant spécifier une composition d'extensions fonctionnelles, qui contiennent le savoir-faire des techniques déjà développées et encapsulées, et son Core, qui contient les parties spécifiques à son cas d'usage. A l'exécution comme illustré dans la figure 3.5 , la composition de ces différentes implémentations est interprétée par le conteneur qui la masque à travers une enveloppe fonctionnelle jouant le rôle de *proxy* pour les consommateurs de service. Cette composition bénéficie pleinement des capacités de dynamisme des SOCM: si par exemple une des extensions fonctionnelles n'est pas encore présente ou invalide dans la plateforme, le fournisseur exposera seulement un sous ensemble des services spécifiés à la conception. Si celle-ci, plus tard, est de nouveau disponible et valide, le conteneur prendra dynamiquement en compte ce changement et exposera la totalité des services.

Comme illustré dans la figure 3.5 , que ce soit le *Core* ou les extensions fonctionnelles, ceux-ci bénéficient de toute les mécaniques de la programmation orientée composant ce qui permet

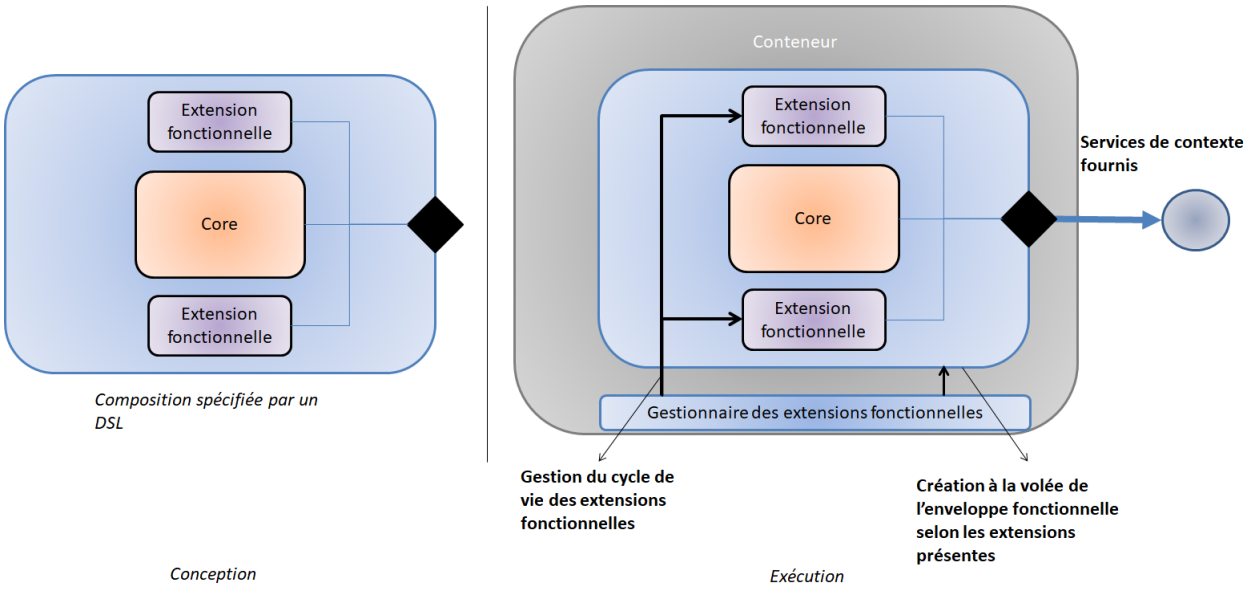

Figure 3.5: Illustration du mécanisme favorisant l'extensibilité

de maintenir un modèle de développement homogène et qui facilite l'intégration transparente de préoccupations non fonctionnelles (persistance, *logging*,...).

3.2.4 Management Autonomique

L'administration du module d'exécution des composants est réalisée par un manager autonomique. L'objectif du manager est de garantir que les services publiés correspondent aux dépendances de service exprimées par les applications.

Il agit de manière globale en venant modifier la topologie du module de contexte en autorisant ou non l'instanciation des différentes entités de contexte. Il peut aussi agir de manière plus locale, par l'intermédiaire de points d'administration génériques exposés par les conteneurs des entités de contexte, en venant par exemple effectuer une reconfiguration des fréquences de synchronisation pour garantir une qualité de service adéquate. Le module d'administration est faiblement couplé au module d'exécution car il ne repose que sur des API ou points d'administration génériques.

3.3 Conclusion

Le chapitre a rappelé les limites des approches actuelles de sensibilité au contexte pour une plateforme de *Fog Computing*. Face à ces limites, nous avons dressés une stratégie de recherche sur plusieurs axes pour aboutir à la proposition d'un modèle à composant spécialisé dans la construction d'un module de contexte. Le modèle à composant de contexte, dont l'unité de base est l'entité de contexte, permet de déléguer à l'environnement d'exécution du composant certaines caractéristiques comme la gestion de la synchronisation et l'extensibilité tout en gardant un modèle de développement simple. L'environnement d'exécution contient en plus une infrastructure permettant d'établir des comportements autonomiques.

La suite de ce manuscrit décrira en détails les différentes contributions de cette thèse puis leur associera dans une seconde partie une implémentation de référence.

Chapter 4

Modèle à composant de contexte

Sommaire

Dans ce quatrième chapitre, notre but est de décrire en profondeur notre proposition de recherche et traiter des différents concepts de notre modèle à composant spécialisé dans la construction d'un module de contexte.

Dans un premier temps, nous allons nous focaliser sur les différents éléments du modèle à composant. L'unité de base de notre modèle à composant est l'entité de contexte, qui est composé de module hautement cohérent implémentant distinctement les spécifications de services proposés par l'entité de contexte et pouvant décrire de manière simple leur logique de synchronisation. Dans un second temps, nous allons décrire l'infrastructure permettant de rendre autonome notre modèle à composant. Nous évoquerons l'architecture globale de la solution, les différentes possibilités de reconfiguration exposées par notre modèle à composant et les différents comportements autonomiques implantés.

4.1 Introduction

Avant de détailler la proposition de recherche du présent manuscrit, une introduction sur les paradigmes de composants et de services logiciels est proposée. Cela est dû au fait que notre approche s'appuie et emprunte beaucoup de vocabulaire lié à ces deux notions.

Comme précisé dans [Hal+11] , ces deux paradigmes souvent confondus, car extrêmement complémentaires, se différencient sémantiquement dans le fait que :

- Dans une approche basée sur les composants logiciels, l'architecte se focalise sur la vue du producteur.
- Dans une approche orientée service, l'architecte se focalise sur la vue du consommateur.

4.1.1 Introduction sur les composants logiciels

Les approches basées sur la programmation orientée composants (*Component-Based Software Engineering* en anglais) sont issues du génie logiciel et préconisent la réalisation d'applications par l'assemblage de composants logiciels [CH01] . De nombreuses définitions de la nature exacte du composant logiciel peuvent être trouvées dans la littérature, nous en retiendrons deux principalement : " A softwre component is a unit of composition with contractually specified interfaces and explicit context dependencies only. A software component can be deployed independently and is subject to composition by third parties. " [SGM02]

"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 composition standard.

A component modedl defines specific interaction and composition standards. A component model implementation is the dedicated set of executable software elements required to support the execution of components that conform to the model.

A software component infrastructure, or framework, is a software system that provides the services needed to develop, deploy, manage and execute applications built using a specific component model implementation. It may itself be organized as a set of interacting software components. " [CH01] reprise par [Kra07].

On retiendra qu'un composant peut être considéré comme une brique fonctionnelle que l'on peut venir composer et déployer indépendamment pour former une application. De plus, un composant d'une approche particulière est programmé pour être conforme à un patron

particulier : un modèle à composant. Le modèle à composant d'une approche particulière sert à définir ce que signifie être un composant de cette approche : comment interagir avec celui-ci, quelles sont ses capacités (cycle de vie ou gestion des configurations). Un *framework* à composant est utilisé pour exécuter les instances des composants adhérant au modèle à composant. Ce *framework* peut prendre en charge à partir du déploiement du composant par exemple la gestion du cycle de vie des composants, la résolution des dépendances [Hal+11] .Actuellement, de nombreux modèles à composants existent et sont utilisés dans des domaines variés. On peut citer comme exemple de modèle à composant les Entreprise Java Bean [Mic] pour supporter le développement d'applications coté serveur distribuées et transactionnelles, le modèle Cilia utilsé pour la gestion de chaine de médiation [Gar12] [Gar+10] ou Fractal [Bru+06] avec ses différents *frameworks* d'exécutions : Julia [Bru+06] en Java ou ou Think [Fas+02] en C.

En résumé, le modèle à composant permet de définir et implanter des types de composants. Ceux-ci sont ensuite pris en charge par le *framework* d'exécution pour créer des unités d'exécution, aussi appelées instances de composant. Les liaisons, toujours orientées entre un producteur et un consommateur, qui existent entre les différentes instances de composant sont aussi créées pour réaliser une composition. Ces liaisons sont souvent décrites explicitement à travers un ADL (*Architecture Description Language* en anglais) [Cle96] et leur réalisation incombe la plupart du temps au *framework* d'exécution. Les ADL sont utilisés dans le modèle à composant pour décrire les différentes liaisons en fonction des propriétés fournies et requises par un type de composant.

Une des techniques les plus utilisées pour la réalisation d'une approche à composant est l'utilisation d'un conteneur. D'une manière abstraite comme représenté dans la figure 4.1 , le conteneur peut être appréhendé comme une coquille entourant le contenu de l'instance. Son rôle est en partie de découpler le contenu de l'instance de différentes préoccupations en les gérants de manière autonome. Elles peuvent concerner les interactions avec les autres instances (via les interfaces fournies ou requises) ou des aspects non fonctionnels (via les interfaces de contrôle). Le conteneur a donc la totale responsabilité de l'exécution du contenu et lui fournit ses différents besoins souvent au moyen des patrons d'inversion de contrôle et d'injection de dépendances [Fow04] .

Les principaux avantages d'une telle approche est qu'elle permet une séparation claire des préoccupations en embarquant du code, souvent non-fonctionnel, dans le conteneur et permet une bonne encapsulation grâce au système d'interface. De plus dans certaines approches, le conteneur est extensible, c'est-à-dire que l'utilisateur peut lui-même venir y ajouter de nouvelles extensions, définies et implémentées par ses soins, pour gérer certaines préoccupations. Ces approches sont souvent qualifiées d'approche à composant extensible, Fractal [Bru+06] et iPOJO [Esc08] [EHL07] en étant des exemples.

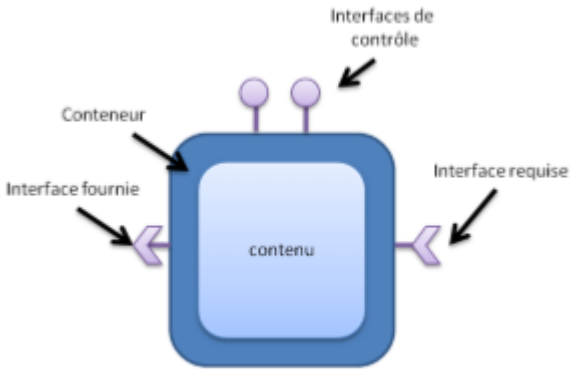

Figure 4.1: Schéma d'une instance de composant et son conteneur associé [Esc08]

4.1.2 Introduction sur les services logiciels

Le paradigme des approches à services [OAS06] s'est construit sur le fait que, en règle générale, des entités (des personnes, organisations ou logiciels) développent des moyens de résoudre des problèmes pour mener à bien leurs activités. Ainsi, lorsque une nouvelle entité rencontre un besoin, il n'est pas rare que celui-ci corresponde à une solution déjà mise en place et offerte par une autre entité. En informatique, cela se traduit par le fait que les besoins d'une entité logicielle pour fonctionner sont remplis par une autre entité logicielle appartenant à un autre utilisateur.

L'objectif des approches à services (ou *Service Oriented Computing* résumé par SOC en anglais) [SG96] est la réalisation d'applications à partir de plusieurs entités logicielles, avec comme principale ambition d'assurer un faible couplage entre ces entités. La notion de service est au centre de ce paradigme et peut se définir de la façon suivante :

« Services are self-describing, platform agnostic computational elements. » [Pap03]

« Un service est une entité logicielle qui fournit un ensemble de fonctionnalités définies dans une description de service. Cette description comporte des informations sur la partie fonctionnelle du service mais aussi sur ses aspects non-fonctionnels. À partir de cette spécification, un consommateur de service peut rechercher un service qui correspond à ses besoins, le sélectionner et l'invoquer en respectant le contrat qui a été accepté par les deux parties. » [Cho09]

L'approche à service en plus de définir la notion de service peut être abordée comme un style architectural. Celui-ci repose sur un patron formalisant les interactions entre trois différents acteurs, comme présenté dans la figure 4.2. Les trois acteurs décrits dans l'approche à services sont :

Le fournisseur de service qui propose un service décrit à l'aide d'une spécification ou description de service.

Le consommateur de service qui utilise des services fournis par des fournisseurs de service.

Le registre de service ou annuaire qui permet aux fournisseurs de services de stocker leur description de service. Celles-ci peuvent ensuite être recherchées et sélectionnées par les consommateurs de services.

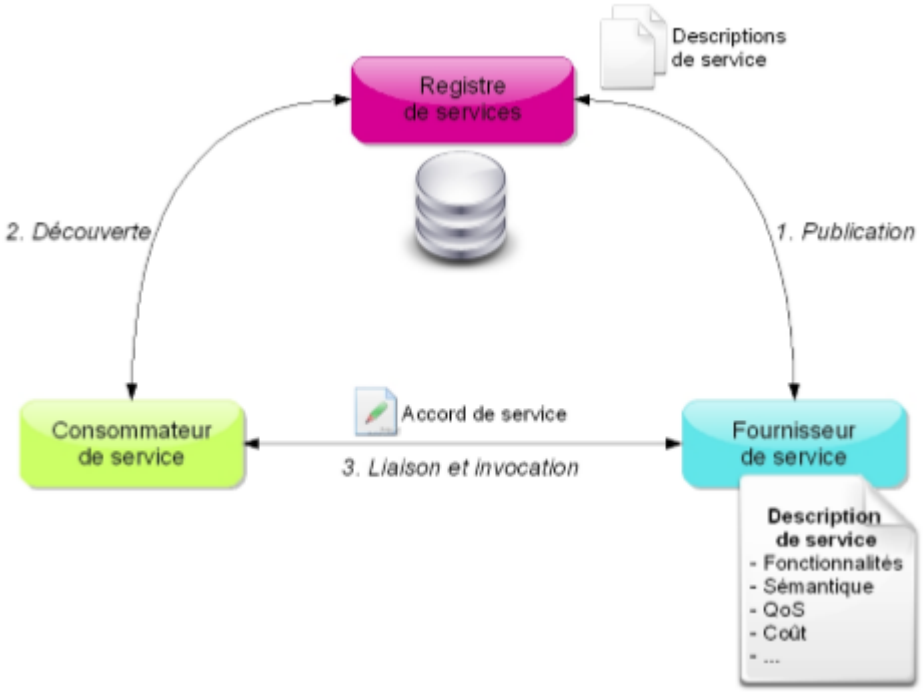

Figure 4.2: Acteurs et Interactions dans l'architecture à service [Cho09]

Les interactions primitives décrites entre les différents acteurs sont les suivantes :

La publication de service: les fournisseurs de service enregistrent la description de service dans le registre de service.

La découverte d'un service: les consommateurs de service interrogent le registre à services pour trouver une description de service conforme à leurs besoins.

La liaison et l'invocation d'un service: une fois que le consommateur a trouvé une description de service correspondant à ses besoins, celui-ci peut se lier au fournisseur de services qui a enregistré la spécification et comment utiliser le service, l'interaction pouvant être locale ou distribuée.

Certaines approches [Esc08], qualifiées de SOC dynamique, ajoutent des interactions supplémentaires pour rendre compte de la volatilité des fournisseurs de services :

Le retrait de service: le fournisseur de services retire sa description de service de l'annuaire pour signifier qu'il n'est plus en mesure de fournir le service décrit.

La notification de retrait ou d'ajout: le registre de services informe les différents consommateurs de l'arrivée ou du retrait d'une description de service qui peut correspondre à leurs besoins.

Si ces interactions supplémentaires sont adoptées, elles permettent aux consommateurs de choisir à l'exécution le fournisseur de service le plus adapté, et d'en changer en cas d'indisponibilité. Cette capacité est souvent décrite avec le terme liaison à retardement (ou *late-binding* en anglais). Ces approches présentent donc comme avantage de pouvoir prendre en compte la dynamique de l'environnement.

Pour la gestion concrète du style architectural présenté précédemment, un environnement d'exécution pouvant supporter les différentes interactions entre les différents acteurs est nécessaire. Cet environnement, souvent qualifié d'architecture orientée service (ou *service-oriented architecture* abrégée par SOA en anglais), propose les mécanismes suivants, illustrés dans la figure 4.3, pour gérer l'intégration et l'exécution des acteurs du SOC :

Les mécanismes de base: ceux-ci assurent les interactions primaires du SOC tel que la publication, la découverte, la composition, la négociation, la contractualisation et l'invocation des services.

Les mécanismes additionnels: ceux-ci permettent de prendre en compte des besoins non-fonctionnels, souvent non spécifiques au SOC, tels que la sécurité, les transactions ou la qualité.

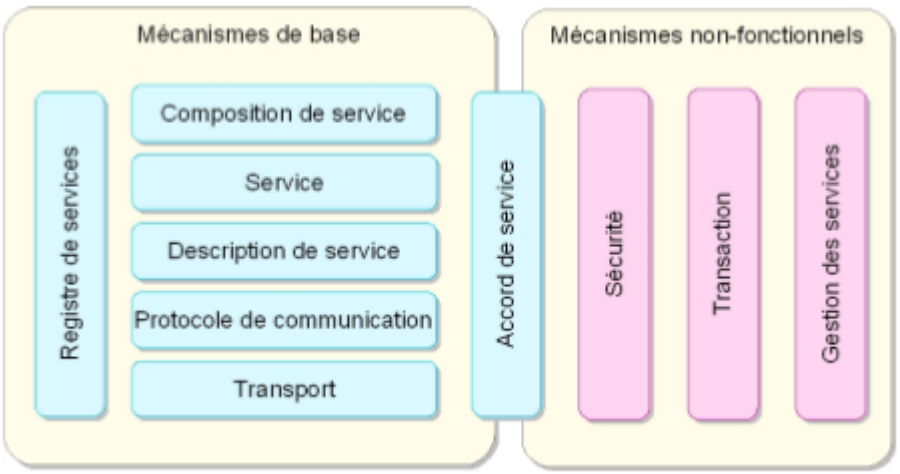

Figure 4.3: Mécanismes nécessaires pour un environnement d'intégration et l'exécution des services logiciels [Cho09]

En plus de la liaison à retardement, il existe d'autres avantages à une telle approche [Sta06] [EHL07]. Le fait de décrire les différentes capacités par une description abstraite de service permet en premier lieu de réduire le couplage entre les consommateurs et fournisseurs de services. Il en résulte que les différents consommateurs de services se voient masquer l'hétérogénéité des différents fournisseurs. Dans un second temps, couplé à la liaison à retardement, cela permet aussi de faciliter la substitution des différents fournisseurs. On peut donc aisément supporter la composition dynamique des différents services à l'exécution [Esc08]. De plus l'approche à service permet aussi de masquer en partie la distribution, en masquant si l'appel à une méthode d'un service est local ou distant [Bar12].

L'architecture orientée service fournit des mécanismes généraux, souvent très techniques pour construire des applications orientées services rendant le travail des développeurs de plus en plus complexe. Pour pallier à cette complexité grandissante, des modèles à composants orientés service (*service-oriented component model* abrégé par SOCM en anglais) ont vu le jour. Ceux-ci permettent d'alléger le travail du développeur en automatisant certaines interactions de l'approche à service comme la composition ou la composition par l'intermédiaire de code embarqué dans leur conteneur. Par exemple, iPOJO [Esc08] et ses différentes extensions, appelées *Handlers*, fournissent un support pour la composition et la découverte par l'intermédiaire du *DependencyHandler* [Esca] ou la publication par l'intermédiaire du *ProvidedServiceHandler* [Escd] comme illustré par la figure 4.4.

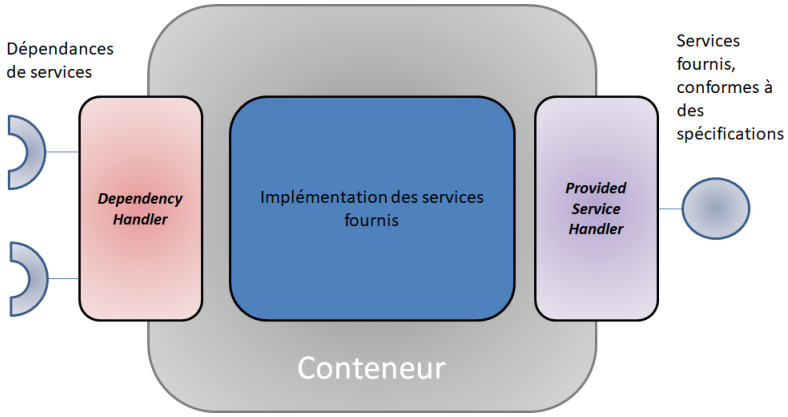

Figure 4.4: Modèle à composant orienté service

4.2 Composants de Contexte

Comme nous l'avons vu dans les parties précédentes, le module de contexte est l'élément central d'une plateforme Fog Computing car la totalité des applications se construisent par rapport aux abstractions de l'environnement qu'il fournit. Celui-ci reste cependant extrêmement difficile à **construire**, **étendre** et **maintenir**.

Le but de notre approche est de fournir une unité d'encapsulation avec une granularité adéquate pour permettre de palier à ces différents challenges et cibler les problématiques spécifiques du contexte. Notre approche fournit donc un modèle de développement, sous la forme d'un modèle à composant spécialisé pour la programmation contexte.

4.2.1 Vers une extension des modèles à composant orienté service

Notre proposition repose en partie sur le fait que le modèle du contexte est défini comme un ensemble de spécifications de services façonné **pour les besoins des applications**. Ces spécifications de services rendent compte des différentes interactions possibles (**lectures** et/ou **actions**) avec les fournisseurs de contexte présents sur la plateforme et aussi des possibles évènements qui peuvent intervenir à l'exécution. A l'exécution, selon la présence des différentes sources de contexte et les besoins de la plateforme, ces services sont publiés ou effacés de l'annuaire de la plateforme. Ces services peuvent ainsi être utilisés de manière opportuniste par les différentes applications adaptatives s'exécutant sur la plateforme.

Notre choix de se baser sur la SOA dynamique a été guidé par les critères que nous avons définis dans le chapitre précédent:

Le faible couplage entre le module applicatif et le module de contexte est assuré grâce à la définition explicite du contrat de service.

La composition dynamique est possible car les consommateurs de service sont avertis de l'arrivée ou du départ des services présents dans la plateforme par l'intermédiaire du registre des services.

Comme nous l'avons vu, les SOCM facilitent le développement des fournisseurs et consommateurs de services. De plus, cette technologie est assez maturs et robuste pour être utilisée à l'échelle industrielle [EBL13]. Ils apportent aussi à notre contribution au regard des critères suivants:

La séparation des préoccupations non-fonctionnelles est assurée en les déléguant au conteneur du composant.

Le support à la composition dynamique est réalisé grâce au conteneur qui automatise les interactions (découverte, publication/retrait de service) avec le registre à service.

Cependant, pour fournir une forme de *Context as a Service*, il faut que les SOCM pallient aux problématiques spécifiquement liées à une plateforme *Fog Computing*.

Comme dit précédemment, à notre avis, la plus-value des plateformes *Fog Computing* réside dans les applications qui s'y exécutent et des actions qu'elles effectuent et non dans toutes les données qui y transitent. Une façon efficace [BLE10] [ECL14] de programmer de telles applications est de les faire réagir à la visibilité des services.

A notre sens, pour faciliter cette méthode de programmation, il faut que chaque service reflète une préoccupation de contexte à propos d'un **unique** fournisseur, comme illustré par la figure 4.5. Chaque service doit donc être indépendant des autres, sans hiérarchie et de faible granularité. Cela permet d'une part de concevoir les services du module de contexte avec une grande souplesse et de pouvoir en ajouter au besoin sans perturber les consommateurs des précédents. Les applications peuvent ainsi établir leurs dépendances de manière précise, cohérente et, à l'exécution, être immédiatement notifiées si une instance de fournisseur apparait et présente toutes les fonctionnalités requises pour être utilisée par l'application. Nous insistons sur le fait qu'utiliser des services pour séparer les différentes préoccupations fonctionnelles d'un fournisseur est un choix très structurant. A contrario des approches utilisant des métadonnées pour taguer les services qui vont seulement associer des données supplémentaires pour refléter une préoccupation, nous permettons l'association à chaque préoccupation d'un ensemble d'actions, ce qui est primordial dans une plateforme *Fog Compting*.

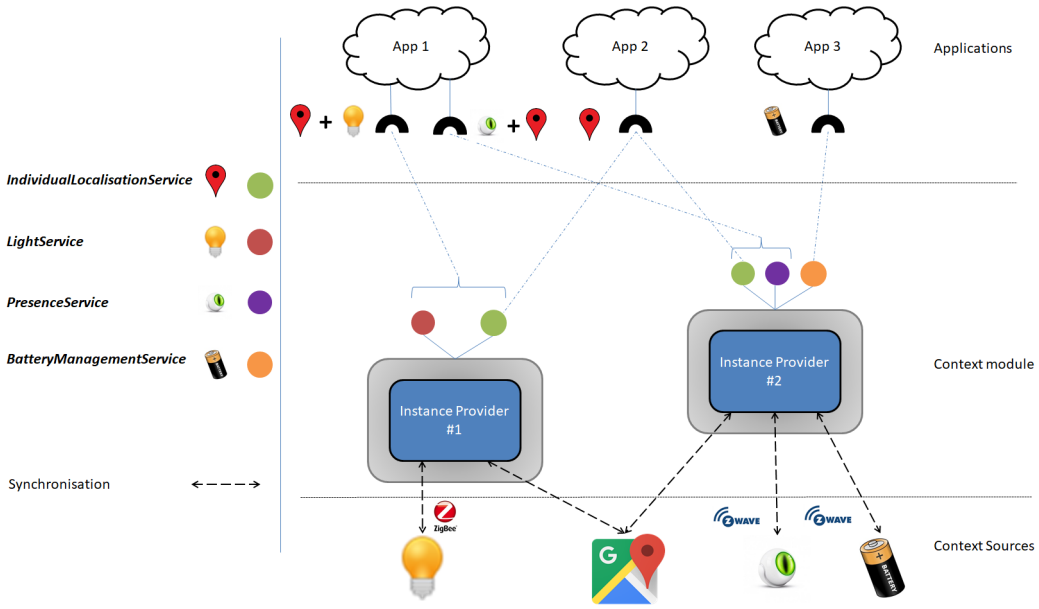

Figure 4.5: Exemple de conception des services de contexte

Ce style de conception rend le développement et l'exécution des fournisseurs plus complexes. D'un point de vue service, les fournisseurs doivent être en mesure d'implémenter et d'exécuter une multitude de services dans un milieu hautement volatile et de veiller à leur synchronisation vis-à-vis des sources disséminées dans l'environnement. Se contenter de laisser libre le développeur face à ces activités donne naissance à des solutions faiblement cohérentes, peu maintenables et difficilement administrables.

La première difficulté concerne la logique de synchronisation avec l'environnement. Si des outils spécifiques ne sont pas proposés au développeur, la logique de synchronisation se retrouve très rapidement entremêlée à la logique métier du service rendant la solution peu lisible et peu maintenable. De plus, chaque développeur aura la charge de veiller à son orchestration et de fournir ses propres points d'administration. Ces derniers risquent donc de ne pas exister ou d'être trop spécifiques pour pouvoir mettre en place un comportement autonomique global.

La seconde difficulté concerne le développement de plusieurs services par un même fournisseur et son exécution. Implémenter plusieurs services conduit souvent à faire appel à un large spectre de compétences, au vu de l'hétérogénéité de la plateforme, ce qui se traduit par un accroissement de dépendances vers des librairies ou des services externes. Si le développement de tel fournisseur n'est pas modulaire, il force les fournisseurs à posséder plusieurs versions des implémentations pour pouvoir s'adapter, ce qui aboutit en définitive à la construction d'un bloc d'une trop forte granularité, peu cohérent, au cycle de vie difficilement gérable sans intervention du développeur et impossible à maintenir. Si celui-ci devient modulaire, cela aide à assurer la séparation des préoccupations obtenue grâce au design des services précédemment évoqué mais n'offre aucune garantie vis-à-vis du dynamisme de la plateforme.

