HAL open science

Cilia : un framework pour le développement d'applications de médiation autonomiques

Denis Morand

To cite this version:

Denis Morand. Cilia : un framework pour le développement d'applications de médiation autonomiques. Informatique ubiquitaire. Université Joseph-Fourier - Grenoble I, 2013. Français. NNT : . tel-00952107

HAL Id: tel-00952107

https://theses.hal.science/tel-00952107v1

Submitted on 26 Feb 2014

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

UNIVERSITÉ DE GRENOBLE

THÈSE

Pour obtenir le grade de

DOCTEUR DE L'UNIVERSITÉ DE GRENOBLE

Spécialité : **Informatique**

Arrêté ministériel : 7 août 2006

Présentée par

Denis Morand

Thèse dirigée par **Pr. Philippe Lalanda**
et codirigée par **Dr. Stéphanie Chollet**

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

Cilia : un framework pour le développement d'applications de médiation autonomiques

Thèse soutenue publiquement le **5 novembre 2013**,
devant le jury composé de :

Pr. Dominique Rieu
Professeur à l'Université Pierre Mendès France, Présidente
Pr. Françoise Baude
Professeur à l'Université de Nice Sophia-Antipolis, Rapporteur
Pr. David R. C. Hill
Professeur à l'ISIMA, Rapporteur
Dr. Julien Ponge
Maître de Conférences à l'INSA Lyon, Examinateur
Benoît Jacquemin
V.P. Schneider Electric, Invité
Pr. Philippe Lalanda
Professeur à l'Université Joseph Fourier, Directeur de thèse
Dr. Stéphanie Chollet
Maître de Conférences à Grenoble INP, Co-Directrice de thèse

Remerciements

Ce travail terminé, je tiens à remercier tous le membres du jury. Tout d'abord mes remerciements vont à Mme Françoise Baude et M. David Hill pour avoir accepté de rapporter mes travaux de thèse. Je remercie également M. Julien Ponge pour avoir examiné mes travaux, M. Benoît Jacquemin pour avoir accepté de faire partie des membres du jury, Mme Dominique Rieu pour avoir présidé le jury de thèse.

Je tiens à remercier aussi mon directeur de thèse M. Philippe Lalanda pour m'avoir accueilli dans l'équipe ADELE et Mme Stéphanie Chollet, co-directrice, pour leur disponibilité, leur confiance, leur soutien, leurs conseils et leur aide tout au long de ce travail. J'ajoute que Stéphanie Chollet a enrichi mes travaux de thèse grâce à ses innombrables idées et discussions. Mes remerciements chaleureux vont à tous les membres de l'équipe ADELE.

Ma pensée la plus émue est adressée à la mémoire du Dr Ronald Schoop, qui a été mon mentor et celui qui m'a permis de démarrer ces travaux de thèse.

J'adresse enfin une pensée chaleureuse aux membres de ma famille, mes parents et mon frère. Mes remerciements les plus chalereux vont à ma femme Claire, qui m'a accompagné, aidé, soutenu et aussi beaucoup encouragé avec Martin tout au long de ces trois années de thèse.

Résumé

CETTE thèse se situe dans le domaine de l'informatique orientée service. Elle propose un ESB (Enterprise Service Bus) autonomique, c'est-à-dire capable de s'autogérer sur un ensemble d'aspects. Cet ESB, nommé Cilia autonomique, permet de construire des applications de médiation sensibles au contexte.

Précisément, la version autonomique de Cilia, que nous proposons, permet l'optimisation de l'utilisation des ressources de la plate-forme d'exécution et l'adaptation dynamique des chaînes de médiation au niveau de la configuration et de la topologie. Notre framework permet également de présenter à tout moment un modèle simplifié des phénomènes liés à l'exécution des chaînes et, ainsi, de faciliter le raisonnement et la prise de décisions d ?adaptation.

Les travaux de cette thèse ont été validés dans le cadre de l'informatique pervasive. En particulier, Cilia autonomique a été utilisé et instrumenté pour la mise en oeuvre d'applications de maintien à domicile et de suivi de la santé des usagers. Les résultats sont disponibles en open source.

Abstract

THIS PhD work takes place within a broader context of service-oriented computing. Precisely, it defines an autonomic Enterprise Service Bus (ESB) with self-management ability regarding certain aspects. This ESB, named autonomic Cilia, allows the simplified development of context-aware mediation applications.

Autonomic Cilia permits the runtime optimization of resources used by its execution machine and the dynamic adaptation of mediation chains, regarding configuration and topology. Our framework can also retrieve at any time a simplified model of the mediation chains execution. Such models enable reasoning and decision making easier to implement for mediation developers.

The work presented in this thesis has been validated in the context of pervasive computing. In particular, autonomic Cilia has been used and instrumented to implement healthcare applications in smart homes. The autonomic Cilia framework is available in open source.

Liste des publications

Les travaux de cette thèse ont été présentés précédement :

Article de revue :

S. Chollet, V. Lestideau, D. Morand, Y. Maurel, P. Lalanda. *Auto-réparation et auto-optimisation des applications pervasives - Un gestionnaire de sélection de dépendances de services basé sur l'Analyse de Concepts Formels*.TSI Informatique autonomique 33, 1-2 (2014) 7-30.

Conférences internationales :

I. Garcia, D. Morand, B. Debbabi, P. Lalanda, P. Bourret. *A Reflective Framework for Mediation Applications*. 12th International Middleware Conference. ACM Proceedings of the 10th International Workshop on Adaptive and Reflective Middleware, 2011, pp. 22-28.

D. Morand, I. Garcia, P. Lalanda. *Autonomic Enterprise Service Bus*. 6th International Workshop on Service Oriented Architectures in Converging Networked Environments. Proceedings of the 2011 IEEE 16th Conference on Emerging Technologies and Factory Automation, 2011, pp. 1-8.

D. Morand, I. Garcia, P. Lalanda. *Towards Autonomic Enterprise Service Bus*. ACM, Proceedings of the 1st Workshop on Middleware and Architectures for Autonomic and Sustainable Computing, 2011, pp. 19-23.

Table des matières

1 Introduction **15**

1 De nouveaux défis 16

1.1 La notion de services intégrés 16

1.2 La notion de services logiciel et d'*Enterprise Service Bus* 17

2 Objectifs de cette thèse 20

3 Structure du document 20

I État de l'art **23**

2 L'approche orientée service et son utilisation **25**

1 La notion de services logiciels 26

1.1 Définitions 26

1.2 Contrat de services 28

2 Architecture à services 29

2.1 SOC : *Service-Oriented Computing* 29

2.2 SOA : Service-Oriented Architecture 31

2.3 Besoins 33

2.4 SOC Dynamique 34

3 Composition de service 35

3.1 Principes 35

3.2 Composition par procédés 36

3.3 Composition Structurelle 37
4 Composants orientés services 39
4.1 Composants 39
4.2 Approche à composants orientés service 41
5 Technologies à services 43
5.1 Caractérisation 43
5.2 Historique : CORBA et Jini 43
5.3 Services Web 46
5.4 UPnP et DPWS 51
5.5 OSGi™et Apache Felix iPOJO 56
5.6 Bilan 61
5.7 Problèmes ouverts 62

3 Intégration des services 63

1 Introduction 64
1.1 Intégration des applications 64
1.2 La médiation 65
1.3 Opérations de médiation 67
1.4 Patron de médiation 67
2 Enterprise Service Bus 72
2.1 Enterprise Application Integration 72
2.2 Définitions 73
2.3 Implantation 76
2.4 Patrons de fédération 77
2.5 Synthèse 79
3 Solutions existantes 80
3.1 Spring Integration 80
3.2 ServiceMix 82
3.3 Mule 84
3.4 Cilia 86
3.5 Comparaisons 93
4 Synthèse 94

4 Informatique autonomique 99

1 Définitions 100
1.1 Les prémisses de l'informatique autonomique 100
1.2 Le manifeste d'IBM 100
1.3 Définitions de l'informatique autonomique 101
1.4 Niveaux de maturité d'un système autonomique 102
2 Inspirations 104
2.1 La biologie 105
2.2 Théorie du contrôle 105
2.3 Autres sources d'inspiration 107
3 Les propriétés auto-* 109
4 Architecture 112
4.1 Présentation 112
4.2 L'architecture générale 113
4.3 Les politiques 116
4.4 Architectures d'éléments autonomiques 117
5 Besoins 119
5.1 L'observation 120
5.2 L'adaptation 123
6 Conclusion 125

II Contribution **127**

5 Proposition **129**

1 Problématique 130
2 Objectifs 131
3 Proposition 133
3.1 Présentation générale 133
3.2 Buts d'administration 135
3.3 Adaptation autonomique des chaînes de médiation 135
3.4 Base de connaissances 137
4 Conclusion 138

6 Implantation et réalisation **139**

1 Rappels 140

1.1 Approche 140
1.2 Apache Felix iPOJO 141
2 Adaptateurs autonomiques 142
2.1 Principes 142
2.2 Extension du langage 143
2.3 Modification de la machine d'exécution de Cilia 143
3 Boucle de contrôle globale 148
3.1 Les *touchpoints* : capteurs et effecteurs 148
3.2 Architecture réflexive 149
4 La surveillance dynamique 154
4.1 Principes 154
4.2 Réalisation 155
4.3 Remontée des données 157
4.4 Audit et événements 159
5 Adaptation dynamique 160
5.1 Création et destruction de composants 160
5.2 Modification topologique 161
5.3 Gestion de la conservation des états 162
6 La base de connaissances 164
6.1 Découverte des chaînes en éxécution 165
6.2 Evénements 166
6.3 Configuration de la base de connaissances 167
6.4 Résumé 168
7 Conclusion 168

7 Validation et évaluation 169

1 Projet MEDICAL 170
2 Cas d'utilisation *Actimétrie* 171
2.1 Description 171
2.2 Module d'accès aux équipements 173
2.3 Module de génération des mesures 174
3 Expérimentation 175
3.1 Plate-forme de validation 175
3.2 Conditions initiales 177

3.3 Cas d'utilisation 1 : Maintien de la qualité de service 178
3.4 Cas d'utilisation 2 : Modification de fonctionnalité 183
4 Evaluation 189
4.1 Description de l'évaluation 189
4.2 Mesure de l'impact du *monitoring* 191
4.3 Mesure de l'impact de la reconfiguration 192
5 Synthèse 192

8 Conclusion et perspectives **195**

1 Conclusion 196
1.1 Contexte 196
1.2 Besoins 196
1.3 Contribution 197
2 Perspectives 198
2.1 Extension des éléments administrés 198
2.2 Auto-gestion de fédération de *framework* Cilia 199
2.3 Couplage avec un répertoire de dépôt 200
2.4 La base de connaissances exposée sous la forme de service 201

Bibliographie **203**

TABLE DES FIGURES

1.1 Infractructure générale. 17

2.1 Patron d'interaction dans l'architecture orientée service. 29
2.2 Mécanismes d'un environnement d'exécution et d'intégration de service. 31
2.3 Architecture orientée service étendue. 33
2.4 Patron d'interaction de l'approche orientée à services dynamique. 34
2.5 Orchestration de services. 36
2.6 Chorégraphie de services. 37
2.7 Composition structurelle. 38
2.8 Composite SCA. 39
2.9 Configuration. 40
2.10 Composant conteneur d'aspects non-fonctionnels. 41
2.11 Modèle de composant orienté service. 42
2.12 Gestionnaire d'instances et registre de services. 42
2.13 CORBA schéma général. 44
2.14 Mécanismes de base et non-fonctionnels pour les services Web. 47
2.15 Liens WSDL et description UDDI. 49
2.16 Réseau UPnP. 52
2.17 Dispositif DPWS. 54
2.18 Réseau DPWS. 55
2.19 Couches OSGi (source [OSGi Alliance, 2007]). 56
2.20 Approche orientée service dynamique OSGi. 58

2.21 OSGi plate-forme d'accueil de iPOJO. 58
2.22 Composant iPOJO. 59
2.23 Assemblage de composants iPOJO. 60
2.24 Interactions entre acteurs. 62
2.25 Producteur et consommateur sont découplés. 62

3.1 Médiation de messages. 66
3.2 Médiation de services. 66
3.3 *Proxy* de services. 68
3.4 Intermédiaires SOAP. 68
3.5 Plate-forme d'exécution. 70
3.6 Architecture de CORBA. 70
3.7 Architecture *hub-and-spoke*. 72
3.8 Architecture ESB. 74
3.9 Approche par composition d'applications. 77
3.10 Fédération d'ESB, style direct. 78
3.11 Fédération d'ESB, style centralisé. 79
3.12 Eléments de *Spring Integration*. 80
3.13 Environnement JBI. 82
3.14 Architecture de Mule. 84
3.15 La plate-forme Cilia. 87
3.16 Cilia, vision générale. 87
3.17 Composant médiateur, selon [Garcia Garza, 2012]. 88
3.18 Couches d'implantation de Cilia. 90
3.19 Réalisation du système réflectif de Cilia. 91
3.20 Cycle de vie d'une application Cilia. 92
3.21 Phases de la conception d'une application Cilia. 92

4.1 Augmentation des fonctionnalités autonomiques. 103
4.2 Niveau de maturité de l'informatique autonomique dans les entreprises. 103
4.3 Sources d'inspiration de l'informatique autonomique. 104
4.4 Système ultra-stable de W. R. Ashby. 106
4.5 Contre réaction. 106
4.6 Arbre des propriétés autonomiques, selon [Sterritt and Bustard, 2003]. 111
4.7 Boucle de contrôle. 112

4.8 Architecture de référence selon IBM. 114
4.9 Architecture distribuée. 117
4.10 Architecture centralisée. 118
4.11 Architecture hiérarchisée. 119

5.1 Vision globale de Cilia autonomique. 133
5.2 Spécification des propriétés autonomiques de Cilia. 134
5.3 Interactions entre boucles de contrôle. 135
5.4 Base de connaissances. 137

6.1 Vision globale de Cilia autonomique. 140
6.2 Composant iPOJO. 141
6.3 RoSe : apparition et disparition d'un proxy. 144
6.4 Adaptateur avec le *framework* RoSe. 145
6.5 L'interface *CiliaContext*. 148
6.6 Architecture réflexive de Cilia. 150
6.7 Bloc fonctionnel du `mediatorManager` et du *monitoring*. 155
6.8 Handlers iPOJO du `mediatorManager`. 156
6.9 Topologie de la chaîne simple. 161
6.10 Topologie de la chaîne simple modifiée. 161
6.11 Gestion des messages dans le `scheduler` Cilia. 163
6.12 Vue générale de la base de connaissances. 168

7.1 Application *Actimétrie*. 170
7.2 Déploiement de l'application *Actimétrie*. 172
7.3 Architecture fonctionnelle du service *Actimétrie*. 173
7.4 Chaîne de médiation *Actimétrie*. 174
7.5 Plate-forme d'*Actimétrie*. 176
7.6 État initial de l'application simulée. 178
7.7 Etat final du cas d'utilisation *Maintien de la QoS*. 182
7.8 Etat initial de la chaîne de médiation obtenue dans le visualisateur. 184
7.9 Phase d'activités du gestionnaire *DevicesManagerPolicies*. 184
7.10 Chaîne de médiation *Actimétrie* » avec photomètre. 187
7.11 Etat initial, cas d'usage amélioration de l'application. 188
7.12 Contrôle de la modification. 189
7.13 Application de mesure du temps de latence. 191

7.14 Chaîne monitorée versus chaîne non monitorée. 191

8.1 Système autonomique. 199

LISTE DES TABLEAUX

2.1 Caractéristiques de Jini et de Corba. 46
2.2 Caractéristiques des services Web. 51
2.3 Caractéristiques de UPnP et de DPWS. 56
2.4 Caractéristiques de OSGi et de iPOJO. 60

3.1 Conception et déploiement. 95
3.2 Assemblage des composants. 95
3.3 Exploitation. 96

4.1 Projets qui ont influencés l'auto-administration, selon [Huebscher and McCann, 2008]. 108

6.1 Variables d'état pour les appels système. 154
6.2 Variables d'état de l'environnement d'exécution. 154
6.3 Variables d'états pour le code métier. 155
6.4 Mots clés pour un filtrage du temps. 158
6.5 Mots clés pour un filtrage de la valeur. 158

7.1 Temps de reconfiguration par opération. 192

Liste des codes sources

6.1 Structure du Cilia DSL (extrait). 142
6.2 Configuration du comportement dynamique d'une instance d'adaptateur (extrait). 143
6.3 Configuration du *framework* RoSe (extrait). 145
6.4 Configuration d'un adapteur-filtre (extrait). 146
6.5 Configuration d'un adapteur-cardinalités (extrait). 147
6.6 Configuration d'un adapteur (extrait). 147
6.7 Interface *CiliaContext* (extrait). 149
6.8 Modèle d'une chaîne de médiation (extrait). 151
6.9 Modèle d'un médiateur (extrait). 151
6.10 Contrôleur de médiateur (extrait). 153
6.11 Interface `IMonitor`. 156
6.12 Contrôleur de médiateur (extrait). 156
6.13 MediatorManager configuration (extrait). 157
6.14 `MonitorHandlerStateVar` (extrait). 158
6.15 Audit de variables du code métier (extrait). 159
6.16 Génération d'événements à partir du code métier (extrait). 160
6.17 Création de la chaîne simple (extrait). 161
6.18 Modification de topologie de la chaîne simple (extrait). 161
6.19 Interface fournie par le service `DataPersistency`. 162
6.20 Interface fournie par le service `DataPersistency`. 163
6.21 `Builder` - Remplacement d'un médiateur (extrait). 164
6.22 Interface `Topology` (extrait). 165
6.23 Base de connaissances (extrait). 167
7.1 DSL Cilia *Actimétrie* 175
7.2 Fonction périodique (extrait). 179
7.3 Enregistrement des événements nœuds (extrait). 180
7.4 Méthode *callback* d'un nœud (extrait). 181
7.5 Trace de l'activité du cas d'usage *Maintien de la QoS* (extrait). 182
7.6 Phase 1 (extrait). 185
7.7 Phase 2 (extrait). 185
7.8 Phase 3 (extrait). 187

1

Introduction

Dans ce premier chapitre, nous présentons le contexte de notre travail, ainsi que les objectifs précis que nous avons poursuivis. Nous détaillons également la structure de ce document de thèse. En particulier, nous montrons que les développements informatiques ont considérablement évolué ces dernières années et qu'ils reposent de plus en plus sur l'intégration de ressources diverses. De nouvelles technologies, les services, et de nouveaux outils, les ESB (*Enterprise Service Bus*), sont apparus pour satisfaire ces besoins. Ils demeurent néanmoins imparfaits et soulèvent de réelles questions, notamment au niveau de l'adaptation et de l'administration.

1 De nouveaux défis

1.1 La notion de services intégrés

Nous sommes entrés dans un monde où l'informatique joue un rôle sociétal majeur. De plus en plus de services sont proposés via une multitude d'équipements électroniques tels que des ordinateurs, des téléphones portables, des tablettes, etc. Ces services reposent sur des ressources informatiques distribuées et variées qu'il convient d'intégrer. Ces ressources incluent classiquement des bases de données, des applications, mais aussi, des capteurs présents dans les environnements de vie et permettant d'acquérir des informations riches et variées de manière automatique et transparente.

Ces nouveaux types de services suscitent dès aujourd'hui de grandes attentes dans des domaines aussi variés que la santé, la domotique, l'énergie, le transport, etc. et ce, malgré un manque évident de maturité et donc, en particulier, de fiabilité et de sécurité.

En effet, la mise en place de ces services soulève de nombreux problèmes. Il convient de gérer des ressources hétérogènes, dynamiques, dont les services ne sont pas les propriétaires (et qui ont donc des cycles de vie indépendants). Il est également souvent nécessaire de gérer des quantités de données importantes. Eric SCHMIDT, CEO de Google, estimait en 2005 que la masse de données disponible électroniquement atteignait le chiffre étourdissant de 5 millions de terabytes (seulement 0,004% étant indexées par Google). Il estimait également que ce chiffre doublait tous les 5 ans.

Les projets de services numériques innovants et à forte valeur ajoutée se multiplient donc avec des résultats très variables. Les échecs sont, à notre avis, liés à la difficulté de clairement définir les attentes et les besoins des usagers, mais aussi, à la complexité technique afférente à ces services.

En guise d'illustration de ces tendances, prenons l'exemple très important aujourd'hui de l'efficacité énergétique. Rappelons qu'en France, le transport et la distribution électrique sont structurés suivant trois niveaux : le réseau de transport HTB [1], le réseau de répartition HTA [2] et le réseau de distribution BT [3].

La HTB est produite par des centrales et alimente des usines ou des installations fortement consommatrices. La HTA s'obtient par transformation de la HTB ou est produite par des dispositifs spécifiques tels que les éoliennes. Elle alimente typiquement des immeubles. La BT, enfin, résulte de la transformation de la HTA et de petits dispositifs tels que des éléments photovoltaïques et alimente les maisons ou les appartements.