Chaque module possède un cycle de vie qui peut être impacté par une multitude de facteur : les capacités de la plateforme, le cycle de vie de la source de contexte qu'il essaye de mimiquer, la résolution de ses dépendances. Ajouté à cela, la disparition de certains services affecte à différents degrés le fournisseur. Cela peut facilement s'appréhender dans notre exemple 4.5, en cas de disparition du *PresenceService* et le *BatteryManagementService* qui traduit la disparation du capteur physique suite à un problème logiciel ou matériel (batterie vide, panne du driver, perte de connexion, etc...), fournir le service de localisation n'a que peu de sens car on continuerait de proposer la localisation d'un capteur qui a disparu, tandis que continuer à fournir le *PresenceService* et *BatteryManagementService* malgré la disparation du module de localisation conserve du sens : le capteur est présent mais ne peut être localisé, et permet d'offrir une continuité de service à l'application trois. On se rend donc compte que certains services correspondent au fonctionnement nominal de ce que l'on essaye de représenter tandis que d'autres peuvent être considérés comme annexes. La tâche du développeur reste donc difficile à réaliser et extrêmement technique. Il force à la génération de beaucoup de glue code pour lier les différentes implémentations et leur garantir un cycle de vie cohérent à l'exécution. L'adaptation à l'exécution des différentes implémentations reste aussi difficile à mener.

Il est donc impératif d'étendre les SOCM pour répondre à ces nouveaux besoins. D'une part, il faut permettre aux développeurs des fournisseurs de contexte d'adopter simplement les différents *patterns* de synchronisation et de mise à jour identifiés par la communauté. D'autre part, il faut proposer un cadre facilitant le développement et l'exécution de fournisseurs multi-services. Nous présenterons dans la suite un modèle à composant proposant ces améliorations.

4.2.2 Service de contexte

Le service de contexte est une entité de premier ordre dans notre approche. Il permet, entre autres, aux applications de raisonner uniquement par rapport à lui pour réduire le couplage avec le module de contexte.

Notre modèle, présenté dans la figure 4.6, permet d'établir des descriptions de service contenant:

- Une interface fonctionnelle qui regroupe un ensemble d'opérations servant à décrire les fonctionnalités du service.
- Un ensemble de variables d'état, identifiées individuellement par leur nom et leur type, qui permet aux consommateurs du service d'effectuer des opérations de filtrage, classement et de recevoir des évènements. Un changement de valeur d'une de ces variables à l'intérieur d'une des instances de service entrainera une notification à tous les consommateurs liés à l'instance de ce service.

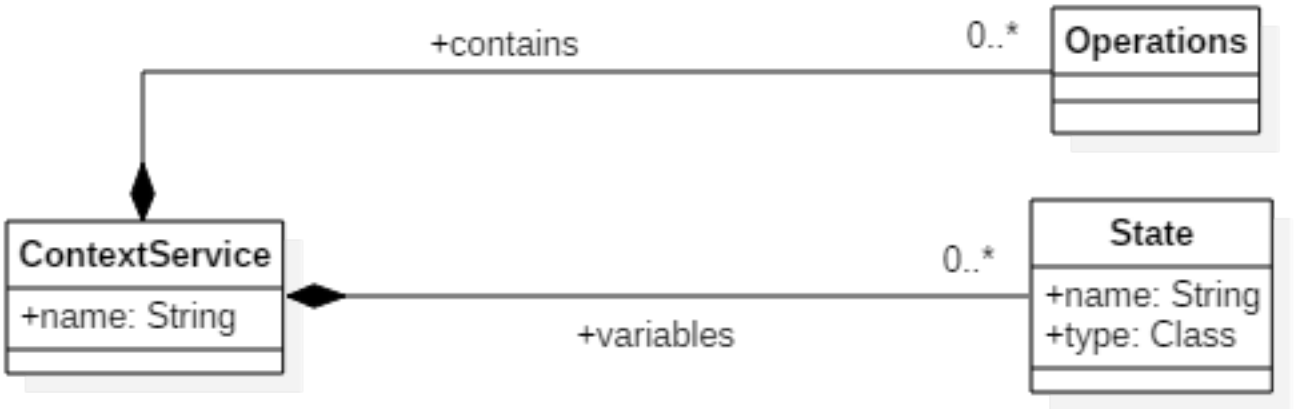

Figure 4.6: Modélisation du contrat de service

Il est à noter que la description de service est un contrat entre le consommateur de services et le fournisseur de services, dans notre cas l'entité de contexte. C'est donc la responsabilité du fournisseur de générer les évènements appropriés en cas de changement d'état.

Pour faciliter la génération des descriptions de service de contexte nous fournissons un DSL, dont le résumé est disponible dans le tableau 4.1. Une illustration de son utilisation est donnée dans la figure 4.7. On notera que les exemples évoqués s'appuient en grande partie sur le langage java car l'implémentation de référence de notre modèle repose sur ledit langage.

```
1 public @ContextService interface GenericDevice{
2    @State(type = String) String DEVICE_SERIAL_NUMBER = "device.serialNumber";
3    String getSerialNumber();
4 }
5
6 public @ContextService interface Thermometer extends GenericDevice {
7    @State(type = Quantity<Temperature> ) String THERMOMETER_CURRENT_TEMPERATURE =
         "thermometer.currentTemperature";
8     Quantity<Temperature> getTemperature();
9 }
```

Figure 4.7: Exemples de définition de service de contexte

L'exemple proposé 4.7 est la définition d'un service représentant un thermomètre. L'annotation *@ContextService* sert à identifier l'interface fonctionnelle du service. Celle-ci propose une opération permettant de récupérer la valeur de la température du thermomètre et hérite d'une opération permettant de connaitre le numéro de série du thermomètre. Une variation de la température peut être écoutée par l'intermédiaire d'un changement de l'état de la variable d'état *THERMOMETER_ CURRENT_ TEMPERATURE* définit par *@State*.

Annotation	Description	Configuration	Description
@ContextService	Permet d'identifier une interface comme étant un service de contexte. L'ensemble de ses méthodes devient de facto l'ensemble des opérations du service	-	-
@State	Permet d'identifier les sources d'évènements du service	*type*	Contient la description du type de la variable d'état

Table 4.1: Résumé du DSL de déclaration de service de contexte

4.2.3 Notion d'entité de contexte

Dans cette partie, nous allons détailler la notion d'entité de contexte et faire apparaitre les concepts fondamentaux (voir figure 4.8) qui lui sont associés. Il est à noter que les modèles présentés dans cette section sont conceptuels et font abstraction des contraintes liées à la technologie de la plateforme d'exécution.

Le but de la notion d'entité de contexte est de proposer un cadre pour les développements des **fournisseurs de contexte**. Une entité de contexte est la description d'un **type** de fournisseur de services de contexte d'une plateforme *Fog Computing*. A l'exécution, une **instance** d'entité de contexte encapsule un état centralisé et synchronisé avec l'environnement, rendu accessible et manipulable par l'ensemble des services fournis par l'instance. L'entité de contexte contient donc le code nécessaire à la réalisation de la logique métier des services fournis et la logique de synchronisation avec l'environnement.

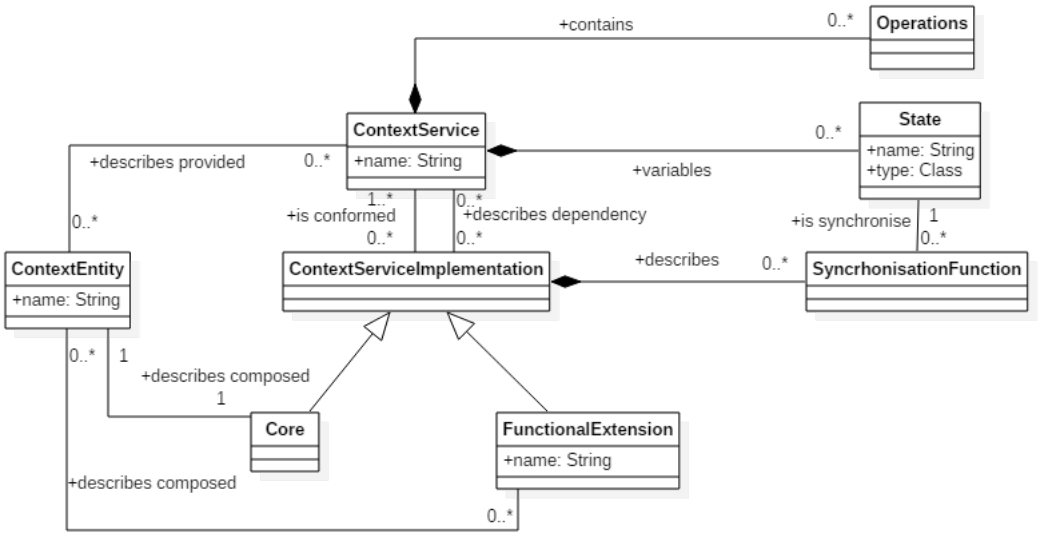

Figure 4.8: Modélisation des éléments représentants une entité de contexte

L'idée principale de notre modèle est de construire ces entités de contexte de façon modulaire, de sorte à ce que la séparation des préoccupations effectuée au niveau de la conception des services se retrouve dans le travail du développeur et le facilite. L'assemblage et la gestion dynamique des instances de module est géré par l'instance de l'entité de contexte ce qui permet de décharger le développeur de cette tâche. Chaque module encapsule une implémentation de service basée sur la philosophie POJO [Fow00] [Esc08], et développée de manière indépendante. Celle-ci peut ainsi décrire ses propres dépendances de code externe sous forme de services, décrire sa logique de synchronisation (voir section 4.2.4) ou préciser d'autres besoins non-fonctionnels à l'aide d'annotations [Esc08]. Il est intéressant de pouvoir différencier les implémentations de service sans lesquelles l'entité de contexte n'as pas de sens et celles que l'on peut considérer comme optionnelles. Cela permet de faciliter le travail de conceptu-

alisation du développeur mais aussi de dégager des propriétés particulières propres aux deux cas. Pour faire écho à cela, nous proposons de packager les implémentations en deux modules différents :

Le Core: c'est le module principal de l'entité de contexte. Il contient les implémentations des services sans lesquelles l'entité de contexte n'a pas de sens. Cette implémentation contient la plus-value technique d'un fournisseur en particulier et se retrouvent donc couplées à des technologies spécifiques. Cela limite son potentiel de réutilisation. A l'exécution, il représente le fonctionnement nominal de l'entité de contexte. Son passage dans l'état invalide 4.10 provoque l'arrêt de l'instance de l'entité de contexte.

Les extensions fonctionnelles: elles sont utilisées pour venir greffer un ensemble de services, et leur implémentation associée, à l'apparition d'une instance d'entité de contexte et de sa mise à l'état valide mais qui peuvent être considérés comme optionnels durant l'exécution. Ces implémentations sont relativement indépendantes des technologies spécifiques du fournisseur. Cela les rend réutilisables au développement par plusieurs fournisseurs et potentiellement remplaçables à l'exécution. A l'exécution, le passage à l'état invalide d'une extension fonctionnelle entraine le retrait du service qui lui était associé mais ne perturbe pas le fonctionnement nominal de l'entité de contexte.

A l'exécution (voir figure 4.9) , l'instance d'entité de contexte se chargera d'effectuer l'instanciation, la configuration et la composition des différents modules de manière **opportuniste**. Chaque module instancié dispose d'un cycle de vie (voir figure 4.10) qui lui est propre et l'instance d'entité de contexte doit en tenir compte pour gérer son propre état ainsi que pour adapter les différents services exposés.

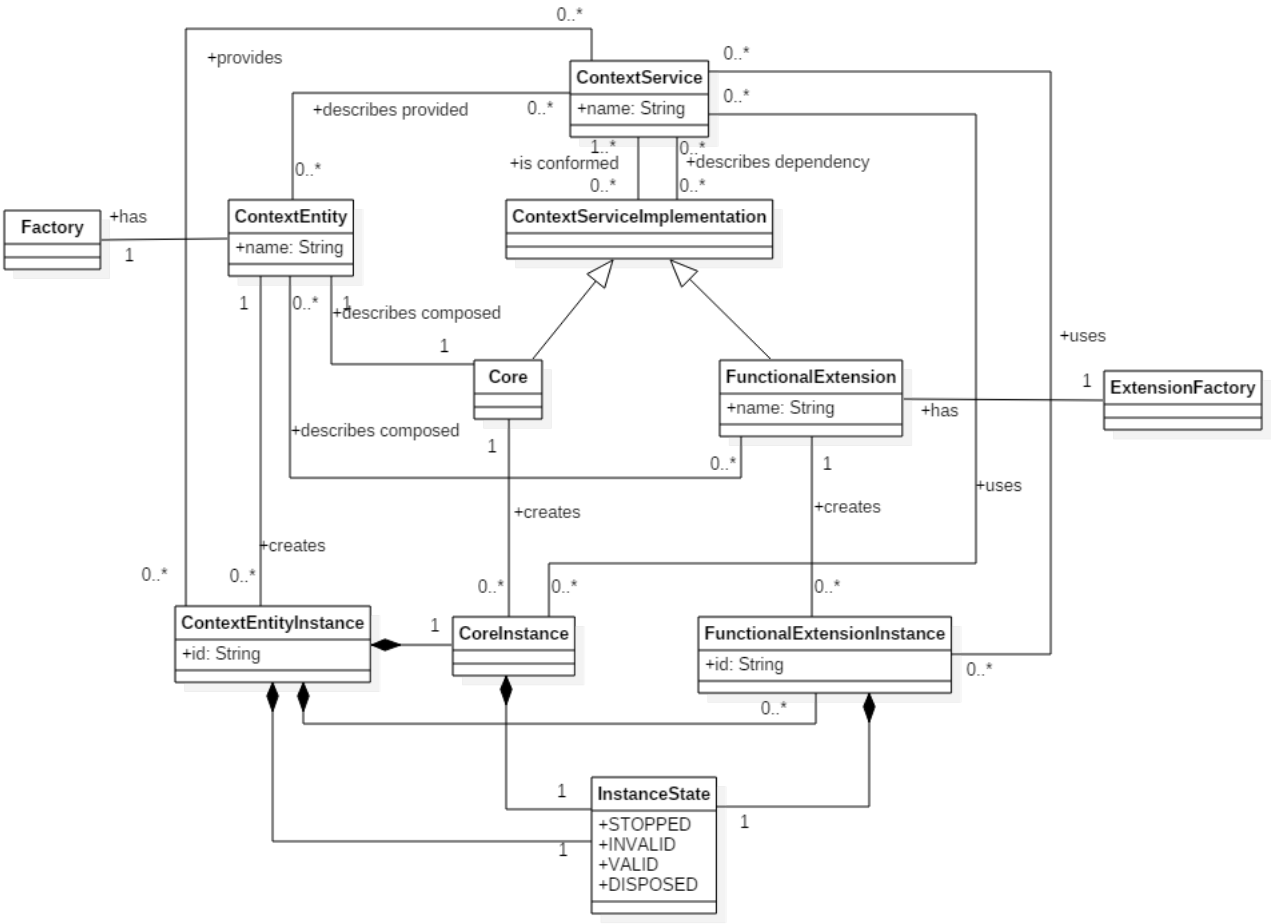

Figure 4.9: Modèle de la relation entre entité de contexte et instance d'entité de contexte

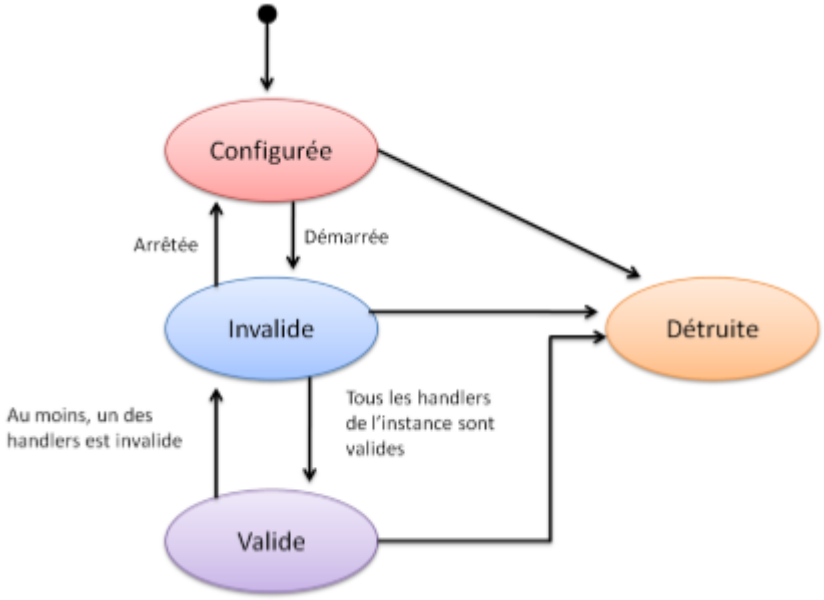

Figure 4.10: Cycle de vie des modules de contexte, calqué sur celui d'iPOJO [Esc08]

Conceptuellement, notre modèle permet donc au fournisseur de définir séparément :

Les fonctionnalités prises en charge: l'entité de contexte fait référence à un ensemble de contrats de service qui sert de référence aux préoccupations de contexte qu'elle prend en charge et à leurs fonctionnalités associées. Les contrats décrivent donc de manière explicite l'ensemble des opérations et des évènements disponibles à la consommation. Cette description est indépendante des techniques ou des technologies nécessaires à sa réalisation.

La réalisation de ces fonctionnalités: l'entité de contexte décrit comment vont être réalisées les fonctionnalités prises en charge. Cette description permet d'assigner à un module de programmation la responsabilité de la réalisation d'une fonctionnalité. Cette description permet à l'entité de contexte de définir quels services seront obligatoirement fournis et ceux considérés comme optionnels et qui viennent enrichir les premiers.

Programmatiquement, une entité de contexte est la réalisation d'une implémentation modulaire de plusieurs contrats de service. Pour être conforme à l'ensemble des services qu'elle peut fournir, une entité de contexte dispose de deux types de modules de programmation, chacun encapsulant une implémentation de service, mais utilisé dans un but différent : le *Core* et les extensions fonctionnelles.

4.2.3.1 Le rôle du Core

Le *Core* est le module principal de l'entité de contexte. Conceptuellement, il permet d'encapsuler l'implémentation des services que le développeur estime nécessaires pour atteindre

dre un fonctionnement nominal de l'entité de contexte. Ce qu'il contient est donc fortement influencé par ce que le développeur essaye de représenter.

Dans notre exemple illustré dans la figure 4.11, le *Core* contient toute la logique des services directement liés aux équipements que l'on essaye de représenter. Mais dans le cadre de la représentation de concepts plus abstraits qu'un équipement, comme par exemple un service de luminosité qui permettrait de gérer la luminosité d'une pièce en s'abstrayant des équipements présents, celui-ci peut aussi contenir des opérations de médiation comme une agrégation des différents capteurs de luminosité de la pièce. Le *Core* présente donc les caractéristiques suivantes :

Au développement: Son implémentation est couplée aux besoins spécifiques des fournisseurs. Cela la rend souvent couplée à des technologies spécifiques ce qui limite son potentiel de réutilisation.

A l'exécution: Le *Core* est le module principal de l'entité de contexte nécessaire à son fonctionnement nominal. En conséquence, l'arrêt de celui-ci provoque l'arrêt de l'entité de contexte.

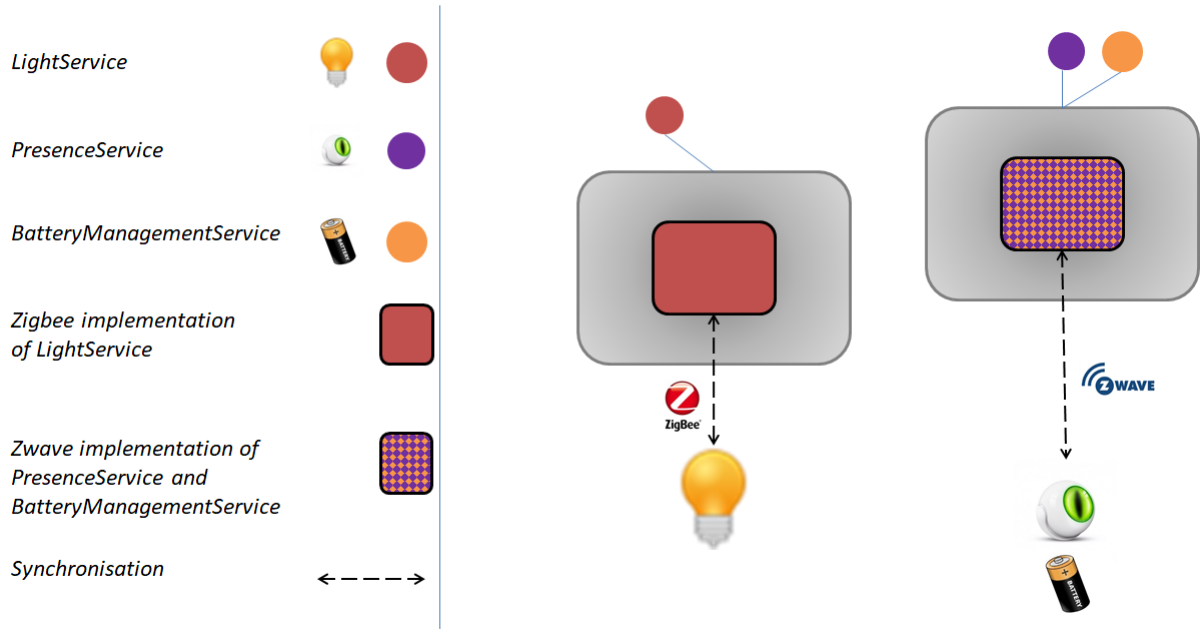

Figure 4.11: Implémentation des services de contexte avec seulement le module *Core*

4.2.3.2 Le rôle des extensions fonctionnelles

La terminologie d'extension fonctionnelle a été choisie car ces modules ont pour but de fournir l'implémentation auto-contenue d'une fonctionnalité, c'est-à-dire d'un service, et de pouvoir

facilement venir la greffer à une entité de contexte et l'ensemble de services déjà implémentés par son *Core*. On souhaite ainsi marquer une différence avec les extensions non-fonctionnelles (appelés par exemple *Handler* dans iPOJO [Esc08]) qui ont pour but de conférer des propriétés non-fonctionnelles (persistance, *logging* par exemple) à une implémentation de service existante.

Dans notre exemple illustré dans la figure 4.13, le service de localisation peut être implémenté comme une extension fonctionnelle. Celle-ci peut être basée sur une interface web permettant à l'utilisateur de placer ses équipements, cette implémentation étant fournie par la plateforme. Mais elle peut être aussi développée selon d'autres techniques comme par exemple la puissance des signaux émis et reçus pour un protocole sans fil, une base de données externe stockant ces informations, etc... Grâce à notre concept d'extension fonctionnelle, le développeur peut très rapidement changer d'implémentation de référence selon les caractéristiques de la plateforme qu'il cible et des propriétés de ce qu'il essaye de représenter.

Une extension fonctionnelle possède les caractéristiques suivantes :

Au développement: elle est indépendante ou faiblement couplée aux technologies spécifiques du fournisseur. Cela permet de la réutiliser entre plusieurs fournisseurs.

A l'exécution: elles sont intégrées comme des extensions de conteneur, utilisées par l'instance d'entité de contexte pour proposer de nouveaux services. Si l'une d'elles se retrouve dans l'état invalide le service qui lui est associé est retiré du registre par l'entité de contexte. L'entité de contexte continue de fournir les services liés à son fonctionnement nominal et ceux des extensions encore valides.

Les extensions fonctionnelles sont aussi un excellent moyen pour permettre à des tierces parties de mettre à disposition leur implémentation de certains services. En effet, les développeurs d'entité de contexte spécifient uniquement leurs besoins en extensions fonctionnelles sans venir perturber le développement de leur *Core*. Cela leur permet de doter très facilement leur entité de contexte de nouvelles fonctionnalités, déjà implémentées et testées, à moindre coût si cela a du sens. Dans une plateforme *Fog Computing*, on imagine très bien à travers notre exemple 4.13 que l'implémentation du service de localisation est fournie par la plateforme pour décharger l'ensemble des fournisseurs d'équipements de cette tâche.

Cela permet donc d'accélérer l'intégration de nouveaux fournisseurs répondant aux besoins des applications au sein du module de contexte, donc son **extension**, tout en maintenant un niveau de la qualité du développement suffisant.

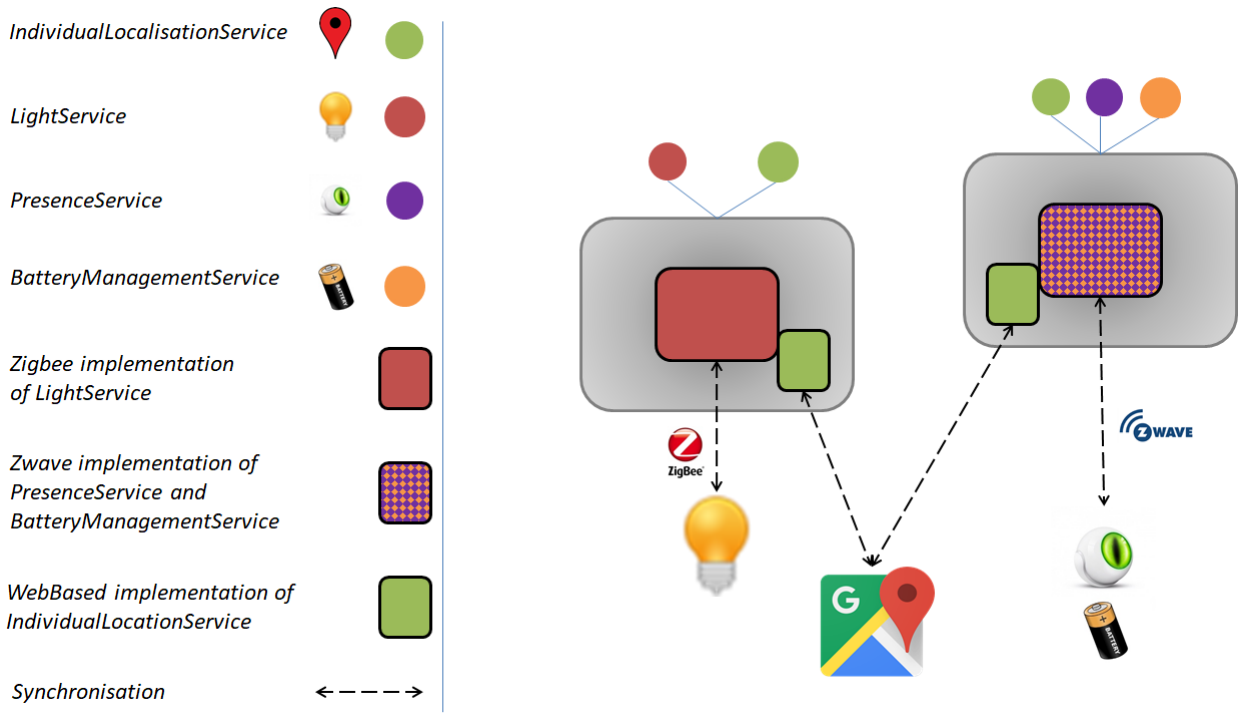

Figure 4.12: Implémentation des services de contexte avec *Core* et extension fonctionnelle.

4.2.3.3 Propriétés du modèle à la conception

Les modules de programmations proposés sont implémentés de manière **simple**, **cohérente** et **homogène**.

La *simplicité*, comme introduit par [Gab91] , met en exergue le fait que la conception, aussi bien pour les interfaces que pour les implémentations, doit rester simple. Cela permet d'éviter toute complexité inutile rendant un système obscur, difficile à comprendre et à maintenir et qui risque d'introduire des erreurs au cours du temps. Pour donner du corps à cette idée, on peut prendre par exemple Apache Felix iPOJO qui propose un modèle de développement uniquement basé sur le concept de POJO [Fow00], concept homogène et commun à tout développeur java. Notre modèle de programmation se veut dans la continuité de cette approche.

Comme résumé dans [Pre92], *un module cohérent* doit idéalement réaliser une seule tâche. Nos modules sont programmés de manière cohérente car il se concentre sur l'implémentation d'un nombre très restreint de services, tendant vers un.

L'implémentation de ces deux types de modules est *homogène* car elle repose sur le même ensemble de concept : programmation d'un service sous la forme d'un POJO, intégration possible des préoccupations non-fonctionnelles sous forme d'extension de conteneur, description

de la logique de synchronisation et description des relations avec d'autres entités de contexte sous forme de dépendances.

Notre modèle se caractérise aussi par une grande **souplesse**. Le développeur garde la main sur le découpage de son fournisseur en choisissant quel service correspond au fonctionnement nominal de son entité de contexte et ceux additionnels. Le concept d'extension fonctionnelle permet de développer indépendamment du fournisseur certaines fonctionnalités et de venir facilement les intégrer sans génération de code supplémentaire.

4.2.3.4 Propriétés du modèle à l'exécution

A l'exécution, comme illustré par la figure 4.9, une fabrique [Gam+95] (*factory* en anglais) est attachée à chaque entité de contexte et pour chaque extension fonctionnelle. Les fabriques permettent respectivement la création d'instances d'entité de contexte, et par effet de bord de son module *Core*, et d'instances d'extensions fonctionnelles. Ces fabriques sont enregistrées comme des services, ce qui permet de les découvrir à l'exécution. Lors de la création d'une instance d'entité de contexte, une configuration minimale contenant un identifiant unique doit lui être passé. La configuration peut s'étendre à d'autres paramètres de l'instance, comme présenté dans [Esc08]. Une fois instanciée, l'instance d'entité de contexte peut traquer les fabriques des extensions fonctionnelles et commencer à instancier les extensions fonctionnelles appropriées. Cela signifie qu'à tout moment une instance d'entité de contexte connait aussi les possibles adaptations liées au remplacement d'une extension fonctionnelle par une autre fournissant le même service. De plus, cela permet aux entités de contexte de supporter une mise à jour à chaud des extensions fonctionnelles sans perturber leur fonctionnement nominal.

A l'exécution, une des spécificités de notre modèle vient du fait que le code implémentant les différents services est encapsulé dans différents modules qui subissent de façon indépendante des contraintes liées au dynamisme de la plateforme lors de leur instanciation. Chaque module de l'instance d'entité de contexte a donc potentiellement un cycle de vie indépendant, calqué sur celui d'iPOJO [Esc08] rappelé en figure 4.10. Ce cycle de vie peut donc être impacté et modifié au cours de l'exécution, comme par exemple avec dépendance de service non résolue qui empêche le module de fournir le service qui lui est associé.

Le rôle de l'instance d'entité de contexte est de venir assurer un cycle de vie cohérent et unifié à tous ces modules. Elle vient réaliser une composition des différentes instances de module, qui peut évoluer dans le temps selon leur état, en créant dynamiquement un *Proxy* [Gam+95] agissant comme une *Facade* [Gam+95] pour l'ensemble de services exposés. Ce *proxy*, nommée enveloppe fonctionnelle, est l'objet fourni lors de liaisons (comme illustré par la figure 4.13), ce qui permet au consommateur de manipuler un unique objet implémentant toute les spécifications requises. Le comportement du proxy est basé sur le patron *Chain Of Responsability* [Gam+95] pour trouver le module à qui déléguer les fonctionnalités con-

sommées. Chaque fois qu'une extension fonctionnelle devient invalide le cycle précédent est répété en ne prenant pas en compte le service qui lui est associé. Ainsi, les services exposés par l'entité de contexte varient selon le temps et peuvent s'adapter selon les caractéristiques de la plateforme.

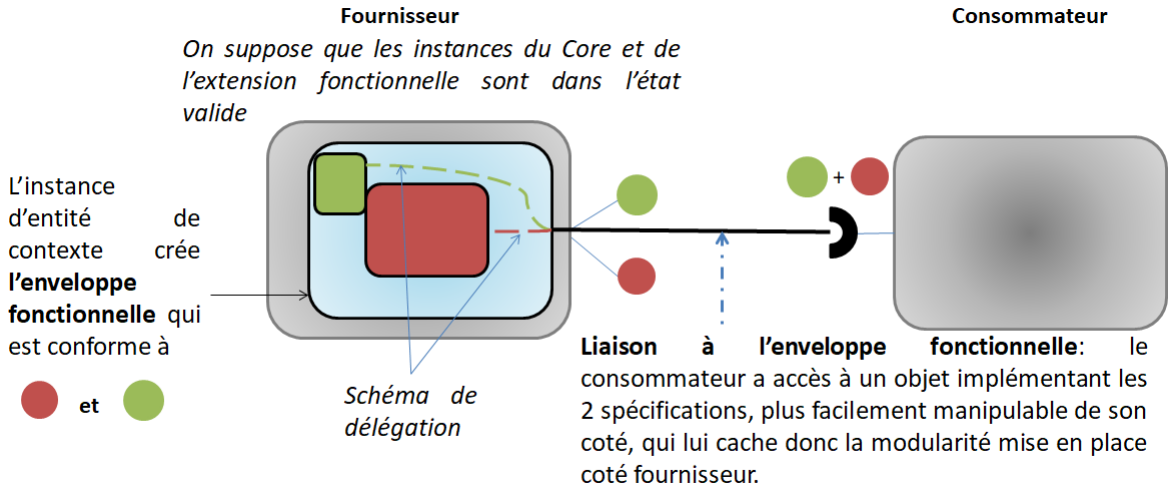

Figure 4.13: Illustration de la création de l'enveloppe fonctionnelle

Par exemple, dans la figure 4.14 , une coupure du réseau peut rendre invalide l'implémentation du service de localisation individuelle. Le rôle de l'instance de l'entité de contexte est dans ce cas-là de détecter ce changement, de venir ajuster les services qu'elle expose pour permettre aux services dont les implémentations sont valides de continuer à être fournis. Cela permet entre autre d'assurer l'exécution continue des applications ne dépendant pas de cette fonctionnalité, comme l'application *3* dans notre exemple. En cas de redémarrage de l'implémentation du service de localisation, l'instance d'entité de contexte réadaptera l'ensemble de services qu'elle fournit. Ceci permettra à l'application *1* et à l'application *2* de reprendre leur exécution.

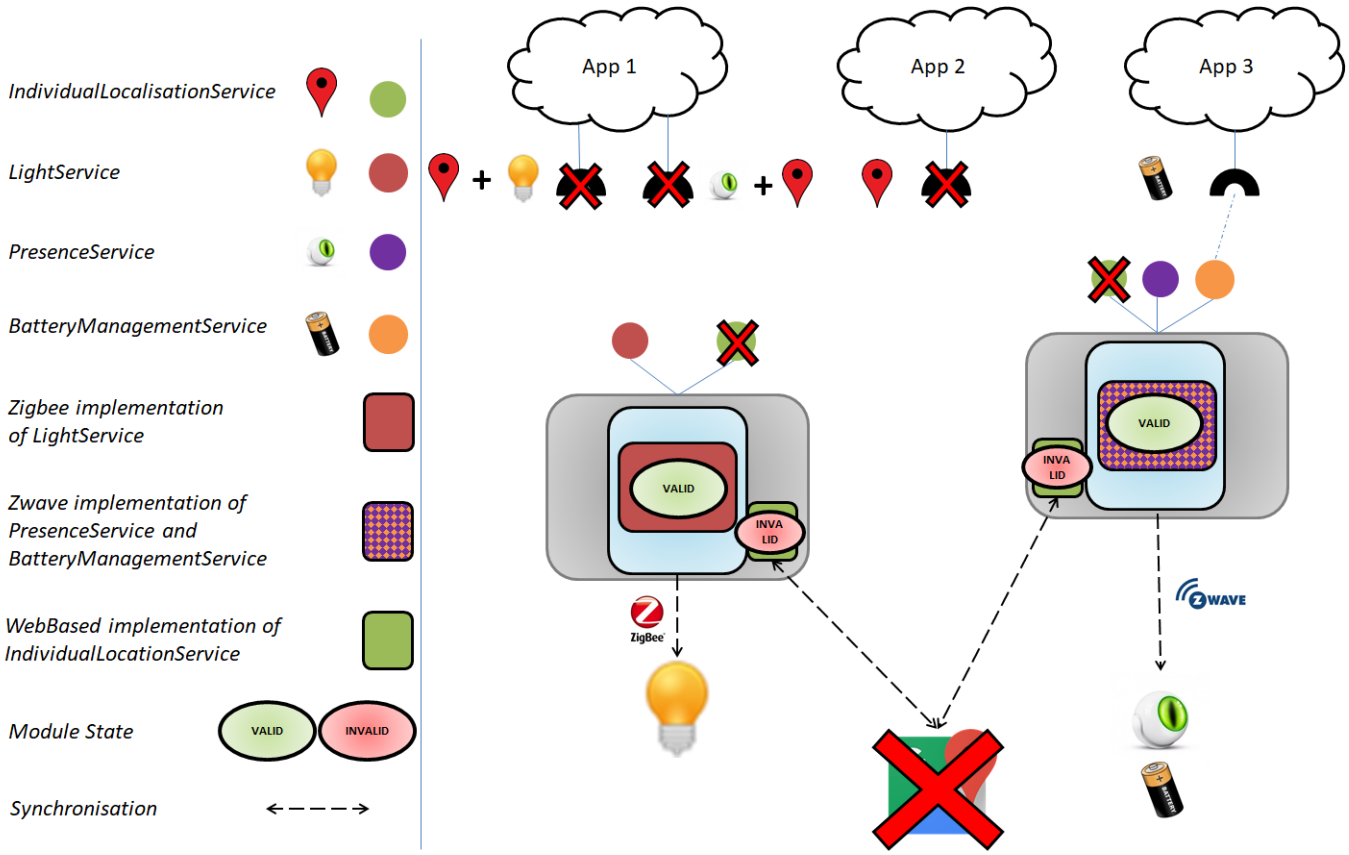

Figure 4.14: Exemple du dynamisme des entités de contexte

4.2.4 Implémentation d'un fournisseur de contexte

L'implémentation d'un service de contexte repose sur un ensemble d'exigences spécifiques. Un des patrons d'implémentation est de maintenir à l'intérieur de l'implémentation du service de contexte une représentation in-memory d'un objet physique de l'environnement et d'essayer de maintenir cette représentation synchronisée. Pour mettre en place ce patron, le développeur se retrouve donc à interagir avec des objets physiques et à développer du code propice à l'erreur concernant les fonctionnalités de synchronisation et la capture d'évènements.

Au regard de ces besoins, nous proposons un DSL permettant d'alléger le travail du développeur. Son modèle et son résumé sont disponibles respectivement dans la figure 4.15 et les deux tableaux 4.2 et 4.3. Notre DSL encourage à adopter le patron précédemment évoqué, tout en essayant de conserver un modèle de développement de la logique business du service sous la forme d'un POJO. Pour cela, nous nous appuyons sur la notion de *State.Field*. Chaque variable d'état définit dans le contrat de service doit correspondre à un champ de l'implémentation, le *State.Field*, et lorsqu'il est valué correspond à la représentation en mémoire de l'environnement. Ces attributs permettent de maintenir la programmation du service au même niveau de simplicité que la programmation orientée objet traditionnelle. Cependant, le développeur a la possibilité d'associer à chacun de ces attributs un ensemble de fonctions, dites de synchronisation. Ces fonctions de synchronisations servent à refléter le fait que chaque tentative de modification de l'état en mémoire doit affecter l'environnement et que chaque évènement se produisant dans l'environnement doit être capturé par l'état en mémoire.

Le développeur dispose des trois principaux patrons de synchronisation identifiés dans la littérature aux travers des fonctions suivantes :

La fonction *pull*: A chaque tentative de lecture de la valeur, cette fonction est appelée. En cas de spécification d'une période, la machine d'exécution du modèle à composant s'occupera de l'appeler périodiquement. Elle a pour effet de venir mettre à jour la valeur de la variable d'état en venant interroger la source de contexte disséminé dans l'environnement. Elle est principalement utilisée pour la programmation des capteurs qui se mettent à jour de manière synchrone et/ou périodique.

La fonction *apply*: A chaque tentative d'écriture de la valeur, cette fonction est appelée. Elle doit avoir pour effet de venir modifier l'environnement, pour refléter une action souhaitée par un consommateur du service. Elle est principalement utilisée pour la programmation des actionneurs.

La fonction *push*: Lorsque qu'un évènement se produit dans l'environnement, celui-ci doit transiter par la méthode push pour venir mettre à jour la valeur de la variable d'état. Elle est principalement utilisée pour la programmation d'objets qui se mettent à jour de manière asynchrone.

L'utilisation de cette approche déclarative confère plusieurs avantages :

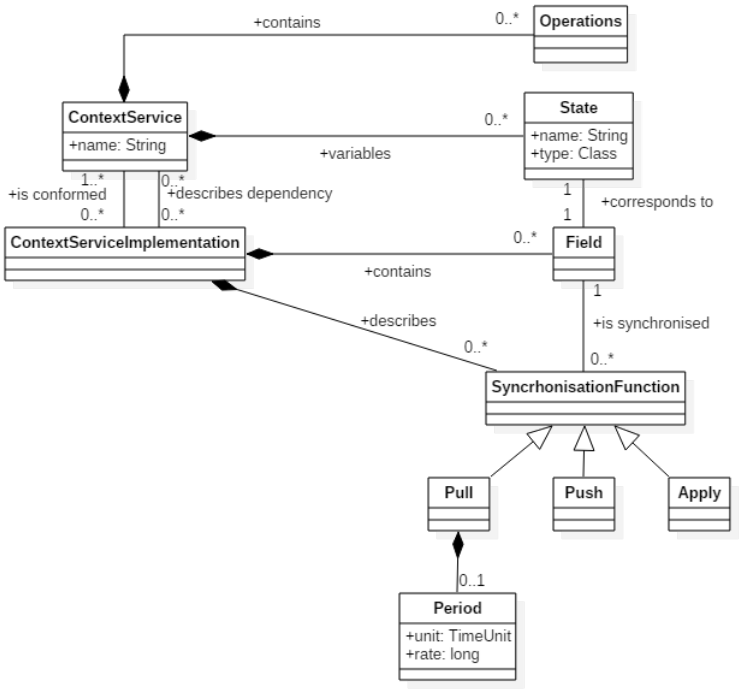

Figure 4.15: Modèle de la description des fonctions de synchronisations

Elle permet l'adoption des patrons de synchronisation identifiés par la communauté ce qui facilite la programmation.

Elle réduit la complexité du programme en déléguant l'orchestration de ces patrons à la machine d'exécution.

Elle facilite les tests et le *debug* de la synchronisation car le code lié à celle-ci est clairement identifié et packagé au sein des patrons précédemment évoqués.

Pour illustrer notre proposition, nous proposons d'étudier l'exemple 4.16 du développement du service de contexte *BinaryLight* s'appuyant sur le protocole Zigbee. En préambule, nous assumerons que le code gérant le protocle Zigbee est encapsulé dans l'interface *ZigbeeDriver*.

```
1  public class ZigbeeBinaryLight implements BinaryLight, ZigbeeDeviceTracker {
2
3  @ContextEntity.State.Field(service = BinaryLight.class, state =
       BinaryLight.POWER_STATUS, value = "false")
4   private boolean powerStatus;
5
6  @Requires private ZigbeeDriver driver;
7
8  @Override public boolean getPowerStatus() {return powerStatus;}
9
10 @Override public void turnOn() { powerStatus = true;}
11
12 @Override public void turnOff() { powerStatus = false;}
13
14 @ContextEntity.State.Apply(service = BinaryLight.class,state = POWER_STATUS)
15  Consumer<Boolean> setPowerStatus = newPowerStatus -> {
16           if (newPowerStatus) {
17                driver.setData(moduleAddress, "1");
18            } else {
19                driver.setData(moduleAddress, "0");
20            }
21  };
22
23 public void deviceDataChanged(String address,Data oldData, Data newData){
24           if(address.compareTo(this.moduleAddress) == 0){
25                        pushPowerStatus(newData.getData());
26           }
27 }
28
29 @ContextEntity.State.Push(service = BinaryLight.class,state = POWER_STATUS)
30 public boolean pushPowerStatus(String data){
31          return data.compareTo("1")==0? true : false;
32 }
```

Figure 4.16: Code java relatif à l'implémentation d'un service de contexte de type *BinaryLight* utilisant le protocole Zigbee

Dans cet exemple, le champ *powerStatus* garde en mémoire l'état courant de la lampe. Celui-ci est annoté grâce à *@ContextEntity.State.Field* et se comporte comme un champ Java normal. Par exemple, l'implémentation des méthodes *turnOn* et *turnOff*, qui allument et éteignent respectivement la lampe, vient modifier directement la valeur du champ. Comme précisé précédemment, toute modification du champ doit entrainer une répercussion sur l'environnement, en utilisant l'actionneur Zigbee approprié. Ici, le code effectuant la synchronisation est effectué à l'intérieur de la méthode annotée avec *@ContextEntity.State.Apply*. A chaque fois que le champ powerStatus est modifié, la fonction *setPowerStatus* est invoquée, ce qui va permettre de déléguer au driver Zigbee l'envoi d'une trame appropriée pour éteindre ou allumer la lampe. Dans notre cas, une lampe Zigbee peut être physiquement allumée ou éteinte par l'utilisation d'un bouton. Ces changements asynchrones sont capturés par la méthode *@ContextEntity.State.Push* qui permet de mettre à jour l'état en mémoire à chacun de ses appels. Cette méthode permet de mener des opérations de médiation en convertissant les données brutes du capteur en un format plus facile à traiter.

Annotation	Description	Configuration	Description
@ContextEntity.State.Field	Permet de marquer un attribut de l'implémentation comme une propriété dont la synchronisation est déléguée à un ensemble de fonctions spécifiques	*Service*	Service de contexte lié à la variable d'état
		State	Nom de la variable d'état.
		Value	Valeur par défaut de la variable d'état
		DirectAccess	Permet de définir si l'on veut utiliser ou non les différentes fonctions de synchronisation associées
@ContextEntity.State.Pull	Permet d'identifier une fonction de synchronisation, relative à une propriété, de type *Pull*. Cette fonction définit le jeu d'actions nécessaires à effectuer pour récupérer une valeur particulière dans la source de contexte. Elle peut être orchestrée soit de manière ponctuelle ou périodique selon les besoins	*Service*	Service de contexte lié à la variable d'état à synchroniser
		State	Nom de la variable d'état à synchroniser
		Period et *Unit*	Si l'on souhaite que la fonction de synchronisation soit périodique, on peut définir une période qui permettra à l'exécution de déterminer la fréquence de synchronisation souhaitée

Table 4.2: Résumé du DSL permettant de décrire les fonctionnalités de synchronisation partie 1

Annotation	Description	Configuration	Description
@ContextEntity.State.Push	Permet d'identifier une fonction de synchronisation, relative à une propriété, de type *Push*. Cette fonction définit le jeu d'actions nécessaires à effectuer lorsqu'une valeur est transmise par la source de contexte à l'entité de contexte	*Service*	Service de contexte lié à la variable d'état à synchroniser
		State	Nom de la propriété à synchroniser
@ContextEntity.State.Apply	Permet d'identifier une fonction de synchronisation, relative à une propriété, de type *Apply*. Cette fonction définit le jeu d'instructions à effectuer lorsqu'une entité de contexte souhaite appliquer un changement particulier relatif à une tentative de modification de valeur d'une propriété à la source de contexte qu'elle représente	*Service*	Service de contexte lié à la variable d'état à synchroniser
		State	Nom de la variable d'état à synchroniser

Table 4.3: Résumé du DSL permettant de décrire les fonctionnalités de synchronisation partie 2

4.2.5 Approvisionnement des fournisseurs de contexte

L'approvisionnement des services de contexte est le processus qui implique le déploiement et l'instanciation des entités de contexte. Ce processus est guidé par un ensemble d'évènements extérieurs : un équipement qui rejoint le réseau, une nouvelle application déployée dans la plateforme ou la demande explicite de l'administrateur de la plateforme.

Une partie de ce processus peut être automatisée par des approches comme RoSe [Bar12] ou MUSIC [Rou+09] qui fournissent un patron pour rendre modulaire et maintenir la découverte des évènements extérieurs à l'exécution. Comme illustré par la figure 4.17, pour être facilement intégrable dans ce type d'approche nous fournissons une API de création d'instance qui permet de réduire grandement la complexité technique de ce type d'approche. Les processus de découverte d'évènements extérieurs émettent des demandes d'instanciation grâce à notre API et laissent au module de contexte la tâche d'instancier et de configurer les entités correspondantes.

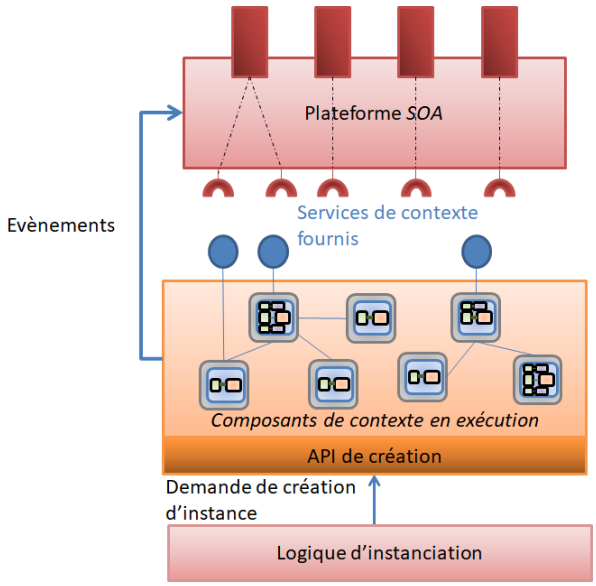

Figure 4.17: Illustration de la présence d'une API aidant à la création d'instance d'entité de contexte

4.2.6 Implémentation d'un consommateur de contexte

La consommation des services est facilitée naturellement dans iPOJO qui fournit un modèle de dépendances très riches [Esc08], comme illustré dans la figure 4.18. Il intègre notamment des options pour gérer la cardinalité, le classement et le filtrage des dépendances.

```
1   @Component(name="LightFollowMeApplication")
2   @Instantiate
3   public class TemperatureRegulationApplication {
4
5   Requires(id="thermometers", optional=true, specification=Thermometer.class,
        filter=Thermometer.THERMOMETER_CURRENT_TEMPERATURE + ">20")
6   private List<Thermometer> thermometers;
7
8   Bind(id="thermometers")
9   public void bindPresence(Thermometer thermometer){
10  \\ implementation
11  }
12
13  Unbind(id="thermometers")
14  public void unbindPresence(Thermometer thermometer){
15  \\ implementation
16  }
17
18  ContextUpdate(specification=Thermometer.class,
        stateId=Thermometer.THERMOMETER_CURRENT_TEMPERATURE)
19  public void updateState(Thermometer thermometer, Quantity<Temperature>
        newTemperature, Quantity<Temperature> oldTemperature){
20  \\ implementation
21  }
22  }
```

Figure 4.18: Exemple d'un composant iPOJO consommateur d'un service de contexte

Une méthode annotée *@Bind* permet au consommateur d'être notifié lors de l'apparition d'un nouveau fournisseur. Son pendant, une méthode annotée par *@Unbind*, notifie le consommateur du départ du fournisseur de service.

Nous avons étendu ce modèle de gestion de dépendances pour faciliter la consommation d'évènements produits par les fournisseurs. Le consommateur peut dorénavant spécifier être intéressé par les évènements liés à une variable d'état particulière. Il dispose pour cela d'une annotation *@ContextUpdate* paramétrée par le nom de la variable d'état, voir la figure 4.18 . A l'exécution, le consommateur sera notifié des changements des fournisseurs de service par l'appel de la méthode annotée.

Annotation	Description	Configuration	Description
@ContextUpdate	Permet de spécifier vouloir être notifié des changements de valeur d'une variable d'état	*specification*	Spécification où la variable d'état est décrite
		stateId	Nom de la variable d'état

Table 4.4: Résumé du DSL permettant de consommer des évènements de contexte

Les applications peuvent donc réagir aux principales évolutions du contexte, qui sont la disponibilité ou le retrait d'un fournisseur de contexte et les changements des variables d'état des services de contexte.

4.3 Autonomie du Modèle

4.3.1 Présentation générale

La boucle de contrôle globale de notre solution, représentée sous la forme d'un manager autonomique dans la figure 4.19, a pour rôle de gérer le module de contexte dans son ensemble et de l'adapter pour satisfaire les besoins des applications. Un choix majeur de notre approche est de limiter l'ensemble des composants en exécution à seulement ce dont les applications ont besoin. Cela permet de préserver les ressources de la plateforme et des équipements de l'environnement. Pour faciliter la mise en place de cette boucle autonomique, nous avons adopté et développé les stratégies suivantes :

Des capteurs permettant d'acquérir des informations sur le module de contexte en exécution comme sa topologie ou d'obtenir des informations plus précises sur un de ses éléments (état du composant, valeur des variables d'états, extensions fonctionnelles présentes).

Des effecteurs permettant de modifier dynamiquement le module de contexte et sa topologie ainsi que ses différents constituants à divers niveaux de granularité.

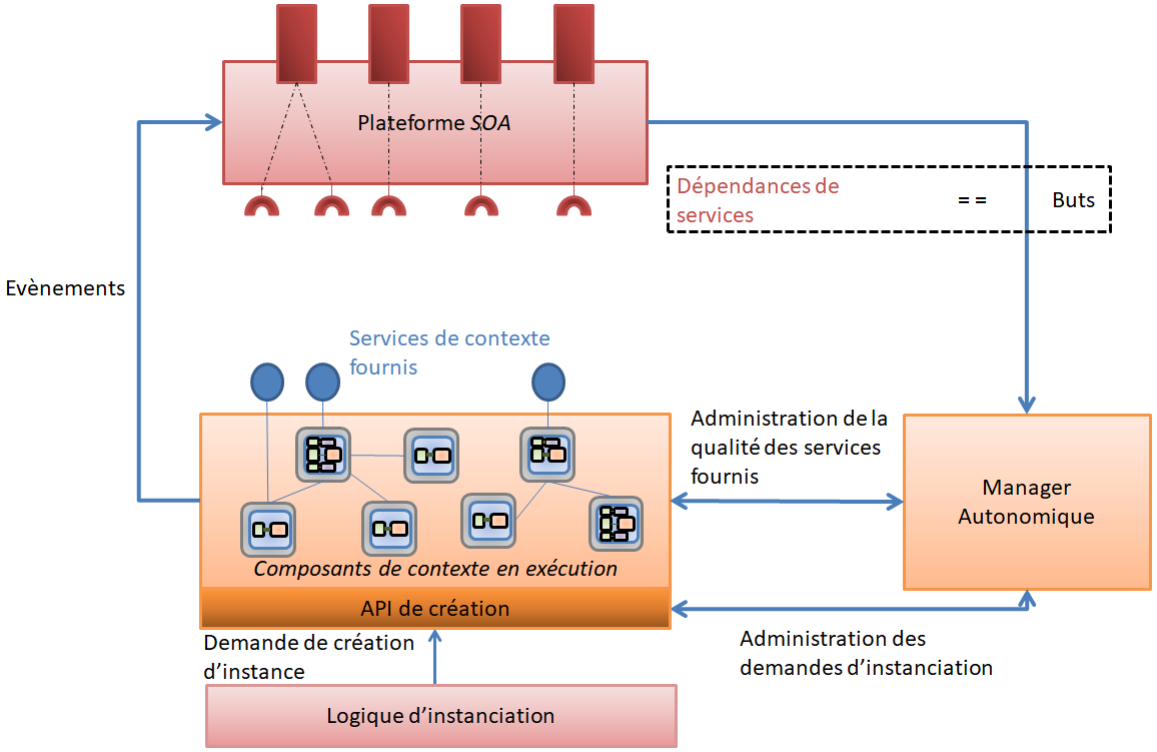

Figure 4.19: Framework à l'exécution de la solution de contexte

Pour effectuer cela, tous les éléments abordés dans la description de notre modèle à composant existent à l'exécution. Ceux-ci sont rendus introspectables par des API

d'administration exposées sous forme de service. Nous allons examiner plus en détail le contenu et les possibilités offertes par ces services d'administration.

4.3.2 Point de contrôle des demandes d'instances

Le premier point de contrôle de notre infrastructure concerne les demandes d'instanciation, comme illustré par la figure 4.20. Ce point de contrôle offre les perspectives suivantes :

Connaitre les demandes d'instanciation et leur état: on peut avoir accès à toute les demandes d'instanciation qui ont été produites jusqu'au moment de l'appel. Cela permet de savoir ce qui a été découvert dans l'environnement. De plus, il est aussi possible de connaître le contenu de la demande d'instanciation, qui correspond aux différents paramètres de configuration souhaités, et si la demande d'instanciation a été prise en compte ou non par le module de contexte.

Modifier le comportement du module de contexte vis-à-vis d'une demande: on peut explicitement demander qu'une demande d'instanciation soit prise en compte par le module de contexte. Le module de contexte va utiliser la factory correspondante pour lancer le processus d'instanciation et créer une instance d'entité de contexte qui va commencer à se synchroniser avec l'environnement. De la même manière, on peut demander la suppression d'une entité de contexte associée à une demande si celle-ci a été précédemment prise en compte par le module de contexte.

Un excellent moyen d'assurer que ce qui s'exécute sur la plateforme est restreint aux besoins des applications est de contrôler la logique d'instanciation. En venant filtrer les demandes d'instanciation et en autorisant uniquement l'instanciation des entités pouvant fournir des services recherchés par les applications, on s'assure que l'empreinte mémoire du module de contexte est réduite au nécessaire. De plus, le fait d'instancier uniquement ce qui est nécessaire permet d'économiser la batterie des équipements fonctionnant avec une synchronisation de type pull. Le fait de contrôler l'instanciation des différentes entités permet aussi de venir modifier la topologie du graphe de contexte et de laisser celui-ci prendre en compte de manière autonomique l'arrivée de nouvelles entités grâce aux propriétés de composition dynamique.