A tous les niveaux de cette chaîne, des services nouveaux permettant une meilleure efficacité énergétique sont élaborés, testés, mis en place. Au niveau HTA, nous trouvons, par exemple, les usines intelligentes et les bâtiments intelligents qui présentent, entre autres, une haute efficacité énergétique et une gestion intelligente des appareils producteurs et consommateurs d'électricité. Au niveau BT, les maisons individuelles et les appartements sont et seront de plus en plus équipés d'une multitude d'équipements électroniques dont la consommation devra être gérée intelligemment en fonction du contexte plus global.

1. Acronyme de Haute Tension B. La tension excède 50 000 volts en courant alternatif ou excède 75 000 volts en courant continu.
2. Acronyme de Haute Tension A. La tension excède 1 000 volts sans dépasser 50 000 volts en courant alternatif ou bien excède 1 500 volts sans dépasser 75 000 volts en courant continu.
3. Acronyme de Basse Tension. La tension excède 50 volts sans dépasser 1 000 volts en courant alternatif ou bien excède 120 volts sans dépasser 1 500 volts en courant continu.

Tous ces services reposent sur l'utilisation des nouvelles technologies de l'information, y compris l'Internet, ce qui n'est pas sans poser d'inquiétants problèmes de sécurité et sur la communication rapide entre les différents éléments de l'infrastructure électrique (représentée sur la figure 1.1 « *Infractructure générale* »). Cela demande ainsi d'intégrer une multitude de ressources qui ne communiquaient pas auparavant et qui sont présentées aujourd'hui sous forme de services Web par exemple (nous reviendrons sur ces aspects techniques plus avant dans cette thèse).

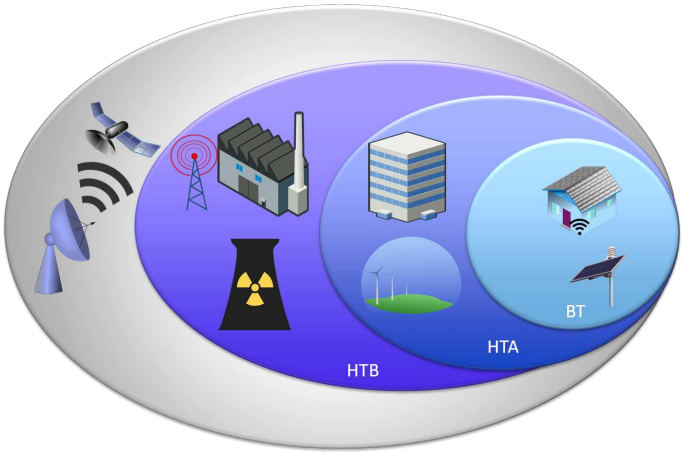


FIGURE 1.1 – Infractructure générale.

Il est ainsi nécessaire d'acquérir des informations riches et variées de manière transparente. Les informations sont issues de dispositifs fixes ou bien mobiles. Parmi les capteurs fixes, nous pouvons citer des dispositifs de mesures des caractéristiques électriques à différents points du réseau, des capteurs d'états de certains dispositifs, des sondes de surveillance avancées, etc. Ces capteurs intelligents communiquent via des supports variés incluant le Wifi, Ethernet, CPL[4], etc. Les dispositifs mobiles sont, par exemple, des ordinateurs, des téléphones portables, des tablettes. Ils permettent d'acquérir des informations et de commander des dispositifs, de piloter soi-même à distance ou non les différents équipements.

De nombreux systèmes existent ou sont en cours de développement ; citons, par exemple, le projet Home de Schneider-Electric. Il s'agit pour ce projet de concevoir des systèmes du bâtiment permettant de réduire les coûts énergétiques. Plusieurs axes de recherche ont été retenus dans ce projet, dont :

- l'adaptation du fonctionnement des équipements à la présence et à l'activité des occupants ;
- l'optimisation de l'approvisionnement énergétique (priorisation des énergies renouvelables) ;
- l'intégration des éléments physiques et des systèmes d'information qui peuvent être lourds et peu flexibles ; ce besoin est présent dans tous les projets liés aux nouveaux batiments. Ces éléments sont distants, de nature différente et, bien sûr, caractérisés par des exigences très différentes.

1.2 La notion de services logiciel et d'*Enterprise Service Bus*

L'intégration de ressources se fait aujourd'hui fréquemment à la l'aide de services logiciels. Ces derniers ont profondément modifié le monde informatique. Lorsque des sociétés ont com-

4. Acronyme de Courant Porteur en Ligne.

mencé à livrer des applications avec une infrastructure, l'objectif était de fournir des solutions métiers pouvant être assemblées et utilisées par un client en suivant le principe d'abonnement.

Conceptuellement, un service logiciel est une fonctionnalité proposée par un fournisseur et offerte à un client. Cette fonctionnalité peut couvrir un vaste spectre d'utilisation qui peut aller de la simple sauvegarde de données à des services plus complexes liés à des processus métiers. Signalons un point important concernant les services : le client l'utilise sans avoir le besoin de le créer et de le détruire au sens informatique du terme ; c'est-à-dire gérer son cycle de vie.

Un service informatique est un composant entièrement décrit par des méta-données. Cette description est publiée et, bien évidemment, comprise par des programmes informatiques. Pour terminer, seule la description du service est nécessaire au client pour qu'il puisse utiliser le service. Un service grâce à sa description rend le consommateur (le client) agnostique de la technologie sous-jacente à son implantation. Par conséquent, le service ne connaît pas le contexte dans lequel il va être utilisé. Pour terminer, le **couplage** entre le fournisseur et le consommateur est faible.

*Par **couplage**, nous entendons le degré de dépendance qu'ont mutuellement deux systèmes. Il existe plusieurs degrés de couplage. Cela va de l'absence de lien entre deux systèmes à une interdépendance complète.*

L'utilisation de services, si elle apporte de nombreux avantages qui seront détaillés par la suite, ne permet pas de résoudre tous les problèmes d'interopérabilité (et donc d'intégration). D'une part, il existe de nombreuses technologies à services utilisant des descriptions et des protocoles de communication différents. D'autre part, rien ne permet de résoudre les divergences syntaxiques et sémantiques entre applications développées par différentes personnes et/ou à différents moments.

Ainsi, de nombreuses entreprises ont placé leurs espoirs dans les nouvelles solutions de type *Enterprise Service Bus* (ESB), qui sont apparues au début des années 2000. Les ESB reposent sur la notion de services logiciels qui permettent d'exposer les différentes ressources d'une infrastructure logicielle de façon homogène.

Les ESB se positionnent, comme les EAI (*Enterprise Application Integration*), d'un point de vue architectural. Ils se situent entre clients et fournisseurs de services et ont pour rôle d'effectuer les opérations de médiation fonctionnelles et non-fonctionnelles. Ils sont néanmoins caractérisés par des exigences quelque peu différentes de celles des EAI. En particulier, un ensemble d'exigences très structurant sur les ESB peut se résumer comme suit. Les ESB doivent être :

- **légers**. C'est certainement l'une des exigences les plus récurrentes. Les entreprises souhaitent autant que possible s'éloigner du syndrome EAI qui va dans le sens d'une gestion centralisée et « lourde » de l'intégration d'applications autour d'un modèle de données pivot. L'intégration d'applications doit plutôt être gérée « à la demande » dans le périmètre des équipes projet.
- **efficaces**. C'est une propriété importante pour tout logiciel dédié à l'intégration. En effet, la qualité du service intégré ne doit pas se dégrader.
- **faciles à installer et à gérer**. Comme les ESB doivent pouvoir être utilisés « à la demande » lors de la création d'un nouveau service intégré, leur installation, leur configuration et leur administration doivent être simples.

- **flexibles**. Les services évoluent dans le temps, d'autant plus lorsqu'ils intègrent des applications indépendantes. De fait, il est important de faciliter l'ajout ou la modification d'opérations de médiation à l'exécution dans le but d'adapter la façon dont les applications s'intègrent.
- **ouverts et faciles à utiliser**. Le modèle de développement et le modèle d'exécution doivent être simples. Le modèle de développement doit en plus être « ouvert ». Une fois encore, il est important d'éviter le retour à des solutions complexes et opaques pour lesquelles le développement et la maintenance de code rendent nécessaires des interventions très souvent onéreuses d'équipes externes spécialisées.
- **adaptés** à la gestion des erreurs. Il s'agit d'une exigence de première importance, d'autant plus que l'intégration d'applications renvoie à des enchaînements d'opérations de médiation impliquant des services distants. De nombreuses erreurs peuvent ainsi se produire au niveau des chaînes de médiation elles-mêmes ou bien lors de la communication avec les services distants (absence de réponse, réponses incorrectes ou partielles...). Une partie importante du code d'intégration est ainsi dédiée à la gestion des erreurs.

On voit que les ESB sont confrontés à des défis majeurs en termes d'exigences. Et ce n'est pas tout ! De plus en plus, afin d'aborder des domaines comme les services pervasifs, les ESB sont amenés à gérer :

- **l'hétérogénéité des ressources**. Les ressources viennent avec des protocoles de communication de plus en plus variés, des langages de programmation distincts et des modèles de données sémantiquement différents, qu'il convient d'assembler au sein d'une même application. Les services de demain seront amenés à inter-opérer avec un environnement de plus en plus riche en capteurs et d'actionneurs.
- **la dynamique des environnements**. Il s'agit d'un défi auquel sont confrontés les concepteurs d'applications. Il convient, en effet, d'adapter les services offerts à l'utilisateur en fonction de son activité et de sa localisation, des capacités et disponibilités des équipements, des caractéristiques ponctuelles de l'espace environnant, etc. L'adéquation entre ressources requises et fournies doit être établie et rétablie dynamiquement selon l'évolution des entités disponibles.
- **les besoins en autonomie**. L'autonomie des applications est un enjeu majeur. Ces applications doivent exhiber des propriétés d'auto-adaptation à leur contexte d'exécution (capacités matérielles, logiciels disponibles, connectivité, présence d'autres équipements, localisation, activité de l'utilisateur...) et de sûreté d'exécution (sécurité, autoréparation).

Les outils et méthodes répondant aux problèmes présentés ci-dessus de façon générique n'existent pas vraiment. Il est dès lors souvent nécessaire de fournir des approches limitées, parfois complètement ad hoc, pour outiller la mise en place et l'administration des services numériques.

Nous terminons cette section par les définitions de l'informatique **uniquitaire** et de l'informatique **pervasive**. M. WEISER a décrit le terme informatique ubiquitaire [Weiser, 1991]. Sa vision correspond à ce que l'on appelle aujourd'hui l'informatique ubiquitaire et pervasive. Dans la suite de ce document, nous utiliserons les termes ubiquitaire et pervasif selon les définitions suivantes :

Informatique ubiquitaire : *environnement où les équipements informatiques sont omniprésents.*

Informatique pervasive : *équipements informatique où les équipements informatiques sont omniprésents et entièrement intégrés.*

2 Objectifs de cette thèse

Cette thèse se situe dans le cadre de l'informatique orientée service et se concentre sur la notion de *Enterprise Service Bus* (ESB). Elle se fonde, plus particulièrement, sur l'ESB **Cilia**, un *framework open source* [5] développé par l'équipe **Adèle** du **Laboratoire d'Informatique de Grenoble**. Il offre des environnements de spécification et d'exécution spécialisés pour la médiation de données et de services.

Cilia se distingue par sa dynamicité, sa modularité et par le fait qu'il soit embarquable. Il reste simple à utiliser et à déployer grâce à l'emploi de DSL (*Domain-Specific Language*) permettant l'implantation directe des EIP [6]. Cilia est développé au-dessus de **Java**, **OSGi**™ et **Apache Felix iPOJO**.

3 Structure du document

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

- le **chapitre 2** présente l'informatique orientée service. Après un certain nombre de définitions, nous présentons les technologies essentielles d'aujourd'hui. Nous montrons ensuite que la mise en œuvre de cette approche est variée en présentant différentes plates-formes : services Web, OSGi™, SCA puis Apache Felix iPOJO. Nous développons également les besoins en intégration liés aux approches à service.
- le **chapitre 3** traite de la problématique d'intégration en informatique. Après un bref positionnement historique, nous décrivons les principaux patrons d'intégration usuellement utilisés aujourd'hui. Nous présentons également les approches et outils existants pour l'intégration de services logiciels, notamment au niveau des ESB. Puis, nous détaillons l'ESB *open source* Cilia, à la base de notre travail.
- le **chapitre 4** traite de l'informatique autonomique. Nous débutons par une définition générale du domaine, puis continuons avec la description des propriétés d'auto-adaptation sous-tendues par cette approche. Avant de conclure, nous présentons l'architecture de référence adoptée dans ce domaine.

5. Libre de droit.
6. Acronyme anglais de *Enterprise Application Integration*. Famille de patrons pour la transformation des données.

La contribution se divise, quant à elle, en trois parties :

- le **chapitre 5** expose la problématique, les objectifs et donne une vision d'ensemble de notre approche. Nous présentons les extensions apportées à Cilia et, plus précisément, l'architecture d'auto-gestion mise en place. Nous introduisons également la notion de variable d'état, qui nous permet de suivre et d'évaluer le fonctionnement d'une chaîne de médiation.
- le **chapitre 6** détaille l'implantation de notre approche. Nous montrons notamment le fonctionnement général ainsi que les détails techniques de notre implantation. En particulier, nous détaillons les différentes boucles de contrôle introduites dans Cilia, ainsi que les techniques utilisées.
- le **chapitre 7** illustre l'utilisation de notre *framework* par un cas d'application : une application de maintien de personnes à domicile. Nous donnons, notamment, deux exemples complets d'auto-administration. Nous construisons un gestionnaire autonomique qui met en valeur les nouvelles propriétés de l'ESB Cilia.

Enfin, le **chapitre 8** synthétise les idées principales de notre proposition. Nous rappelons les points principaux de notre contribution et nous décrivons pour conclure les perspectives envisagées pour de futurs travaux.

Première partie

État de l'art

2

L'approche orientée service et son utilisation

Dans la première partie, nous allons présenter l'architecture logicielle la plus aboutie de nos jours : l'approche à services. Après avoir introduit les services logiciels, nous étudierons les acteurs et les interactions qui en découlent. Dans une deuxième partie, nous explorerons les besoins technologiques pour réaliser l'approche à services. Les services logiciels peuvant être composés, nous présenterons, dans la troisième partie, les deux styles de composition pour des services logiciels qui sont la composition par procédés et la composition structurelle. Dans la quatrième partie, nous rappellerons les principes de l'approche à composants pour mieux introduire l'approche à composants orientés services. Dans la cinquième partie, nous étudierons les technologies mettant en œuvre les principes de l'approche à services. Nous détaillerons les technologies comme les services Web, OSGi™ et Apache Felix iPOJO. Avant de conclure, nous ferons une comparaison des différentes technologies présentées.

1 La notion de services logiciels

1.1 Définitions

Un service est proposé par un **fournisseur** dans le but de répondre à un besoin d'un **consommateur**. Une capacité d'un fournisseur est mise à la disposition de consommateurs. Dans le cadre de services informatiques, la définition proposée par M. P. Papazoglou est la plus souvent citée :

> « *Services are self-describing, platform agnostic computational element.* » [Papazoglou, 2003]

Cette définition met en avant deux propriétés fondamentales d'un service :
- seule la connaissance de la description est nécessaire et suffisante pour qu'un consommateur puisse l'utiliser ;
- le service ne connaît ni sa plate-forme d'exécution ni le contexte client dans lequel il va être utilisé.

La description du service est la clé de la réussite des services informatiques. Ils réduisent significativement le nombre de données échangées entre les deux parties (le consommateur et le fournisseur). La description du service est suffisante pour qu'un consommateur puisse l'utiliser ; le fournisseur et le consommateur n'échangeront ainsi aucune information relative à la plate-forme d'exécution, à l'implantation et à la réalisation de leur intégration.

Le consortium OASIS [1] propose une toute autre définition des services :

> « *A service is a mechanism to enable access to one or more capabilities, where the access is provided by using a prescribed interface and is exercised consistent with constraints and policies as specified by the service composition. A service is accessed by means of a service interface where the interface comprises the specifics of how to access the underlying capabilities. There are no constraints on what constitutes the underlying capability or how access is implemented by the service provider. A service is opaque in that its implementation is typically hidden from the service consumer for (1) the information and behaviour models exposed through the service interface and (2) the information required by service consumers to determine whether a given service is appropriate for their needs* » [OASIS, 2006a]

Cette définition complète la précédente. Elle s'attache à définir le rôle de la description du service : c'est une **interface**, qui fournit la **spécification** du service à laquelle le service doit se **conformer**. La description du service doit être indépendante de l'implantation et suffisamment précise pour que le consommateur puisse déterminer si le service correspond à son besoin. Elle explicite la politique d'accès (qui peut être une politique de restriction) et doit être autonome ; c'est-à-dire sans lien avec d'autres interfaces. Pour le consortium OASIS, un service est soit local soit distant ; il peut donc être exécuté sur une plate-forme distante ou importé sur une plate-forme locale.

1. Acronyme de *Organization for the Advancement of Structured Information Standards*, `http://www.oasis-open.org/home/index.php`. OASIS est en charge du développement, de la convergence et de l'adoption de standard utilisant la technologie XML.

Ces précédentes définitions nous amène à comprendre qu'un service n'a pas d'état, il doit fournir tous les paramètres nécessaires à son exécution. Introduisons maintenant la définition de A. ARSANJANI qui liste les interactions entre les différentes parties :

> *« A service is a software resource (discoverable) with an externalized service description. This service description is available for searching, binding, and invocation by a service consumer. Services should be ideally be governed by declarative policies and thus support a dynamically reconfigurable architectural style. »* [Arsanjani, 2004]

Nous découvrons les trois interactions fondamentales :
- la première qui consiste à **découvrir les descriptions** des services et à **identifier** le service qui correspond aux besoins ;
- la seconde qui est **l'établissement de la liaison** avec le service choisi ;
- la troisième qui est l'**invocation** du service.

Ces interactions entre le fournisseur de service et le consommateur permettent de construire des applications flexibles. En effet, modifier le critère de sélection permet de changer le service qui sera invoqué. Un service peut apparaître ou disparaître et sa description peut être ajoutée et/ou retirée. Par conséquent, les architectures des applications clientes peuvent évoluer d'une exécution à une autre. Les applications ainsi construites sont donc flexibles et agiles pour suivre des évolutions fonctionnelles.

M. P. PAPAZOGLOU et W. HEUVEL définissent un service comme :

> *« [. . .] services, which are well defined, self-contained modules that provide standard business functionality and are independent of the state or context of other services. »* [Papazoglou and Heuvel, 2007]

Cette définition précise le service informatique. Un service fournit une fonctionnalité **métier**. Il est **défini**, **auto-contenu** et **indépendant** de l'état des autres services.

Sur la base de ces définitions précédentes, nous concluons avec la définition suivante :

> *« Un service est une entité logicielle qui fournit un ensemble de fonctionnalités défini dans une description de service. Cette description comporte des informations sur la partie fonctionnelle du service mais aussi sur ses aspects non-fonctionnels. A partir de cette spécification, un consommateur de service peut rechercher un service qui correspond à ses besoins, le sélectionner et l'invoquer en respectant le contrat qui a été accepté par les deux parties »* [Chollet, 2009]

Cette définition ajoute la notion de **contrat**, résultat d'une négociation entre le client et le fournisseur du service. Le but du contrat est de prévenir les conflits avant l'exécution et de garantir les fonctionnalités d'exécution.

1.2 Contrat de services

Nous avons précédemment introduit la notion de contrat entre le service fournisseur et le service consommateur. Nous allons explorer cette notion en commençant par la définition de L. TOUSEAU :

> *« Un accord de niveau de service est un contrat entre au moins deux parties, dont les clauses portent sur la qualité du (ou des) service(s) faisant l'objet du contrat »* [Touseau, 2010]

En anglais, le *Service Level Agreement* (SLA) [Aiello et al., 2005] est un contrat de niveau de service, il permet une compréhension commune des deux parties, que sont le fournisseur et le consommateur. Il apporte des obligations de qualité du service pour le fournisseur et de résultat pour le consommateur. Le contrat de niveau de service porte également sur les modalités d'exécution et sur les responsabilités de chaque partie. Cet accord de service, accepté par le fournisseur et le consommateur, doit être passé avant toute collaboration entre les deux parties. Deux objectifs peuvent alors être atteints :

1. la suppression des conflits d'exécution ;
2. la garantie des fonctionnalités rendues par l'application.

La réalisation de ces objectifs requiert d'expliciter, de qualifier, de quantifier et de mesurer les contraintes et les attentes. Prenons un exemple : un accord entre un client et un serveur porte sur le délai minimum entre deux invocations de méthode. Pour être mesurable, ce délai doit être exprimé avec une unité (heures, minutes, secondes, millisecondes) et une précision (exprimée en pourcentage). Si l'accord n'est pas respecté, la politique de pénalité est appliquée (la requête est mise en file d'attente et elle n'est pas traitée immédiatement).