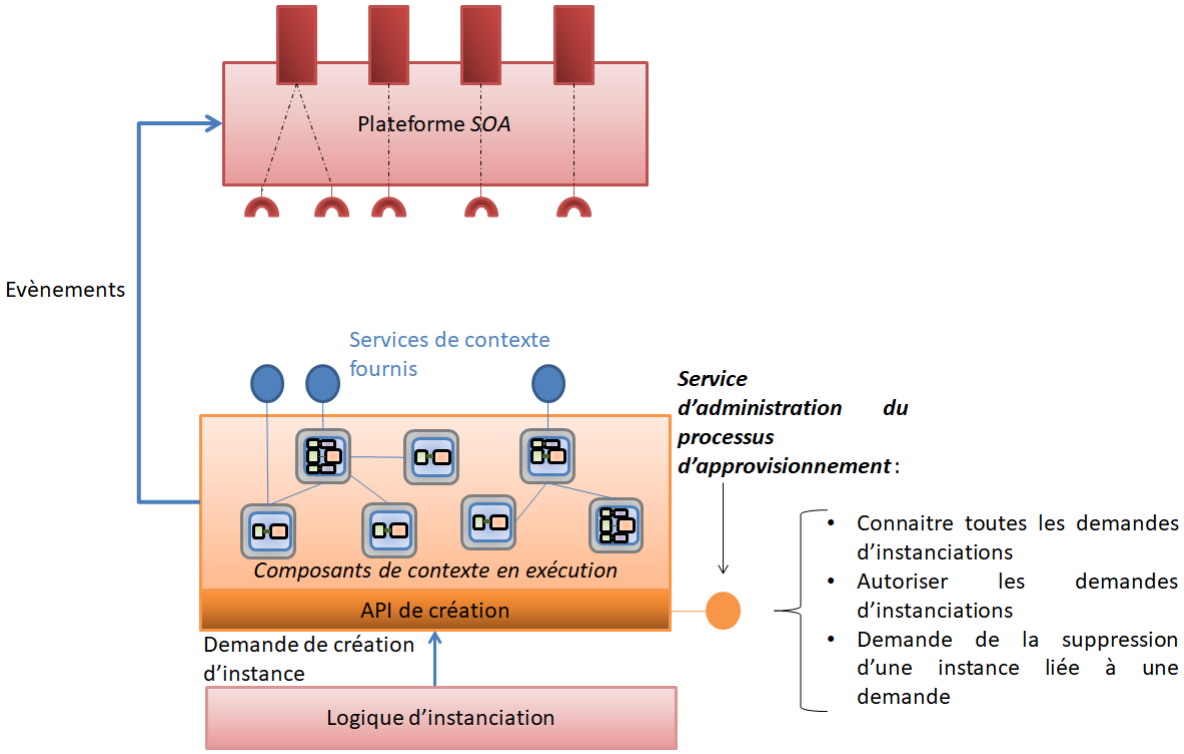

Figure 4.20: Service d'administration des demandes d'instanciation

4.3.3 Point de contrôle des entités instanciées

Le second point de contrôle de notre infrastructure concerne les composants de contexte en exécution, comme illustré par la figure 4.21. Ce point de contrôle offre les perspectives suivantes :

Connaitre la topologie du module de contexte: le service d'administration permet d'obtenir la topologie du module de contexte à l'instant de l'appel. Il retourne l'identifiant des différents nœuds ainsi que les liens qu'il possède avec les autres nœuds du graphe de contexte.

Obtenir la description d'une instance d'entité de contexte: à partir de l'identifiant unique d'une instance d'entité de contexte, le service d'administration peut retourner une description de l'instance au moment de l'appel. Cette description permet à un administrateur de retrouver les concepts du modèle à composant de contexte. Elle permet de connaitre l'état du cycle de vie dans lequel se trouve l'instance ainsi que celui de ses différents modules. On y retrouve aussi des informations sur les différents modules. La description du Core contient la liste des services qu'il doit fournir, la valeur des variables d'état et si des fonctions de synchronisation leur correspondent avec dans le cadre de la fonction pull la fréquence de synchronisation utilisée. La description d'une extension fonctionnelle contient sensiblement les mêmes informations que celle du Core. La différence vient du fait qu'elle y ajoute une liste contenant les extensions fonctionnelles

présentes dans l'environnement d'exécution en capacité de la remplacer car fournissant les mêmes services. Cela est fait pour faciliter le travail de l'administrateur.

Reconfiguration au niveau module: le service d'administration offre la possibilité de reconfigurer une instance d'entité de contexte au niveau module en permettant le remplacement à chaud d'une instance d'extension fonctionnelle. Il suffit à l'administrateur de seulement spécifier l'implémentation de la nouvelle extension fonctionnelle au service d'administration et le module de contexte prend en charge dynamiquement la destruction de l'ancienne instance et la création de la nouvelle, sans perturber le fonctionnement nominal de l'instance d'entité de contexte.

Reconfiguration au niveau paramètre: le service d'administration offre aussi la possibilité de reconfigurer plus finement une entité de contexte en venant modifier les fréquences de synchronisations de la logique de synchronisation.

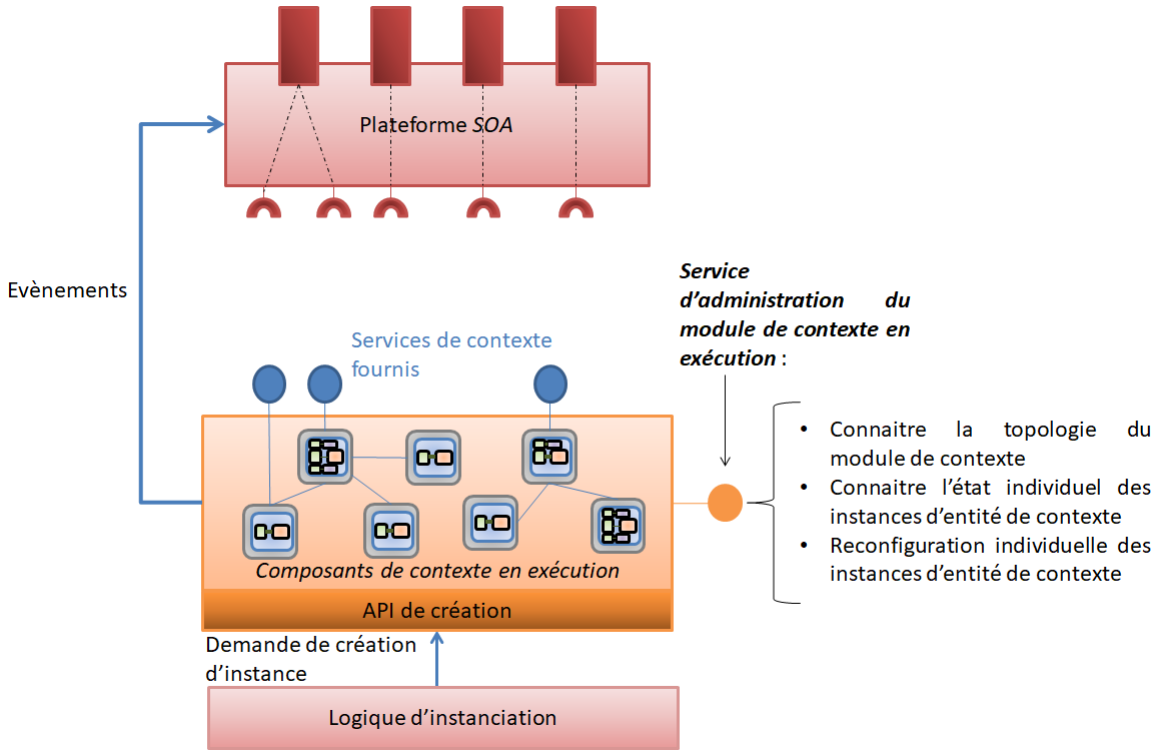

Figure 4.21: Service d'administration du module de contexte

4.3.4 Comportement autonomique

L'administration du module d'exécution des composants est réalisée par un manager autonomique. L'objectif du manager est de garantir que les services publiés correspondent aux dépendances de service exprimées par les applications. Le manager doit donc gérer le dynamisme lié aux applications qui peuvent apparaitre et disparaitre durant l'exécution et

celui lié à l'environnement où de nouvelles sources de contexte peuvent aussi être découvertes, tombées en panne ou supprimées.

Chaque fois qu'une application apparait le manager autonomique inspecte ses dépendances de services de contexte et les compare aux composants en exécution sur la plateforme. Si ceux-ci ne fournissent pas les services requis, le manager inspecte toutes les demandes d'instanciation non satisfaites et autorise celles qui peuvent fournir les services requis par la nouvelle application.

Cependant l'instanciation d'une entité ne garantit en rien sa mise à l'état valide et le fait qu'elle va fournir les services désirés. Lorsque le manager instancie une entité de contexte, celui-ci vérifie que les modules concernant les services attendus sont à l'état valide. Dans le cas où un des modules concernés ne se trouve pas dans l'état valide, le manager avertit l'administrateur de la plateforme avec un message d'erreur disponible sur une interface Web. Si le module invalide est une extension fonctionnelle, l'administrateur peut sélectionner une adaptation proposée par le manager, toujours à travers l'interface Web, pour venir remplacer l'extension défectueuse. Le manager prendra en charge la destruction de la précédente extension fonctionnelle et son remplacement.

En cas de départ d'une application, le manager autonomique inspecte le module de contexte pour déterminer quelles entités sont utilisées ou non par les applications continuant de s'exécuter sur la plateforme. Une fois cela déterminé, le manager prend la responsabilité de détruire les instances non utilisées par l'intermédiaire du service d'administration du processus d'approvisionnement de la plateforme. Cela permet de préserver les ressources de la plateforme et de l'environnement.

4.4 Conclusion

Dans ce chapitre nous avons présenté une vision approfondie de notre proposition. Précisément, nous avons mis en lumière comment notre proposition répond aux différents objectifs de recherche que nous nous étions fixés. Nous résumerons cela dans le tableau suivant :

Facilité d'utilisation	*Développeur d'application*	*Faible Couplage*	Utilisation de SOA. Proposition d'une définition de service contenant les actions disponibles (remontées de données et commandes sur les actionneurs) ainsi que des variables qu'on peut écouter et qui traduisent les changements des états, donc les évènements. A l'exécution, ajout d'un DSL permettant de facilement consommer les différentes fonctionnalités décrites (union de service, évènements)
		Support à la composition dynamique	Utilisation de SOA et d'un développement à base de composants orientés service qui permet une composition dynamique structurelle et comportementale de manière autonome [Esc08].
	Développeur du module de contexte	*Adoption des patterns existants*	Proposition d'un DSL permettant de décrire les différentes formes de synchronisation (*push/pull*) ainsi que leurs variantes (périodique/ponctuelle) trouvées dans la littérature.
		Séparation des préoccupations	Développement à base de SOCM extensible permettant une séparation claire des préoccupations. A l'exécution une partie du conteneur s'occupe des besoins spécifiques du contexte : gestion du cycle de vie des fonctions de synchronisation, propagation des évènements, expositions des services.

Table 4.5: Résumé des propriétés du modèle à composant de contexte partie 1

Support à l'extensibilité	*Modularité*	Les fonctionnalités sont définies de façon modulaire à base de service. Chaque définition est donc indépendante.
	Souplesse	Le développeur garde la maitrise sur son travail et est libre de choisir quel module utiliser ou réutiliser dans son développement.
	Homogénéité	Quels que soient les modules utilisés, ceux-ci bénéficient des mêmes mécaniques de programmation SOCM. Les mêmes concepts sont réutilisés et manipulés par le développeur.
Autonomie	*Modularité*	Les fonctionnalités sont définies de façon modulaire à base de service et implémentées avec un SOCM. Cela permet de les déployer à l'exécution selon les besoins.
	Administrable	La logique d'instanciation des différents modules est contrôlable et les différentes instances à l'exécution peuvent être reconfigurées à différents niveaux.
	Management autonomique externe	Présence d'un manager autonomique externe qui prend en compte les besoins des applications pour fournir les services requis.

Table 4.6: Résumé des propriétés du modèle à composant de contexte partie 2

Chapter 5

Implantation et Validation

Sommaire

Dans ce cinquième chapitre, notre but est de présenter en détails l'implémentation de notre proposition et d'évaluer la viabilité de celle-ci sur un exemple de taille réelle.

Dans un premier temps nous nous intéresserons à l'implémentation de notre solution qui repose en grande partie sur le SOCM Apache Felix iPOJO. Nous aborderons les différentes briques technologiques sur lesquelles nous nous sommes appuyés et nous détaillerons les extensions que nous leur avons apportées. Nous décrirons aussi la mise en place de la dimension autonomique de notre approche. Dans un second temps, nous nous intéresserons à la validation de notre approche qui s'est déroulée en grande partie sur la plateforme domotique iCASA. Nous aborderons brièvement la plateforme en question et son domaine d'application. Nous nous intéresserons par la suite à son module de contexte et sa restructuration grâce à notre approche. On étudiera ensuite comment se traduit la restructuration du module de contexte sur le développement d'une application particulière : un système intelligent de suivi lumineux. Puis dans un dernier temps, nous quantifierons les impacts à l'exécution de notre approche.

5.1 Implantation de la proposition

Dans le chapitre précédent, nous avons pu établir les détails de notre proposition qui se présente sous la forme d'un modèle à composant spécialisé dans la représentation des éléments de contexte ainsi que la machine d'exécution qui lui est associée. Nous avons aussi pu aborder les comportements autonomiques souhaitables pour un tel modèle et la mise en place de points d'administration permettant de les établir.

Durant cette description, nous nous sommes efforcés de fournir une description relativement indépendante d'une plateforme ou d'une technologie particulière. Dans ce chapitre nous présenterons l'implémentation de référence de notre modèle à composant.

Pour cela, nous commencerons par présenter les technologies sur lesquelles nous nous sommes appuyés pour le développement de notre implémentation de référence. Nous aborderons ensuite les évolutions apportées à ces technologies pour permettre la mise en place de notre modèle à composant. Pour finir, nous détaillerons les outils relatifs au management autonomique d'un module de contexte d'une plateforme *Fog Computing*.

5.1.1 Technologies utilisées

Nous avons entièrement développé notre modèle à composant de contexte, baptisé **CReAM** pour **C**ontext **Re**presentation **A**nd **M**anagement, et la machine d'exécution qui lui est associée. Notre implémentation se base sur la pile technologique illustrée par la figure 5.1 . Elle contient principalement les technologies à service OSGi^{TM} [All] [Hal+11] et Apache Felix iPOJO [Escc] que nous allons détailler par la suite.

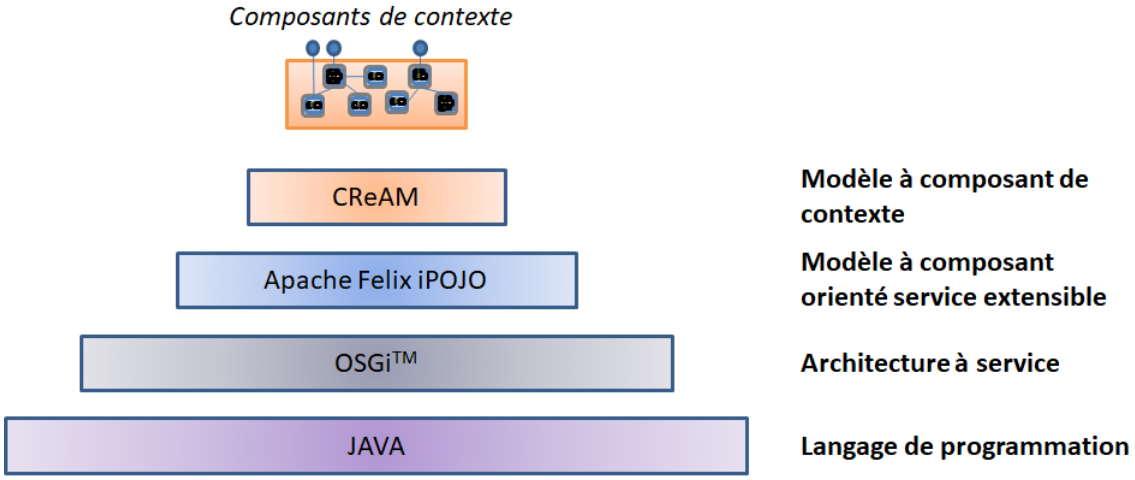

Figure 5.1: Technologies utilisées pour l'implémentation de *CReAM*

Le développement d'une solution entièrement *ad-hoc* dans notre cas semble peu judicieux compte tenu de la complexité des mécanismes et interactions se déroulant à l'exécution et de par l'étendue des travaux déjà réalisés par la communauté SOA et SOCM. Notre choix a donc été dicté par la recherche d'un environnement d'exécution supportant l'architecture orientée service dynamique et permettant de développer de façon modulaire et sous forme de composant orienté service. Nous nous sommes dirigés vers les technologies précédentes car elles offrent ces supports et ont atteint un fort niveau de maturité et de robustesse : leur utilisation s'étend dans les milieux industriels. De plus l'équipe ADELE possède une grande expertise de ces deux technologies, une plateforme domotique les utilisant, iCASA [Lal+15], ainsi qu'un panel d'applications domotiques. Ces éléments pourront être utilisés pour faciliter la validation de notre solution.

5.1.1.1 OSGi™

Notre machine d'exécution est basée sur OSGi™, une plateforme mettant en place l'architecture à service dynamique. Elle offre, entre autres, la possibilité de construire des applications modulaires découpées bundles, l'unité de modularité logique [Hal+11] de la plateforme, et de venir les déployer de manière dynamique. On peut donc ainsi ajouter ou retirer du code à la volée. Comme illustré par la figure 5.2 , la plateforme OSGi™ est principalement constituée de quatre couches :

La couche module: Elle définit les mécanismes de base permettant la mise en place de la modularité. Ceux-ci sont l'unité de modularité de la plateforme, et la politique d'exportation ou importation de code dans les différents modules.

La couche cycle de vie: Elle associe à chaque module un cycle de vie qui permet sa gestion dynamique. On ajoute ainsi la possibilité d'installer, d'initialiser, d'arrêter et de mettre à jour chaque bundle au cours de l'exécution.

La couche service: Elle permet de mettre en place l'architecture à service dynamique étudiée précédemment 4.1.2. La plateforme fournit l'annuaire et les API d'interaction.

La couche sécurité: Elle fournit des mécanismes pour protéger et sécuriser l'exécution des différents modules sur la plateforme.

La technologie OSGi™ a été choisie car elle supporte parfaitement l'architecture à service et le dynamisme. De plus la possibilité de mettre à jour dynamiquement certaines parties du code permet de faciliter l'administration. De plus, une des caractéristiques des applications *Fog Computing* est la réactivité ce qui exclut un certain nombre de technologies consommatrices de puissance de calcul tels les web services. En OSGi™, le coût d'un appel de service est approximativement le même que celui d'un appel de méthode Java ce qui nous permet d'obtenir la réactivité souhaitée.

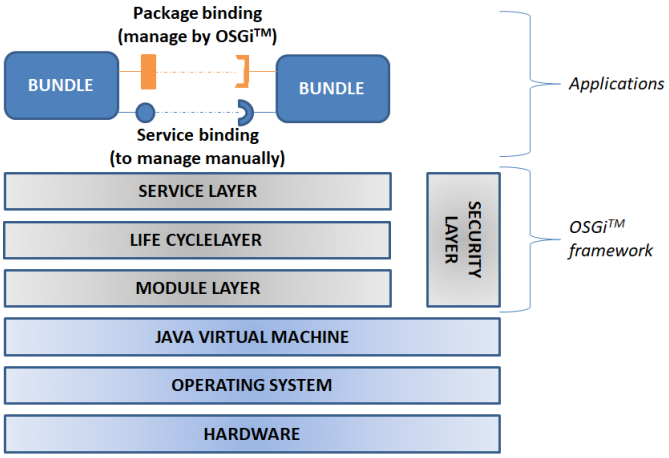

Figure 5.2: Architecture de la plateforme OSGi^{TM} inspiré de [Esc08]

5.1.1.2 Apache Felix iPOJO

Apache Felix iPOJO [Esc08] est un des modèles à composants orienté service fonctionnant au-dessus d'OSGi^{TM}. Il a pour but principalement de palier aux difficultés techniques liées à la mise en place de l'architecture à service pour les consommateurs et fournisseurs de services. Il vise donc principalement à masquer la complexité du développement d'applications dynamiques en proposant un modèle de développement simple, en facilitant la gestion du cycle de vie des composants et son impact sur les interactions de l'approche à service.

Pour cela, Apache Felix iPOJO utilise la technologie de conteneur comme illustré par la figure 5.3. Un composant Apache Felix iPOJO est donc composé d'un code métier sous forme de POJO ainsi que d'un conteneur servant à gérer les aspects non-fonctionnels. Un conteneur contient un ensemble de *handlers* qui correspondent à du code, pouvant être appelé en interception sur un attribut et ses modifications ou un appel de méthode, produit dans le but de s'occuper d'un ou plusieurs aspects non-fonctionnels. Les *handlers* les plus utilisés sont ceux s'occupant du cycle de vie du composant, de la gestion des dépendances et de leur injection dans le code fonctionnel, de la publication des services et des propriétés qui leurs sont associées et celui permettant d'introspecter l'architecture du composant.

Le choix de la technologie Apache Felix iPOJO a été fait car c'est le modèle à composant orienté service au-dessus d'OSGi^{TM} le plus complet et performant en termes d'interception de code, de service et d'injection [Escb] tout en conservant un modèle de développement simple.

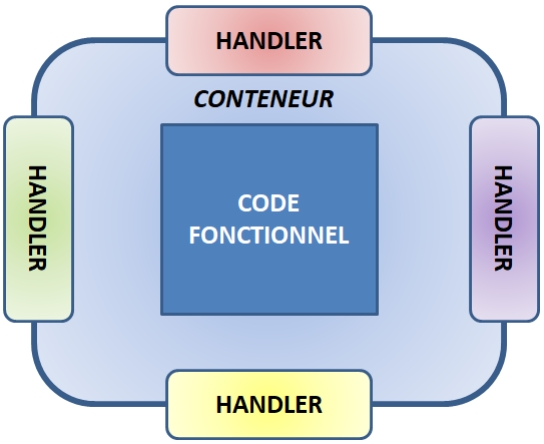

Figure 5.3: Architecture d'un composant Apache Felix iPOJO

De plus, une de ses forces est la grande extensibilité de son modèle sur plusieurs points. On peut par exemple étendre à moindre coût son processus de compilation, venir adapter les stratégies de fournitures de service ou les types de *handlers* constituant la membrane des composants. Ce dernier point nous intéresse particulièrement car il permet dans notre cas d'intégrer de façon transparente pour le développeur d'entités de contexte la gestion des extensions fonctionnelles ou les mécanismes de synchronisation.

5.1.2 Architecture générale

La contribution de cette thèse est le modèle à composant spécialisé dans la représentation du contexte appelé *CReAM*. La modèle à composant *CReAM* propose un ensemble d'outils implémentant et permettant l'exécution des concepts énoncés dans le chapitre précédent. Le point central, comme illustré par la figure 5.4, de notre implémentation est constitué d'annotations directement intégrables par le développeur d'entités de contexte dans son code, celles-ci étant proposées dans le module *Cream-core*. Celles-ci permettent d'identifier les parties *Core*, les extensions fonctionnelles et de les associer au sein d'une entité de contexte. On y retrouve aussi le DSL permettant de spécifier les besoins en synchronisation de chaque implémentation.

Brièvement, le modèle à composant *CReAM* inclue les projets suivants :

Cream-core: ce projet contient les concepts du modèle à composant exposés sous forme d'annotations Java.

Cream-ipojo-manipulator: ce projet contient une extension intégrée au processus de compilation d'Apache Felix iPOJO pour faciliter la création de bundles contenant des éléments de notre modèle à composant.

Cream-ipojo-runtime: ce projet contient l'infrastructure servant à étendre le modèle Apache Felix iPOJO pour le faire fonctionner selon les attentes décrites dans notre modèle à composant.

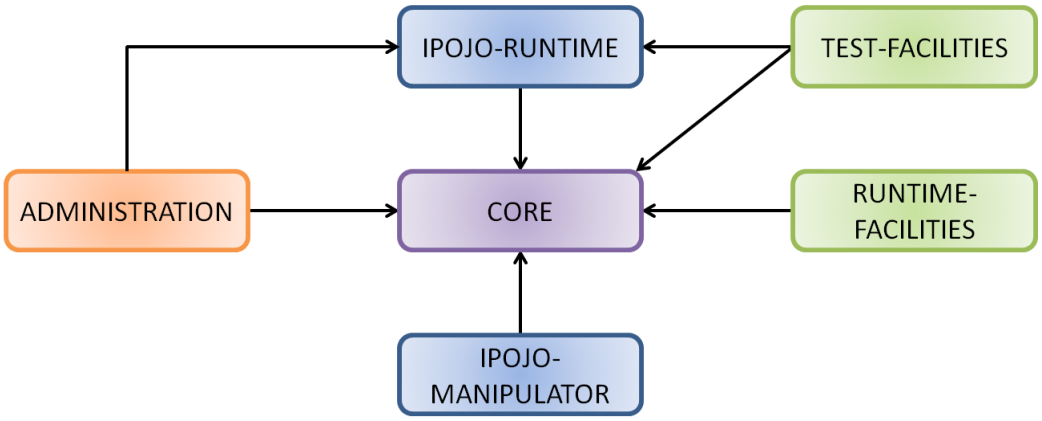

Figure 5.4: Graphe de dépendance du projet *CReAM*

Cream-administration: ce projet contient la description des services d'administration et leurs implémentations. L'implémentation des services d'administration est couplée à l'implémentation de référence basée sur Apache Felix iPOJO.

Cream-test-facilities: ce projet contient des facilités pour mettre en place des tests d'intégration relatifs aux développements d'un module de contexte.

Cream-runtime-facilities: ce projet contient des facilités relatives à la consommation des services de contexte. Il permet d'enrichir sur certains points spécifiques le modèle de dépendances d'Apache Felix iPOJO.

La suite du chapitre explique plus en détails certains des projets listés précédemment et leur apport particulier vis-à-vis de notre proposition.

5.1.3 Cream-core

Le projet *Cream-core* implémente les concepts présentés dans la proposition. Il permet de déclarer les différents éléments d'une entité de contexte et de les assembler, tout cela sous forme d'annotations Java donc directement intégrées dans le langage de programmation. Il contient aussi le langage de description de synchronisation présenté dans le chapitre précédent. Les différentes annotations présentes sont résumées dans le tableau suivant 5.2.

Annotation	Description	Configuration	Description
@ContextEntity	Permet de déclarer une entité de contexte	*coreServices*	Services de contexte implémentés par le core
@FunctionalExtender	Permet de déclarer un fournisseur d'extension fonctionnelle	*contextServices*	Services de contexte implémentés par l'extension fonctionnelle
@FunctionnalExtension	Permet d'ajouter à une entité de contexte une extension fonctionnelle	*id*	Id associé à l'extension fonctionnelle
		contextServices	Services de contexte fournis par l'extension fonctionnelle
		implementation	Classe de l'extension fonctionnelle

Table 5.1: Résumé des annotations relatives à la construction d'une entité de contexte

Dans l'exemple suivant 5.5, on reprend la construction de la lampe Zigbee qui a été initiée dans le chapitre précédent 4. On utilise l'annotation *@ContextEntity* pour déclarer une entité de contexte et immédiatement lui associer la classe correspondant à son *Core*. La configuration passée dans le paramètre *coreServices* correspond à l'ensemble des services fournis par le *Core*. Dans notre cas, il se limite au service défini par l'interface *BinaryLight*.

```
1 @ContextEntity(coreServices={BinaryLight.class})
2 public class ZigbeeBinaryLight implements BinaryLight, ZigbeeDeviceTracker {
3
4 // ZigbeeBinaryLight code
5 }
```

Figure 5.5: Code java relatif à la déclaration d'une entité de contexte et de son *Core*

L'ajout d'une extension fonctionnelle est illustré par l'extrait 5.6. Dans ce cas on désire rendre notre implémentation du service *BinaryLight* localisable. Pour cela, on peut utiliser l'implémentation par défaut fourni par la plateforme du service Localisable sous la forme d'une extension fonctionnelle identifiée par sa classe d'implémentation *WebUIBasedLocalisableExtension*. Le fait d'associer un id à chaque extension est utile pour les reconfigurations à l'exécution comme nous le verrons dans la partie 5.1.5. Si d'autres implémentations par défaut sont proposées, le développeur aura simplement à changer la configuration implementation avec la nouvelle classe souhaitée, tout cela sans impacter le code spécifique produit pour la réalisation du service *BinaryLight*.

```java
@ContextEntity(coreServices={BinaryLight .class})
@FunctionnalExtension(id= "location",
    contextServices={Localisable.class},implementation=WebUIBasedLocalisableExtension.class)
public class ZigbeeBinaryLight implements BinaryLight, ZigbeeDeviceTracker {

// ZigbeeBinaryLight code
}
```

Figure 5.6: Code Java relatif à l'ajout d'une extension fonctionnelle

Du côté de la plateforme, pour mettre à disposition une extension fonctionnelle il suffit d'annoter une classe avec *@FunctionnalExtedender* comme illustrer par l'extrait 5.7. La configuration *contextServices* étant l'ensemble de services que l'extension fonctionnelle est capable de fournir à une entité de contexte. Le nom de la classe annoté correspond à la configuration implementation de l'annotation *@FunctionnalExtension*.

```java
@FunctionnalExtender(contextServices={Localisable.class})
public class WebUIBasedLocalisableExtension implements Localisable {

// Localisable web based code
}
```

Figure 5.7: Code Java relatif à la fourniture d'une extension fonctionnelle

Comme précisé dans le chapitre précédent nous considérons que la logique de découverte et d'instanciation est programmée à l'instar du framework RoSe [Bar12]. Pour faciliter l'approvisionnement, pour chaque type d'entité de contexte nous avons associé une *factory* exposée comme un service dans l'environnement d'exécution. L'interface utilisée comme service d'approvisionnement est celui décrit dans l'extrait suivant 5.8. Elle permet de créer ou détruire des instances d'entité de contexte d'un type précis.

```java
/** A factory object used to create context entities of the specified type
    @param <E> The entity type */
public interface Entity<E> {

    public Set<String> getInstances();

    /** Return the created instance of the context entity */
    public E getInstance(String id);

    /** Creates a new instance of the context entity */
    public void create(String id, Map<String, Object> initializationParams);

    public void create(String id);

    public void delete(String id);

    public void deleteAll();
}
```

Figure 5.8: Description du service relatif à l'approvisionnement d'une entité de contexte

Le projet *Cream-core* est indépendant des technologies utilisées pour l'exécution des entités de contexte. Il en résulte que différentes implémentations, couplées à une technologie d'exécution, peuvent être réalisées. Nous présenterons dans la partie suivante notre implémentation de référence couplée aux technologies Apache Felix iPOJO et OSGi^{TM}.

5.1.4 Cream-iPOJO-*

Les projets *Cream-iPOJO-manipulator* et *Cream-iPOJO-runtime* contiennent l'implémentation de référence de notre modèle. Celle-ci repose sur les technologies Apache Felix iPOJO et OSGi^{TM} pour les raisons évoquées dans les sections précédents 5.1.1.1 5.1.1.2.

L'artefact, à la sortie de la conception, dans un développement basé sur OSGi^{TM} est le *bundle*. Ce *bundle* est de manière physique un jar qui contient le code Java compilé ainsi que des méta-informations stockées dans un fichier nommé *MANIFEST*. Les méta-informations sont utilisées pour décrire les entités de contexte et leurs constituants. Il existe donc un fort couplage entre ces méta-informations et le code java produit.

Pour éviter toute source d'erreur au moment de la production de cet artefact, nous nous sommes appuyés sur la technologie Apache Maven [Fouc]. Apache Maven permet la génération d'artefacts au travers de l'utilisation de *plugins* capables d'automatiser certains traitements

(compilation, ajout/modification de fichier, génération de documentation, etc...) sur un projet en développement.

Notre implémentation de référence utilise deux *plugins*. Le premier est le *maven-bundle-plugin* [Foub] pour générer le bundle OSGi^{TM} et les méta-informations propres à OSGi^{TM}. Le second est le *maven-ipojo-plugin* utilisé pour instrumenter le bytecode des POJOs et générer les méta-informations propres à Apache felix iPOJO. Pour inclure dans les méta-informations les descriptions propres à notre modèle à composant, nous avons étendu le processus de manipulation d'Apache Felix iPOJO, comme illustré par l'extrait 5.9 avec le projet *Cream-iPOJO-manipulator*. De plus, cela permet à chacun de nos modules (*Core* et extension fonctionnelle) d'être instrumentés selon la méthode de Apache Felix iPOJO ce qui permet de faciliter l'interception de code ou l'appel de méthode du POJO sans passer par de la réflexion.

```
<plugin>
      <groupId>org.apache.felix</groupId>
      <artifactId>maven-ipojo-plugin</artifactId>
      <version>1.12.0</version>
      <dependencies>
                  <dependency>
                        <groupId>fr.liglab.adele.cream</groupId>
                        <artifactId>cream.ipojo.manipulator</artifactId>
                        <version>0.1-SNAPSHOT</version>
                  </dependency>
       </dependencies>
</plugin>
```

Figure 5.9: Intégration de l'extension de compilation Apache Felix iPOJO au sein d'un projet Apache Maven

Une fois les *bundles* créés, il reste à les déployer et les exécuter. Pour la partie déploiement, aucun outil spécifique n'a été mis en place en supplément de ceux déjà disponibles sur la plateforme d'exécution OSGi^{TM}. Il existe par exemple des commandes au niveau *Shell* pour déployer des bundles ou des fichiers de configuration à insérer dans une distribution.

Pour la partie exécution, nous avons délégué certains aspects de notre approche à Apache Felix iPOJO. Les tâches de création des *Factories* associées au *Core* et aux extensions fonctionnelles et de leurs expositions en tant que service sont naturellement prises en charge par Apache Felix iPOJO.

Le projet *cream-iPOJO-runtime* contient deux *handlers*, automatiquement intégrés à chaque conteneur d'instance entité de contexte pour gérer certains aspects de notre modèle à composant.

Le premier *handler*, illustré dans la figure 5.10 sous le nom de *SynchronisationHandler*, a pour rôle de prendre en charge les préoccupations liées à la synchronisation évoquées dans la partie 4.2.4. Pour cela, il gère :

Un cache des variables d'états: Le *handler* contient un cache des valeurs des variables d'état. Il l'actualise grâce aux différentes fonctions de synchronisation fournies par le développeur. Ce cache est utilisé pour injecter la valeur d'un champ annoté *@State.Field* dans le POJO.

L'orchestration des synchronisations périodiques: Le handler a pour tâche d'appeler périodiquement les fonctions *@Pull* périodiques. Il profite de leur appel pour mettre à jour le cache de l'état. Ces périodes sont reconfigurables à l'exécution.

La propagation des évènements: Chaque variable d'état est exposée sous forme d'une propriété de service. Lorsque le *handler* détecte une mise à jour de son cache, il utilise le *ProvidedServiceHandler*, *handler* fourni par Apache Felix iPOJO responsable de l'exposition des services, pour mettre à jour la propriété de service concernée.

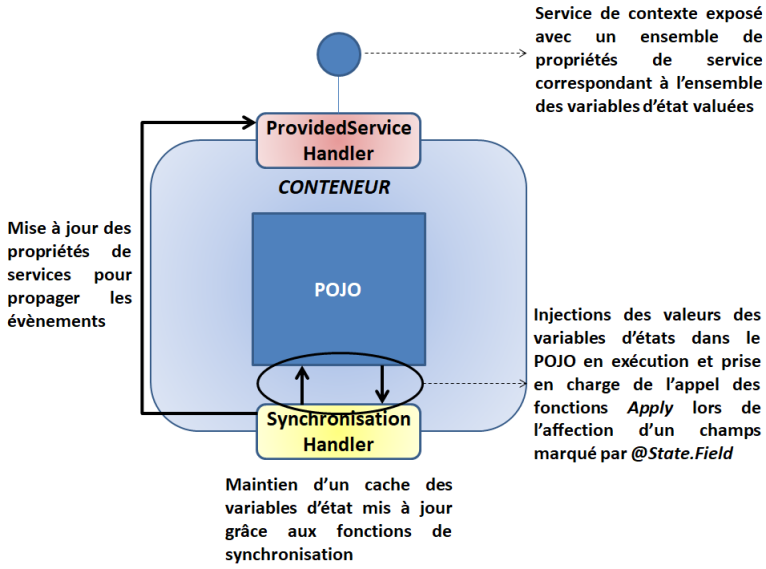

Figure 5.10: Rôle du *handler* de synchronisation

Le second *handler*, illustré dans la figure 5.11 sous le nom de *FunctionnalExtensionrHandler*, permet la gestion dynamique des extensions fonctionnelles. Pour cela il gère :

La découverte des extensions fonctionnelles: Le *handler* traque les différentes fabriques d'extensions fonctionnelles présentes dans l'environnement d'exécution, exposées sous forme de service. Il garde une référence sur les fabriques d'extensions fonctionnelles capables de fournir les services décrits au moyen des annotations *@FunctionnalExtension*. De cette manière, il connait en permanence tous les types d'extensions fonctionnelles pouvant être utilisées lors d'une reconfiguration.

La création des instances d'extension fonctionnelle: Le *handler* est en charge de créer et configurer les instances d'extension fonctionnelle. Il utilise les informations contenues dans l'annotation *@FonctionnalExtension* pour sélectionner le type d'extension à créer et vérifie que celle-ci peut fournir les services demandés. Comme décrit par la figure 5.11, chaque instance d'extension fonctionnelle possède son propre conteneur et donc ses propres handlers définis pour ses besoins particulier.

Le management dynamique des instances créées: Le management dynamique des instances d'extension fonctionnelle implique plusieurs tâches. Tout d'abord, il implique la gestion du cycle de vie des instances créées. Les instances d'extension fonctionnelles ne peuvent être valides que si l'instance du *Core* l'est. Lorsque le *Core* et une extension sont valides, le *handler* est en charge de propager les variables d'états valuées par l'instance d'extension fonctionnelle dans les propriétés de services exposées par le *ProvidedService-Handler*. Lorsque le *Core* est valide mais qu'une extension fonctionnelle devient invalide, le *handler* réagit en retirant les services et les propriétés de service liées à l'extension invalide en créant un *ServiceController* [Escd] virtuel . Le dernier point du management dynamique est de permettre la reconfiguration du type d'extension fonctionnelle utilisée pour fournir un service. Le *handler* prend en charge la destruction de la précédente instance, si elle existe. Il assure ensuite la création de la nouvelle instance si une *factory* de celle-ci existe et que le type d'extension peut fournir les services spécifiés.

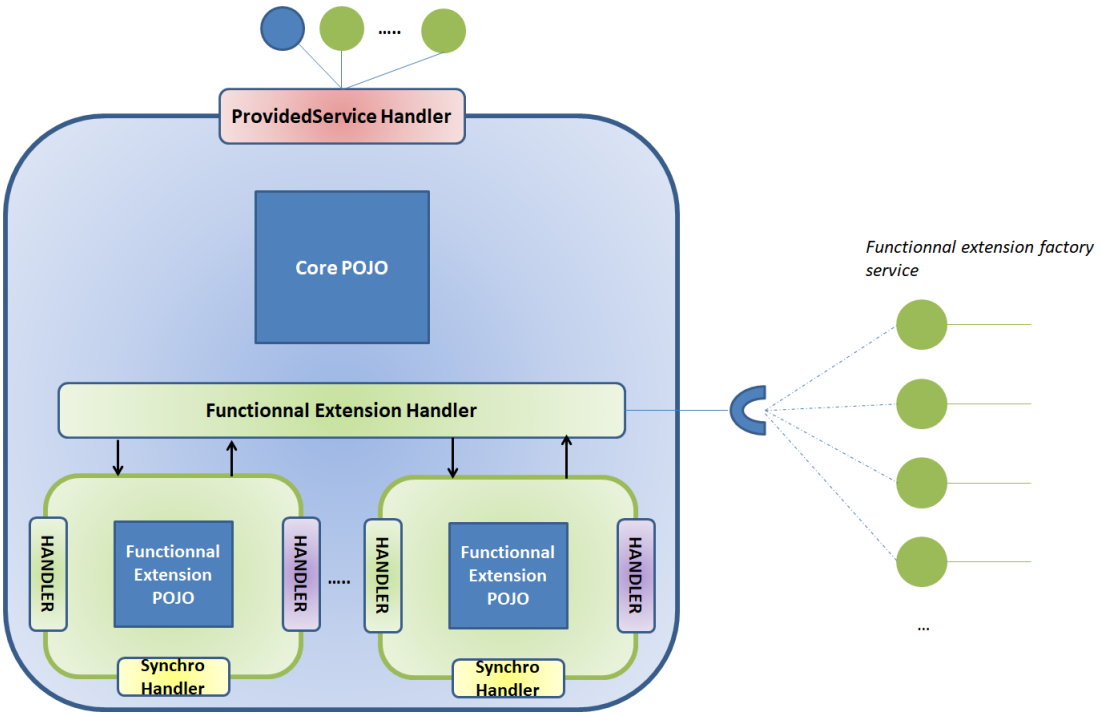

Figure 5.11: *Handler* de gestion des extensions fonctionnelles

Le dernier point notable de notre implémentation concerne la génération de l'enveloppe fonctionnelle. Celle-ci a pour but de donner l'impression aux consommateurs d'obtenir une référence vers un unique POJO, implémentant une multitude de services, alors que celui-ci

est en fait un ensemble de POJO implémentant individuellement un ou plusieurs services. Le *ServiceProviderHandler* peut-être configuré avec plusieurs stratégies pour retourner une référence lorsqu'un consommateur cherche à se lier (phase de *binding*) avec un fournisseur. La stratégie par défaut [Escd] est de retourner la même instance de POJO pour tous les consommateurs, mais il en existe d'autres permettant par exemple de retourner une nouvelle instance du POJO par consommateur.

Pour la création de l'enveloppe, nous avons créé notre propre stratégie. Celle-ci permet de retourner un objet *Proxy* [Orab] implémentant uniquement les interfaces des services fournis. Celui-ci est donc re-généré à chaque fois que l'ensemble de services fournis change. Le schéma de délégation de l'enveloppe fonctionnelle est ensuite pris en charge par chacun des conteneurs, celui du Core et ceux des extensions fonctionnelles, qui agissent comme les *ConcreteHandlers* du patron *Chain Of Responsability* [Gam+95] . Il est à noter que les conteneurs sont génériques tandis que les objets responsables de traiter les appels sont spécifiques. Pour réaliser l'appel, nous générons à la volée grâce à la librairie *ASM* [Con], le morceau de code responsable de l'appel de la méthode concrète. Cela évite de passer par des techniques reposant sur de la réflexion, reconnues comme coûteuses. Chaque conteneur possède donc un schéma de délégation propre aux services pour lesquels il a été instancié. D'autres choix de librairies de génération de code [Win] auraient pu être envisagés mais *ASM* est présente de base dans une distribution Apache Felix iPOJO.

5.1.5 Cream-administration

Le projet *Cream-administration* contient les interfaces des services d'administration du modèle à l'exécution et une implémentation de référence de ces services. Comme décrit par la figure 5.12, les services d'administration sont uniquement couplés au projet *Cream-core*. Cela permet donc de construire les outils de management indépendamment de la technologie utilisée pour maintenir un modèle du contexte à l'exécution.

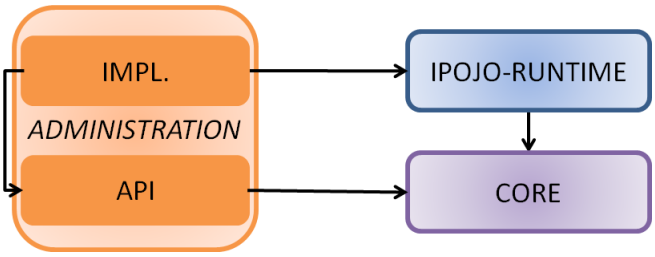

Figure 5.12: Graphe de dépendances centré sur le projet *Cream-administration*

Le contrat de service, présenté dans la figure 5.13, permet donc d'exposer les fonctionnalités décrites dans la partie 4.3.3.Ce service permet, entre autres, d'obtenir la topologie du module *(Ligne 5)* de contexte et des informations précises sur chacun de ses nœuds *(Ligne 8)*, de venir changer d'extension fonctionnelle *(Ligne 12)* ou de reconfigurer les fréquences de synchronisation des différentes variables d'état *(Ligne 16)*. On notera que le service d'administration

retourne uniquement des objets immutables, assimilables à une photo du modèle de contexte à l'instant de l'appel à ses consommateurs pour éviter toute fuite de référence due à la nature dynamique du module de contexte.

```
1  /** Administration Service of A running Context Module **/
2  public interface AdministrationService {
3
4     /** Return a set of ImmutableContextEntity that represente the topology of the
          context module. Each ImmutableContextEntity contains informations about its
          Core and Functionnal Extensions of an underlying context entity instance**/
5      Set<ImmutableContextEntity> getContextEntities();
6
7     /** Return a snapshot of context entity instance identified by the id parameter
          **/
8      ImmutableContextEntity getContextEntity(String id);
9
10     /** Reconfigure the periodic param frequency of a particular context state of a
           particular context entity instance
11 The previous configuration can be know throught the use of the getContextEntity()
       or getContextEntities() method**/
12     void reconfigureContextEntityFrequency(String contextEntityId, String
           contextStateId, long frequency, TimeUnit unit);
13
14 /** Reconfigure the used functionnal extension by a context entity instance
15 The possible configuration can be know through the use of the getContextEntity() or
       getContextEntities() method**/
16     void reconfigureContextEntityComposition(String contextEntityId,String
           functionnalExtensionId,String functionnalExtensionImplementation);
17 }
```

Figure 5.13: Service d'administration du module de contexte en exécution

Le projet d'implémentation contient un unique composant implémentant le service décrit ci-dessus 5.13 en utilisant les propriétés de reconfiguration de Apache Felix iPOJO et celles prises en charge par les *handlers* précédemment présentés.

5.1.6 Management Autonomique

Cette section présente les différents outils relatifs à l'administration et au management autonomique qui ont été mis en place autour de notre modèle à composant. Ceux-ci se basent principalement sur les interfaces des services d'administration fournis par le projet *Cream-administration* ce qui permet de les faire évoluer sans impacter l'implémentation du modèle à composant et vice-versa.

5.1.6.1 Outils d'administration

En complément du support à l'exécution de notre modèle à composant, nous proposons plusieurs outils à l'exécution pour faciliter l'administration et le monitoring du module de contexte. Ces outils permettent entre autres de visualiser et de reconfigurer le module de contexte à chaud.

Le premier outil est un service d'instructions dans le *Shell* de la plateforme, comme illustré par la figure 5.14. Il permet de connaitre l'état des éléments du module de contexte et de venir effectuer les opérations de reconfiguration disponibles à l'exécution. On peut ainsi grâce à lui reconfigurer les fréquences de synchronisation ou remplacer certaines implémentations d'extension fonctionnelle par de nouvelles.

Figure 5.14: Shell d'administration du module de contexte en exécution

Le second outil développé est un outil de *monitoring* de la topologie du module de contexte. Celui-ci se présente sous la forme d'une interface *web* dont un aperçu est donné dans la figure 5.15. Il permet de connaitre la topologie du module de contexte en cours d'exécution et d'obtenir des informations sur chacun des nœuds du graphe de contexte.

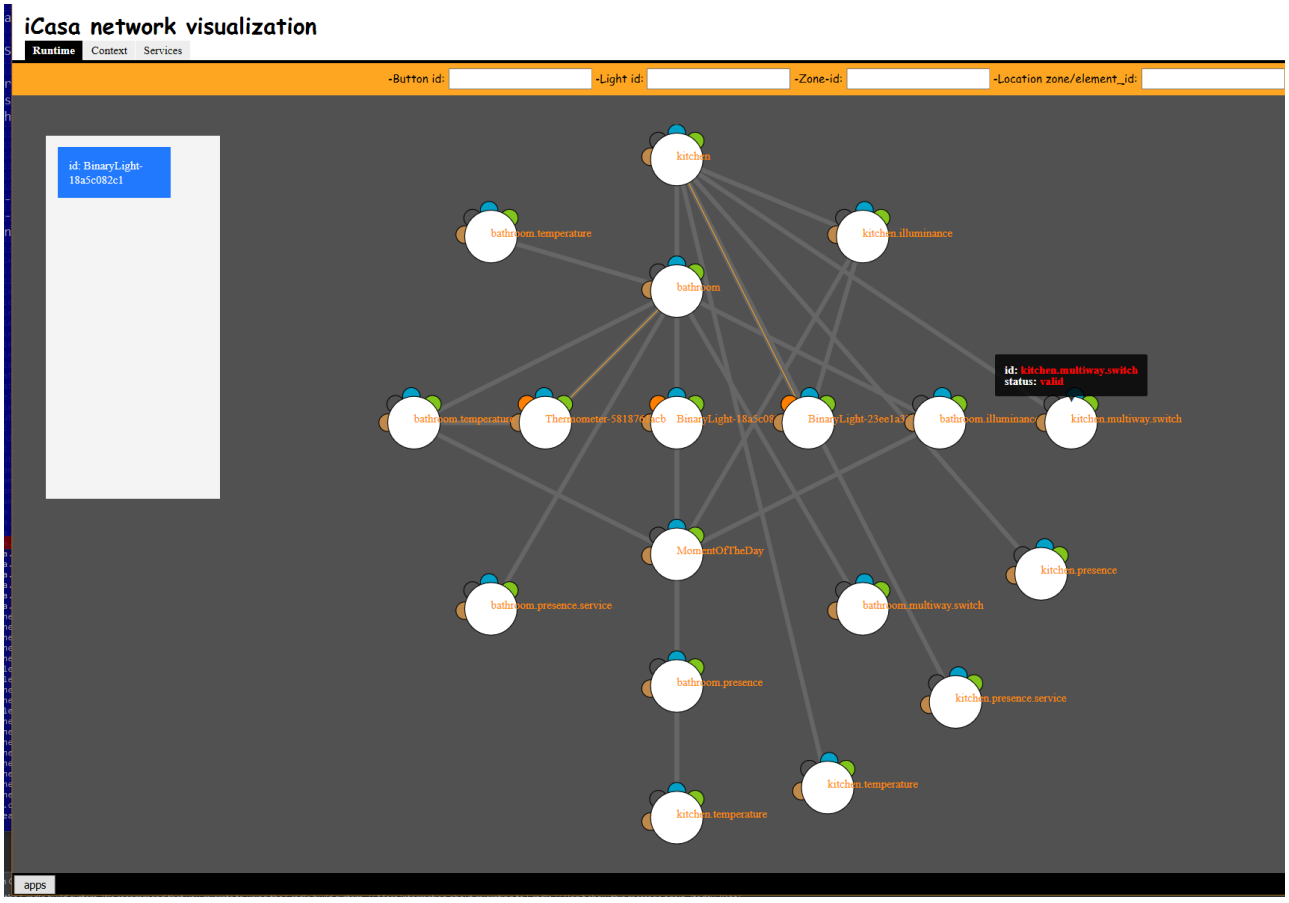

Figure 5.15: Outil de visualisation du contexte

5.1.6.2 Manager autonomique

Nous avons développé un manager autonomique, sous la forme d'un composant Apache Felix iPOJO, dont le but est d'essayer de garantir que les services publiés par le module de contexte se limitent aux besoins des applications.

Nous avons assumé que les applications s'exécutant sur la plateforme de Fog Computing étaient développées selon le modèle proposé par Apache Felix iPOJO. Chaque application peut ainsi être constituée de plusieurs composants et utiliser des services de contexte pour s'exécuter. Comme nous l'avons vu, Apache Felix iPOJO rends naturellement autonomique le conteneur d'un composant au regard de la dynamique des services. Ainsi les applications peuvent s'adapter en bénéficiant des mécanismes de late-binding et de substituabilité de service obtenue grâce aux annotations proposées par Apache Felix iPOJO. Notre manager assume ainsi que les applications peuvent patienter jusqu'à l'apparition d'un service de contexte.

Le manager développé est un composant Apache Felix iPOJO reposant sur l'architecture MAPE-K [KC03]. Il dispose comme senseurs et effecteurs sur le module de contexte des points d'administration évoqués dans la partie 4.3 et des points d'introspection fournis par iPOJO. Il fonctionne de la manière suivante

La partie *Knowledge*: Elle contient les buts du manager représentés par les services de contexte dont les applications ont besoin pour fonctionner, les services de contexte déjà présents sur la plateforme ainsi que les services de contexte possiblement présents sur la plateforme.

La partie *Monitoring*: Pour peupler la base de connaissance, cette partie se sert de l'introspection pour connaitre les besoins des composants applicatifs, de l'API d'administration du module de contexte 4.3.3 pour connaitre les services présents à l'exécution. Les services potentiellement présents sont déduits des demandes d'instance non traitées obtenues grâce au point d'administration évoqué dans la partie 4.3.2 et des extensions fonctionnelles invalides.

La partie *Analysis*: Cette partie permet d'analyser trois cas distincts. Les trois cas sont :

- *Cas 1*: il y a plus de services en exécution que nécessaire.
- *Cas 2*: Il manque des services pour permettre l'exécution d'une application et ceux-ci peuvent être fournis par une demande d'instanciation pas encore traitée.
- *Cas 3*: Il manque des services pour permettre l'exécution d'une application et ceux-ci peuvent être fournis par une reconfiguration d'extension fonctionnelle.

La partie *Planning*: Cette partie est relativement simple car notre manager est seulement réactif et non proactif. Pour les différents cas, elle prévoit :

- *Cas 1*: La destruction des instances d'entité de contexte inutilisées.

- *Cas 2*: L'autorisation de la prise en compte d'une demande d'instanciation pouvant fournir le service, à condition que le cas 3 ne soit pas actif.
- *Cas 3*: L'avertissement de l'administrateur de la plateforme pour qu'il procède à une reconfiguration de l'entité de contexte possédant l'extension fonctionnelle invalide.

La partie *Execution*: cette partie se charge d'effectuer les actions précédemment définies grâce aux différents effecteurs.

5.1.7 Synthèse de l'implantation

Cette première partie a permis de présenter les points les plus importants de l'implémentation de référence de notre modèle à composant de contexte. On a pu constater que nous possédons tous les outils nécessaires pour concevoir, exécuter et administrer un module de contexte mettant en œuvre les préceptes avancés dans notre proposition.

Cette implémentation de référence est disponible en libre accès sous Apache Licence 2 [Foua] sur le site GitHub à l'*url* suivant: `https://github.com/aygalinc/Context-SOCM`. Les résultats des différents *builds*, effectués par un serveur d'intégration continu [Tra], sont disponibles à l'adresse suivante: `https://travis-ci.org/aygalinc/Context-SOCM`. Les métriques complètes des différents projets ont été produites par SonarQube [Sonb], un outil de management de qualité de code. Elles sont disponibles à l'adresse suivante: `https://sonarcloud.io/dashboard?id=fr.liglab.adele.cream%3Acontext.socm.reactor`.

Le tableau suivant 5.2 résume les éléments majeurs de notre implémentation :

Artefact	Nombre de ligne de code	Nombre de ligne de tests
Cream-core	847	71
Cream-iPOJO-manipulator	1100	63
Cream-iPOJO-runtime	2700	1365
Cream-administration	574	862
Cream-test-facilities	787	-
Cream-runtime-facilities	391	183

Table 5.2: Nombre de ligne de code du projet *CReAM*

C'est cette implémentation de référence qui servira à tester et valider l'approche de cette thèse dans la section suivante.

5.2 Validation

La problématique générale de cette thèse est de simplifier le développement d'un module de contexte tout en le dotant naturellement de capacités autonomiques. Notre proposition a abouti sur :

- Un modèle à composant et sa machine d'exécution spécialisée dans le support du développement d'un module de contexte et fournissant des points d'adaptation nécessaires à la gestion des instances de composant.
- Un manager autonomique présent à l'exécution permettant une gestion efficace en termes de fourniture de service du module de contexte.

La validation de cette proposition reposera sur des développements effectués en rapport avec la plateforme iCasa [IMAb]. Pour cela, nous présenterons brièvement le domaine d'application visé par la plateforme. Nous présenterons ensuite la plateforme et ses capacités. Pour finir, nous donnerons un aperçu des tests effectués et de leurs visées.

5.2.1 Environnement de validation

5.2.1.1 Smart home

Le domaine du Smart home, ou maison intelligente en français, a pour but de fournir des services à haute valeur ajoutée aux résidents d'une maison [Ald03] [Jeu05]. Ces services sont en rapport avec le confort, la sécurité, la gestion de l'énergie ou la santé des occupants. Pour permettre le développement et l'exécution de ces services, la résidence est parsemée d'objets communicants et peut aussi compter sur sa connexion avec le monde extérieur. Une approche populaire [ECL14] [BLE10] est d'héberger en partie ces applications sur une plateforme domotique, comme illustré par la figure 5.16.

Nous nous sommes intéressés à ce domaine car la plateforme domotique correspond tout à fait à une des applications du *Fog Computing* [AH14]. Pour la suite de la validation, nous avons décidé d'appliquer nos travaux à une plateforme domotique particulière, la plateforme iCasa [Lal+15] [IMAb], que nous allons présenter dans la section suivante.