Nous avons précédemment présenté les objectifs généraux de contrat de niveau de service. Ils ont été classés en quatre niveaux [Beugnard et al., 1999] selon l'ordre suivant :

- **le niveau syntaxique** : il porte sur la signature des méthodes, ce que l'on retrouve généralement dans les interfaces des langages de programmation (interfaces Java) et dans les *Langages de Description des Interfaces*[2]. Les signatures des opérations correspondent aux noms des méthodes, avec leurs paramètres d'entrée et de sortie ainsi que les exceptions qui peuvent être levées durant l'exécution de ces opérations.
- **le niveau comportemental** : il enrichit le niveau précédent, auquel il apporte des conditions sur le comportement de chaque opération. Ces conditions sont de type booléennes. Elles sont appelées pré-conditions, post-conditions et invariants. Par exemple, UML[3] formalise les contraintes par le langage formel OCL[4].
- **le niveau synchronisation** : ce niveau de contrat s'attache à spécifier le comportement global entre appels de méthodes, en spécifiant la synchronisation. Plusieurs types d'appel peuvent être effectués avec ou sans contrainte, séquentiellement ou parallèlement.
- **le niveau qualité de service** : il repose sur les niveaux de contrat précédent. Il définit les accords de niveau de services en qualifiant et quantifiant la livraison d'un point de vue du service, du domaine d'application. Les deux parties s'entendent sur la qualité de livraison du service ainsi que les fonctionnalités exposées sous forme d'interfaces. Donnons deux exemples issus de domaines d'application différents :

2. En anglais, *Interface Description Language* (IDL).
3. Acronyme de *Unified Modeling language*.
4. Acronyme de *Object Constraint Language*.

1. pour les serveurs Web : on retrouve classiquement le nombre de connexions simultanées, le temps de réponse moyen et temps de réponse maximal ;
2. pour les communications réseaux : on définit généralement le temps de réponse, le débit, le temps de latence, le nombre d'erreurs de connexions maximum par unité de temps.

2 Architecture à services

2.1 SOC : *Service-Oriented Computing*

L'approche orientée service, en anglais *Service-Oriented Computing* (SOC), est un style architectural, au même titre que l'approche orientée objet ou l'approche orientée composant. Elle n'est pas une architecture de référence d'implantation, mais un modèle d'interactions entre trois acteurs : le **fournisseur** de services, le **consommateur** de services et le **courtier** de services.

Microsoft propose la définition suivante de l'approche orientée service :

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

Ce modèle d'interactions permet l'intégration des différentes fonctionnalités, qui sont exposées sous forme de service.

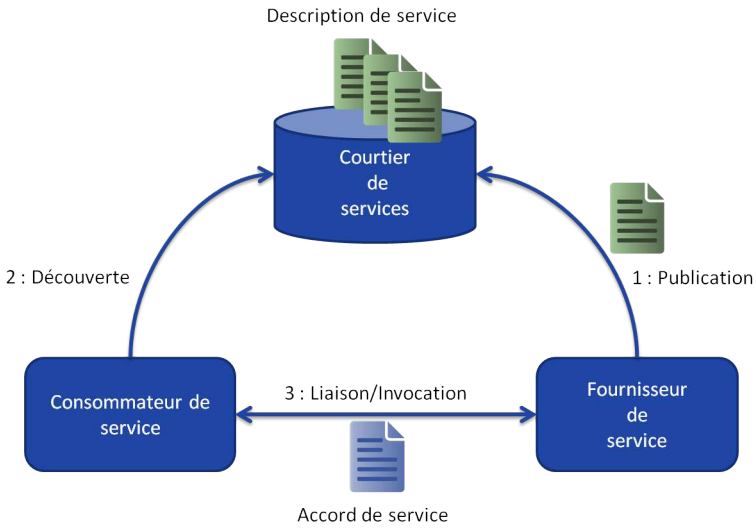


FIGURE 2.1 – Patron d'interaction dans l'architecture orientée service.

Comme illustré dans la figure 2.1 « *Patron d'interaction dans l'architecture orientée service* », l'approche orientée service est basée sur trois acteurs :

– **le fournisseur de services** : il représente une entité logicielle, une personne, un service, une organisation. Il décrit son service dans une **spécification de service**. Cette

description rassemble toutes les propriétés fonctionnelles nécessaires à sa sélection et, de manière facultative, des propriétés non-fonctionnelles telles que des métriques de coûts, de performance, de temps de réponse, d'attributs de sécurité, de disponibilité, tout autant que des propriétés liées à sa communication (sa localisation, ses protocoles de communication et ses restrictions). Le fournisseur de services met à disposition sa spécification de service.
- **le courtier de services** :il fournit les mécanismes pour publier, découvrir et enregistrer les spécifications de service. Il est aussi appelé **annuaire de services** ou encore registre de services.
- **le consommateur de services** : il interroge le courtier de services pour obtenir les services disponibles qui correspondent à son besoin. Après avoir sélectionné le service, un accord de service [Aiello et al., 2005] est réalisé entre le fournisseur et le consommateur. Après accord, le fournisseur peut effectuer la liaison et invoquer les fonctionnalités du service.

Ce style architectural est aussi un modèle d'interactions, qui comprend :
- **la publication de service** : le fournisseur de services enregistre sa description de service ainsi que des informations nécessaires à sa localisation ;
- **la découverte de service** : le consommateur de services découvre les services qui correspondent à ses besoins. Il interroge le courtier de services ;
- **la liaison et l'invocation d'un service** : ultime étape, après la sélection du service et après accord de service, le fournisseur peut effectuer la liaison et invoquer les fonctionnalités du service.

Nous avons décrit les acteurs et les interactions constituant l'architecture orientée service. Attachons nous maintenant à lister quelques caractéristiques ayant été à l'origine de la réussite (tant au niveau académique qu'industriel) de ce style architectural :
- **le masquage de la localisation du service** : le consommateur ne connaît pas à l'avance la localisation du service fournisseur. L'interaction est locale ou distante.
- **le faible couplage** : la seule information partagée entre le fournisseur et le consommateur est la **description de service**. Dans le cas des interactions distantes (au travers d'un réseau informatique), le faible couplage est dépendant du style de communication entre le consommateur et le fournisseur. Par exemple, l'utilisation d'appel de méthodes distantes de type RPC [5] est un couplage fort. Ce degré de couplage est principalement lié au fait que le consommateur doit connaître les signatures des méthodes du fournisseur. Une interaction par échange de messages ne fait pas perdre la propriété de faible couplage ; car le consommateur n'a pas la nécessité de connaître les signatures des méthodes du fournisseur.
- **la substituabilité** : ce style architectural offre la propriété de liaison tardive ; la liaison entre le consommateur et le fournisseur est réalisée au moment de l'exécution. De plus, un service est sans état. Il est tout à fait possible de remplacer entre deux invocations un fournisseur de service par un autre.
- **l'augmentation de l'abstraction de l'application et l'agilité d'une application** : les entreprises doivent s'adapter en permanence à la réaction du marché, elles subissent des acquisitions, des fusions, des scissions, des changements technologiques. Par conséquent, les applications doivent évoluer rapidement. L'application orientée service est constituée par une agrégation de **descriptions de services**. Ces descriptions sont exprimées dans un langage de haut niveau, indépendant de la technologie. Elles permettent

5. Acronyme de *Remote Procedure Call*.

une agilité de construction à la volée (à la demande du consommateur) de l'application (agilité de construction d'application cliente).
- **le masquage de l'hétérogénéité technologique** : le consommateur n'a pas la connaissance du détail de l'implantation, de la plate-forme d'exécution du fournisseur de services, du langage de programmation. Ce masquage de l'hétérogénéité technologique apporte les propriétés nécessaires à la réutilisabilité et à l'évolutivité des services fournisseurs (agilité d'évolution des services fournisseurs).

Le monde industriel est en permanence à la recherche de fiabilité et de sécurité pour l'exécution des applications. Les propriétés liées aux dynamismes et à la substituabilité sont sources d'erreurs. Elles doivent être utilisées avec précaution. Sans maîtrise des différents scenarii possibles, les possibilités d'architectures peuvent être nombreuses et sont, par conséquent, sources d'erreurs si toutes les éventualités ne sont pas validées.

2.2 SOA : Service-Oriented Architecture

Comme expliqué dans la section précédente, l'approche orientée service est un style architectural qui permet de construire des applications à partir de description de services publiées et découvertes. La réalisation des interactions suivant l'approche orientée service nécessite la mise en place d'un environnement d'intégration et d'exécution des services. Les technologies qui réalisent ces interactions, sont appelées architectures orientées service, en anglais *Service-Oriented Architecture* (SOA) [Papazoglou and Georgakopoulos, 2003].

Comme le montre la figure 2.2 « *Mécanismes d'un environnement d'exécution et d'intégration de service* », les mécanismes technologiques qui permettent de réaliser ces environnements d'intégration et d'exécution, sont organisés en deux catégories :
- **les mécanismes de base** permettent la publication, la découverte, la composition, la négociation, la composition, la contractualisation ainsi que l'invocation des services ;
- **les mécanismes additionnels** prennent en charge les aspects non-fonctionnels, tels que la sécurité, la gestion des transactions, le dynamisme.

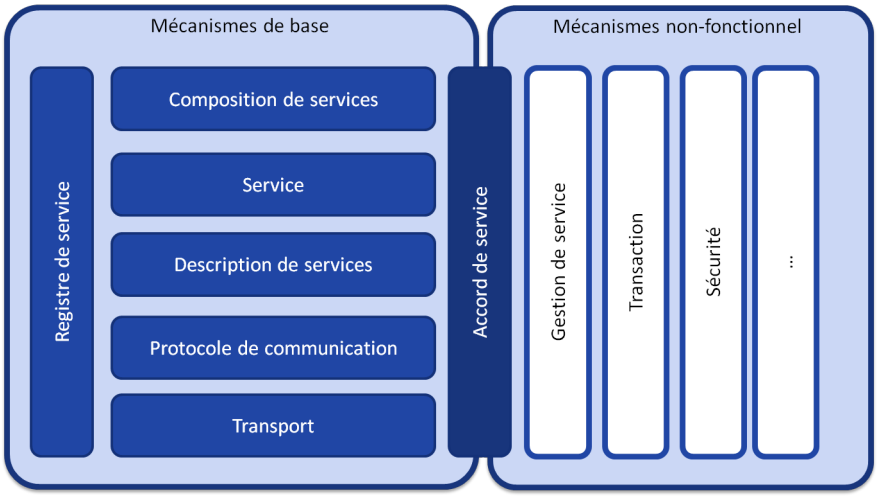


FIGURE 2.2 – Mécanismes d'un environnement d'exécution et d'intégration de service.

Les mécanismes de base sont présents dans toutes les technologies SOA, les mécanismes non-fonctionnels sont optionnels et extensibles selon l'objectif, le type d'application ou le métier. Nous allons détailler la première catégorie nécessaire à l'environnement d'intégration et d'exécution des architectures orientées service, à savoir les **mécanismes de base** :

- les couches **transport** et **protocole de communication** assurent la communication entre les acteurs. Pour des applications distribuées, nous retrouvons le protocole de communication ainsi que les requêtes et les réponses émises ;
- la couche **description de service** permet la description du service dans un langage de description spécifique. Elle permet d'exposer à un consommateur les capacités fonctionnelles et non-fonctionnelles du service et la description de son invocation ;
- la couche **service** assure l'exécution des services ;
- le **registre de services** permet la publication et la découverte de service correspondant au besoin du consommateur ;
- la **composition de services** fournit les mécanismes d'assemblage de services dans une application (voir section 3 « *Composition de service* », page 35).

L'**accord de service**, quant à lui, est à l'intersection entre les **mécanismes de base** et les **mécanismes non-fonctionnels**. Il représente le contrat entre le consommateur et le service. Ce contrat passé entre les deux parties explicite les fonctionnalités que ce service doit rendre, et des propriétés non-fonctionnelles qui permettent de satisfaire la qualité du service rendu.

La seconde catégorie, appelée **mécanismes non-fonctionnels**, ajoute de la valeur à l'environnement d'exécution. Il existe des applications distribuées nécessitant de la **sécurité**. La sécurité est une propriété non-fonctionnelle, qui doit être traitée suivant plusieurs aspects : l'authentification, l'autorisation ainsi que l'intégrité et la confidentialité des messages. D'autres applications peuvent nécessiter de la **fiabilité**. La fiabilité est elle aussi une propriété non-fonctionnelle. Elle peut se décrire comme la faculté de servir un service consommateur indépendamment de l'activité et du volume de traitement du service fournisseur. Nous pouvons encore continuer la liste des propriétés non-fonctionnelles avec la gestion transactionnelle, la performance, la qualité de service, la disponibilité, etc.

L'approche orientée service impacte avantageusement le cycle de vie du logiciel. Une application constituée de services doit évoluer pour suivre les besoins du consommateur, mais aussi toutes raisons autres que technologiques : par exemple, la fusion de sociétés ou l'acquisition de société. Cette description de service est le cœur de l'architecture orientée service. En effet, elle est indépendante des technologies réalisant le service mais aussi de celle de la plate-forme d'exécution. Ces applications sont, par conséquent, agiles pour suivre l'évolution des exigences ainsi que le changement d'infrastructure. De même la composition (voir section 3 « *Composition de service* », page 35) et toutes les propriétés précédemment décrites réduisent le temps des phases : le développement, les tests, l'intégration et la livraison (déploiement) du cycle de vie du logiciel.

Nous allons, dans la section 4 « *Composants orientés services* », page 39, décrire des technologies largement utilisées tant au niveau académique qu'industriel ; citons les deux technologies phare de l'approche orientée service :

- les services Web pour des applications distribuées ;
- OSGi™ [6] pour construire des applications orientées service de type local.

Comme habituellement en développement logiciel, le choix de la technologie à utiliser pour la construction d'application orientée service dépend du domaine d'application, du métier et des objectifs attendus.

6. Acronyme de *Open Service Gateway Initiative* : `http://www.osgi.org`

2.3 Besoins

Les applications orientées service sont des applications flexibles. Leurs réalisations font émerger des problématiques de construction et de maintenance. Pour répondre à ces besoins, M. P. PAPAZOGLOU [Papazoglou and Heuvel, 2007] a défini une SOA étendue nommée xSOA[7] s'appuyant sur les primitives de base de l'approche orientée service qui sont la publication, la découverte, la sélection, la liaison et la spécification. Ces éléments sont représentés dans la figure 2.3 « *Architecture orientée service étendue* », qui est une pyramide constituée de trois niveaux :

- les mécanismes de base et les capacités des services ;
- la composition des services ;
- l'administration des systèmes.

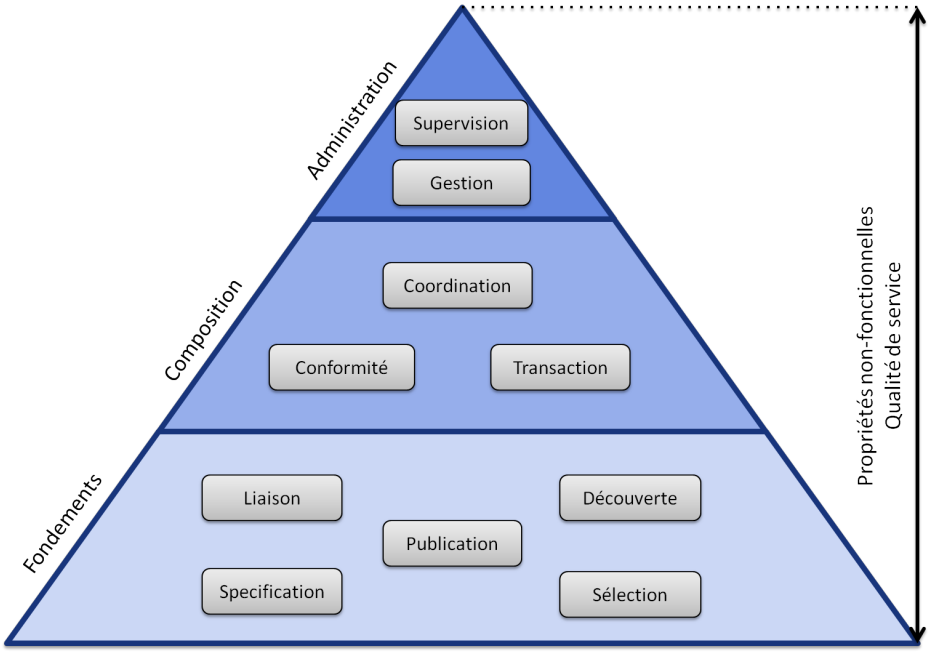


FIGURE 2.3 – Architecture orientée service étendue.

Le premier niveau de la pyramide fournit les **fondements** : les fonctionnalités communes à tout l'environnement d'exécution, ce niveau apporte l'infrastructure d'exécution pour connecter des services et systèmes hétérogènes.

Le deuxième niveau **composition** s'appuie sur le premier niveau, il offre les fonctionnalités pour le support de la composition de services et permet la création de services composites. Les services composites respectent les mécanismes de base présentés dans la section 4 « *Composants orientés services* », page 39. Le support de la composition de services nécessite de composer, coordonner des services et de vérifier la conformité de la composition.

Le dernier niveau de la pyramide administration apporte, aux fonctionnalités précédentes, les fonctionnalités d'**administration** des applications construites par composition de services. Ce sommet de la pyramide apporte les fonctionnalités telles que le déploiement, la surveillance et l'administration.

7. Acronyme de *Extended Service Oriented Architecture*.

Une architecture orientée service doit être capable de gérer, à tous les niveaux de la pyramide, des propriétés non-fonctionnelles (la sécurité, la qualité de service, etc.).

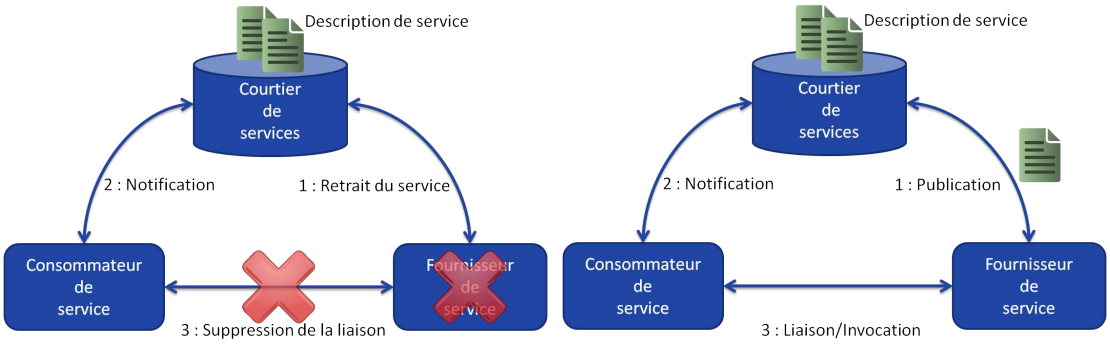
2.4 SOC Dynamique

Lorsque l'application est réalisée en changeant l'architecture, cette dernière est dite dynamique [Oreizy, 1998]. Décliné dans l'approche orientée service, l'architecture de l'application peut changer avec le départ et l'arrivée de services fournisseur. Pour construire de telles applications réactives à l'environnement, l'approche orientée service a été adaptée.

Lorsque le dynamisme est issu de l'environnement [Escoffier, 2008], l'approche à service, appelée approche à service dynamique, requiert deux primitives d'interactions supplémentaires. Le rôle du courtier de services est de faire le lien entre le fournisseur et le consommateur. A ce courtier incombe d'offrir ces deux nouvelles primitives :

- le **retrait d'un service** : lorsqu'un service n'est plus disponible, le fournisseur informe de son indisponibilité en retirant sa description de service du courtier de services ;
- la **notification** : cette primitive permet d'informer tous les consommateurs du départ et de l'arrivée d'un service. Ils peuvent traiter conséquemment le départ du service ou traiter une arrivée de service, qui peut correspondre à un besoin attendu.

La figure 2.4(a) illustre le départ d'un service et les interactions associées alors que la figure 2.4(b) montre l'arrivée d'un service.



(a) Départ de service.

(b) Arrivée d'un service

FIGURE 2.4 – Patron d'interaction de l'approche orientée à services dynamique.

Dans une architecture SOC dynamique, le dynamisme est traité autant du côté fournisseur que du côté consommateur et, grâce à l'ajout de deux nouvelles primitives, tous les moyens sont maintenant disponibles pour rendre l'architecture adaptable dynamiquement en cours d'exécution.