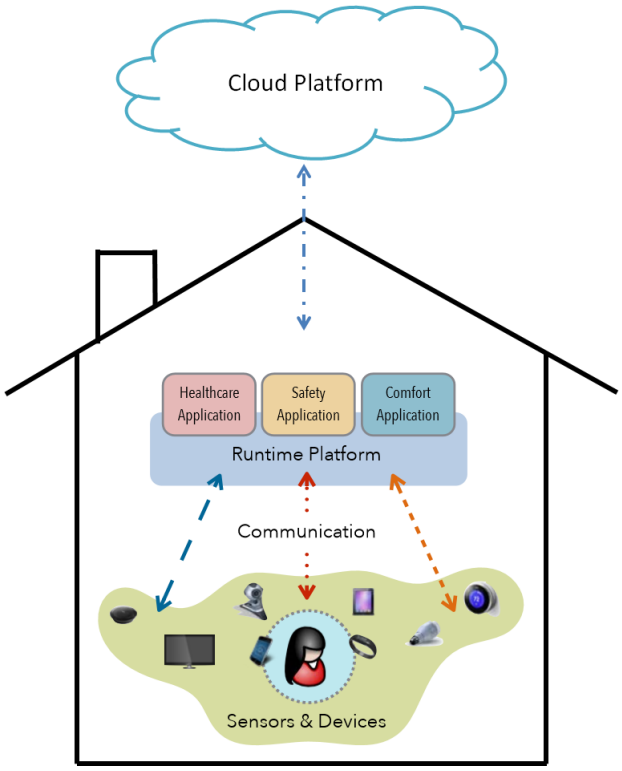

Figure 5.16: Architecture du Smart Home

5.2.1.2 Plateforme iCasa

La plateforme iCasa [Lal+15] [IMAb], qui se qualifie de plateforme ubiquitaire pour le domaine Smart Home, a pour but de faciliter le développement, le test, l'exécution et l'administration d'applications du domaine précédemment étudié.

Pour cela elle propose principalement trois outils, comme illustré dans la figure 5.17

Un environnement de développement basé sur un plug-in Eclipse [IMAa]. Celui-ci facilite la construction d'applications basées sur Apache Felix iPOJO, leur compilation et leur déploiement dans l'*environnement d'exécution*.

Un environnement d'exécution basé sur les technologies OSGi™ et Apache Felix iPOJO pour supporter l'exécution dynamique d'applications java. Il est composé de plusieurs middlewares supplémentaires facilitant la découverte de ressources distantes (middleware RoSe [Bar12]), la médiation (middleware Cilia [Gar12]) ou un serveur d'applications Web (middleware Wisdom [Esce]). Il offre aussi le support de plusieurs services techniques tels que du scheduling ou de la persistance. Pour finir, il contient aussi un module de contexte offrant des abstractions sous forme de services pour les zones, équipements de l'habitation ou l'utilisateur lui-même. Il supporte aussi l'exécution d'un *simulateur d'environnement Smart Home*.

Un simulateur d'environnement Smart Home qui contient des implémentations simulées liées au module de contexte et qui permet leur instanciation/destruction et certaines interactions au travers d'une interface Web. Il permet aussi la définition de scénarios au travers de scripts pour faciliter le test des applications précédemment développées.

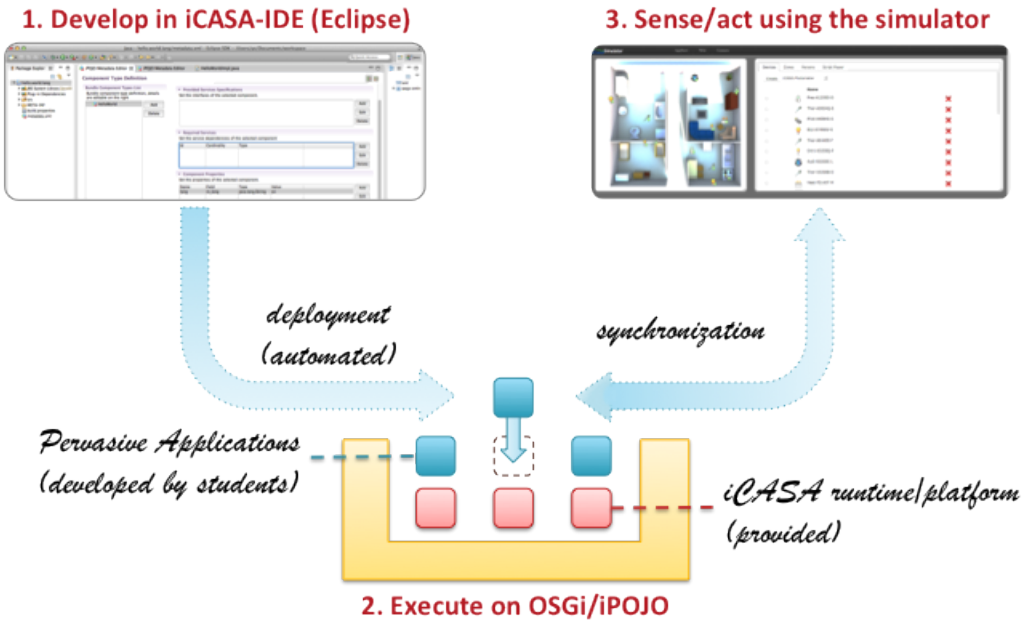

Figure 5.17: iCasa Workflow [IMAb]

Plusieurs applications ont pu ainsi être développées grâce à la plateforme iCasa. On retrouve l'application d'actimétrie présentée précédemment (voir section 2.3.1) ou une application de suivi lumineux intelligent.

Nous avons retenu cette plateforme pour effectuer notre validation car elle repose en partie sur des technologies communes avec notre implémentation de référence ce qui permet d'intégrer notre solution à moindre coût. De plus, elle possède déjà un module de contexte et plusieurs applications ce qui permettra d'évaluer l'impact en termes de développement de notre solution. De plus, l'équipe ADELE est, en partenariat avec *Orange Labs*, l'auteur du développement de cette plateforme ce qui nous permet de bénéficier d'une expertise préalable solide et un accès aux différentes bases de code.

5.2.1.3 Plan de test

Nous avons choisi d'effectuer deux scénarios pour mesurer l'impact de notre proposition. Dans les deux scénarios nous comparons une implémentation existante utilisant seulement Apache Felix iPOJO (qualifiée de *Référence* dans les différents graphiques) et une implémentation utilisant la solution *CReAM*.

Les deux scénarios sont les suivant :

Refonte du module de contexte de la plateforme iCasa: nous allons mettre en place notre solution pour redévelopper l'ensemble des services proposés par l'environnement d'exécution iCasa et leur implémentation dans le module de contexte défini par le simulateur.

Développement d'une application d'éclairage intelligent: Le second scénario offre la comparaison entre deux développements de la même application d'éclairage intelligent, l'un étant réalisé au-dessus du modèle de contexte *ad-hoc* défini par Apache Felix iPOJO tandis que l'autre profite du module de contexte développé en suivant notre solution.

Dans les deux scénarios notre évaluation portera sur l'effort lié à la conception pour mettre en place, utiliser ou étendre le module de contexte. Nous avons choisi de mesurer les métriques [Sona] suivantes qui seront calculées par le logiciel SonarQube en version 6.2 [Sonb]

- Le nombre de lignes de code
- La dette technique: cette métrique est liée à l'effort nécessaire à la réparation de toutes les sources potentielles d'erreurs dans le code.
- La complexité cyclomatique : cette métrique reflète le nombre de chemins indépendants possibles dans chaque méthode. Elle donne une indication sur la facilité à comprendre un projet, à le maintenir et le tester.

Une amélioration de la conception d'une solution particulière se traduit par la baisse de ses différentes métriques.

5.2.2 Restructuration de la plateforme iCASA

5.2.2.1 Contexte

Cette première évaluation concerne la plateforme iCasa et le simulateur qui lui est associé. Les fonctionnalités fournies par ces projets sont les suivantes :

- Un ensemble d'abstractions correspondant aux équipements, utilisateurs, pièces de l'habitation et leurs implémentations simulées.
- Une interface web servant de tableau de bord pour visualiser les différents éléments simulés.
- Un langage de script permettant d'instancier dynamiquement les diverses implémentations simulées.

Dans l'approche *Référence*, le module de contexte fut développé de manière *ad-hoc* avec comme seul support la technologie Apache Felix iPOJO. Il est difficile de l'étendre car son évolution s'est faite au gré des besoins des développeurs qui n'ont appliqué que peu de patterns reproductibles d'une extension à une autre.

Nous avons appliqué notre solution pour rebâtir entièrement l'environnement d'exécution iCasa et son simulateur. Dans un premier temps, nous avons tout utilisé simplement le fait de concevoir chaque service de contexte pour refléter une préoccupation particulière et notre DSL de synchronisation. Le résultat de cette première restructuration est noté *Approche* dans la section suivante. Dans un second temps, nous avons utilisé le concept d'extension fonctionnelle pour packager certaines préoccupations de contexte, en particulier celle concernant le positionnement des objets grâce à l'interface web. L'implémentation des extensions fonctionnelles est disponible dans la partie environnement d'exécution de l'implémentation. Ce second résultat est noté *ApprocheAvecFEx* dans la section suivante.

5.2.2.2 Résultats

Les résultats de l'évaluation sont donnés dans les figures suivantes :

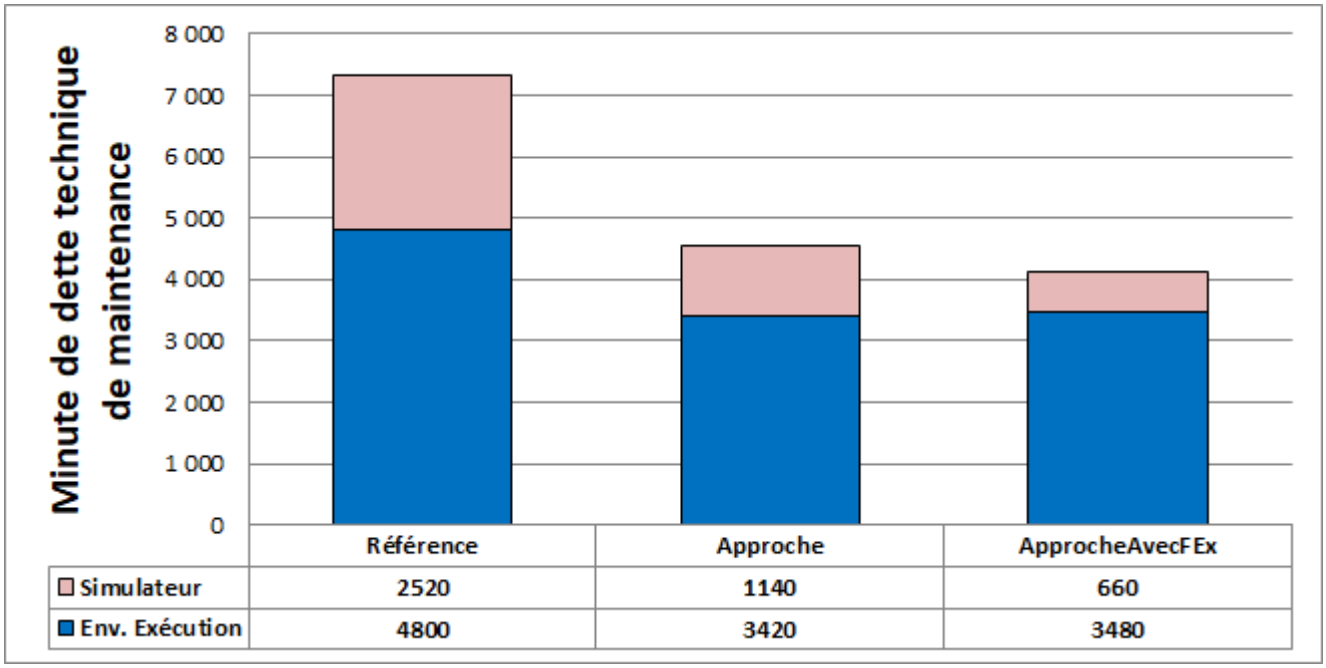

Figure 5.18: Evaluation de la dette technique des différentes implémentations de iCASA

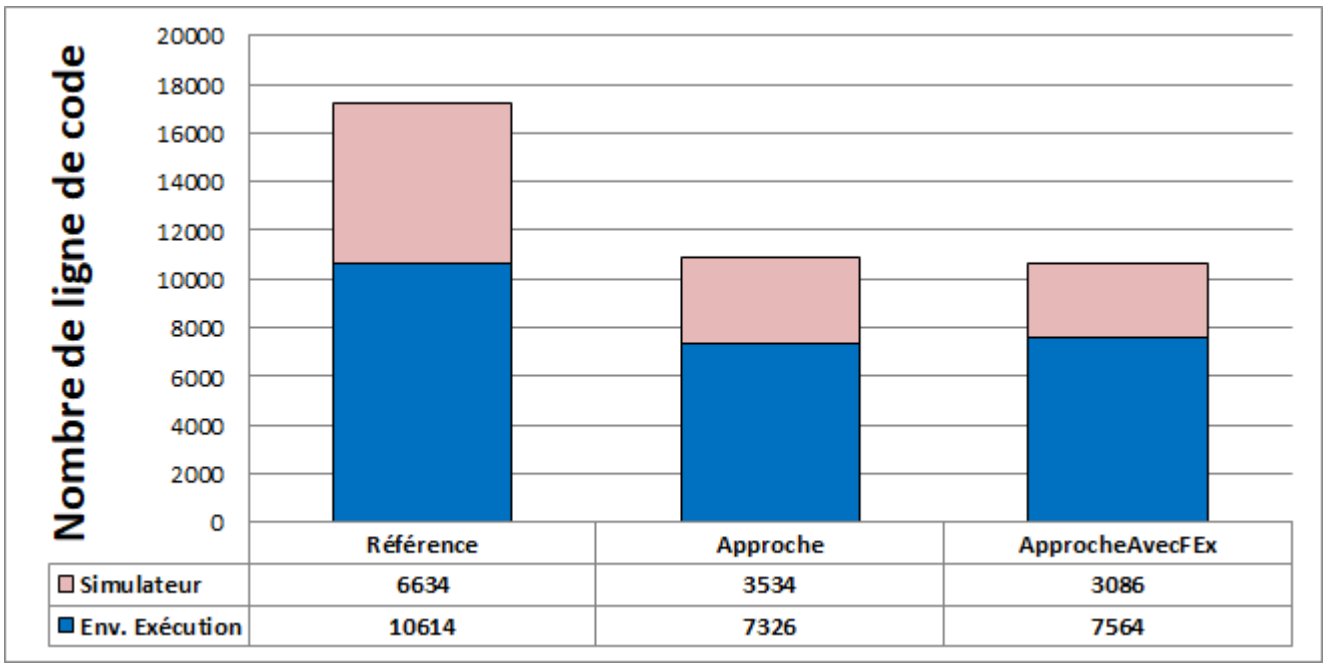

Figure 5.19: Evaluation du nombre de ligne de code des différentes implémentations de iCASA

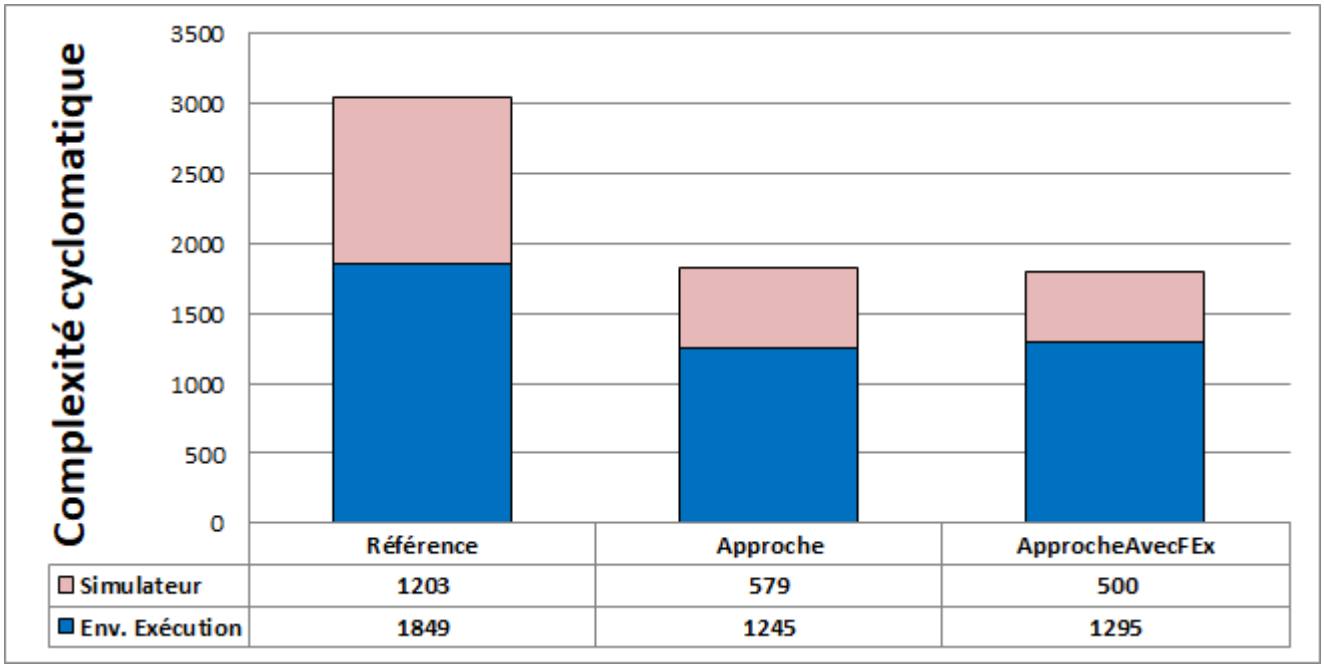

Figure 5.20: Evaluation de la complexité cyclomatique des différentes implémentations de iCASA

5.2.2.3 Analyse

On peut constater une diminution de toutes les métriques grâce à l'utilisation de notre approche. Cela est dû principalement aux annotations fournies qui permettent de traiter beaucoup plus de concepts de manière déclarative. Le développeur évite ainsi de produire le code associé à ces concepts. Ce code ne disparait pas mais est dorénavant embarqué dans le conteneur des composants de contexte.

On a aussi pu noter l'apport à la conception de la notion d'extension fonctionnelle. Nous avons pu noter que l'implémentation Approche avait un nombre élevé de code dupliqué, 6.3 %, contrairement à l'implémentation de référence qui est de 2.7 %. Cela s'explique par les limites des technologies Apache Felix iPOJO. Celle-ci ne supporte pas totalement l'héritage entre composants, principalement pour les champs annotés. Dans l'implémentation *ApprocheAvecFEx*, nous sommes tombés à un pourcentage de code dupliqué de 1.3 %, ce qui se ressent aussi sur la diminution de la dette technique. De plus même si Apache Felix iPOJO pouvait supporter l'héritage, notre solution permet de garder un design plus flexible en promouvant la composition au lieu de l'héritage [Gam+95].

Cette première évaluation permet de conclure à l'efficacité de notre approche qui facilite le développement d'un logiciel plus testable, maintenable et extensible.

5.2.3 Développement d'une application d'éclairage intelligent

5.2.3.1 Contexte

Cette seconde évaluation compare deux versions d'une application de suivi lumineux intelligent, appelée *Light Follow Me*. La comparaison s'effectue entre une version de l'application développée au-dessus de la plateforme Référence tandis que l'autre est effectuée au-dessus de la plateforme *ApprocheAvecFEx*.

L'application *Light Follow Me* a pour but d'allumer ou d'éteindre les différentes lampes de l'habitation en fonction de la présence ou non de l'utilisateur dans chaque pièce. Nous avons sélectionné cette application, car bien qu'elle puisse être perçue comme simple, elle correspond aux caractéristiques des applications *Fog Computing* présentées précédemment dans la section 2.3. En effet, c'est une application domotique essentielle qui ne nécessite aucun raisonnement complexe, qui doit pouvoir faire face au dynamisme de l'environnement (les capteurs et lampes peuvent apparaitre ou disparaitre pendant l'exécution) et qui doit réagir au changement de l'environnement en l'influençant directement au travers d'actionneurs.

Dans l'implémentation *Référence*, l'application avait la charge de calculer l'information de présence par zone. Pour cela, elle raisonnait directement au-dessus des informations des capteurs de présence et de leur localisation.

En utilisant notre approche, nous avons choisi d'externaliser ce calcul d'information en créant un service de contexte de présence par zone et son entité de contexte correspondant. Ce service de présence peut ainsi être partagé par plusieurs applications et évoluer indépendamment de la logique métier de l'application *Light Follow Me*.

5.2.3.2 Résultats

Les résultats de la comparaison des deux implémentations de l'application *Light Follow Me* sont disponibles dans la figure 5.21.

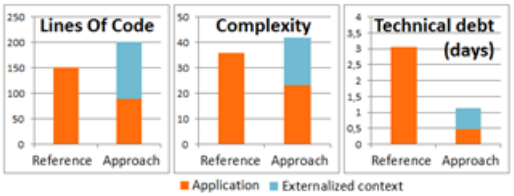

Figure 5.21: Résultat de la comparaison des deux implémentations de l'application *Light Follow Me*

5.2.3.3 Analyse

On peut constater que l'externalisation du service de présence produit plus de lignes de code et augmente la complexité. Ceci est dû au fait qu'en plus de la production de l'implémentation du service de présence, il faut aussi fournir sa logique d'approvisionnement. Celle-ci reposant sur une API java, elle est doit donc être programmée. Cependant, cette augmentation peut être partagée par plusieurs applications. Si nous analysons uniquement le code métier de l'application, celui-ci est approximativement divisé par 2 pour toutes les métriques. De ce fait, il devient plus facilement testable, maintenable et son évolution s'en trouve facilitée.

On a pu noter que durant cette extension du module de contexte, nous n'avons pas utilisé le concept d'extension fonctionnelle. On a pu constater au cours de nos travaux que lors d'une extension que l'on peut qualifier de « verticale », c'est-à-dire où l'on ajoute des services de plus en plus abstraits, ceux-ci se focalisent souvent une unique préoccupation de contexte ce qui permet de se passer du contexte d'extension fonctionnelle durant leur implémentation. Cependant nous constatons aussi que la construction de ces services abstraits est facilitée car les services des couches inférieures implémentent de multiples préoccupations de contexte qui peuvent cette fois bénéficier du concept d'extension fonctionnelle.

Pour résumer, externaliser le contexte ajoute une tâche additionnelle au développement et l'architecture du module de contexte s'en retrouve compliquée. Ce coût peut cependant être mutualisé et partagé par plusieurs applications. De plus, le développement de nouvelles applications, au-dessus de ces services plus abstraits, s'en trouve facilité.

5.2.4 Evaluation des coûts

Cette section présente les principaux coûts d'exécution de notre solution. Comme on a pu le voir dans le chapitre précédent, nous avons enrichit le modèle à composant Apache Felix iPOJO pour la partie conception mais avons alourdi son exécution. Il est donc intéressant pour un développeur de savoir quel coût d'exécution il va devoir supporter et anticiper pour bénéficier de la souplesse de notre modèle par rapport à un SOCM classique.

Les principaux impacts à l'exécution de notre solution concernent la mémoire et le temps d'appels des méthodes des services. En effet, pour utiliser notre solution les artefacts logiciels embarquant les différentes implémentations et API doivent maintenant être présents dans la plateforme *Fog Computing* ce qui impactera son empreinte mémoire. Le fait d'intégrer une couche de délégation supplémentaire, par l'intermédiaire de l'enveloppe fonctionnelle, lors de l'appel d'une méthode d'un service fait obligatoirement apparaitre un *overhead* en termes de performance. Le but de cette partie est donc de quantifier ces deux *overhead*.

Pour produire les différentes mesures de cette section, nous avons utilisé l'utilitaire JProfiler [JPr] qui permet de monitorer une JVM.

5.2.4.1 Mémoire

Pour utiliser notre solution dans une plateforme OSGi^{TM}, l'utilisateur doit déployer dans son environnement d'exécution a minima le *bundle Cream-core* et *Cream-iPOJO-runtime* en plus du *bundle ipojo-core*. La taille des *bundles* de notre solution est récapitulée dans le tableau 5.3.

Bundle	Taille en KiloOctet
Cream-core	127.3
Cream-iPOJO-runtime	105.5
Cream-admin-API	8.5
Cream-admin-impl	15.3
Cream-runtime-facilities	18.3

Table 5.3: Détails de la taille des différents projets de la solution *CReAM*

On peut donc constater que notre solution possède une empreinte mémoire faible qui peut être facilement absorbée par une solution de *Fog Computing*.

5.2.4.2 Performance

Nous avons aussi évalué le surcoût au moment de l'appel d'une méthode qu'entraine le fait de passer par l'enveloppe fonctionnelle et son schéma de délégation par rapport à un appel de méthode d'un service Apache Felix IPOJO. Pour cela, nous nous sommes appuyé sur le *microbenchmark* décrit dans le diagramme de séquence présenté dans la figure 5.22. Nous avons effectué les trois mesures suivantes :

Cas 1: Coût d'un appel d'une méthode d'un service fourni par une instance de composant Apache Felix iPOJO.

Cas 2: Coût d'un appel d'une méthode d'un service fournis par le *Core* d'une instance d'entité de contexte (cas *MethodBelongsToCoreService* sur la figure 5.22).

Cas 3: Coût d'un appel d'une méthode d'un service fournis par une extension fonctionnelle d'une instance d'entité de contexte (cas *Else* sur la figure 5.22).

Les résultats du *microbenchmark* font apparaitre qu'un appel sur une instance de composant Apache Felix iPOJO prend une micro seconde tandis qu'un appel sur une méthode contenue dans le *Core* prend deux microsecondes. Un appel sur une méthode contenue dans une extension fonctionnelle prend quatre microsecondes. On précisera que le schéma de délégation au niveau des extensions (transition *8 : Select Functional Extension to call* sur la figure 5.22) repose principalement sur une *HashMap* [Oraa]. Ce choix de la délégation peut être caractérisé par un surcoût de performance amorti en O(1), ce qui théoriquement n'handicape pas les performances de l'appel en fonction du nombre d'extensions fonctionnelles attachées à une entité.

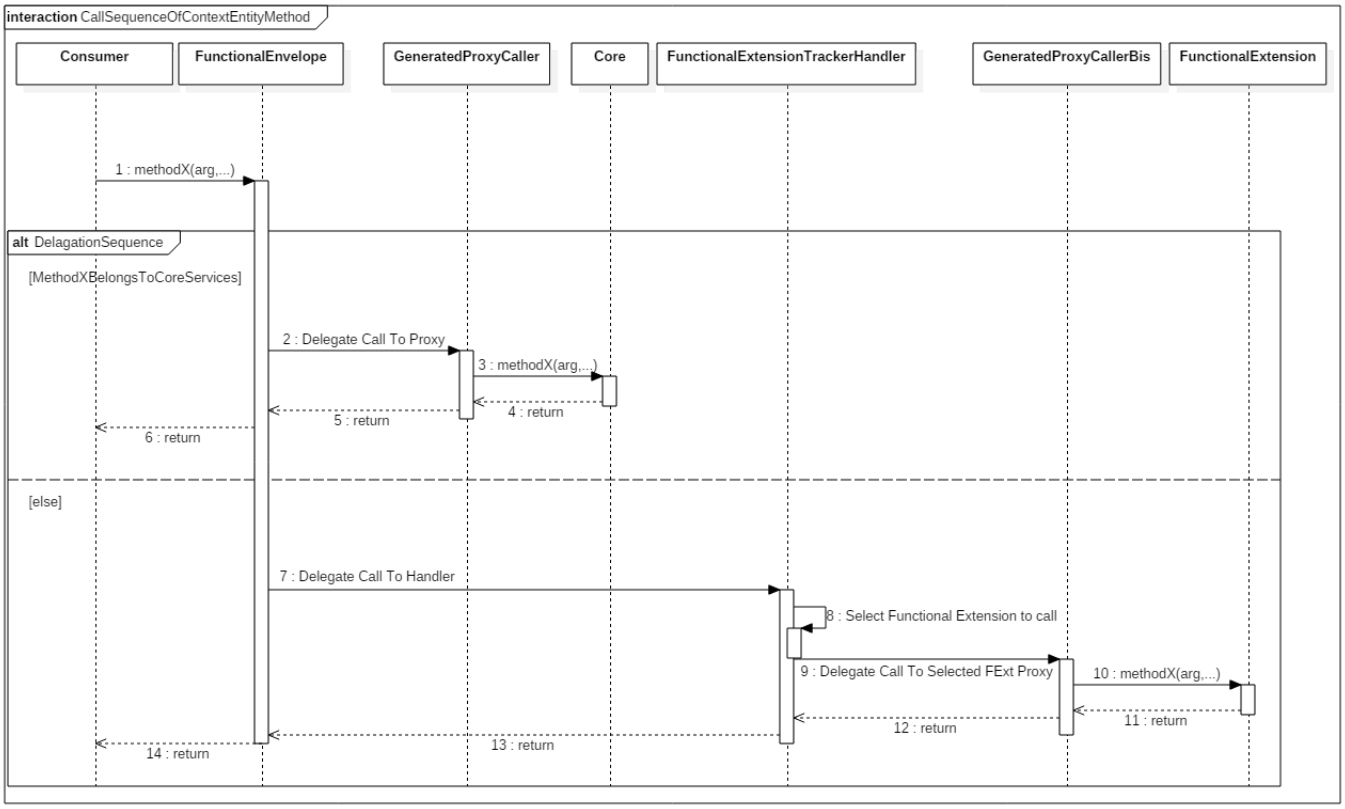

Figure 5.22: Diagramme de séquence de la délégation d'un appel à une instance d'entité de contexte

5.2.5 Synthèse de la validation

Ces différentes évaluations ont permis de montrer la faisabilité de la mise en place de notre solution sur des exemples de tailles réelles. De plus, comme nous avons pu l'observer notre solution impacte de manière favorable la production de code liée à un module de contexte. Nous sommes cependant conscients des limites et des biais de cette première évaluation. Elle ne permet pas en effet d'apporter la preuve de la réelle efficacité de notre approche. Pour cela, il aurait fallu mettre entre les mains de centaines de développeurs indépendants notre solution et analyser leurs productions ainsi que leur courbe d'apprentissage pour conclure de manière plus formelle. Cependant, les chiffres produits sont encourageants et confirment dans un premiers temps la viabilité d'une telle approche.