L'ajout dans le niveau **fondations** de la pyramide de l'architecture orientée service de ces deux primitives, qui sont le **retrait** de service et la **notification** de **disparition** et d'**apparition**, permet la construction d'applications réactives à l'évolution de leur environnement. Ceci n'est pas sans conséquence dans les niveaux supérieurs. Le niveau **comportement** doit évoluer pour maintenant gérer des applications composites de services dynamiques. La difficulté augmente pour composer, coordonner, vérifier la conformité des services qui peuvent apparaître et disparaître pendant l'exécution. De même dans le niveau **administration**, le déploiement, la surveillance et l'administration sont impactés par ce dynamisme.

Transversalement, la qualité de service et l'ensemble des propriétés non-fonctionnelles doivent aussi prendre en compte cet aspect dynamique. La difficulté de la gestion du dynamisme fonctionnel et non-fonctionnel croît avec les niveaux de la pyramide depuis sa base jusqu'à son sommet. Citons la technologie OSGi™ [8], qui est un bon exemple de plate-forme logicielle gérant le cycle de vie des services et leur dynamisme. Aujourd'hui, peu d'intergiciels offrent l'ensemble des couches nécessaires à la réalisation de la SOA dynamique.

3 Composition de service

Comme nous l'avons vu précédemment, l'approche à services autorise la composition de services pour la construction d'application. Cette section va présenter les principes de la composition de services et les deux styles de composition.

3.1 Principes

De la composition de services résulte un service appelé **service composite**. Le service composite est aussi un service, il possède les mêmes caractéristiques qu'un autre. La composition peut être effectuée soit de manière hiérarchique soit de manière récursive. D'une manière générale, la composition est un processus qui, à partir d'une spécification abstraite, permet d'obtenir une composition concrète. Ce processus a été défini par J. Yang et M. P. PAPAZOGLOU [Yang and Papazoglou, 2004] selon quatre phases allant de la définition abstraite à l'exécution :

1. la phase de **définition abstraite** de la composition de services consiste :
   - à identifier les fonctionnalités que l'application issue de la composition doit remplir ;
   - à identifier les fonctionnalités que doit apporter chaque participant à la composition, qui sont les fonctionnalités connues et attendues ;
   - à spécifier les interactions entre les participants.
2. la phase de **planification** consiste à rechercher dans un registre et à identifier des services correspondant aux besoins fonctionnels de l'application. Les aspects conformité et compatibilité des services sont traités lors de cette étape. Cette phase liste les candidats potentiels à la réalisation de la composition concrète des services ;
3. la phase de **construction**, en fonction de la stratégie fournie, sélectionne les fournisseurs parmi la liste des candidats potentiels. Si un service composite est construit, la description du service réalisé est publiée dans le courtier de services ;
4. la phase d'**exécution** de l'application réalise la composition concrète. Les services sont déployés sur la plate-forme d'exécution, les liaisons avec les services sont établies et l'invocation est réalisée.

Outre cet ordonnancement de phases qui explicite les tâches à effectuer, la composition de services reste une tâche difficile à définir et longue à réaliser. Composer des services consiste à entrelacer les séquences d'interactions entre les services, faisant apparaître des problèmes de séquencements et de synchronisation.

Dans certains cas, la composition de services nécessite la connaissance métier. Ces tâches métier ne sont pas toutes automatisées, certaines restent manuelles, tandis que les autres

8. Acronyme de *Open Service Gateway Initiative*.

peuvent entraîner des problèmes d'interopérabilité voire des problèmes liés aux flux de communication, ce sont généralement des problèmes de synchronisation. Ces derniers peuvent être résolus par des mécanismes de type médiation. La complexité de l'approche à service rajoute de la complexité dans la mise en œuvre de la composition de services.

Il existe deux manières d'appréhender l'analyse et la réalisation de la composition de services : en portant l'analyse sur la spécification de la réalisation ainsi que sur la logique de la coordination ou bien en portant l'analyse sur les liaisons entre services. De ces deux possibilités, il en résulte deux styles composition : la composition par procédés et la composition structurelle.

3.2 Composition par procédés

La composition de services par procédés consiste à définir un ordre d'enchaînement des invocations des services [Peltz, 2003]. Ce flux d'enchaînement ordonné est généralement représenté sous forme de graphe orienté et décrit dans un langage spécialisé de haut niveau interprété par un moteur d'exécution. Ce moteur d'exécution prend en charge les points suivants :

- exécuter les invocations dans l'ordre spécifié et ensuite communiquer avec les services concrets ;
- assurer le routage des données entre les services ;
- maintenir l'état de l'activité ;
- maintenir l'état du service composite ;
- gérer les erreurs.

Il existe deux familles de composition par procédés l'**orchestration** de services et la **chorégraphie** de services.

L'**orchestration de services**, voir figure 2.5 « *Orchestration de services* », exprime les conditions et les enchaînement des invocations. C'est le modèle des interactions que doit suivre le service composite pour réaliser sa fonctionnalité. C'est une vision centralisée sur les services, il représente la vision interne du service composite. L'orchestration décrit les étapes internes et les opérations internes (la transformation des données, l'alignement syntaxique et sémantique) réalisées entre les interactions du service composite [Peltz, 2003].

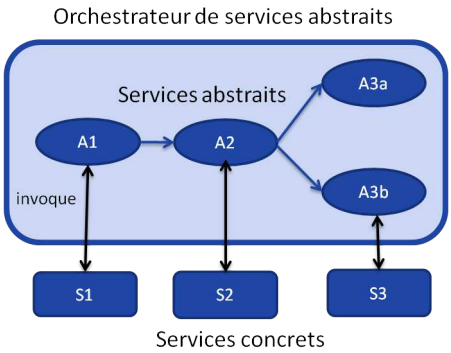


FIGURE 2.5 – Orchestration de services.

La **chorégraphie de services**, illustrée par la figure 2.6 « *Chorégraphie de services* » avec trois services S1, S2 et S3, décrit la manière dont un ensemble de services collaborent pour obtenir un but commun.

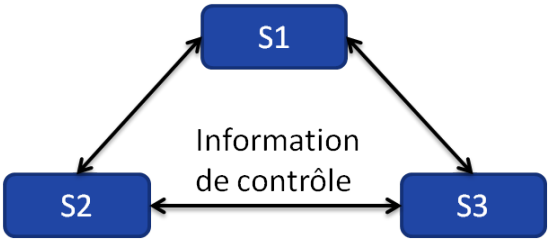


FIGURE 2.6 – Chorégraphie de services.

Dans ce cas, ce qui est rendu visible, est la vision globale des services et des interactions [Austin et al., 2004]. Pour cela la chorégraphie spécifie les interactions des services et les dépendances entre ces interactions. La chorégraphie est utilisée lorsque la modélisation des interactions entre les services est complexe (comme des transactions commerciales).

L'avantage de la composition par procédés est double. Il permet, d'une part, d'expliciter l'enchaînement ordonné d'invocations des services dans un langage de haut niveau et, d'autre part, de séparer le moteur d'exécution de la description de la composition. Une application qui utilise la composition par procédés, sépare donc les fonctionnalités des services de leur contrôle. Ce type de composition sépare les préoccupations.

De même, la composition des services se fait sans connaissance du fonctionnement interne ; seule la description du service est utilisée pour créer, définir et spécifier le composite. Ce type de composition est particulièrement bien adapté pour les services distribués tels que les services Web. Les services Web seront présentés dans la section 5.3 « *Services Web* », page 46.

Citons deux exemples de technologies de composition par procédés, toutes les deux basées sur la technologie XML[9] : WS-CDL[10] [Kavantzas et al., 2005], normalisé par le W3C, est un langage de description de chorégraphie de services Web et WS-BPEL[11] [OASIS, 2007], normalisé par OASIS, est un langage d'orchestration de services Web.

Cette approche de composition est difficile à réaliser lorsqu'il y a prise en compte des propriétés non-fonctionnelles [Chollet, 2009].

3.3 Composition Structurelle

Nous avons vu que la composition par procédés offre un mode de contrôle externe aux services. Nous allons maintenant détailler un mode de composition basé sur un contrôle interne aux services. La composition structurelle est un type de composition utilisable lorsque les **services** et les **interactions** sont clairement identifiés. Elle décrit une composition comme étant un assemblage de services à partir de leurs **implantations**.

Pour que la composition soit valide, il est nécessaire que l'assemblage résolve les dépendances sémantiques et syntaxiques. C'est la raison pour laquelle le développeur doit définir la logique de coordination (éventuellement fournie sous la forme d'une classe Java) pour exprimer le fonctionnement de la composition. Les propriétés non-fonctionnelles peuvent ainsi être résolue par l'implantation.

9. Acronyme de *eXtensible Marckup Language*.
10. Acronyme de *Web Service Choreography Description Language*.
11. Acronyme de *Web Service Business Process Execution Language*.

La figure 2.7 « *Composition structurelle* » illustre un exemple avec trois classes Java C1, C2 et C3. Les propriétés non-fonctionnelles des liaisons C1/C2 et C1/C3 peuvent être implantées dans la classe Java C1.

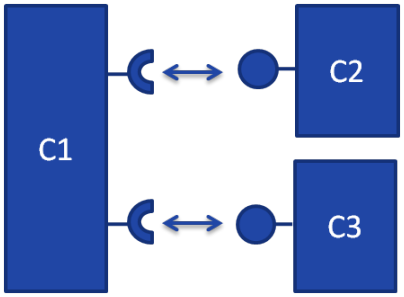


FIGURE 2.7 – Composition structurelle.

Il est aussi possible de décrire la composition structurelle avec un langage de description d'architecture (ADL [12]). L'ADL a pour objectif de décrire l'architecture logicielle. Dans le cadre de l'approche orientée service, l'ADL est utilisé pour décrire les composants constituant une application. Les composants seront présentés dans la suite de ce document. L'ADL est partagé par l'architecte (celui qui définit et modélise), le développeur (celui qui implémente) et l'intégrateur (celui qui assemble).

Notons que la communication entre services n'est pas externalisée dans un moteur d'exécution comme pour la composition par procédés (voir section 3.2 « *Composition par procédés* », page 36). Par conséquent, la communication de la composition structurelle est rapide, efficace et légère en empreinte mémoire.

Citons deux exemples de composition structurelle : la spécification SCA [13] pour la composition structurelle de services Web et iPOJO [14] pour la composition structurelle de services OSGi. OSGi et iPOJO sont détaillés dans la suite de ce document (voir section 5.5 « *OSGi™ et Apache Felix iPOJO* », page 56), nous présentons ici les principes généraux de SCA.

SCA est un ensemble de spécifications destinées à la composition de services Web. Un composant SCA réalise un ou plusieurs services et fournit des opérations. Il peut dépendre d'autres services et, à son tour, exposer toute une partie de son comportement comme service composite. Un composant SCA peut déclarer une ou plusieurs propriétés nécessaires pour la configuration de l'instance. La composition est indépendante de la technologie utilisée pour l'implantation et de la logique métier (voir figure 2.8 « *Composite SCA* »). Citons Apache Tuscany [Apache Software Fundation, 2008] qui implante l'infrastructure SCA.

12. Acronyme de *Architecture Description Language*.
13. Acronyme de *Service Component Architecture*.
14. Acronyme de *Injected Plain of Java Object*.

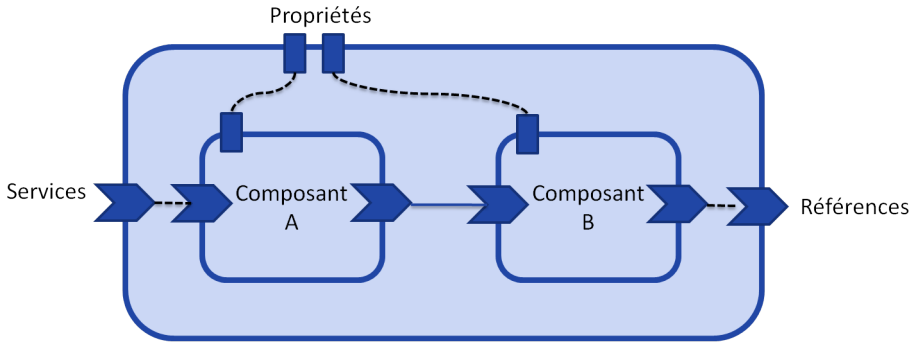


FIGURE 2.8 – Composite SCA.

4 Composants orientés services

4.1 Composants

Avant de présenter les composants orientés services, revenons sur les principes de l'**approche orientée objet**.Un objet est défini par trois propriétés : état, comportement et entité. Un **état** regroupe et fournit toutes les valeurs instantanées des données (appelées aussi attributs). Un **comportement** fournit des opérations qui peuvent être déclenchées par des appels de méthodes ou des événements. Pour terminer, l'**entité** est le nom et l'emplacement mémoire (ce que l'on appelle l'instance). Les objets fournissent le mécanisme d'encapsulation ; c'est-à-dire une séparation entre interface et implantation. La notion d'héritage et de polymorphisme est un mécanisme qui permet la réutilisation et l'adaptation de type. Une classe est un ensemble d'objets ayant des données communes (attributs) et disposant des mêmes méthodes.

Les applications construites suivant l'approche orientée objet présentent des difficultés pour satisfaire les qualités de réutilisabilité et d'évolutivité. A cela deux raisons principales :

1. les interfaces formalisent uniquement la notion de service fourni. Il n'existe pas de formalisme pour exprimer la notion de service requis. Par conséquent, l'évolutivité est difficile à obtenir sans modifier la stabilité du système.
2. la relation structurelle entre les objets ne peut pas être explicitement décrite. Il n'existe pas de description globale de l'application ; la réutilisabilité est difficile à obtenir principalement parce qu'il est difficile d'identifier les objets réutilisables.

L'**approche orientée composant** succède à l'approche orientée objet. Son objectif est de dépasser les limitations de réutilisabilité et d'évolutivité de l'approche orientée objet. L'approche orientée composant met en œuvre la construction d'applications par assemblage de composants [Cervantes and Hall, 2004]. L'idée sous-jacente est de déplacer une partie du travail du **développement logiciel** vers l'**intégration logicielle**. Ce déplacement permet de couvrir un plus large spectre du cycle de vie du logiciel. Cette séparation des préoccupations - développement et intégration - permet à l'industrie du développement logiciel de réduire les coûts ainsi que les temps de mise à disposition des applications sur le marché.

Attachons nous à la description d'un composant. Il n'existe pas de définition consensuelle d'un composant. Néanmoins, la définition la plus couramment citée dans la littérature est celle de C. SZYPERSKI.

> *« A software component is a unit of composition which contractually specified interfaces and explicit context dependencies only. A software component can be deployed independently and is subject to composition by third party. »* [Szyperski, 2002]

Cette définition met en avant les concepts importants qui caractérisent les composants :
- les composants définissent explicitement les interfaces qui expriment les fonctionnalités fournies ainsi que leurs dépendances ;
- pour être une unité autonome de composition, les composants fournissent des propriétés de configuration.

Dans un modèle à composant, on trouve les **types**, les **composants** et les **fabriques**. Les types de composants décrivent la spécification des interfaces, fournissent l'implantation et les aspects non-fonctionnels. Un composant est un type configuré. La fabrique permet la création du composant à partir du type du composant et de sa configuration. Il est souvent utilisé le terme instance de composant à la place de composant.

Plus le modèle du composant est riche plus les interfaces fonctionnelles supportent des mécanismes de communication évolués (synchrone, asynchrone et événementielle). Les dépendances sont décrites au niveau du type du composant, elles ne peuvent, par conséquent, n'être résolues qu'au niveau des composants.

Les composants nécessitent un environnement d'exécution permettant leur création et leur composition. Ici aussi, suivant les capacités de l'environnement d'exécution et les mécanismes de composition peuvent être plus ou moins riches [Cervantes and Hall, 2004].

L'approche orientée composant fournit donc les concepts de composant et de composition. L'approche est complétée avec un langage pour décrire l'architecture logicielle de l'application. Ce langage de haut niveau est appelé ADL, il présente deux caractéristiques [Krakowiak, 2009] :
- il est partagé par les développeurs (phase implantation) et les concepteurs logiciels (phase d'assemblage et d'intégration) ;
- il peut être associé à divers outils qu'ils soient graphiques ou non.

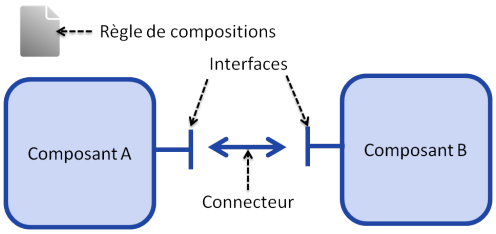


FIGURE 2.9 – Configuration.

La figure 2.9 « *Configuration* » illustre les trois notions présentes dans un ADL, comme défini par S. KRAKOWIAK [Krakowiak, 2009] :
- les **composants** sont la logique métier : l'unité de composition pour constituer une application. Ils sont composables à partir de leurs interfaces qu'elles soient requises et/ou fournies ;
- les **connecteurs** sont les liaisons réalisées entre les différents composants (d'interface à interface) et fournissent la communication (synchrone, asynchrone, directe, événementielle). De même plusieurs patrons de communication peuvent être utilisés (client/serveur, publication/souscription) ;

– la **configuration** est un ensemble de composants liés par les connecteurs. Les règles de composition définissent la manière dont est assemblée la configuration, ainsi que les contraintes pour qu'une interface requise puisse être liée à une interface fournie.

Une application n'est pas définie uniquement avec des termes fonctionnels ; des préoccupations non-fonctionnelles doivent être prise également en compte (comme la sécurité et l'audit). Pour aider à la prise en compte de ces aspects, une approche couramment utilisée consiste à séparer les préoccupations fonctionnelles de celles non-fonctionnelles, le tout encapsulé dans un composant. Ce composant est alors un conteneur, illustré par la figure 2.10 « *Composant conteneur d'aspects non-fonctionnels* ».

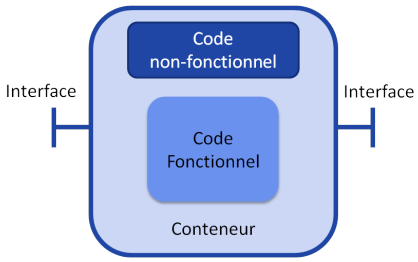


FIGURE 2.10 – Composant conteneur d'aspects non-fonctionnels.

En résumé, l'approche à composant répond aux besoins initiaux de couverture du cycle de vie du logiciel et offre une réutilisabilité et évolutivité supérieure à l'approche objet. De plus, la substituabilité d'un composant par un autre est une propriété essentielle d'une approche à composant. Elle est utilisée pour faire évoluer une application [Campbell and Grady, 1999].

L'approche à composant dans un environnement pervasif n'est pas adaptée, principalement par un manque de dynamisme de l'ADL. Même s'il existe des ADL dynamiques (qui permettent de créer des instances pendant l'exécution de l'application), leurs capacités sont souvent limitées à une séquence de patrons prédéfinis, restreignant les possibilités de gestion du dynamisme des composants (*e.g.*, ArchJava [Morrison et al., 2004]).

4.2 Approche à composants orientés service

L'approche à composants orientés service proposée par [Cervantes and Hall, 2004] introduit l'approche orientée service dans l'approche orientée composant.

La **disponibilité dynamique** est la situation où des fonctionnalités fournies par des composants (qui forment une application) deviennent disponibles ou indisponibles. Les applications pour supporter ces changements de fonctionnalités doivent prendre en compte ces situations. Bien évidemment, les modèles à composant supportent ces situations d'apparition et la disparition de composants. Cette situation est réalisée par l'implémentation du composant. Il en résulte un mélange de propriétés fonctionnelles et de logique de gestion de la disponibilité dynamique. L'approche composant orienté service introduit la notion de disponibilité dynamique dans le modèle à composant. Elle permet aux applications de s'adapter aux changements des applications et sépare les préoccupations de gestion de la disponibilité dynamique de celle du code fonctionnel.

Un **composant orienté service** est un composant conteneur d'aspects fonctionnels et non-fonctionnels dont les connecteurs suivent le style architectural de l'approche orientée service. Le code fonctionnel est exécuté à l'intérieur d'un conteneur qui va prendre en charge les propriétés non-fonctionnelles. Cette approche permet la simplification de l'écriture des ap-

plications orientées service. Le développeur n'a pas à prendre en compte la gestion de la disponibilité dynamique ainsi que le cycle de vie du composant. Ces tâches sont déléguées au code non-fonctionnel.

La figure 2.11 « *Modèle de composant orienté service* » représente le modèle de composant orienté service selon H. CERVANTES [Cervantes, 2004]. Ce modèle rajoute au composant conteneur des interfaces de contrôle, des propriétés de service ainsi que des dépendances de déploiement. Les interfaces de contrôle permettent la gestion de la reconfiguration dynamique, les propriétés de services sont associées à l'implantation du composant. Les dépendances de déploiement assurent les dépendances de ressources gérées par l'environnement d'exécution.