Nous avons aussi pu quantifier l'*overhead* en termes de performance et de mémoire de notre solution par rapport à l'utilisation d'un SOCM seul. Nous estimons que celui-ci peut être considéré comme négligeable pour une plateforme de *Fog Computing* et est compensé par la réduction de l'effort et la difficulté de la tâche du développement d'un module de contexte. De plus le module de contexte à l'exécution est maintenant doté de capacité autonomique, absente auparavant.

Conclusion et perspectives

Les premières étoiles se montraient. Bientôt toutes ses amies lointaines parsemèrent le ciel. Le poisson aussi est mon ami, dit-il tout haut. Pourtant faut que je le tue. Heureusement qu'on est pas obligé de tuer les étoiles !

Ernest Hemingway - Le vieil homme et la mer

5.3 Résumé des travaux de thèse

Le développement d'applications sensibles au contexte est une des clés de l'établissement durable de la vision de l'informatique pervasive. Cependant la mise en place de ce type d'application reste extrêmement complexe. D'une part, celles-ci peuvent s'exécuter sur un nombre extrêmement varié d'infrastructures allant d'un serveur distant hébergé dans des infrastructures *Cloud Computing* jusque, à contrario, à des passerelles *Fog Computing* au plus près de l'utilisateur. D'autre part, ces applications sont par essence capables de s'adapter durant leur cycle de vie aux différentes variations de l'environnement engendrées par sa dérive naturelle ou par les interactions de l'utilisateur. Pour en simplifier le développement et l'exécution, l'approche la plus populaire a été de déléguer à l'infrastructure d'exécution la tâche de fournir un modèle de l'environnement et d'en créer une instance synchronisée et accessible par les applications. Un large spectre de techniques a été exploré pour la réalisation de ces tâches mais celles-ci doivent être mises en perspective avec les capacités et contraintes de l'infrastructure ciblée ainsi que les attentes des applications s'y exécutant.

Dans cette thèse, nous nous sommes particulièrement intéressés à la construction d'un module de contexte au sein d'une infrastructure *Fog Computing*. La composante applicative d'une telle plateforme a des exigences précises et identifiées en termes de remontée de données et possibilité d'actions pour pouvoir réagir à tout évènement de l'environnement et tenter de l'influencer à travers le réseau d'actionneurs disponibles. Toutes les données ne sont pas obligatoirement consommées à l'intérieur de la plateforme mais peuvent être remontées, après avoir subi plusieurs opérations de médiation, à des infrastructures *Cloud Computing* pour des traitements ayant des visées à plus long terme. De son côté, le module de contexte est, lui, directement en prise avec le monde physique, sa dynamique, ses besoins constants d'évolution et d'intégration. Son développement soulève aujourd'hui de nombreux défis tels que :

Le code est extrêmement complexe: Le code du module de contexte doit d'une part supporter l'intégration et la synchronisation de sources de contexte distantes, volatiles et hétérogènes disséminées dans l'environnement. De plus, il doit être capable de s'adapter à l'apparition ou la disparition des sources de contexte pour pouvoir proposer des entités ou éléments plus abstraits.

L'ajout de nouveaux fournisseurs et besoins est une tâche infinie: L'apparition de nouvelles applications, l'évolution des besoins de celles déjà présentes ou l'évolution des technologies des sources de contexte conduisent le développeur du module de contexte à ajouter perpétuellement de nouvelles fonctionnalités aux fournisseurs déjà présents ou à intégrer de nouveaux fournisseurs dans la plateforme.

L'autonomie est nécessaire: Un des buts majeurs de la plateforme *Fog Computing* est de combler les besoins des applications tout en essayant de préserver au maximum les ressources de l'environnement. Pour cela, la solution de contexte doit être continuellement administrée. Cependant, la haute technicité des opérations d'administration et la

rapidité à laquelle celles-ci doivent être effectuées couplée à l'absence d'administrateur qualifié font de l'autonomie une caractéristique nécessaire à ce type de solution.

Nos différentes contributions s'articulent autour du canevas architectural, rappelé dans la figure 5.23, qui permet la conception d'applications interagissant avec l'environnement au travers d'un ensemble de services exposé dynamiquement selon leurs besoins. Ces contributions fournissent la base pour orchestrer le développement d'un module de contexte de manière simple, extensible et assurer une exécution autonome. Un détail des différentes contributions est proposé par la suite.

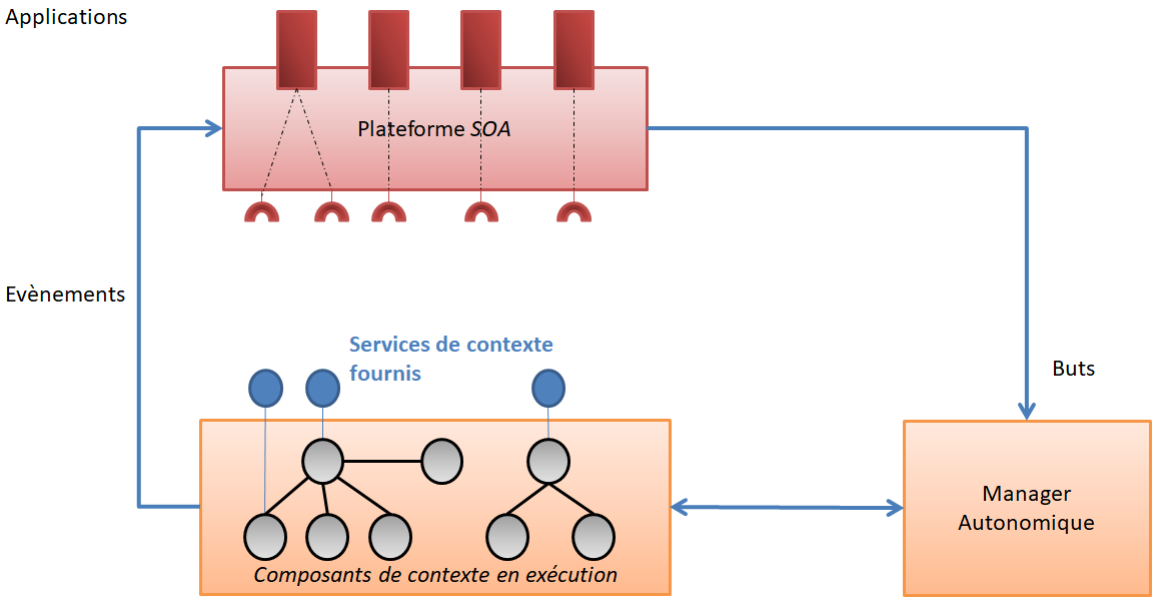

Figure 5.23: Rappel du canevas architectural de notre solution

Context as a Service: Nous avons proposé que le contexte soit modélisé comme un ensemble de services. Cela permet aux applications d'exprimer clairement et simplement leurs besoins en termes d'actions, de remontée de données et d'évènements en se basant sur les différentes descriptions de services proposées par la plateforme. A l'exécution, cet ensemble de services est exposé selon l'état des ressources de l'environnement et des besoins des applications. Cela permet entre autres aux applications d'être immédiatement notifiées de l'apparition, disparation ou modification d'un fournisseur de service. Nous avons ainsi pu constater que cette architecture présente comme avantage de masquer l'hétérogénéité des technologies de communication, des techniques de raisonnement et de médiation aux applications tout en autorisant une composition dynamique au sein du module de contexte pour satisfaire leurs besoins.

Un modèle à composant spécialisé dans le développement des services de contexte: Nous avons proposé un modèle à composant répondant aux contraintes du développement de services de contexte. Il s'appuie tout d'abord sur les principes éprouvés des SOCM pour ce qui a trait à la publication et la consommation de service. Il intègre principalement deux nouveautés. La

première est la faculté donnée au développeur de décrire simplement les fonctions contenant la logique de synchronisation avec une source de contexte et leur orchestration au moyen d'un DSL. La seconde est la capacité à construire des fournisseurs multi-services de façon modulaire tout en conservant un modèle de développement simple. En plus des avantages en termes de lisibilité de code, de souplesse et de séparation des préoccupations, cela permet à la plateforme de mettre à disposition un ensemble d'implémentations réutilisables qui conduit à accélérer l'intégration de nouveaux fournisseurs de contexte. A l'exécution, les conteneurs des instances de composants de contexte exposent des points d'introspection et de reconfiguration des précédents ajouts ce qui permet de rendre le modèle entièrement administrable.

Un management autonomique: Nous avons proposé de mettre en place un manager autonomique servant à guider la composition dynamique du module dans le but de satisfaire les besoins des applications. Son rôle est de limiter l'instanciation des composants de contexte aux seuls besoins des applications pour éviter une perte de ressources dans des processus de synchronisation coûteux. Il peut donc autoriser la création d'instances de composant de contexte en fonction de ce qui a été découvert dans l'environnement, détruire les instances inutilisées ou reconfigurer partiellement les module utilisés lors de l'implémentation des instances.

Ces différentes contributions ont été pleinement implémentées au sein du modèle à composant *CReAM* disponible en open source à l'adresse: `https://github.com/aygalinc/Context-SOCM`. Ces travaux ont pu être intégrés à la plateforme domotique iCASA pour en vérifier la viabilité.

5.4 Perspectives

5.4.1 Enrichissement de la plateforme iCasa

Un des travaux à court terme vise à étendre le module de contexte de la plateforme iCasa avec de nouveaux services de contexte pour plusieurs raisons. Actuellement, nous nous sommes focalisés sur l'intégration, avec succès, de plusieurs technologies et types de capteur et d'actionneur, ceux-ci étant soit atteignables localement par la plateforme ou aux travers d'autres plateformes pervasives. Ce choix a été fait car le module de contexte de iCasa déjà présent ne possédait que cet ensemble d'abstractions. Nous n'avons, à l'opposé, développé que peu de services de contexte plus abstrait permettant par exemple de manipuler des grandeurs physiques par zone. Un élargissement vertical du module de contexte de la plateforme serait donc une plus-value pour celle-ci et permettrait la mise en place de nouvelles applications.

De plus, ces développements additionnels permettraient de venir renforcer notre partie validation. D'une part, s'ils sont effectués par des développeurs indépendants cela permettrait de faire disparaitre certains biais de confirmation dont est victime notre validation. D'autre part, de nouvelles métriques seraient accessibles comme la courbe d'apprentissage liée à notre solution ce qui permettrait de conclure de manière plus formelle sur l'utilisabilité de notre solution. Enfin, ceux-ci permettraient aussi d'identifier s'il existe des manques dans les concepts avancés par notre approche.

5.4.2 Qualité de contexte

Comme nous avons pu le voir la qualité de contexte est un des enjeux majeurs des solutions sensibles au contexte. Les plateformes de *Fog Computing* seront donc amenées à supporter des mécanismes intégrant cette notion pour livrer des services plus efficients et augmenter la satisfaction de l'utilisateur final.

D'une part, ces travaux soulèvent des questions du côté des producteurs de service de contexte. On peut aisément étendre le modèle présenté précédemment et associer à chaque variable d'état une métadonnée reflétant la qualité de l'information. Les différentes métadonnées de qualité sont depuis longtemps connues [Bel+12]. L'interrogation demeure sur la façon d'assigner une valeur à ces métadonnées de qualité. La façon d'attribuer une valeur à un paramètre varie selon la nature de celui-ci et l'entité où le paramètre est utilisé [Bel+12]. La valeur peut soit être configurée en dur comme la résolution d'un capteur particulier, soit programmée de manière spécifique comme la confiance dans un service de présence par pièce qui dépend du nombre de dispositifs validant une présence dans la pièce divisé par le nombre total de dispositifs capables de mesurer l'information de présence [Ran+04], soit programmée de manière générique comme par exemple la fraicheur d'une information qui s'actualise de manière automatique à chaque fois que l'information est mise à jour. Dans certains cas, c'est

le fournisseur de service qui est chargé d'effectuer le calcul [Ran+04] tandis que dans d'autres c'est une tierce partie qui a la responsabilité d'évaluer l'information de qualité [Fil+10]. Une piste intéressante dans notre cas pourrait être de proposer un modèle de développement du calcul de la qualité de contexte, avec une unité de modularité appropriée qui puisse être intégrée dans le conteneur de l'instance d'entité de contexte. Cela permettrait au fournisseur de contexte de bien séparer les préoccupations de contexte de son code métier mais aussi à une tierce partie de venir intégrer des méthodes de calcul de qualité de contexte de façon transparente auprès des fournisseurs.

D'autre part, une partie de ces travaux concernent aussi les consommateurs de contexte. Une des limites de nos travaux est que nous nous sommes surtout focalisés sur la facilitation de la production de service de contexte et que nous nous nous sommes reposés sur la syntaxe des dépendances de Apache Felix iPOJO pour la consommation de ces services. Or ce système de dépendances n'a pas de mécanisme spécifique pour préciser facilement la consommation de services selon certains niveaux de qualité. Un premier pas pour l'amélioration du système de dépendance pourrait être d'intégrer un mécanisme permettant au développeur, de manière déclarative, d'associer à une dépendance de service de contexte un niveau d'exigence de qualité de contexte. Des inspirations sont bien sûr à chercher du coté des mécanismes SLA [KL03] [Lud+03] tout en gardant en tête les limites des similitudes entre la QoS et la QoC [BS03].

5.4.3 Vers un module de contexte plus autonomique

Une perspective sur le long terme peut être de rendre le module de contexte *self-aware*. Un système *self-aware* [Kou+17] en informatique est un système capable d'apprendre d'un modèle servant à capturer des connaissances particulières à propos du système en question ou de son environnement, de raisonner en utilisant ces modèles en accord avec des buts de haut niveau puis de prendre des décisions en accord avec ces buts. Ces décisions ont pour effet de modifier la structure interne du système ou son environnement. Cette nouvelle approche de l'informatique autonomique est relativement récente et pourrait être porteuse d'avenir quant à l'amélioration des middlewares de contexte.

L'introduction de la qualité de contexte, présentée dans la perspective précédente, soulève déjà de nombreux challenges dans le développement et l'exécution du module de contexte. Mais de nouvelles perspectives s'ouvrent aussi pour l'administration autonomique du module de contexte. Comme nous l'avons évoqué tout au long de ce document, l'objectif principal du module de contexte est de satisfaire les besoins des applications. Si dorénavant, les besoins sont exprimés en termes de fonctionnalité (interface de service) et de qualité de contexte, un management autonomique beaucoup plus évolué que celui présenté dans ces travaux est nécessaire. En effet, le manager autonomique doit maintenant savoir comment impacter la qualité de contexte, donc posséder un modèle de la structure interne du module de contexte, estimer, voire anticiper, le coût des adaptations sur l'environnement pour en économiser les ressources au maximum, en s'inspirant de [Kou+16]. C'est sur ces points où nous pensons que

les approches, architectures, modèles et leurs formalismes associés développés par la communauté self-aware [Kou+15] [Kou+16] [Kou+17] peuvent aider à atteindre un nouveau niveau d'autonomie pour les solutions de contexte. La faisabilité de la mise en place d'une telle approche est actuellement en développement au sein de l'équipe ADELE et est présentée dans plusieurs travaux [LGC17] [Ger+17].

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.

[AH14] Mohammad Aazam and Eui-Nam Huh. "Fog Computing and Smart Gateway Based Communication for Cloud of Things." In: *2014 International Conference on Future Internet of Things and Cloud, FiCloud 2014, Barcelona, Spain, August 27-29, 2014*. 2014, pp. 464–470. DOI: 10.1109/FiCloud.2014.83. URL: https://doi.org/10.1109/FiCloud.2014.83.

[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/.

[All07] OSGi Alliance. *OSGi Service Platform Release 4*. 2007, pp. 74–206. URL: http://www.osgi.org/Main/HomePage.

[AM00] Gregory D. Abowd and Elizabeth D. Mynatt. "Charting past, present, and future research in ubiquitous computing." In: *ACM Trans. Comput.-Hum. Interact.* 7.1 (2000), pp. 29–58. DOI: 10.1145/344949.344988. URL: http://doi.acm.org/10.1145/344949.344988.

[Arm+10] Michael Armbrust et al. "A view of cloud computing." In: *Commun. ACM* 53.4 (2010), pp. 50–58. DOI: 10.1145/1721654.1721672. URL: http://doi.acm.org/10.1145/1721654.1721672.

[Ayg+16a] Colin Aygalinc et al. "A model-based approach to context management in pervasive platforms." In: *2016 IEEE International Conference on Pervasive Computing and Communication Workshops, PerCom Workshops 2016, Sydney, Australia, March 14-18, 2016*. 2016, pp. 1–6. DOI: 10.1109/PERCOMW.2016.7457109. URL: https://doi.org/10.1109/PERCOMW.2016.7457109.

[Ayg+16b] Colin Aygalinc et al. "Autonomic Management of Pervasive Context." In: *2016 IEEE International Conference on Autonomic Computing, ICAC 2016, Wuerzburg, Germany, July 17-22, 2016*. 2016, pp. 241–242. DOI: 10.1109/ICAC.2016.64. URL: https://doi.org/10.1109/ICAC.2016.64.

[Ayg+16c] Colin Aygalinc et al. "Autonomic Service-Oriented Context for Pervasive Applications." In: *IEEE International Conference on Services Computing, SCC 2016, San Francisco, CA, USA, June 27 - July 2, 2016*. 2016, pp. 491–498. DOI: 10.1109/SCC.2016.70. URL: https://doi.org/10.1109/SCC.2016.70.

[Ayg+16d] Colin Aygalinc et al. "Service-Oriented Autonomic Pervasive Context." In: *Service-Oriented Computing - 14th International Conference, ICSOC 2016, Banff, AB, Canada, October 10-13, 2016, Proceedings*. 2016, pp. 795–809. DOI: 10.1007/978-3-319-46295-0_56. URL: https://doi.org/10.1007/978-3-319-46295-0_56.

[Ayg+16e] Colin Aygalinc et al. "Service-oriented context for pervasive applications." In: *2016 IEEE International Conference on Pervasive Computing and Communication Workshops, PerCom Workshops 2016, Sydney, Australia, March 14-18, 2016*. 2016, pp. 1–2. DOI: 10.1109/PERCOMW.2016.7457080. URL: https://doi.org/10.1109/PERCOMW.2016.7457080.

[Bal+02] Rajesh Krishna Balan et al. "The case for cyber foraging." In: *Proceedings of the 10th ACM SIGOPS European Workshop, Saint-Emilion, France, July 1, 2002*. 2002, pp. 87–92. DOI: 10.1145/1133373.1133390. URL: http://doi.acm.org/10.1145/1133373.1133390.

[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.

[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.

[Bel+12] Paolo Bellavista et al. "A survey of context data distribution for mobile ubiquitous systems." In: *ACM Comput. Surv.* 44.4 (2012), 24:1–24:45. DOI: 10.1145/2333112.2333119. URL: http://doi.acm.org/10.1145/2333112.2333119.

[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.

[Ber96] Philip A. Bernstein. "Middleware: A Model for Distributed System Services." In: *Commun. ACM* 39.2 (1996), pp. 86–98. DOI: 10.1145/230798.230809. URL: http://doi.acm.org/10.1145/230798.230809.

[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.

[BFA05] Arthur H. van Bunningen, Ling Feng, and Peter M. G. Apers. "Context for Ubiquitous Data Management." In: *2005 International Workshop on Ubiquitous Data Management (UDM 2005), 4 April 2005, Tokyo, Japan*. 2005, pp. 17–24. DOI: 10.1109/UDM.2005.7. URL: https://doi.org/10.1109/UDM.2005.7.

[BG01] Jean Bézivin and Olivier Gerbé. "Towards a Precise Definition of the OMG/MDA Framework." In: *16th IEEE International Conference on Automated Software Engineering (ASE 2001), 26-29 November 2001, Coronado Island, San Diego, CA, USA*. 2001, pp. 273–280. DOI: 10.1109/ASE.2001.989813. URL: https://doi.org/10.1109/ASE.2001.989813.

[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.

[BLE10] Jonathan Bardin, Philippe Lalanda, and Clément Escoffier. "Towards an Automatic Integration of Heterogeneous Services and Devices." In: *5th IEEE Asia-Pacific Services Computing Conference, APSCC 2010, 6-10 December 2010, Hangzhou, China, Proceedings*. 2010, pp. 171–178. DOI: 10.1109/APSCC.2010.89. URL: https://doi.org/10.1109/APSCC.2010.89.

[Bon+12] Flavio Bonomi et al. "Fog computing and its role in the internet of things." In: *Proceedings of the first edition of the MCC workshop on Mobile cloud computing, MCC@SIGCOMM 2012, Helsinki, Finland, August 17, 2012*. 2012, pp. 13–16.

DOI: 10.1145/2342509.2342513. URL: http://doi.acm.org/10.1145/2342509.2342513.

[Bou14] Pierre Bourret. "Modèle à Composant pour Plate-forme Autonomique. (Component model for Autonomic-Ready plateform)." PhD thesis. Grenoble Alpes University, France, 2014. URL: https://tel.archives-ouvertes.fr/tel-01214962.

[Bru+04] Eric Bruneton et al. "An Open Component Model and Its Support in Java." In: *Component-Based Software Engineering, 7th International Symposium, CBSE 2004, Edinburgh, UK, May 24-25, 2004, Proceedings*. 2004, pp. 7–22. DOI: 10.1007/978-3-540-24774-6_3. URL: https://doi.org/10.1007/978-3-540-24774-6_3.

[Bru+06] Eric Bruneton et al. "The FRACTAL component model and its support in Java." In: *Softw., Pract. Exper.* 36.11-12 (2006), pp. 1257–1284. DOI: 10.1002/spe.767. URL: https://doi.org/10.1002/spe.767.

[Bru+11] Julien Bruneau et al. *Design-driven Development of Safety-critical Applications: A Case Study In Avionics*. Technical Report. INRIA, 2011. URL: https://hal.inria.fr/inria-00638203.

[BS03] Thomas Buchholz and Michael Schiffers. "Quality of Context: What It Is And Why We Need It." In: *In Proceedings of the 10th Workshop of the OpenView University Association: OVUA'03*. 2003.

[Cas+11] Damien Cassou et al. "Leveraging software architectures to guide and verify the development of sense/compute/control applications." In: *Proceedings of the 33rd International Conference on Software Engineering, ICSE 2011, Waikiki, Honolulu , HI, USA, May 21-28, 2011*. 2011, pp. 431–440. DOI: 10.1145/1985793.1985852. URL: http://doi.acm.org/10.1145/1985793.1985852.

[CBC10] Damien Cassou, Julien Bruneau, and Charles Consel. "A tool suite to prototype pervasive computing applications." In: *Eigth Annual IEEE International Conference on Pervasive Computing and Communications, PerCom 2010, March 29 - April 2, 2010, Mannheim, Germany, Workshop Proceedings*. 2010, pp. 820–822. DOI: 10.1109/PERCOMW.2010.5470550. URL: https://doi.org/10.1109/PERCOMW.2010.5470550.

[CF03] Ekaterina Chtcherbina and Marquart Franz. "Peer-to-peer coordination framework (p2pc): Enabler of mobile ad-hoc networking for medicine, business, and entertainment." In: *Proceedings of International Conference on Advances in Infrastructure for Electronic Business, Education, Science, Medicine, and Mobile Technologies on the Internet, L'Aquila, Italy*. 2003, pp. 883–891.

[CFJ03] Harry Chen, Tim Finin, and Anupam Joshi. "An ontology for context-aware pervasive computing environments." In: *Knowledge Eng. Review* 18.3 (2003), pp. 197–207. DOI: 10.1017/S0269888904000025. URL: https://doi.org/10.1017/S0269888904000025.

[CH01] WT Council and GT Heineman. "Component-based Software Engineering Putting the Pieces Together." In: *Addison Weysley* (2001).

[Che+04] Harry Chen et al. "SOUPA: Standard Ontology for Ubiquitous and Pervasive Applications." In: *1st Annual International Conference on Mobile and Ubiquitous Systems (MobiQuitous 2004), Networking and Services, 22-25 August 2004, Cambridge, MA, USA*. 2004, pp. 258–267. DOI: 10.1109/MOBIQ.2004.1331732. URL: https://doi.org/10.1109/MOBIQ.2004.1331732.

[Cho09] Stéphanie Chollet. "Orchestration de services hétérogènes et sécurisés. (Orchestration of heterogeneous and secured services)." PhD thesis. Joseph Fourier University, Grenoble, France, 2009. URL: https://tel.archives-ouvertes.fr/tel-00544420.

[Chu36] Alonzo Church. "An unsolvable problem of elementary number theory." In: *American journal of mathematics* 58.2 (1936), pp. 345–363.

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

[Cle96] Paul C Clements. "A survey of architecture description languages." In: *Software Specification and Design, 1996., Proceedings of the 8th International Workshop on*. IEEE. 1996, pp. 16–25.

[CLK08] Guanling Chen, Ming Li, and David Kotz. "Data-centric middleware for context-aware pervasive computing." In: *Pervasive and Mobile Computing* 4.2 (2008), pp. 216–253. DOI: 10.1016/j.pmcj.2007.10.001. URL: https://doi.org/10.1016/j.pmcj.2007.10.001.

[CM00] Paul Castro and Richard R. Muntz. "Managing context data for smart spaces." In: *IEEE Personal Commun.* 7.5 (2000), pp. 44–46. DOI: 10.1109/98.878537. URL: https://doi.org/10.1109/98.878537.

[CMD99] Keith Cheverst, Keith Mitchell, and Nigel Davies. "Design of an object model for a context sensitive tourist GUIDE." In: *Computers & Graphics* 23.6 (1999), pp. 883–891. ISSN: 0097-8493. DOI: http://dx.doi.org/10.1016/S0097-8493(99)00119-3. URL: http://www.sciencedirect.com/science/article/pii/S0097849399001193.

[Con] OW Consortium. *ASM website*. URL: http://asm.ow2.org/.

[Dav89] Fred D. Davis. "Perceived Usefulness, Perceived Ease of Use, and User Acceptance of Information Technology." In: *MIS Quarterly* 13.3 (1989), pp. 319–340. URL: http://misq.org/perceived-usefulness-perceived-ease-of-use-and-user-acceptance-of-information-technology.html.

[De 11] Vincenzo De Florio. "Robust-and-evolvable resilient software systems: Open problems and lessons learned." In: *Proceedings of the 8th workshop on Assurances for self-adaptive systems*. ACM. 2011, pp. 10–17.

[Dey01] Anind K. Dey. "Understanding and Using Context." In: *Personal and Ubiquitous Computing* 5.1 (2001), pp. 4–7. DOI: 10.1007/s007790170019. URL: https://doi.org/10.1007/s007790170019.

[Dij82] Edsger W. Dijkstra. "On the Role of Scientific Thought." In: *Selected Writings on Computing: A personal Perspective*. New York, NY: Springer New York, 1982, pp. 60–66. ISBN: 978-1-4612-5695-3. DOI: 10.1007/978-1-4612-5695-3_12. URL: https://doi.org/10.1007/978-1-4612-5695-3_12.

[EBL13] Clément Escoffier, Pierre Bourret, and Philippe Lalanda. "Describing Dynamism in Service Dependencies: Industrial Experience and Feedbacks." In: *2013 IEEE International Conference on Services Computing, Santa Clara, CA, USA, June 28 - July 3, 2013*. 2013, pp. 328–335. DOI: 10.1109/SCC.2013.82. URL: https://doi.org/10.1109/SCC.2013.82.

[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.

[EHL07] Clément Escoffier, Richard S. Hall, and Philippe Lalanda. "iPOJO: an Extensible Service-Oriented Component Framework." In: *2007 IEEE International Conference on Services Computing (SCC 2007), 9-13 July 2007, Salt Lake City, Utah, USA*. 2007, pp. 474–481. DOI: 10.1109/SCC.2007.74. URL: https://doi.org/10.1109/SCC.2007.74.

[Ena+13] Quentin Enard et al. "Design-driven development methodology for resilient computing." In: *CBSE'13, Proceedings of the 16th ACM SIGSOFT Symposium on Component Based Software Engineering, part of Comparch '13, Vancouver, BC, Canada, June 17-21, 2013*. 2013, pp. 59–64. DOI: 10.1145/2465449.2465458. URL: http://doi.acm.org/10.1145/2465449.2465458.

[Esca] Clement Escoffier. *Dependency Handler*. URL: http://felix.apache.org/documentation/subprojects/apache-felix-ipojo/apache-felix-ipojo-userguide/describing-components/service-requirement-handler.html.

[Escb] Clement Escoffier. *iPOJO Benchmark*. URL: http://felix.apache.org/documentation/subprojects/apache-felix-ipojo/apache-felix-ipojo-userguide/ipojo-faq.html.