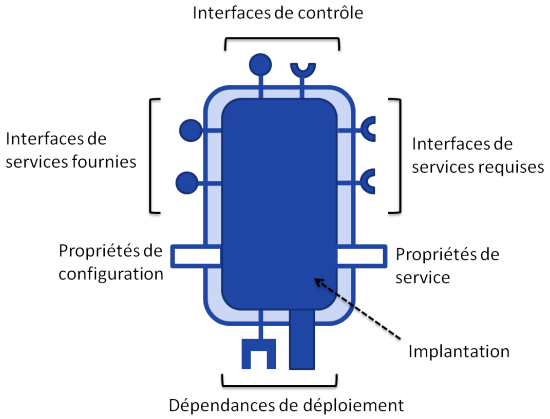


FIGURE 2.11 – Modèle de composant orienté service.

La figure 2.12 « *Gestionnaire d'instances et registre de services* » illustre selon H. CERVANTES [Cervantes, 2004] l'approche orientée composant. Chaque composant est géré par un gestionnaire de composants. Il est un conteneur composant en charge de la gestion du cycle de vie du composant en fonction de la validité des services requis. L'ensemble des interactions suit l'approche orientée service.

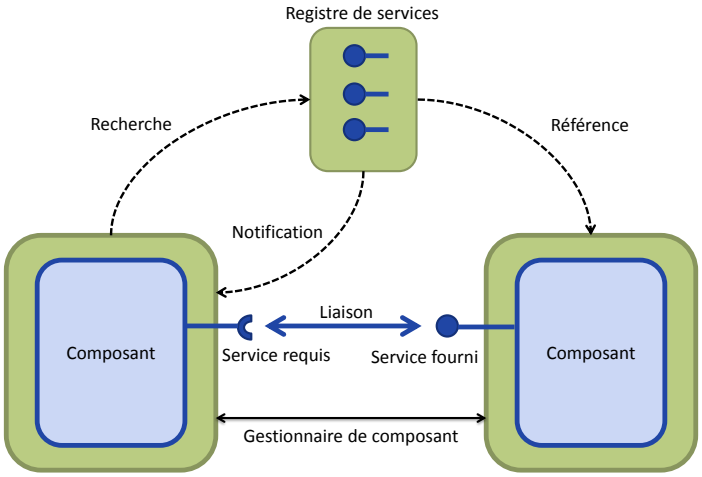


FIGURE 2.12 – Gestionnaire d'instances et registre de services.

5 Technologies à services

5.1 Caractérisation

Les précédents chapitres ont permis de définir les services informatiques, le paradigme de l'architecture orientée services et ses besoins technologiques nécessaires à sa réalisation. Cette partie va présenter les technologies orientées services en les caractérisant suivant les critères suivant :

- la **spécification** : comment sont décrites les fonctionnalités ainsi que les propriétés non-fonctionnelles des services ?
- le **type de courtier** : quel est le type de courtier (local, distribué ou fédération de courtiers) ?
- la **publication de services** : comment un fournisseur effectue une publication de service ?
- le **retrait de service** : comment un fournisseur peut retirer son service du courtier ?
- la **découverte de service** : quel type de découverte des spécifications de service est spécifié ?
- la **notification** : quel est le type de notifications qui sont émises au courtier de services lors d'un retrait et/ou d'une arrivée de service ?
- la **composition** : quel type de composition est supporté ?
- le **protocole de liaison** : quel est le protocole de liaison des services qui est utilisé ?
- l'**implantation** : quel langage de programmation permet de réaliser le service ?
- l'**interopérabilité** : quelle est la portée de l'interopérabilité ?
- le **type de couplage** : quel est le type du couplage ? Est-il fort ou faible ?

5.2 Historique : CORBA et Jini

5.2.1 CORBA

CORBA [Object Management Group (OMG™), 2008] est l'abréviation de *Common Object Request Broker Architecture*, il est l'un des standard issus du consortium OMG [15]. CORBA est une architecture logicielle pour le développement de composants communicant par l'intermédiaire de l'ORB [16].

Un ORB est un bus logiciel dans lequel les objets émettent des requêtes et reçoivent des réponses de manière transparente. Ce bus logiciel fournit la technologie pour qu'un objet puisse invoquer à distance les méthodes d'un autre objet.

Créée en 1992, cette spécification est écrite pour résoudre les problèmes entre services hétérogènes, supportant différents langages de programmation et environnements d'exécution. Le standard propose un langage de description d'interface (IDL [17]) pour la description des interfaces des composants. Des compilateurs permettent de générer le langage IDL à partir d'un langage cible donné.

Pour assurer l'homogénéité de la communication avec l'ORB, pendant la phase de compilation du langage IDL, il est ajouté de manière transparente le code d'appel des méthodes distantes et le traitement des résultats.

15. Acronyme de *Object Management Group*, `http://www.omg.org/`
16. Acronyme de *Object Request Broker*.
17. Acronyme de *Interface Description Language*.

En 1996, la version 2 de CORBA spécifie le protocole de communication IIOP [18] et, en 2002, la version 3 de CORBA spécifie des services communs tels que le nommage, le cycle de vie, la notification d'événements et, notamment, l'annuaire d'objets. Cet annuaire d'objets apporte à CORBA une approche orientée service. Il a la fonction de courtier de services telle que décrite dans l'approche orientée service.

La description de service est faite dans le langage IDL. La liaison et l'invocation utilisent généralement le protocole IIOP (Java RMI [19] est l'autre possibilité). La publication ainsi que le retrait d'un type de service dans l'annuaire d'objets passe par l'ORB. De même qu'un consommateur recherche un service dans l'annuaire de services en utilisant l'ORB, il invoque le service par les méthodes décrites dans l'interface IDL.

Grâce à son mécanisme de référentiel des interfaces, CORBA garde trace des objets, plus précisément, la signature des méthodes déployées sur l'ORB. L'interface d'invocation dynamique permet de rechercher et de localiser l'objet pour invoquer la méthode distante. Ce mécanisme permet l'établissement d'un lien flexible entre les consommateurs et fournisseurs au sein d'une application statique ; car les services sont décrits en IDL et compilés. La figure 2.13 « *CORBA schéma général* » illustre les éléments de CORBA version 2.

CORBA, qui fournit un couplage fort entre les consommateurs et fournisseurs, ne propose actuellement pas la spécification de composition de services.

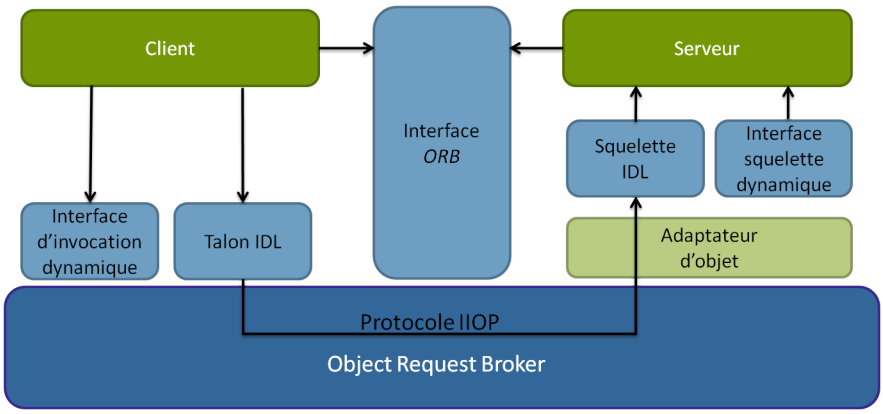


FIGURE 2.13 – CORBA schéma général.

Il existe des implantations commerciales comme Visibroker4 [20], qui fournit l'implantation de CORBA 2.3, et aussi des libres comme Mico [21], qui fournit l'implantation de CORBA 2. Aucunes n'implantent l'ensemble de la spécification CORBA.

5.2.2 Jini

Jini est une spécification proposée en 1999 par Sun MicroSystems (maintenant, Oracle Corporation). Jini permet le développement d'applications constituées de dispositifs communiquant sur les réseaux, dans un environnement pervasif suivant l'approche orientée service dynamique. Jini repose sur le langage Java et offre la possibilité de télécharger du code à travers le réseau.

18. Acronyme de *Internet Inter Orb Protocol*.
19. Acronyme de *Remote Method Invocation*.
20. `http://www.microfocus.com/products/visibroker/index.aspx`
21. `http://www.mico.org/`

Jini définit trois entités :
- une infrastructure pour l'administration des dispositifs sur les réseaux dans un environnement pervasif ;
- un modèle de programmation pour le développement d'applications ;
- des services, qui sont des entités logicielles utilisables par l'ensemble des acteurs.

Les services sont décrits par une interface Java, dont les méthodes définissent l'ensemble des fonctionnalités du service et les signatures des méthodes explicitent la syntaxe d'appel.

Le courtier de services Jini est un service technique. Il fonctionne comme une table d'associations « Objets/Interfaces », il est appelé service ***lookup***. En plus de sa fonction de table d'associations, ce service technique gère l'apparition et la disparition de service. Il est aussi en charge de propager des événements aux clients qui lui sont abonnés. Nous allons expliciter dans les prochains paragraphes ces événements. Plusieurs services *lookup* peuvent exister au sein d'une même infrastructure. Ils peuvent communiquer les uns avec les autres. La capacité de communication entre services *lookup* associée avec la gestion des événements (abonnement et propagation) permet la fédération des services *lookup*.

La gestion de la disponibilité des services au sein de l'infrastructure est gérée selon le principe suivant. Les services sont enregistrés dans le service lookup avec un début et une date de fin de validité. C'est ce que l'on appelle un bail. Un service, qui voit son bail arriver à terme, peut demander au service *lookup* de renouveler son bail (une nouvelle date de fin de validité). Si le service *lookup* ne reçoit pas de demande de renouvellement, il supprime de la table d'associations le service qui a son bail invalide.

La spécification définit trois types d'événements du service *lookup* : **apparition de service**, **disparition de service** et **modification de service**. Un service client peut s'abonner auprès de ce service pour recevoir ces événements. Pour s'enregistrer auprès du service *lookup*, un fournisseur doit :
- connaître l'adresse du service *lookup* ou le rechercher sur le réseau ;
- implanter les interfaces du service ;
- enregistrer l'objet réalisant l'interface ainsi qu'un ensemble d'attributs.

Pour consommer un service, il faut :
- connaître son adresse ou la rechercher sur le réseau ;
- rechercher le service *lookup* ou s'enregistrer auprès du service *lookup* pour être notifié de son apparition, de sa disparition et/ou de sa modification de service.

La liaison et l'invocation utilisent le protocole Java RMI. Java RMI permet la communication entre applications Java exécutées sur une même machine virtuelle (*Java Virtual Machine*) ou bien sur des machines virtuelles différentes. Java RMI est un mécanisme RPC [22] , le couplage est fort entre le fournisseur et le consommateur (voir section 2.1 « *SOC : Service-Oriented Computing* », page 29).

Jini ne propose de spécification pour la composition de services.

Le projet Apache River [23] fournit une implémentation de Jini. L'interopérabilité entre versions de services est assurée par l'infrastructure d'exécution (le *runtime* Apache River) ainsi que par la spécification Jini.

22. Acronyme de *Remote Procedure Call*.
23. `http://river.apache.org/`

5.2.3 Synthèse

CORBA et Jini (*c.f.* table 2.1 « *Caractéristiques de Jini et de Corba* ») sont deux technologies assez anciennes, qui permettent de construire des applications suivant le paradigme de l'approche à services. CORBA ne peut pas permettre la construction d'applications dynamiques par l'absence de découverte passive. La technologie Jini, quant à elle, permet la construction d'applications dynamiques. Cependant, ces deux technologies ne permettent pas la composition de services et fournissent un couplage fort.

CRITÈRES	CORBA	JINI
Spécification	IDL	Interface Java
Type de courtier	Fédération de courtiers distants d'un service CORBA	Centralisé et généralement distant sous forme de service Jini
Publication de service	Manuel par appel de méthode du courtier	Manuel par appel de méthode du courtier
Retrait de service	Manuel par appel de méthode du courtier	Automatique par expiration du bail
Découverte de services	Découverte active	Découverte active et passive, service de recherche
Notification	Pas de notification	Appel de méthode
Composition	Non proposé	Non proposé
Protocole de liaison	Principalement IIOP et Java RMI	Java RMI
Type de couplage	Fort (IIOP)	Fort (Style RPC et sérialisation Java)
Implantation	Tous les langages, génération de squelette et talon à partir de l'IDL	Java

TABLE 2.1 – Caractéristiques de Jini et de Corba.

5.3 Services Web

Les services Web (*c.f.* table 2.14 « *Mécanismes de base et non-fonctionnels pour les services Web* », page 47)sont la technologie qui a reçu la plus large acceptation pour la mise en œuvre de l'architecture orientée service. Principalement, ils permettent de rendre disponibles et d'intégrer des applications hétérogènes sur l'Internet ou l'intranet.

Les services Web utilisent des technologies largement utilisées et standardisées par le W3C et OASIS. Ils apportent une facilité dans le partenariat et la collaboration technologique inter-entreprises ou intra-entreprise. Pour les services Web, les standard utilisés sont au nombre de quatre :

1. *eXtensible Markup Language* (XML) [Bray et al., 1997] ;
2. *Simple Object Access Protocol* (SOAP) [Gudgin et al., 2003] ;
3. *Universal Description, Discovery and Integration* (UDDI) [OASIS, 2004] ;
4. *Web Service Description Language* (WSDL) [Chinnici et al., 2007].

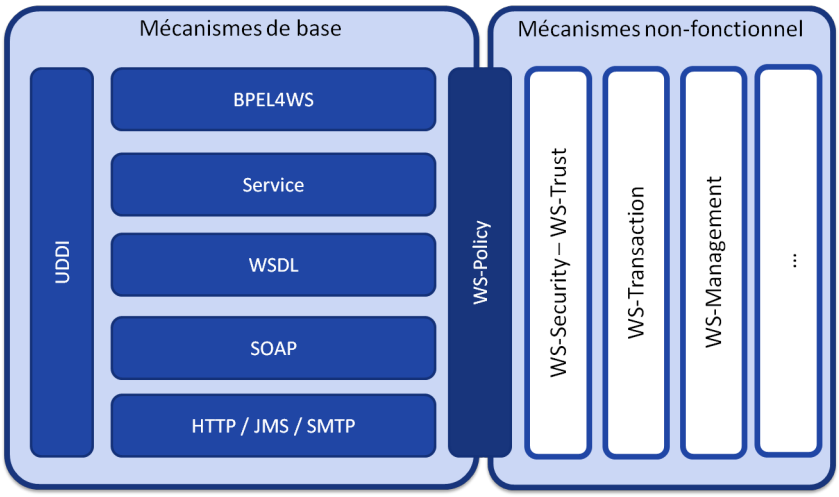


FIGURE 2.14 – Mécanismes de base et non-fonctionnels pour les services Web.

Nous allons étudier les trois standard pour les services Web : WSDL, SOAP et UDDI.

5.3.1 La description de services : WSDL

Les services Web sont décrit avec un langage standard de description : le *Web Service Description Language* (WSDL) [Chinnici et al., 2007]. Ce langage permet d'exprimer à la fois les caractéristiques fonctionnelles ainsi que les propriétés non-fonctionnelles des services. Les caractéristiques fonctionnelles comprennent les opérations qui définissent le comportement général du service, alors que les caractéristiques non-fonctionnelles décrivent principalement les caractéristiques de l'environnement d'hébergement. WSDL est basé sur une grammaire XML.

Le WSDL peut être séparé en deux parties distinctes :
- la **partie abstraite** est la description des interfaces du service. On retrouve les opérations supportées par le service, les paramètres et les types abstraits de données ;
- la **description concrète** permet de lier les interfaces à une adresse réseau avec un protocole de communication ainsi que la définition concrète des structures.

La balise `<definition>` encapsule le document WSDL et donne le nom de la description du service Web. Les balises `<type>`, `<messages>`, `<part>`, `<portType>` et `<operations>` permettent de décrire la partie. Les balises `<binding>`, `<port>` spécifient le protocole réseaux, elles sont la concrétisation de la partie abstraite de `<portType>` et de `<operation>`. Quant à la balise `<service>`, elle fournit l'adresse réseau du service Web décrit (voir figure 2.15 « *Liens WSDL et description UDDI* » page 49).

Le WSDL permet la description de quatre types de patron d'interactions de services Web. Ces opérations correspondent au besoin de communication entrante et sortante d'un service Web :
- **unidirectionnelle** : l'opération peut recevoir un message, mais ne renvoie pas de réponse ;
- **question-réponse** : l'opération peut recevoir une requête et doit retourner une réponse ;
- **notification** : l'opération peut envoyer un message mais n'attend pas de réponse ;
- **réponse sollicitée** : l'opération peut envoyer une requête et attend une réponse.

Il existe actuellement beaucoup de compilateurs WSDL, qui permettent de générer dans un langage cible des proxies de service Web (*e.g.*, WSDL2Java de Axis [Apache Software Fundation, 2013]

pour la production de code Java et WSD2H de gSoap [Van Engelen and Gallivan, 2002] pour la production de code C ou C++).

Le WSDL permet de décrire la partie fonctionnelle, tout autant que les patrons d'interactions entre les services Web. Les aspects non-fonctionnels ne peuvent pas être décrits avec un WSDL, ils nécessitent un langage indépendant ou une combinaison de plusieurs spécifications. Citons, par exemple, WS-Policy [Bajaj et al., 2006] généralement associé à WS-PolicyAttachment [Vedamuthu et al., 2007] pour décrire la qualité de service d'un service Web.

Seule la version 2.0 est approuvée par le W3C, néanmoins la version 1.1 qui est une note du W3C reste la plus utilisée à ce jour et, par conséquent, est un standard de fait.

Les services Web utilisent généralement deux méthodes pour récupérer la description du service fournisseur :
- une méthode statique : le consommateur récupère la description du service par une méthode ad hoc. La description du service récupérée contient, entre autres, l'URL[24] du service fournisseur ;
- une méthode dynamique : le consommateur recherche l'annuaire de service UDDI (voir section 5.3.2 « *L'annuaire de service Web : UDDI* »), puis la description de service dans l'annuaire.

5.3.2 L'annuaire de service Web : UDDI

Universal Discovery, Description and Integration (UDDI) est le standard utilisé pour le registre de services (courtier de services) pour les services Web. UDDI est un standard OASIS décrit par un WSDL. UDDI est un répertoire public, qui fournit un mécanisme pour sauvegarder et récupérer les informations sur les services.

UDDI est consultable de trois manières :
- les **pages blanches** permettent de retrouver à partir du nom des informations d'entreprise (les coordonnées, la description succincte) ;
- les **pages jaunes** sont l'annuaire thématique organisé selon une classification normalisée ;
- les **pages vertes** sont l'annuaire technique.

UDDI décrit quatre types d'entités :
- les ***businessEntity***, contenant la description des entreprises (destinée à des humains) ;
- les ***businessService***, contenant les informations non techniques des services fournis par les entreprises ;
- les ***bindingTemplates***, contenant les informations techniques pour accéder au service (URL et la définition abstraite du WSDL) ;
- les ***tModel***, contenant la partie implémentation des services (la définition concrète du WSDL).

La figure 2.15 « *Liens WSDL et description UDDI* » montre les liens entre le WSDL et la description UDDI.

24. Acronyme de *Uniform Resource Locator*.

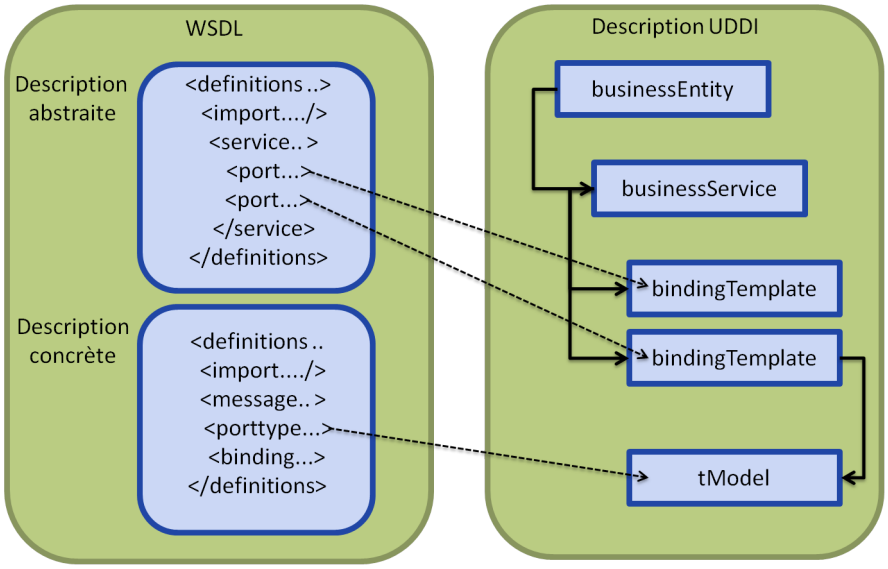


FIGURE 2.15 – Liens WSDL et description UDDI.