[Escc] Clement Escoffier. *iPOJO website*. URL: http://felix.apache.org/documentation/subprojects/apache-felix-ipojo.html.

[Escd] Clement Escoffier. *Provided Service Handler*. URL: http://felix.apache.org/documentation/subprojects/apache-felix-ipojo/apache-felix-ipojo-userguide/describing-components/providing-osgi-services.html.

[Esce] Clement Escoffier. *Wisdom framework website*. URL: http://wisdom-framework.org/.

[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.

[Eva11] Dave Evans. "The internet of things: How the next evolution of the internet is changing everything." In: *CISCO white paper* 1.2011 (2011), pp. 1–11.

[Fas+02] Jean-Philippe Fassino et al. "Think: A Software Framework for Component-based Operating System Kernels." In: *Proceedings of the General Track: 2002 USENIX Annual Technical Conference, June 10-15, 2002, Monterey, California, USA*. 2002, pp. 73–86. URL: http://www.usenix.org/publications/library/proceedings/usenix02/fassino.html.

[FF98] David Franklin and Joshua Flaschbart. "All gadget and no representation makes jack a dull environment." In: *Proceedings of the AAAI 1998 Spring Symposium on Intelligent Environments*. 1998, pp. 155–160.

[Fil+10] José Bringel Filho et al. "Modeling and Measuring Quality of Context Information in Pervasive Environments." In: *24th IEEE International Conference on Advanced Information Networking and Applications, AINA 2010, Perth, Australia, 20-13 April 2010*. 2010, pp. 690–697. DOI: 10.1109/AINA.2010.164. URL: https://doi.org/10.1109/AINA.2010.164.

[Foua] Apache Software Foundation. *Apache license website*. URL: https://www.apache.org/licenses/LICENSE-2.0.

[Foub] Apache Software Foundation. *Maven bundle plugin website*. URL: http://felix.apache.org/documentation/subprojects/apache-felix-maven-bundle-plugin-bnd.html.

[Fouc] Apache Software Foundation. *Maven website*. URL: http%20://maven.apache.org.

[Fou+12] François Fouquet et al. "Kevoree: une approche model@ runtime pour les systèmes ubiquitaires." In: *UbiMob2012*. 2012.

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

[Fow04] Martin Fowler. *Inversion of Control Containers and the Dependency Injection pattern*. 2004. URL: http://www.martinfowler.com/articles/injection.html.

[FT00] Roy T Fielding and Richard N Taylor. *Architectural styles and the design of network-based software architectures*. University of California, Irvine Doctoral dissertation, 2000.

[Gab91] Richard Gabriel. "The rise of "worse is better"." In: *Lisp: Good News, Bad News, How to Win Big* 2.5 (1991).

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

[Gam+95] Erich Gamma et al. "Design patterns: Elements of reusable object-oriented software." In: *Reading: Addison-Wesley* 49.120 (1995), p. 11.

[Gar+02] David Garlan et al. "Project Aura: Toward Distraction-Free Pervasive Computing." In: *IEEE Pervasive Computing* 1.2 (2002), pp. 22–31. DOI: 10.1109/MPRV.2002.1012334. URL: https://doi.org/10.1109/MPRV.2002.1012334.

[Gar+10] Issac Garcia et al. "Towards a Service Mediation Framework for Dynamic Applications." In: *5th IEEE Asia-Pacific Services Computing Conference, APSCC 2010, 6-10 December 2010, Hangzhou, China, Proceedings*. 2010, pp. 3–10. DOI: 10.1109/APSCC.2010.90. URL: https://doi.org/10.1109/APSCC.2010.90.

[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.

[Ger+17] Eva Gerbert-Gaillard et al. "A self-aware approach to context management in pervasive platforms." In: *2017 IEEE International Conference on Pervasive Computing and Communications Workshops, PerCom Workshops 2017, Kona, Big Island, HI, USA, March 13-17, 2017*. 2017, pp. 256–261. DOI: 10.1109/PERCOMW.2017.7917568. URL: https://doi.org/10.1109/PERCOMW.2017.7917568.

[GPZ05] Tao Gu, Hung Keng Pung, and Daqing Zhang. "A service-oriented middleware for building context-aware services." In: *J. Network and Computer Applications* 28.1 (2005), pp. 1–18. DOI: 10.1016/j.jnca.2004.06.002. URL: https://doi.org/10.1016/j.jnca.2004.06.002.

[GR14] Luis Miguel Vaquero Gonzalez and Luis Rodero-Merino. "Finding your Way in the Fog: Towards a Comprehensive Definition of Fog Computing." In: *Computer Communication Review* 44.5 (2014), pp. 27–32. DOI: 10.1145/2677046.2677052. URL: http://doi.acm.org/10.1145/2677046.2677052.

[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/.

[Gu+04] Tao Gu et al. "An ontology-based context model in intelligent environments." In: *Proceedings of communication networks and distributed systems modeling and simulation conference*. Vol. 2004. San Diego, CA, USA. 2004, pp. 270–275.

[Gun14] Ozan Necati Gunalp. "Continuous deployment of pervasive applications in dynamic environments. (Déploiement continue des applications pervasives en milieux dynamiques)." PhD thesis. Grenoble Alpes University, France, 2014. URL: https://tel.archives-ouvertes.fr/tel-01215029.

[GW13] Kurt Geihs and Michael Wagner. "Context-awareness for self-adaptive applications in ubiquitous computing environments." In: *Context-Aware Systems and Applications, Springer* (2013), pp. 108–120.

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

[Hal+12] Svein O. Hallsteinsen et al. "A development framework and methodology for self-adapting applications in ubiquitous computing environments." In: *Journal of Systems and Software* 85.12 (2012), pp. 2840–2859. DOI: 10.1016/j.jss.2012.07.052. URL: https://doi.org/10.1016/j.jss.2012.07.052.

[Hal10] Svein Hallsteinsen. "MUSIC vision and solutions." In: *Rapport technique, IST Music* (Jan. 2010), p. 40.

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

[HBS02] Albert Held, Sven Buchholz, and Alexander Schill. "Modeling of context information for pervasive computing applications." In: *Proceedings of SCI* (2002), pp. 167–180.

[Hel+05] Sumi Helal et al. "The Gator Tech Smart House: A Programmable Pervasive Space." In: *IEEE Computer* 38.3 (2005), pp. 50–60. DOI: 10.1109/MC.2005.107. URL: https://doi.org/10.1109/MC.2005.107.

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

[HIR03] Karen Henricksen, Jadwiga Indulska, and Andry Rakotonirainy. "Generating context management infrastructure from high-level context models." In: *In 4th International Conference on Mobile Data Management (MDM)-Industrial Track*. Citeseer. 2003.

[HIR08] Peizhao Hu, Jadwiga Indulska, and Ricky Robinson. "An Autonomic Context Management System for Pervasive Computing." In: *Sixth Annual IEEE International Conference on Pervasive Computing and Communications (PerCom 2008), 17-21 March 2008, Hong Kong*. 2008, pp. 213–223. DOI: 10.1109/PERCOM.2008.56. URL: https://doi.org/10.1109/PERCOM.2008.56.

[HNB97] Richard Hull, Philip Neaves, and James Bedford-Roberts. "Towards Situated Computing." In: *First International Symposium on Wearable Computers (ISWC 1997), Cambridge, Massachusetts, USA, 13-14 October 1997, Proceedings*. 1997, pp. 146–153. DOI: 10.1109/ISWC.1997.629931. URL: https://doi.org/10.1109/ISWC.1997.629931.

[Hof+03] Thomas Hofer et al. "Context-Awareness on Mobile Devices - the Hydrogen Approach." In: *36th Hawaii International Conference on System Sciences (HICSS-36 2003), CD-ROM / Abstracts Proceedings, January 6-9, 2003, Big Island, HI, USA*. 2003, p. 292. DOI: 10.1109/HICSS.2003.1174831. URL: https://doi.org/10.1109/HICSS.2003.1174831.

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

[IAC16] Unai Alegre Ibarra, Juan Carlos Augusto, and Tony Clark. "Engineering context-aware systems and applications: A survey." In: *Journal of Systems and Software* 117 (2016), pp. 55–83. DOI: 10.1016/j.jss.2016.02.010. URL: https://doi.org/10.1016/j.jss.2016.02.010.

[IEE08] IEEE. *ISO/IEC/IEEE Standard for Systems and Software Engineering - Software Life Cycle Processes*. 2008.

[IMAa] IMAG. *iCasa-IDE website*. URL: http://adeleresearchgroup.github.io/iCasa-IDE/.

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

[IS03] Jadwiga Indulska and Peter Sutton. "Location Management in Pervasive Systems." In: *ACSW Frontiers 2003, 2003 ACSW Workshops - the Australasian Information Security Workshop (AISW) and the Workshop on Wearable, Invisible, Context-Aware, Ambient, Pervasive and Ubiquitous Computing (WICAPUC), Adelaide, South Australia, February 2003*. 2003, pp. 143–151. URL: http://crpit.com/confpapers/CRPITV21WIndulska.pdf.

[JBB07] Christophe Jacquet, Yolaine Bourda, and Yacine Bellik. "A component-based platform for accessing context in ubiquitous computing applications." In: *Journal of Ubiquitous Computing and Intelligence* 1.2 (2007), pp. 163–173.

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

[Jon07] Martin Jonsson. "Sensing and Making Sense : Designing Middleware for Context Aware Computing." PhD thesis. Royal Institute of Technology, Stockholm, Sweden, 2007. URL: http://nbn-resolving.de/urn:nbn:se:kth:diva-4432.

[JPr] JProfiler. *JProfiler website*. URL: https://www.ej-technologies.com/products/jprofiler/overview.html.

[KC03] Jeffrey O. Kephart and David M. Chess. "The Vision of Autonomic Computing." In: *IEEE Computer* 36.1 (2003), pp. 41–50. DOI: 10.1109/MC.2003.1160055. URL: https://doi.org/10.1109/MC.2003.1160055.

[Kin+06] Jeffrey King et al. "Atlas: A Service-Oriented Sensor Platform: Hardware and Middleware to Enable Programmable Pervasive Spaces." In: *LCN 2006, The 31st Annual IEEE Conference on Local Computer Networks, Tampa, Florida, USA, 14-16 November 2006*. 2006, pp. 630–638. DOI: 10.1109/LCN.2006.322026. URL: https://doi.org/10.1109/LCN.2006.322026.

[Kjæ07] Kristian Ellebæk Kjær. "A survey of context-aware middleware." In: *Proceedings of the 25th conference on IASTED International Multi-Conference: Software Engineering*. ACTA Press. 2007, pp. 148–155.

[KL03] Alexander Keller and Heiko Ludwig. "The WSLA Framework: Specifying and Monitoring Service Level Agreements for Web Services." In: *J. Network Syst. Manage.* 11.1 (2003), pp. 57–81. DOI: 10.1023/A:1022445108617. URL: https://doi.org/10.1023/A:1022445108617.

[Kou+15] Samuel Kounev et al. "Model-driven Algorithms and Architectures for Self-Aware Computing Systems (Dagstuhl Seminar 15041)." In: *Dagstuhl Reports* 5.1 (2015), pp. 164–196. DOI: 10.4230/DagRep.5.1.164. URL: https://doi.org/10.4230/DagRep.5.1.164.

[Kou+16] Samuel Kounev et al. "A Model-Based Approach to Designing Self-Aware IT Systems and Infrastructures." In: *IEEE Computer* 49.7 (2016), pp. 53–61. DOI: 10.1109/MC.2016.198. URL: https://doi.org/10.1109/MC.2016.198.

[Kou+17] Samuel Kounev et al., eds. *Self-Aware Computing Systems*. Springer International Publishing, 2017. ISBN: 978-3-319-47472-4. DOI: 10.1007/978-3-319-47474-8. URL: https://doi.org/10.1007/978-3-319-47474-8.

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

[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.

[Leh80] Meir M Lehman. "Programs, life cycles, and laws of software evolution." In: *Proceedings of the IEEE* 68.9 (1980), pp. 1060–1076.

[LGC17] Philippe Lalanda, Eva Gerbert-Gaillard, and Stéphanie Chollet. "Self-Aware Context in Smart Home Pervasive Platforms." In: *2017 IEEE International Conference on Autonomic Computing, ICAC 2017, Columbus, OH, USA, July 17-21, 2017*. 2017, pp. 119–124. DOI: 10.1109/ICAC.2017.1. URL: https://doi.org/10.1109/ICAC.2017.1.

[LSD10] Fei Li, Sanjin Sehic, and Schahram Dustdar. "COPAL: An adaptive approach to context provisioning." In: *IEEE 6th International Conference on Wireless and Mobile Computing, Networking and Communications, WiMob 2010, Niagara Falls, Ontario, Canada, 11-13 October, 2010*. 2010, pp. 286–293. DOI: 10.1109/WIMOB.2010.5645051. URL: https://doi.org/10.1109/WIMOB.2010.5645051.

[Lua+15] Tom H. Luan et al. "Fog Computing: Focusing on Mobile Users at the Edge." In: *CoRR* abs/1502.01815 (2015). URL: http://arxiv.org/abs/1502.01815.

[Luc08] David Luckham. "The Power of Events: An Introduction to Complex Event Processing in Distributed Enterprise Systems." In: *Rule Representation, Interchange and Reasoning on the Web, International Symposium, RuleML 2008, Orlando, FL, USA, October 30-31, 2008. Proceedings*. 2008, p. 3. DOI: 10.1007/978-3-540-88808-6_2. URL: https://doi.org/10.1007/978-3-540-88808-6_2.

[Lud+03] Heiko Ludwig et al. "A Service Level Agreement Language for Dynamic Electronic Services." In: *Electronic Commerce Research* 3.1-2 (2003), pp. 43–59. DOI: 10.1023/A:1021525310424. URL: https://doi.org/10.1023/A:1021525310424.

[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.

[McC93] John McCarthy. "Notes on Formalizing Context." In: *Proceedings of the 13th International Joint Conference on Artificial Intelligence. Chambéry, France, August 28 - September 3, 1993.* 1993, pp. 555–562. URL: http://www-formal.stanford.edu/jmc/context3/context3.html.

[Mey09] John F Meyer. "Defining and evaluating resilience: A performability perspective." In: *Proceedings of the International Workshop on Performability Modeling of Computer and Communication Systems (PMCCS).* Vol. 1. 2009, pp. 561–566.

[Mic] Sun Microsystems. *Enterprise JavaBeans Technology.* URL: http://java.sun.com/products/ejb/index.jsp.

[Moo98] Gordon E Moore. "Cramming more components onto integrated circuits." In: *Proceedings of the IEEE* 86.1 (1998), pp. 82–85.

[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.

[MTD08] Atif Manzoor, Hong Linh Truong, and Schahram Dustdar. "On the Evaluation of Quality of Context." In: *Smart Sensing and Context, Third European Conference, EuroSSC 2008, Zurich, Switzerland, October 29-31, 2008. Proceedings.* 2008, pp. 140–153. DOI: 10.1007/978-3-540-88793-5_11. URL: https://doi.org/10.1007/978-3-540-88793-5_11.

[OAS06] OASIS. *Reference Model for Service Oriented Architecture 1.0.* 2006.

[OMG01] OMG. *Model Driven Architecture.* 2001.

[Oraa] Oracle. *HashMap documentation.* URL: https://docs.oracle.com/javase/8/docs/api/java/util/HashMap.html.

[Orab] Oracle. *Proxy Javadoc.* URL: https://docs.oracle.com/javase/8/docs/api/java/lang/reflect/Proxy.html.

[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.

[Par72] David Lorge Parnas. "On the Criteria To Be Used in Decomposing Systems into Modules." In: *Commun. ACM* 15.12 (1972), pp. 1053–1058. DOI: 10.1145/361598.361623. URL: http://doi.acm.org/10.1145/361598.361623.

[Pas09] Nearchos Paspallis. "Middleware-based development of context-aware applications with reusable components." PhD thesis. University of Cyprus, 2009. URL: https://nearchos.github.io/phd/paspallis_phd_thesis_2009-final.pdf.

[Pat+14] Milan Patel et al. "Mobile-edge computing introductory technical white paper." In: *White Paper, Mobile-edge Computing (MEC) industry initiative* (2014).

[Per+14a] Charith Perera et al. "A Survey on Internet of Things From Industrial Market Perspective." In: *IEEE Access* 2 (2014), pp. 1660–1679. DOI: 10.1109/ACCESS.2015.2389854. URL: https://doi.org/10.1109/ACCESS.2015.2389854.

[Per+14b] Charith Perera et al. "Context Aware Computing for The Internet of Things: A Survey." In: *IEEE Communications Surveys and Tutorials* 16.1 (2014), pp. 414–454. DOI: 10.1109/SURV.2013.042313.00197. URL: https://doi.org/10.1109/SURV.2013.042313.00197.

[Pie+08] Stefan Pietschmann et al. "CroCo: Ontology-Based, Cross-Application Context Management." In: *Third International Workshop on Semantic Media Adaptation and Personalization, SMAP 2008, Prague, Czech Republic, December 15-16, 2008*. 2008, pp. 88–93. DOI: 10.1109/SMAP.2008.10. URL: https://doi.org/10.1109/SMAP.2008.10.

[Pre92] Roger S Pressman. *Software engineering: a practitioner's approach*. Palgrave Macmillan, 1992.

[PRL09] Mikko Perttunen, Jukka Riekki, and Ora Lassila. "Context representation and reasoning in pervasive computing: a review." In: *International Journal of Multimedia and Ubiquitous Engineering* 4.4 (2009).

[Ran+04] Anand Ranganathan et al. "MiddleWhere: A Middleware for Location Awareness in Ubiquitous Computing Applications." In: *Middleware 2004, ACM/IFIP/USENIX International Middleware Conference, Toronto, Canada, October 18-20, 2004, Proceedings*. 2004, pp. 397–416. DOI: 10.1007/978-3-540-30229-2_21. URL: https://doi.org/10.1007/978-3-540-30229-2_21.

[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.

[Rea89] Chris Reade. *Elements of functional programming*. International computer science series. Addison-Wesley, 1989. ISBN: 978-0-201-12915-1.

[Rei+08] Roland Reichle et al. "A Comprehensive Context Modeling Framework for Pervasive Computing Systems." In: *Distributed Applications and Interoperable Systems, 8th IFIP WG 6.1 International Conference, DAIS 2008, Oslo, Norway, June 4-6, 2008. Proceedings*. 2008, pp. 281–295. DOI: 10.1007/978-3-540-68642-2_23. URL: https://doi.org/10.1007/978-3-540-68642-2_23.

[RFL05] Mark Raskino, Jackie Fenn, and Alexander Linden. "Extracting value from the massively connected world of 2015." In: *Gartner Research, Tech. Rep. G00125949* (2005).

[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.

[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.

[Rou+09] Romain Rouvoy et al. "MUSIC: Middleware Support for Self-Adaptation in Ubiquitous and Service-Oriented Environments." In: *Software Engineering for Self-Adaptive Systems [outcome of a Dagstuhl Seminar]*. 2009, pp. 164–182. DOI: 10.1007/978-3-642-02161-9_9. URL: https://doi.org/10.1007/978-3-642-02161-9_9.

[RPM99] Nick Ryan, Jason Pascoe, and David Morse. "Enhanced reality fieldwork: the context aware archaeological assistant." In: *Bar International Series* 750 (1999), pp. 269–274.

[Sat+09] Mahadev Satyanarayanan et al. "The Case for VM-Based Cloudlets in Mobile Computing." In: *IEEE Pervasive Computing* 8.4 (2009), pp. 14–23. DOI: 10.1109/MPRV.2009.82. URL: https://doi.org/10.1109/MPRV.2009.82.

[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.

[SB05] Quan Z. Sheng and Boualem Benatallah. "ContextUML: A UML-Based Modeling Language for Model-Driven Development of Context-Aware Web Services." In: *2005 International Conference on Mobile Business (ICMB 2005), 11-13 July 2005, Sydney, Australia*. 2005, pp. 206–212. DOI: 10.1109/ICMB.2005.33. URL: https://doi.org/10.1109/ICMB.2005.33.

[Sch+12] Fabio A. Schreiber et al. "PerLa: A Language and Middleware Architecture for Data Management and Integration in Pervasive Information Systems." In: *IEEE Trans. Software Eng.* 38.2 (2012), pp. 478–496. DOI: 10.1109/TSE.2011.25. URL: https://doi.org/10.1109/TSE.2011.25.

[SDA99] Daniel Salber, Anind K. Dey, and Gregory D. Abowd. "The Context Toolkit: Aiding the Development of Context-Enabled Applications." In: *Proceeding of the CHI '99 Conference on Human Factors in Computing Systems: The CHI is the Limit, Pittsburgh, PA, USA, May 15-20, 1999*. 1999, pp. 434–441. DOI: 10.1145/302979.303126. URL: http://doi.acm.org/10.1145/302979.303126.

[SG02] João Pedro Sousa and David Garlan. "Aura: an Architectural Framework for User Mobility in Ubiquitous Computing Environments." In: *Software Architecture: System Design, Development and Maintenance, IFIP 17th World Computer Congress - TC2 Stream / 3rd IEEE/IFIP Conference on Software Architecture (WICSA3), August 25-30, 2002, Montréal, Québec, Canada.* 2002, pp. 29–43.

[SG96] Mary Shaw and David Garlan. *Software architecture - perspectives on an emerging discipline.* Prentice Hall, 1996. ISBN: 978-0-13-182957-2.

[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.

[SL04] Thomas Strang and Claudia Linnhoff-Popien. "A context modeling survey." In: *Workshop Proceedings.* 2004.

[SML01] Michael Samulowitz, Florian Michahelles, and Claudia Linnhoff-Popien. "CAPEUS: An Architecture for Context-Aware Selection and Execution of Services." In: *New Developments in Distributed Applications and Interoperable Systems, IFIP TC6 / WG6.1 Third International Working Conference on Distributed Applications and Interoperable Systems, September 17-19, 2001, Kraków, Poland.* 2001, pp. 23–39. DOI: 10.1007/0-306-47005-5_3. URL: https://doi.org/10.1007/0-306-47005-5_3.

[Sona] Sonarsource. *SonarQube metrics doc.* URL: https://docs.sonarqube.org/display/SONAR/Metric+Definitions.

[Sonb] Sonarsource. *SonarQube website.* URL: https://www.sonarqube.org/.

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

[Sta06] Michael Stal. "Using Architectural Patterns and Blueprints for Service-Oriented Architecture." In: *IEEE Software* 23.2 (2006), pp. 54–61. DOI: 10.1109/MS.2006.60. URL: https://doi.org/10.1109/MS.2006.60.

[SW14] Ivan Stojmenovic and Sheng Wen. "The Fog Computing Paradigm: Scenarios and Security Issues." In: *Proceedings of the 2014 Federated Conference on Computer Science and Information Systems, Warsaw, Poland, September 7-10, 2014.* 2014, pp. 1–8. DOI: 10.15439/2014F503. URL: https://doi.org/10.15439/2014F503.

[Tig+09] Jean-Yves Tigli et al. "WComp middleware for ubiquitous computing: Aspects and composite event-based Web services." In: *Annales des Télécommunications* 64.3-4 (2009), pp. 197–214. DOI: 10.1007/s12243-008-0081-y. URL: https://doi.org/10.1007/s12243-008-0081-y.

[Tra] TravisCI. *Travis website.* URL: https://travis-ci.org/.

[Tur38] Alan M. Turing. "Systems of Logic Based on Ordinals." PhD thesis. Princeton University, NJ, USA, 1938. DOI: 10.1112/plms/s2-45.1.161. URL: https://doi.org/10.1112/plms/s2-45.1.161.

[Van+13] Sebastian VanSyckel et al. "Configuration Management for Proactive Adaptation in Pervasive Environments." In: *7th IEEE International Conference on Self-Adaptive and Self-Organizing Systems, SASO 2013, Philadelphia, PA, USA, September 9-13, 2013*. 2013, pp. 131–140. DOI: 10.1109/SASO.2013.28. URL: https://doi.org/10.1109/SASO.2013.28.

[Van13] Sebastian VanSyckel. "Middleware-based system support for proactive adaptation in pervasive environments." In: *2013 IEEE International Conference on Pervasive Computing and Communications Workshops, PERCOM 2013 Workshops, San Diego, CA, USA, March 18-22, 2013*. 2013, pp. 425–426. DOI: 10.1109/PerComW.2013.6529534. URL: https://doi.org/10.1109/PerComW.2013.6529534.

[Van15] Sebastian VanSyckel. "System Support for Proactive Adaptation." PhD thesis. University of Mannheim, 2015. URL: https://ub-madoc.bib.uni-mannheim.de/39016.

[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+04] Xiaohang Wang et al. "Ontology Based Context Modeling and Reasoning using OWL." In: *2nd IEEE Conference on Pervasive Computing and Communications Workshops (PerCom 2004 Workshops), 14-17 March 2004, Orlando, FL, USA*. 2004, pp. 18–22. DOI: 10.1109/PERCOMW.2004.1276898. URL: https://doi.org/10.1109/PERCOMW.2004.1276898.

[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.

[WG00] Zhenyu Wang and David Garlan. "Task-driven computing." In: *Computer Science Department* (2000), p. 695.

[Wie92] Gio Wiederhold. "Mediators in the Architecture of Future Information Systems." In: *IEEE Computer* 25.3 (1992), pp. 38–49. DOI: 10.1109/2.121508. URL: https://doi.org/10.1109/2.121508.

[Wil85] Maurice Wilkes. *Memoirs of a computer pioneer*. Massachusetts Institute of Technology, 1985.

[Win] Rafael Winterhalter. *Byte buddy website*. URL: http://bytebuddy.net/#/.

Résumé — L'informatique pervasive promeut une vision d'un cadre dans lequel un patchwork de ressources hétérogènes et volatiles est intégré dans les environnements du quotidien. Ces ressources, matérielles ou logicielles, coopèrent de manière transparente, souvent aux travers d'applications, pour fournir des services à haute valeur ajoutée adaptés à chaque utilisateur et son environnement, grâce à la notion de contexte. Ces applications sont déployées dans un large spectre d'environnements d'exécution, allant d'infrastructures distantes de *Cloud Computing* jusqu'au plus près de l'utilisateur dans des passerelles *Fog Computing* ou directement dans les capteurs du réseau. Dans ces travaux, nous nous intéressons spécifiquement au module de contexte d'une plateforme *Fog Computing*. Pour faciliter la conception et l'exécution des applications *Fog Computing*, une approche populaire est de les bâtir au dessus d'une plateforme adoptant l'architecture à service, ce qui permet de réduire leur complexité et simplifie la gestion du dynamisme. Dans nos travaux, nous proposons d'étendre cette approche en modélisant le contexte comme un ensemble de descriptions de services, disponible à la conception, et exposé dynamiquement par le module de contexte à l'exécution, selon les besoins des applications et l'état de l'environnement. Ce module est programmé à l'aide d'un modèle à composant spécifique. L'unité de base de notre modèle à composant est l'entité de contexte, qui est composée de modules hautement cohérents implémentant distinctement les spécifications des services proposés par l'entité de contexte. Ces modules peuvent décrire de manière simple leur logique de synchronisation avec les sources de contexte distantes grâce à un langage dédié à ce domaine. A l'exécution, les instances d'entitées de contexte sont rendues introspectables et reconfigurables dynamiquement, ce qui permet, grâce à un manager autonomique externe, de veiller à la satisfaction des besoins des applications. Nous avons développé une implémentation de référence de ce modèle à composant, nommée CReAM, qui a pu être utilisée dans la passerelle domotique iCASA, développée en partenariat avec *Orange Labs*.

Mots clés : informatique pervasive, sensibilité au contexte, fog computing, approche orientée service, modèle à composant, génie logiciel.

Université Grenoble Alpes, LIG

Abstract — Pervasive computing promotes environments where a patchwork of heterogeneous and volatile resources are integrated in daily life. These hardware and software resources cooperate in a transparent way, through applications, in order to provide high value-added services. These services are adapted to each user and its environment, via the notion of context. Pervasive applications are now widely distributed, from distant cloud facilities down to Fog Computing gateway or even in sensors, near the user. Depending on the localization, various forms of context are needed by the applications. In this thesis, we focus on the context module at Fog Level. In order to simplify the design and execution, Fog applications are built on top of a service-oriented platform, freeing the developer of technical complexity and providing a support to handle the dynamism. We propose to extend this approach by providing the context as a set of service descriptions, available at design to the application developer. At runtime, depending on the context sources availability and on application current needs, context services are published or withdrawn inside the platform by the context module. We tailor a specific component model to program this context module. The base unit of our component model is called context entity. It is composed of highly coherent modules, implementing distinctly each service description proposed by the underlying context entity. These modules can simply describe their synchronization logic with context sources thanks to a domain specific language. At runtime, context entity instances can be introspected and reconfigured. An external autonomic manager uses these properties to match dynamically the context services exposed by the context module to the application needs. We have developed a reference implementation of our work, called CReAM, which can be used in a smart home gateway called iCASA, developed in a partnership with Orange Labs.

Keywords: pervasive computing, context-awareness, fog computing, service oriented architecture, component model, software engineering.

Université Grenoble Alpes, LIG