UDDI est un standard pour les services Web, mais reste à ce jour peu utilisé.

WS-Inspection [Ballinger et al., 2001] est un standard en cours de développement par IBM et Microsoft, il permet la découverte des services Web. Ce standard repose sur les technologies WSDL et SOAP. Il pourrait être un candidat au remplacement de UDDI.

5.3.3 SOAP : le protocole de messagerie des services Web

SOAP était à l'origine l'acronyme de *Simple Object Access Protocol*, maintenant il est un nom commun. C'est un protocole de messagerie pour l'échange de messages entre deux entités logicielles. Il repose sur le langage XML.

SOAP est spécifié pour un environnement distribué agnostique de la technologie de transport des messages. De fait, il décrit le format d'un message et nullement son mode de livraison. Le message doit être encapsulé dans un protocole de niveau transport. La couche transport la plus couramment utilisée avec SOAP est HTTP [25], d'autres protocoles peuvent être utilisés tels que FTP [26], SMTP [27] et même RMI [28].

SOAP est entièrement défini à la fois par son modèle de communication ainsi que par la structure des messages. Le modèle de communication de SOAP repose sur le mode client/serveur avec deux styles possibles :

- le **style RPC** [29] : la requête cliente est exprimée comme un appel de méthode, avec les paramètres d'entrée et la réponse contenant une valeur. Ce mode nécessite un fort couplage de communication entre le client et le serveur.
- le **style message** (Document) : un document entier est envoyé et, optionnellement, un document de réponse est reçu. Le serveur doit parser le document et le client doit lui aussi parser le document réponse.

25. Acronyme de *Hyper Text Transfert Protocol*.
26. Acronyme de *File Transfer Protocol*.
27. Acronyme de *Simple Mail Transfer Procotol*.
28. Acronyme de *Remote Method Invocation*.
29. Acronyme de *Remote Procedure Call*.

En ce qui concerne la structure d'un message, SOAP est constitué de trois entités : une enveloppe `<Envelop>` contenant optionnellement un en-tête `<Header>` et d'un corps `<Body>`.

L'enveloppe SOAP `<Envelop>` est la balise qui encapsule tous les messages. Elle sert à décrire ce qu'il y a dans le message et comment le décoder et indique l'ensemble des règles d'encodage et de décodage de l'application XML avec les schémas XML.

L'en-tête SOAP `<Header>` est un mécanisme d'extensibilité du protocole SOAP. Il permet d'ajouter des fonctionnalités telles que la sécurité ou encore les transactions. Elle fournit des informations sur le contenu du message SOAP. La balise `<Header>` apporte aussi des informations d'adressage. Tout message SOAP commence par l'adresse de l'émetteur (celui qui crée le message) et se termine par l'adresse du receveur. Il est possible de définir une adresse intermédiaire servant d'adresse de livraison du message sous certaines conditions non-fonctionnelles telles que la sécurité.

Le corps SOAP `<Body>` contient les données spécifiques. Dans le cadre des services Web, cette balise apporte toutes les informations sur les méthodes à invoquer avec leurs paramètres dans le style RPC ou le document à échanger dans le style message.

5.3.4 Synthèse

Les services Web (*c.f.* table 2.2 « *Caractéristiques des services Web* », page 51) ont démocratisé les applications orientées service ; ils offrent un couplage faible (pas de connaissance de structures internes ou de contexte) entre le consommateur et le fournisseur. Ils permettent la collaboration entre des fournisseurs et des consommateurs provenant de différentes entreprises.

Cependant, UDDI, SOAP et WSDL sont des spécifications qui n'explicitent pas comment les données XML sont représentées en mémoire pour être transformées en requête/réponse de communication (*marshalling*) ainsi que les opérations inverses (*unmarshalling*). Il apparaît des problèmes d'interopérabilité d'implantation aux trois points clés de l'approche orientée service : la découverte, la définition et la communication (les requêtes émises et les réponses reçues). Un exemple d'interopérabilité est la représentation en mémoire : des dates, des nombres à virgules flottantes et des nombres primitifs. Comparons la plate-forme .NET [30] et Java. Le XML schéma normalisé par le W3C définit des nombres primitifs non signés : `xsd:unsignedInt`, `xsd:unsignedLong`, `xsd:unsignedShort` et `xsd:unsignedByte`. Sur une plate-forme .NET les nombres primitifs `uint`, `ulong`, `ushort` et `ubyte` sont directement associés aux types `xsd` précédemment cités. Pour la plate-forme Java les types non signés ne sont pas des types primitifs. Des informations peuvent être perdues lors de leur représentation en mémoire. De même, pour les nombres à virgule flottante .NET fournit une précision de cinq chiffres après la virgule et Java une précision de six chiffres. Les dates sont aussi source de non interopérabilité. XML Schema défini `xsd:dateTime` réalisé par la méthode `System.dateTime` en .NET avec une précision de quatre chiffres pour l'année et sept chiffres pour les millisecondes. Pour Java la méthode est `java.util.Date` avec une précision de cinq chiffres pour l'année et trois chiffres pour les millisecondes.

Une autre raison de rupture d'interopérabilité est l'émergence et l'évolution rapide des standards. Ces standards évoluent, les applications ne peuvent pas suivre ces évolutions (technologies non encore disponibles).

30. `http://www.microsoft.com/net`

Les standards ne peuvent pas être précis sur tous les points de leur spécification, il existe des zones floues. Elles amènent à une interprétation particulière et, bien évidement, non partagée entre le consommateur et le fournisseur. Toute interprétation particulière est source de non-interopérabilité.

WS-I[31] [OASIS, 2013] est un consortium d'industriels qui s'attache à résoudre les problèmes d'interopérabilité des standard WS-*, en fournissant des profils interopérables. Un profil interopérable est une liste explicitant : des spécifications avec leur version, des recommandations pour l'implantation, des clarifications sur les zones floues des spécifications ainsi que des outils pour contrôler la conformité au profil implanté.

Pour terminer, citons deux familles de technologies facilitant le développement d'applications basées sur des services Web cliente et fournisseurs : côté libre, Apache Axis2/Java[32] et Apache Axis2/C[33] et, côté commercial, IBM WebSphere[34].

CRITÈRES	SERVICES WEB
Spécification	WSDL 1.1 ou WSDL 2.0
Type de courtier	Serveur UDDI
Publication de service	Requêtes UDDI (SOAP/HTTP1.1) ou API
Retrait de service	Requêtes UDDI (SOAP/HTTP1.1) ou API
Découverte de services	Requêtes UDDI (SOAP/HTTP1.1) ou API
Notification	UDDI service notification (SOAP/HTTP1.1) ou API
Composition	BPEL4WS
Protocole de liaison	SOAP, protocoles de transport HTTP, JMS, SMTP
Type de couplage	Faible
Implantation	Tous les langages, génération automatique de *proxies* à partir de la description WSDL

TABLE 2.2 – Caractéristiques des services Web.

5.4 UPnP et DPWS

5.4.1 UPnP

Universal Plug and Play (UPnP) [UPnP Forum, 2008b] est un ensemble de spécifications de UPnP Forum [UPnP Forum, 2008a], initiées en 1999 par un ensemble d'industriels, dont Microsoft. UPnP permet la communication entre dispositifs hétérogènes sur un réseau local SOHO[35], filaire ou sans fil, facile à utiliser et flexible. Les équipements connectés à ces réseaux peuvent être des dispositifs hétérogènes tout autant que des ordinateurs portables.

31. Acronyme de *Web Service Interoperability Organization*.
32. `http://axis.apache.org/axis2/java/core/`
33. `http://axis.apache.org/axis2/c/`
34. `http://www-01.ibm.com/software/fr/websphere/`
35. Acronyme de *Small Office Home Office*.

Ces dispositifs UPnP sont définis pour être sans configuration et sans pilote[36] installé. Le réseau est de type pair à pair[37]. UPnP repose sur une architecture orientée service dynamique, les technologies utilisées sont issues des standard très largement utilisés tels que TCP[38], UDP[39], HTTP et SOAP. La figure 2.16 « *Réseau UPnP* » illustre un réseau UPnP.

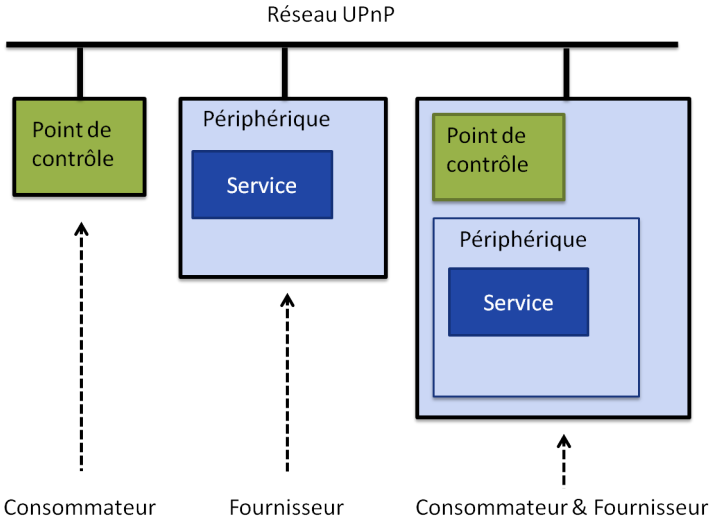


FIGURE 2.16 – Réseau UPnP.

UPnP définit deux entités :

- le **périphérique** est un conteneur de services. Les services peuvent être des services composites (voir section 3.1 « *Principes* », page 35). Le service est constitué : d'une table d'états, d'un serveur de contrôle et d'un serveur d'événements. Les états sont les valeurs des variables caractérisant le service. Le serveur de contrôle effectue la mise à jour des variables d'actions. Le serveur d'événements effectue l'émission d'événements aux périphériques abonnés.
- le **point de contrôle** est en charge de la découverte et de la disparition des périphériques mais aussi des descriptions de services. Il récupère les valeurs des variables d'états et gère la souscription aux abonnements à d'autres périphériques, il invoque des actions pour contrôler le service.

Un dispositif UPnP peut être une **entité périphérique** et, dans ce cas, il est **consommateur** de services. S'il est un **point de contrôle**, il est alors **fournisseur** de service. Un dispositif UPnP peut supporter les deux entités : périphériques et point de contrôle (il est à la fois fournisseur de services et consommateurs).

Nous allons maintenant expliciter les mécanismes généraux des spécifications UPnP. Le mécanisme d'arrivée et de départ de périphérique sur le réseau est basé sur le principe de notification générale. Le point de contrôle émet une trame HTTP en multicast pour signaler l'arrivée du périphérique (trame `HELLO` dans la spécification) et une autre pour signaler le départ du réseau (trame `BYE`). Les abonnements à des événements de périphériques sont à durée limitée (principe de bail voir section 5.2.2 « *Jini* », page 44). Il est à la charge du point de contrôle souscripteur de renouveler son abonnement auprès du point de contrôle de l'événement source.

36. En anglais, *driver*.
37. En anglais, *peer-to-peer*.
38. Acronyme de *Transmission Control Protocol*.
39. Acronyme de User Datagram Protocol.

Le protocole de découverte permet de découvrir des **périphériques** puis, de récupérer la liste des descriptions des services hébergées par chacun d'eux. Chaque **périphérique** est fournisseur de services, chaque **point de contrôle** est consommateur de service. La fonction courtier de services de l'approche orientée service est apportée par le protocole de découverte générale de périphérique et de recherche de services.

La fonction courtier de services n'est pas centralisée, elle est assurée par l'ensemble des dispositifs connectés sur le réseau. Cette architecture se justifie de par les objectifs de UPnP qui sont : un protocole léger et facile à utiliser, sans configuration pour un réseau pair à pair qu'il soit filaire ou non filaire ainsi que des dispositifs hétérogènes et dynamiques. UPnP ne traite pas les propriétés non-fonctionnelles notamment la sécurité.

UPnP Forum fournit deux familles d'outils :
- des **outils croisés** pour le développement des dispositifs et des services (plusieurs langages cibles et système d'exploitation (C/C++/Java et Linux/Windows/Android) ;
- des **outils de certifications** à la spécification UPnP. Ces outils permettent la certification aux spécifications UPnP des dispositifs et des services.

Ces deux familles d'outils permettent l'interopérabilité totale entre dispositifs hétérogènes de la famille UPnP.

5.4.2 DPWS

Les services Web intègrent un ensemble de spécifications plus ou moins complexes permettant de définir de riches et complexes fonctions pour réaliser des applications distribuées. Partant de ce constat et du besoin de rendre interopérables des dispositifs à ressources limitées, Microsoft a défini un ensemble de spécifications de services Web pour ces dispositifs à ressources contraintes. Le contour fonctionnel a été limité aux possibilités :
- d'**envoyer et de recevoir des messages** sécurisés à partir et à destination de services Web ;
- de **découvrir** dynamiquement des services Web ;
- de **décrire** un service Web ;
- de **souscrire** et recevoir des événements d'un service Web.

L'ensemble des spécifications sélectionnées parmi les WS-* disponibles fournit un profil appelé DPWS [OASIS, 2009], il est l'acronyme de *Device Profile for Web Services*. Ce profil est une liste de spécifications de services Web, mais aussi des **recommandations de niveaux implantations** (par exemple, le protocole de transport de message SOAP est HTTP 1.1 et doit utiliser l'encodage de transfert type *chuncked*[40]).

DPWS repose sur une architecture orientée services et, tout comme UPnP, définit deux entités :
- ***device***, qui prend en charge la partie découverte et échange de méta-data ;
- ***hosted services***, qui sont les applications services Web hébergées par le device.

Le protocole WS-Discovery [OASIS, 2009] spécifie l'arrivée et le départ d'un dispositif DPWS sur le réseau. Il utilise le mode UDP[41] *multicast* pour diffuser à tous les présents l'arrivée d'un nouvel équipement (émission d'une trame `HELLO` dans la spécification DPWS). La spécification définit un mode appelé *discovery-proxy*. Il permet la découverte des dispositifs sur des réseaux TCP/IP interconnectés par des routeurs (passerelles), la limite de la décou-

40. Le mode chuncked de HTTP 1.1 permet l'envoi par morceaux consécutifs des messages de taille non connue à l'avance.
41. Acronyme de *User Datagram Protocol*.

verte reposant sur UDP est dépassée. Un dispositif DPWS quitte le réseau en émettant une trame UDP [42] *multicast* (BYE).

Le protocole WS-Discovery permet de découvrir et ou de rechercher des *devices* DPWS présents sur le réseau, mais ne découvre pas des services. La description retournée par les *devices* DPWS contient la liste des descriptions WSDL des services de type *hosted services*.

La figure 2.17 « *Dispositif DPWS* » schématise le bloc fonctionnel d'un dispositif DPWS. Il est constitué d'une entité découverte des services qui écoute et émet des messages SOAP en UDP, d'un moteur d'exécution SOAP qui écoute et émet des messages SOAP en utilisant la couche transport HTTP. Ce moteur est aussi en charge des opérations de *marschalling* et *unmarchalling*. Le service d'événements est en charge de la souscription, de la notification et de la gestion du bail des services abonnés.

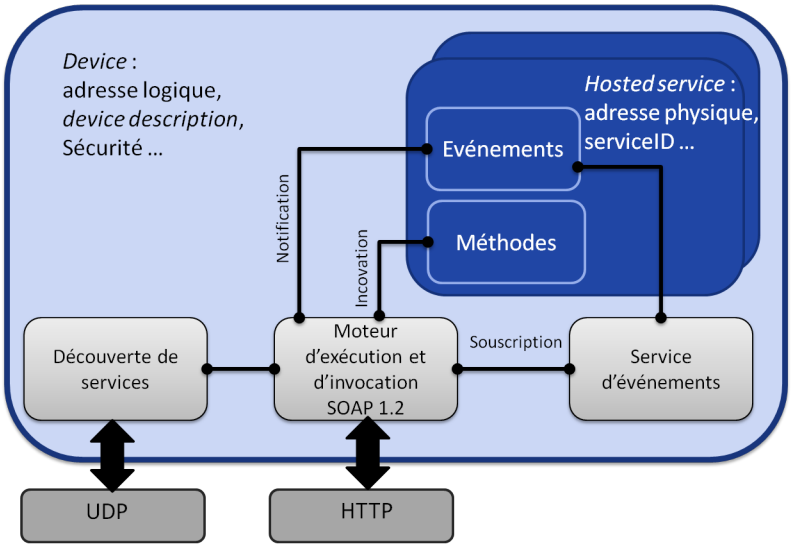


FIGURE 2.17 – Dispositif DPWS.

Un device DPWS ne rend pas visible son adresse transport. Seule une adresse logique est exposée (l'adresse du Device de la figure 2.17 « *Dispositif DPWS* »), il est donc impossible d'invoquer une opération d'un *hosted service* (voir figure 2.17 « *Dispositif DPWS* ») sans avoir précédemment découvert le dispositif DPWS par WS-Discovery. Ce mécanisme est l'implantation de la spécification, qui demande qu'un device DPWS ne doit pas avoir d'URL fixe. Par conséquent, la composition de services DPWS n'est pas impossible mais elle est rendue très difficile, BPEL4WS [Jordan et al., 2007] nécessite des évolutions. La figure 2.18 « *Réseau DPWS* » schématise un réseau de dispositifs DPWS.

DPWS résout les problèmes d'interopérabilité en reprenant le principe de WS-I [43] ; c'est-à-dire que le standard DPWS est un profil contenant une liste de spécifications et une clarification des zones floues des spécifications. Cependant, il n'existe pas à ce jour d'outil de contrôle de la conformité de l'implantation d'un dispositif DPWS à une version particulière de DPWS.

42. Les trames UDP *multicast* possèdent un compteur configuré par l'initiateur de la trame. Ce compteur est décrémenté de un à chaque passage de routeur. Lorsqu'il est décrémenté et qu'il devient nul, le routeur détruit la trame.

43. Acronyme de *Web Service Interoperability* ; `http://www.ws-i.org/`

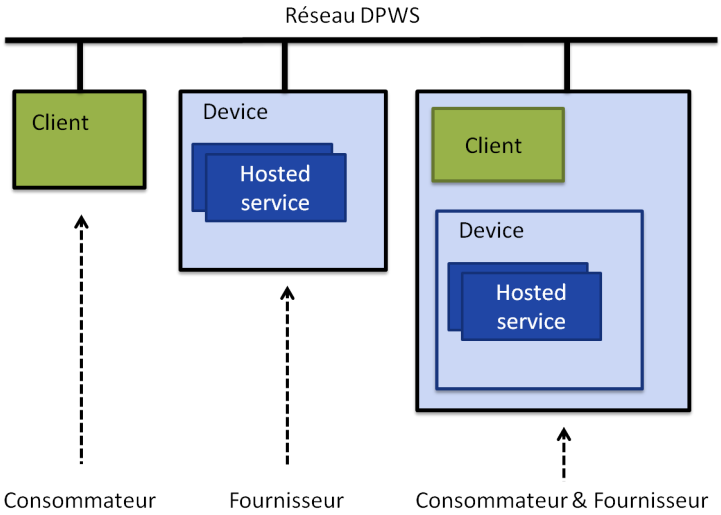


FIGURE 2.18 – Réseau DPWS.

DPWS est une technologie issue des services Web, les problèmes d'interopérabilité d'implantation aux trois niveaux des interactions, qui sont la découverte, la spécification et la communication, restent ceux des services Web précédemment décrits dans la section 5.3 « *Services Web* », page 46. Cependant, la notion de profil apporte des problèmes d'interopérabilité entre versions standardisées. DPWS de version 1.0 n'est pas interopérable avec DPWS de version 1.1. DPWS 1.0 normalise la découverte et la recherche de services par WS-Adressing [Box et al., 2004] de 2004 de Microsoft, quant à DPWS 1.1, lui normalise la version 1.0 de OASIS de WS-Adressing [Gudgin et al., 2006]. Les espaces de noms de ces deux spécifications ne sont pas identiques, rendant donc incompatibles les requêtes SOAP liées à la découverte.

Windows Vista intègre la pile DPWS. Pour les dispositifs, il existe actuellement deux piles libres de droits, disponibles pour les langages C et Java qui sont respectivement WS4D [44] et SOA4D [45].

5.4.3 Synthèse

UPnP et DPWS permettent la construction d'applications dynamiques pour des réseaux locaux et peuvent être implantés dans des dispositifs peu onéreux et à faibles ressources. UPnP et DPWS (*c.f.* table 2.3 « *Caractéristiques de UPnP et de DPWS* », page 56) proposent des solutions pour quitter le réseau proprement, toutefois dans le contexte de réseau domotique ou SOHO, il n'existe pas d'infrastructure assurant la continuité d'alimentation électrique. Sur coupure secteur du dispositif connecté au réseau DPWS ou UPnP, le dispositif n'a pas le temps d'effectuer sa séquence normale de départ du réseau. Si le concepteur de l'application cliente n'a pas pris en compte cette éventualité, le comportement du service consommateur n'est pas déterministe (invocation d'une méthode d'un dispositif inexistant).

DPWS apporte des concepts de sécurité manquant à UPnP et permet de développer des services riches et complexes (*c.f.* table 2.3 « *Caractéristiques de UPnP et de DPWS* », page 56).

44. `http://ws4d.e-technik.uni-rostock.de`
45. `https://forge.soa4d.org/`

CRITÈRES	UPnP	DPWS
Spécification	Propriétaire UPnP	WSDL 1.1 étendu
Type de courtier	Distribué entre tous les dispositifs	Distribué entre tous les dispositifs
Publication de service	Diffusion générale d'événements type arrivée sur le réseau	Diffusion générale d'événements type arrivée sur le réseau
Retrait de service	Diffusion générale d'événements type départ du réseau	Diffusion générale d'événements type départ du réseau
Découverte de services	Découverte active et passive	Découverte active et passive
Notification	Diffusion générale d'événements (mode *multicast*) départ du réseau	Diffusion générale d'événements (mode *multicast*) départ du réseau
Composition	Non supportée	Possible
Protocole de liaison	SOAP 0.9 -1.1/HTTP 1.1	SOAP 1.2/HTTP 1.1
Type de couplage	Fort (style RPC)	Faible (style Document)
Implantation	Tous les langages	Tous les langages

TABLE 2.3 – Caractéristiques de UPnP et de DPWS.

5.5 OSGi™et Apache Felix iPOJO

5.5.1 OSGi™

OSGi™Alliance [OSGi Alliance, 2007] créée en 1999 est composé d'industriels et d'académiques (Adobe, IBM). Leur objectif est de proposer une spécification permettant de simplifier la construction d'applications dynamiques, de gérer le cycle de vie du logiciel et de fournir des interactions entre applications faiblement couplées. OSGi en est à la version 5.

L'approche prise par OSGi est l'approche à service dynamique. Le cycle de vie de l'application prend en compte le déploiement, l'exécution et se termine par la désinstallation des applications. Ces applications, appelées *bundles*, sont téléchargeables sans arrêt et sans redémarrage de la plate-forme OSGi.

La figure 2.19 « *Couches OSGi (source [OSGi Alliance, 2007])* » page 56 est extrait de la documentation OSGi™Alliance et présente les différentes couches OSGi.

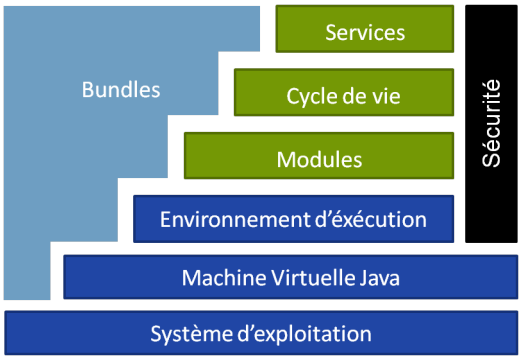


FIGURE 2.19 – Couches OSGi (source [OSGi Alliance, 2007]).

OSGi est composé de couches logicielles et d'entités comme suit :

- les ***bundles*** sont les composants OSGi écrits par des développeurs ;
- les **services** mettent en relation les *bundles* suivant l'approche orientée service ;
- le **cycle de vie** fournit les API pour installer, désinstaller, démarrer et arrêter les *bundles* ;
- les **modules** définissent le chargement et la politique de partage des classes Java ; c'est-à-dire les classes privées aux *bundles* et les classes partagées avec d'autres *bundles* ;
- la **sécurité** gère les aspects non-fonctionnels liés à la sécurité ;
- l'**environnement d'exécution** définit quelles sont les méthodes et classes disponibles dans la plate-forme d'exécution.

OSGi définit une description de services par deux entités :

- un **contrat de niveau syntaxique** caractérisant le pourtour fonctionnel ainsi que la syntaxe d'appel. Ce contrat fournit la liste des méthodes avec leurs signatures et les exceptions qui peuvent être levées.
- un **dictionnaire de propriétés** permettant de caractériser le fournisseur. OSGi ne donne aucune contrainte de taille et de contenu sur ce dictionnaire.

Cette description est publiée dans un annuaire de service, appelé *service registry*.

Un *bundle* peut être fournisseur de services ou consommateur de services. Lorsqu'un *bundle* désire offrir des services à d'autres *bundles*, il doit :

- **implémenter** les méthodes correspondantes à l'interface ;
- **enregistrer** la classe qui implémente l'interface dans le *service registry* ;
- **enregistrer** un dictionnaire de propriétés. Ce dictionnaire peut être vide.

Un *bundle* qui désire consommer un service, doit soit rechercher le service dans le *service registry* (ce mode est dit actif), soit attendre un événement système généré par la plate-forme OSGi (c'est le mode passif). Pour le mode passif de recherche, le consommateur de service fait une demande explicite auprès du *service registry* pour être notifié sur le départ, l'arrivée et/ou la modification du service qui l'intéresse.

La plate-forme OSGi génère les trois événements, suivant ces conditions :

- **départ de service**, lorsque le fournisseur retire du *service registry* son service, il devient donc indisponible à tous les consommateurs ;
- **arrivée de service**, lorsque le fournisseur enregistre son service dans le *service registry* ;
- **modification de service**, lorsque le fournisseur modifie les propriétés du service enregistrées dans le *service registry*.

L'invocation d'un service se fait par appel direct de méthode Java. La figure 2.20 «*Approche orientée service dynamique OSGi*» page 58 représente l'approche orientée service dynamique de OSGi.

Les trois couches Modules/Cycle de vie/Services permettent le partage des classes, des instances et des services suivant l'approche orientée service. Il est à la charge du développeur de faire le partage des services au niveau bundle et de gérer manuellement les interactions entre les services. Cette gestion manuelle des interactions demande au développeur du bundle une connaissance approfondie des événements système de la plate-forme OSGi. Le code du service consommateur doit les prendre en compte et être capable de réagir correctement au départ, à l'arrivée ou à la modification d'un service et cela, à tout instant, pendant l'exécution du service.

La composition est de type structurel gérée par le développeur ; OSGi ne fournit pas d'ADL[46].

46. Acronyme de *Architecture Description Language*.

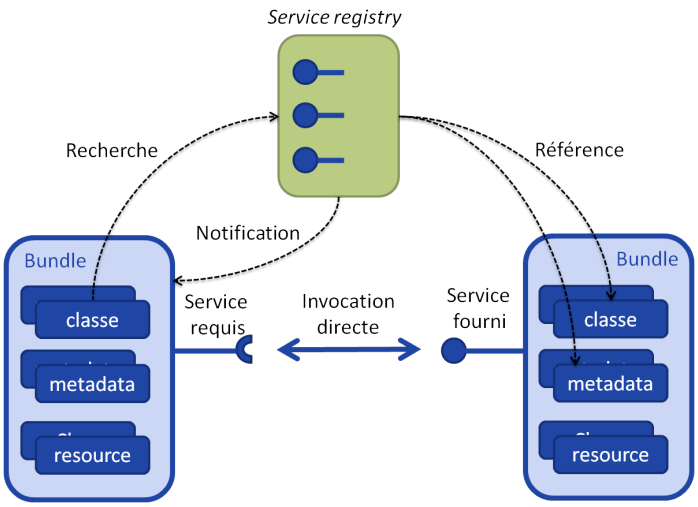


FIGURE 2.20 – Approche orientée service dynamique OSGi.

5.5.2 Apache Felix iPOJO

Apache Felix iPOJO [Escoffier et al., 2007, Escoffier, 2008] est l'acronyme de *Injected Plain Old Java Object*, développé par l'équipe Adèle du Laboratoire d'Informatique de Grenoble. iPOJO est une plate-forme qui s'appuie sur celle d'OSGi (voir la figure 2.21 « *OSGi plate-forme d'accueil de iPOJO* »).

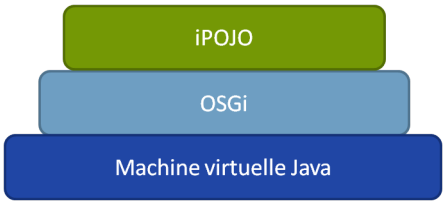


FIGURE 2.21 – OSGi plate-forme d'accueil de iPOJO.

L'objectif de iPOJO est de simplifier la construction des applications à services dynamiques en offrant la composition structurelle de services. Pour cela, iPOJO repose sur quatre socles technologiques :

1. le principe de l'approche orientée composant à service ; c'est-à-dire un composant conteneur (séparation du code fonctionnel de celui non-fonctionnel, encapsulé dans un conteneur). Les liaisons dynamiques entre composants suivent également le principe de l'approche orientée services ;
2. la manipulation du code des classes du composant. Le code du composant peut être intercepté. De même, la manipulation de code des classes permet l'injection de code. iPOJO définit deux méthodes d'injection : l'injection de méthode et l'injection de variable. La manipulation de code nécessaire pour l'injection de code et l'interception de code peut se faire lors de la compilation ou pendant l'exécution du composant ;
3. les composants iPOJO sont décrits (en XML, par annotations ou bien par API) ;
4. iPOJO est un modèle extensible. Le code non-fonctionnel est géré sous forme de *handler*. La plate-forme iPOJO fournit des *handlers* techniques sélectionnables suivant le besoin de l'application. Il est possible de développer ses propres *handlers*.

La figure 2.22 « *Composant iPOJO* » illustre un composant iPOJO constitué d'une agrégation de quatre handlers et d'un code fonctionnel POJO [47].

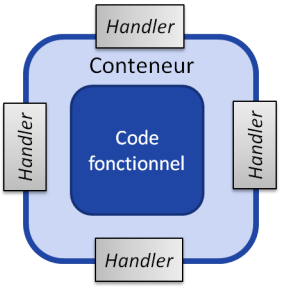


FIGURE 2.22 – Composant iPOJO.

Par défaut, iPOJO fournit quatre handlers techniques, qui sont :
- **un *handler* de publication de services** : il gère tous les aspects nécessaires à la publication de services fournis ;
- **un *handler* de dépendances de services** : il gère la politique de dépendance(s) et le cycle de vie du composant en fonction de la disponibilité du ou des fournisseurs ;
- **un *handler* de gestion du cycle de vie du composant** : il prend en charge le cycle de vie de l'instance du composant en fonction des événements de démarrage et d'arrêt ;
- **un *handler* de configuration** : il réalise la configuration du composant.

Pour publier un service iPOJO, il faut :
- **déclarer** le composant et donner un nom pour qu'il soit manipulé par iPOJO ;
- donner le **nom de la classe Java** contenue dans le composant POJO ;
- donner le **nom de l'interface Java** qui définit les fonctionnalités et la syntaxe d'appel ;
- optionnellement, donner des **propriétés du service fourni**.

Pour consommer un service, un composant iPOJO doit :
- donner l'**interface Java qu'il requiert** ;
- optionnellement, **donner un filtre LDAP** [48] pour ajouter des contraintes de sélection sur les propriétés du service requis ;
- choisir parmi trois modes le **type de dépendance** : statique, dynamique ou priorité dynamique.

iPOJO permet d'instancier un composant de manière :
- statique par XML ou par annotation dans le code source Java ;
- ou dynamique par API iPOJO ou service OSGi.

La figure 2.23 « *Assemblage de composants iPOJO* » illustre des composants iPOJO avec les *handlers* techniques nécessaires.

47. Acronyme de *Plain Of Java Object* ; `http://www.martinfowler.com/bliki/POJO.html`.
48. Acronyme de *Lightweight Directory Access Protocol*.

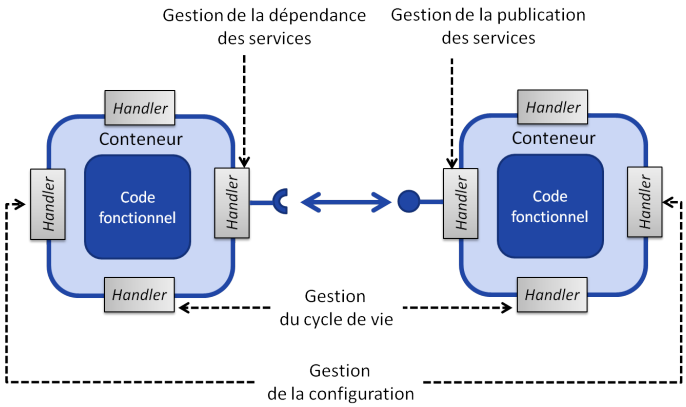


FIGURE 2.23 – Assemblage de composants iPOJO.

iPOJO permet de créer des composants implantant des services de manière simple, il permet de s'affranchir de tous les aspects techniques de OSGi. Le développeur ne prend pas en compte la gestion technique de la plate-forme OSGi (*bundles*, événements, etc.). iPOJO fournit un langage pour la description des composants multi-technologies et ajoute sa propre machine d'exécution au dessus la plate-forme OSGi. Cette plate-forme assure la gestion du cycle de vie des composants iPOJO ainsi que les applications dynamiques.

5.5.3 Synthèse

Le tableau ci-dessous synthétise les différentes caractéristiques d'OSGi et d'iPOJO.

CRITÈRES	OSGi	iPOJO
Spécification	Interface Java + dictionnaire de propriétés (pas de propriété non-fonctionnelle)	Interface Java + dictionnaire de propriétés (pas de propriété non-fonctionnelle), extension possible
Type de courtier	Centralisé, *service registry* d'OSGi	Contextuel, pas de service annuaire centralisé
Publication de service	Manuelle	Automatique par le *handler* de publication de service
Retrait de service	Manuel	Manuel
Découverte de services	Passive et active	Passive et active
Notification	Evénements envoyés aux consommateurs	Automatique, effectuée par le *handler* de dépendances de service
Composition	Structurelle sans ADL	Structurelle avec description de services requis et fournis
Protocole de liaison	Direct : appel de méthode Java	Direct : appel de méthode Java
Type de couplage	Fort	Fort
Implantation	Java	Java

TABLE 2.4 – Caractéristiques de OSGi et de iPOJO.

OSGi est une puissante plate-forme d'administration des bundles, il n'y a pas d'architecture explicite de l'application : il n'offre pas de composition structurelle.

iPOJO utilise la plate-forme OSGi ; par conséquent, il hérite de toutes ses qualités et il compense les manques d'OSGi en permettant la composition structurelle et les liaisons dynamiques entre les services. Il est aussi un modèle extensible.

5.6 Bilan

Les chapitres précédents nous ont permis de passer en revue les différents paradigmes proposés pour résoudre les besoins de réutilisabilité et d'évolution des applications logicielles. L'**approche orientée services** apporte la construction flexible d'applications grâce à la découverte de la description de services, aux propriétés de **faible couplage** et de **liaison tardive** ainsi que par l'agnosticisme de l'environnement d'exécution.

Les applications dynamiques peuvent, elles aussi, hériter des mêmes propriétés, en ajoutant deux primitives d'interactions au paradigme de l'approche orientée service, qui sont le retrait et la notification de services. Les applications sont réalisées par l'agrégation de descriptions de services. La technologie utilisée permet ou non la composition, qu'elle soit structurelle ou fonctionnelle. L'approche orientée service apporte des gains de temps de mise sur le marché des applications constituées de services. Néanmoins, cette approche reste difficile à implanter du fait du nombre de couches logicielles nécessaires. Le manque de maturité des outils et la relative lenteur d'exécution augmentent la difficulté de mise en œuvre de ces d'applications.

La notion de service nécessite une nouvelle démarche de conception des applications. Quel est le processus métier que l'on doit décrire et jusqu'à quel niveau de détails ? Comment identifier les services ? Un service ne doit pas être un ensemble d'API comme ceux d'une bibliothèque logicielle.

Une architecture orientée service doit être acceptée à tous les niveaux d'une entreprise : des dirigeants pour autoriser à exposer à des clients et/ou à des partenaires des services métiers, des architectes pour définir des applications orientées services, des développeurs pour le style de programmation et l'implantation ou l'utilisation des technologies à services, des intégrateurs pour assembler l'application.

Les différentes technologies réalisent l'approche orientée services en fonction des besoins, qui peuvent être de type fonctionnel ou non-fonctionnel. Les **services Web** sont particulièrement bien adaptés pour la construction d'applications distribuées sur Internet ou intranet. Toutefois, les problèmes d'interopérabilités demeurent, mais ils ont été déplacés au niveau de l'implantation. Les différends concernant les standards entre OASIS et le W3C pourraient à terme faire perdre aux services Web leur point fort d'interopérabilité.

UPnP est spécifié pour des applications sur des réseaux de type SOHO sans configuration de dispositifs garantis interopérables. La fonction métier réalisant le service ne peut pas être complexe. **DPWS** est aussi défini pour les réseaux SOHO et permet de mieux gérer les propriétés non-fonctionnelles ainsi que de pouvoir héberger, sur des dispositifs, des fonctions métiers complexes.

OSGi et **iPOJO** sont des technologies pour des applications dynamiques centralisées sur une plate-forme d'exécution Java. OSGi suit les principes de l'approche orientée service pour la construction de ces applications. La plate-forme OSGi apporte aussi une gestion standardisée du cycle de vie logiciel, prenant en compte le déploiement, l'exécution et la désinstallation des unités d'exécution. Quant à iPOJO, il est un complément indispensable à OSGi ; il apporte

une simplification dans le développement des applications dynamiques. Cette simplification est due à deux propriétés : tout d'abord, le type du composant iPOJO, séparant les propriétés fonctionnelles et non-fonctionnelles. De plus, iPOJO supporte la composition structurelle, manquante à la plate-forme OSGi.

5.7 Problèmes ouverts

La figure 2.24 « *Interactions entre acteurs* » montre les interactions possibles d'une application constituées de deux consommateurs et deux fournisseurs de service. Le nombre d'interactions augmente suivant le carré du nombre de fournisseurs (suivant une loi en O^2). L'application peut rapidement devenir difficile à maîtriser.

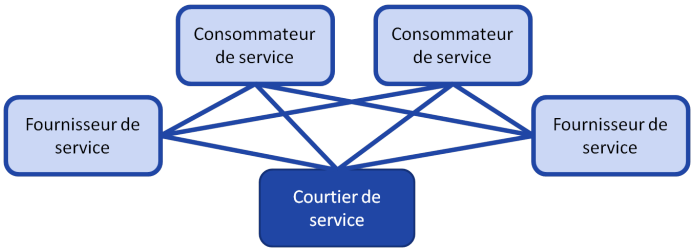


FIGURE 2.24 – Interactions entre acteurs.

Nous avons vu dans le section 3.2 « *Composition par procédés* », page 36, que le moteur d'exécution prend en charge le routage des données entre les services. Ces données transmises nécessitent un alignement syntaxique et sémantique entre les services. Un tel moteur d'exécution prenant en charge ces alignements est difficile à réaliser. Il en va de même pour la prise en compte par le moteur d'exécution des propriétés non-fonctionnelles.

La spécification de service apporte un lien fort entre le client et le fournisseur de service, avec les propriétés de couplage lâche et de liaison tardive. L'évolution des services [Papazoglou, 2008] est une tâche difficile à gérer durant le cycle de vie d'une application. Il est impossible de prévoir toutes les évolutions d'un service dès sa conception.

Le faible couplage permet aux fournisseurs et aux consommateurs de services d'évoluer indépendamment les uns des autres en continuant de garantir la même fonction métier. Deux axes de réduction du couplage sont possibles ; soit étudier le style de communication entre les deux participants, soit définir une architecture. En effet, une communication asynchrone par messages est plus faiblement couplée qu'une communication synchrone par messages. Le producteur du message doit connaître les détails que le consommateur attend pour communiquer (la signature des méthodes, les paramètres, les erreurs). Le producteur doit aussi connaître la localisation du fournisseur.

Intercaler un intermédiaire entre le producteur et le consommateur ; c'est-à-dire couper le lien direct entre les deux parties (voir figure 2.25 « *Producteur et consommateur sont découplés* ») est une architecture qui permet aussi de réduire le couplage.

FIGURE 2.25 – Producteur et consommateur sont découplés.

3

Intégration des services

Dans la première partie, après avoir mis en avant le concept d'intégration, nous étudierons les opérations de médiation et les différentes formes d'intégration. Dans la seconde partie, nous présenterons la dernière génération d'environnement d'intégration permettant l'interopérabilité des services. La troisième partie sera focalisée sur la présentation de quatre solutions existantes dont la solution Cilia qui sera plus particulièrement détaillée. Avant de conclure, nous ferons un comparatif des quatre technologies présentées.

1 Introduction

1.1 Intégration des applications

Nous définissons **l'intégration des applications** comme :

> *un ensemble de méthodes et de moyens permettant la construction de nouvelles applications à partir d'applications ou de dispositifs déjà existants.*

Cette activité d'intégration est aujourd'hui d'une importance capitale pour les entreprises. Ces dernières sont en évolution permanente ; elles doivent être de plus en plus réactives aux évolutions du marché, aux acquisitions, aux scissions, aux changements technologiques, etc. Ces évolutions peuvent être issues de demandes internes, comme le regroupement de sites d'exploitation, le changement de politiques de la direction générale ou encore les évolutions de la législation locale en vigueur.

Le **système d'information** des entreprises - nœud des échanges financiers, techniques, métier - doit évoluer régulièrement. Il est constitué d'outils qui permettent aux acteurs d'une entreprise de communiquer, de facturer, de stocker, de traiter et de diffuser de l'information. Ces systèmes d'information sont constitués d'applications intégrées ; la tâche d'intégration de ces applications est difficile à réaliser et à maintenir. Les sources de difficultés sont, comme nous l'avons déjà énoncé, l'évolution permanente des systèmes d'information. Si l'on observe les types d'application, on constate qu'il en existe deux :

- les applications patrimoniales [1] (en anglais, *legacy*) ;
- les applications nouvellement développées et/ou évolutives.

Les **applications patrimoniales** sont difficilement évolutives (voire pas du tout) ; car soumises à des difficultés d'ordre technique ou financier (un coût de maintenance élevé, un manque de compétences techniques. . .). Les **applications évolutives** offrent une architecture technique permettant des évolutions rapides. Leur agilité d'évolution n'étant pas les mêmes ; des incompréhensions syntaxique et sémantique des données peuvent apparaître.

Revenons à l'architecture d'intégration employée. Elle peut être définie au fil de l'eau (anarchiquement) ou rationnellement. Une architecture d'intégration développée anarchiquement induit des coûts élevés de maintenance et diminue la qualité générale du système d'information. Nous pouvons citer comme principales causes :

- un **fort couplage** entre applications et/ou une communication en mode point à point, induisant un coût de maintenance élevé ;
- une **évolution** d'une application peut affecter le fonctionnement correct d'une application patrimoniale. Une panne générale du système d'information peut, par exemple, être provoquée par une régression d'une application ;
- la **sécurité** n'ayant pas de modèle général, elle peut générer des dysfonctionnements dus, par exemple, à une incompatibilité des systèmes d'authentification ;
- la **surveillance** du bon fonctionnement est difficilement réalisable d'une manière homogène : chaque système apporte ses propres méthodes et outils de surveillance qui permettent de quantifier et qualifier leur fonctionnement individuellement et non du système intégré.

1. L'expression « application patrimoniale » désigne des applications déjà existantes.

1.2 La médiation

Nous allons étudier les principes et les solutions proposées pour obtenir une architecture d'intégration rationnelle et efficace. G. WIEDERHOLD a défini la notion de médiateur pour l'informatique :

> *« a software module that exploits encoded knowledge about some sets or subsets of data to create information for a higher layer of applications »* [Wiederhold, 1992]

Cette définition est à situer dans son contexte historique où la médiation était utilisée pour intégrer des données et des ressources provenant de **systèmes d'information**. Le médiateur selon G. WIEDERHOLD consiste à définir une entité logicielle entre le client, qui interroge un ensemble de sources de données distribuées et hétérogènes, et le serveur, qui répond à la requête. Le but est d'interroger un seul système centralisé et homogène pour le client (l'initiateur de la requête), alors que les sources sont distribuées et hétérogènes. En fonction de la requête du client, le médiateur sélectionne la source d'information. Le médiateur est, par conséquent, spécifique à un domaine d'application donné. On notera qu'un médiateur nécessite d'avoir la connaissance de la **syntaxe** et de la **sémantique** des données dès lors qu'il veut les intégrer. Si l'on résume cette définition, la couche de médiation apporte les caractéristiques essentielles, qui sont les suivantes :

- le **découplage** entre les sources de données hétérogènes et les applications (distribuées et/ou autonomes) ;
- l'ajout **d'aspects non-fonctionnels** entre les applications communicantes ;
- la réduction du **nombre de connexions** entre les applications.

De nos jours, d'autres domaines nécessitent l'intégration des applications : citons, par exemple, la **domotique** [2] ; son défi consiste à intégrer des capteurs-actionneurs et des applications dans un environnement dynamique et hétérogène.

L'intégration a pour objectif de fournir des services à fortes valeurs ajoutées que ce soit à grande échelle, comme les systèmes d'information des entreprises, ou bien dans le domaine de l'informatique pervasive. De plus, quelles que soient les technologies utilisées, l'intégration de services nécessite une couche de **médiation**. La médiation consiste en une série de traitements à effectuer pour permettre la communication entre deux applications.

La figure 3.1 « *Médiation de messages* » est un exemple pour la médiation de messages entre deux applications. Nous exprimons cette médiation sous la forme d'un diagramme de séquences. Illustrons avec un exemple en prenant deux applications A et B qui doivent échanger des informations par messagerie. L'application A émet et reçoit des messages de données au format XML. L'application B émet et reçoit des messages au format CSV [3]. Le protocole de communication entre l'application A et B est assuré par le protocole de messagerie. Par contre, le contenu du message émis par l'application A (*i.e.* XML) n'est pas celui attendu par l'application B (*i.e.* CSV). Nous pouvons affirmer que ces deux services ne sont pas alignés syntaxiquement ; en effet, les informations échangées ne sont pas comprises par les deux applications A et B. Pour résoudre ce problème, nous utilisons le mécanisme de la médiation de messages. Son rôle est d'offrir et d'implanter l'alignement syntaxique entre les applications A et B tout en conservant ces deux applications telles quelles.

2. Lorsque les technologies de l'information et de la communication sont appliquées à une maison on parle de domotique.
3. Acronyme de *Comma-Separated Value*.

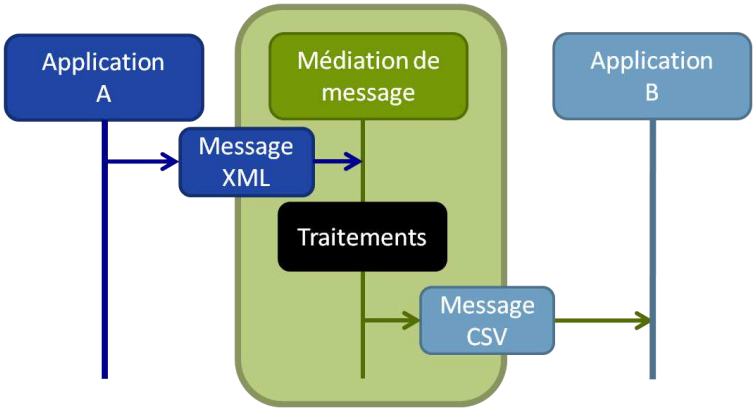


FIGURE 3.1 – Médiation de messages.

L'application A émet un message au format XML à l'entité de *Médiation de message*. Cette entité remplit deux rôles :

- en premier, la génération d'un nouveau message aligné syntaxiquement et sémantiquement. Les données seront mises au format CSV.
- en second, l'acheminement du nouveau message au service destinataire. Dans notre cas, ce sera le service B.

Nous remarquons que pour transformer des données XML en CSV, il est nécessaire que la sémantique des tags XML et celles des lignes/colonnes du CSV soit connue de la fonction traitement.

Il est à noter que la médiation ne se résume pas uniquement à la fonction de médiation de messages entre applications ; elle peut être aussi un service médiateur entre le client et le service. La figure 3.2 « *Médiation de services* » illustre la médiation entre un client et un service fournisseur. La médiation assure la sécurité des échanges. Pour les services Web, ce médiateur de sécurité est en charge de sécuriser les échanges des messages SOAP selon la spécification WS-Security [OASIS, 2006b].

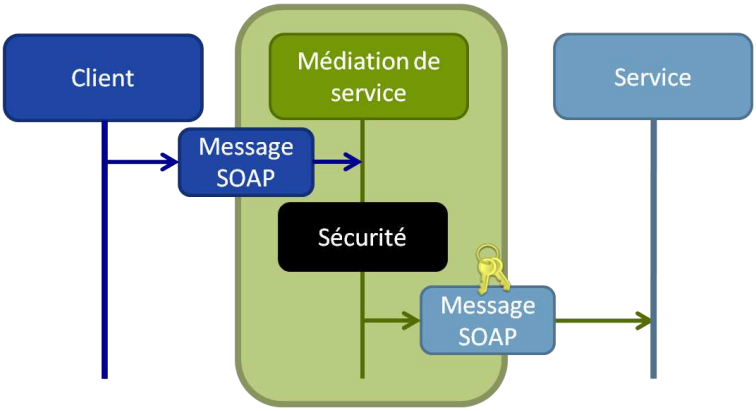


FIGURE 3.2 – Médiation de services.

Les applications évoluent ; par conséquent, la couche de médiation doit elle aussi subir des évolutions. Le coût d'évolution de la couche de médiation est induit par l'évolution du client ou du fournisseur. Et, il faut s'attacher à réduire ce coût. La couche de médiation doit donc offrir des capacités d'évolution (peu onéreuses en temps de développement et de moyens humains et/ou matériels).

1.3 Opérations de médiation

Les opérations de médiation couramment réalisées et listées ci-après sont extraites de notre article [Morand et al., 2011b] :

- **la communication** permet l'interopérabilité entre applications utilisant des protocoles différents ; il s'agit principalement de la transformation de protocoles.
- **l'alignement syntaxique** permet d'aligner le format des données échangées entre applications. Lorsque cela est possible un format pivot peut être utilisé afin de diminuer le nombre de transformations des données.
- **l'alignement sémantique** : le but est de lier des applications ayant des différences dans la sémantique des méthodes, des données et du comportement dans un même contexte. Cette fonction est difficile à réaliser et repose sur des solutions basées sur des liens ontologiques [Shvaiko and Euzenat, 2013].
- **l'ajout de propriétés non-fonctionnelles** permet d'ajouter ou de prendre en compte des propriétés dans les échanges entre applications, par exemple, la sécurité, la disponibilité, la performance... Les propriétés non-fonctionnelles ne peuvent pas toujours être facilement réalisées de manière distribuée entre toutes les applications. La sécurité est une propriété non-fonctionnelle difficilement réalisable et coûteuse en performance si elle est réalisée de manière distribuée entre toutes les applications.
- **la persistance des données** permet de garder une trace de tous les échanges entre applications. Cette fonction peut servir d'aide à la détection et au diagnostic de pannes. La persistance des données peut être aussi une exigence forte en fonction du type d'application intégrée et de la législation en vigueur.
- **la surveillance des échanges** rassemble les données de la couche de médiation et alimente un système de supervision. Les données collectées permettent de quantifier la qualité des échanges entre applications. La couche de médiation peut être aussi supervisée, dans ce cas, elle fournit un ensemble de données caractérisant son fonctionnement, sa configuration, sa qualité de service... Ces données seront elles aussi collectées. Généralement, la supervision utilise deux flux de communication pour la remontée d'informations : un premier pour les données liées aux échanges de messages et un second pour les données caractérisant la couche de médiation.
- **l'ajout de code métier**, comme l'accès à une base de données, est aussi une des possibilités d'utilisation de la couche de médiation. Cette fonction d'ajout est essentielle ; elle permet de répondre rapidement à une nouvelle exigence technique ou fonctionnelle. Cependant, elle apporte une grande confusion : il n'y a plus de séparation du code métier et du code technique. La maintenance et l'évolution sont difficiles à réaliser et, par conséquent, le niveau de robustesse de l'application résultant de l'intégration peut diminuer.

1.4 Patron de médiation

La fonction médiation peut être réalisée par deux patrons d'architecture logicielle :

1. soit par une **fonction de médiation**, appelée *proxy* [4] ;
2. soit par une **infrastructure (plate-forme) d'exécution**.

Nous allons maintenant détailler ces deux patrons qui permettent la réalisation de la médiation.

4. Mot anglais pour désigner un composant logiciel qui se place entre deux autres et assure leurs communications.

1.4.1 Style *Proxy*

Le *proxy* est le style de médiation le plus basique et simple à mettre en œuvre. Ce patron a été identifié pour la première fois dans le domaine de la programmation répartie [Shapiro, 1986]. Son application est lorsqu'un client demande un service fourni par un objet qui est éventuellement distant.

La figure 3.3 « *Proxy de services* » illustre une médiation type *proxy* entre un service client et un service fournisseur. Dans ce cas, le *proxy* virtualise le service ainsi que sa description. Il réalise alors deux fonctions :

- il est client du service fournisseur : le service présente sa réelle description ;
- il est le fournisseur de service pour le client : il présente une description virtuelle du service au client.

FIGURE 3.3 – *Proxy* de services.

Le client invoque le *proxy* qui effectue le ou les traitements d'adaptation et qui invoque, à son tour, les opérations réelles du service. Ce patron est facilement réalisable pour gérer la compatibilité non **ascendante**[5] de service. Le *proxy* prend alors en charge les opérations non supportées et/ou modifiées. Généralement, ce cas d'utilisation est une position d'attente permettant d'éviter la synchronisation de l'évolution (et du déploiement) des services ainsi que des applications clientes.

Dans un autre usage, ce style de médiation opère comme une passerelle utilisée pour appliquer un ensemble de traitements sur les messages et/ou opérations entrants et sortants. Généralement, ce sont des applications de type boîte noire ; le *proxy* implémente les mêmes interfaces que l'application fournisseur. Le *proxy* intercepte les messages/opérations entrants, les traite puis les transmet (ou invoque les opérations) à l'application terminale. Cette méthode reste limitée en capacité d'évolution. Dans ce cadre d'utilisation, le *proxy* est particulièrement bien adapté pour ajouter une propriété **non-fonctionnelle** comme la traçabilité des échanges.

Nous allons présenter un exemple de mécanisme couramment utilisé pour faire du chaînage des services Web, qui illustre les principes du patron *proxy*. La figure 3.4 « *Intermédiaires SOAP* » montre le mécanisme de chemin des messages SOAP.

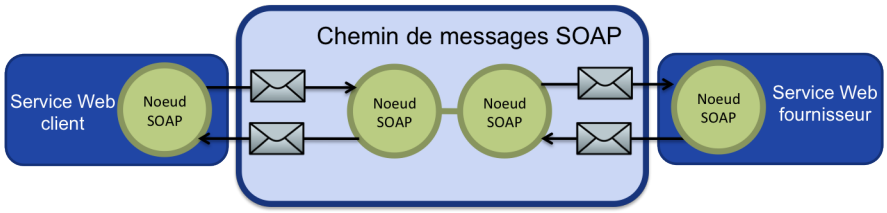


FIGURE 3.4 – Intermédiaires SOAP.

Nous avons plutôt vu dans ce manuscrit que le protocole de messagerie pour les services Web est SOAP. Comme tout protocole de messagerie, SOAP définit dans un message des sections indépendantes les unes avec les autres (en-tête et corps du message). SOAP définit trois

5. Une compatibilité de service est dite ascendante lorsque la version du service implémente toutes les opérations de la version précédente, par contre, le contexte d'exécution doit rester le même.

types de nœuds : un nœud initiateur de message, un ou plusieurs nœuds intermédiaires et un nœud récepteur du message. Les nœuds initiateur et récepteur traitent le message complet SOAP (en-tête et corps du message) alors que le(s) nœud(s) intermédiaire(s) ne traite(nt) que la partie en-tête. Par exemple, il est en charge du filtrage de message, de la persistance, ou bien encore de la sécurité.

Pour les services Web, le principe des nœuds intermédiaires SOAP offre la technologie nécessaire à la réalisation de chaînage de *proxies*. Ces derniers sont au niveau de la couche transport. Or, les protocoles de communication nécessitent des adresses de type réseau ou des adresses logiques. Tous les *proxies* chaînés doivent nécessairement avoir une adresse différente (exigence de la couche transport). L'ajout d'une évolution sur un nœud intermédiaire peut nécessiter le déploiement d'évolution sur d'autres nœuds. Ces évolutions peuvent êtres identiques et, dans ce cas, le déploiement est alors le même pour tous les nœuds. Par contre, si les évolutions ne sont pas identiques pour tous, le déploiement sera difficile. Il faudra identifier les nœuds qui doivent évoluer et connaître leur adresse.

De cet exemple de chaînage de *proxies*, on note deux points :

1. la **robustesse** de l'exécution de l'application réalisée par le chaînage des *proxies* n'est pas facile à maintenir au fur et à mesure des évolutions applicatives ;
2. la **cohérence** des propriétés non-fonctionnelles est difficilement réalisable et, par conséquent, moins facilement maintenable. Ce point met en évidence la difficulté d'implantation des propriétés **non-fonctionnelles** pour des applications distribuées.

Pour conclure, le médiateur style *proxy* offre une simplicité de mise en œuvre et, comme nous l'avons dit plus tôt dans cette section, **le médiateur est spécifique à un domaine d'application donné**. La configuration des composants est réalisable avec beaucoup d'efforts et, par conséquent, ils sont difficilement réutilisables.

1.4.2 Style plate-forme d'exécution

Le style d'architecture **plate-forme d'exécution** permet de dépasser les limites du style *proxy* et notamment de permettre :

- le passage à l'échelle ;
- de réaliser des composants réutilisables ;
- d'étendre les possibilités d'application du principe de **séparation des préoccupations** [6] ;
- de reconfigurer des composants ;
- la réalisation des propriétés **non-fonctionnelles**.

Pour le style architectural plate-forme d'exécution, le principe général de mise en relation du client et du fournisseur est le même que celui du style architectural proxy comme décrit dans la section précédente. La figure 3.5 « *Plate-forme d'exécution* » page 70 illustre les différents éléments d'une plate-forme d'exécution.

6. Démarche de conception qui consiste à séparer dans différentes unités d'exécution les aspects indépendants ou faiblement couplés et à traiter séparément chacun de ces aspects.

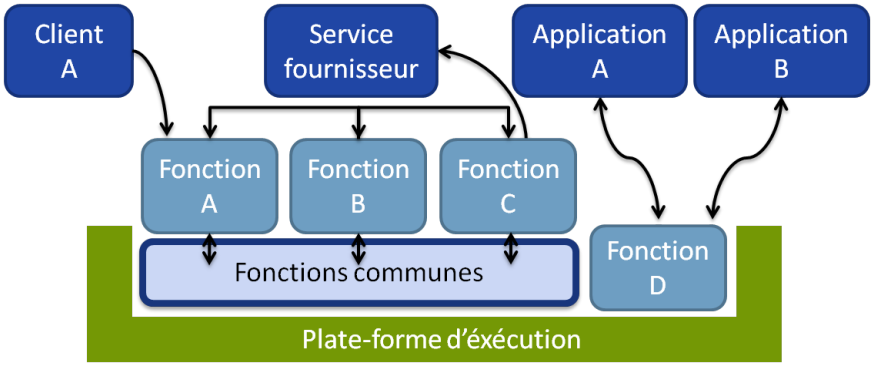


FIGURE 3.5 – Plate-forme d'exécution.

La plate-forme d'exécution a en charge le cycle de vie pour chaque fonction déployée. La plate-forme configure, instancie et détruit les fonctions selon les besoins internes ou externes à l'application. Illustrons ce style de médiation par la technologie CORBA et ses différentes entités. La séparation des préoccupations, mise en évidence dans la figure 3.6 « *Architecture de CORBA* », par les fonctions communes facilite la réutilisabilité des fonctions.

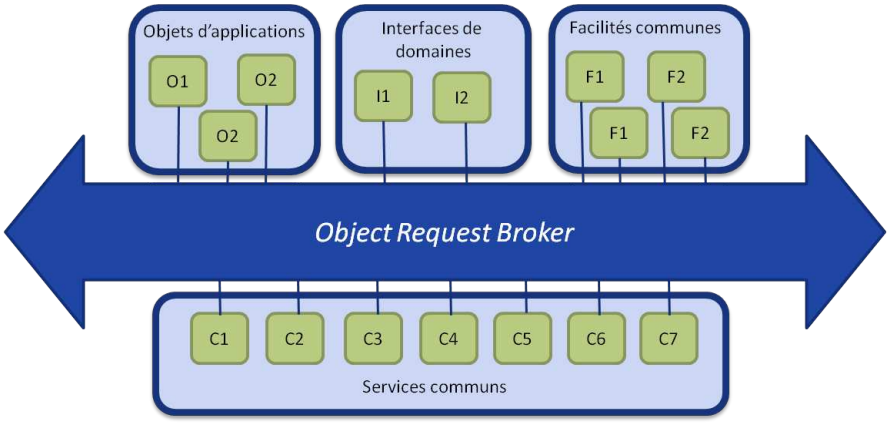


FIGURE 3.6 – Architecture de CORBA.

CORBA est constitué des blocs fonctionnels suivants :

- l'*Object Request Broker* est le bus logiciel, l'infrastructure de communication ;
- les objets d'applications [7] sont des applications mises en places par les développeurs ;
- les interfaces de domaines [8], ce sont des *frameworks* spécialisés pour des domaines d'activités (finances, simulation des données, transport, comptabilité...) ;
- les facilités communes [9], ce sont des *frameworks* pour des traitements courants (interfaces utilisateurs, impression...) ;
- les services communs [10] sont indépendants du domaine d'activité (nommage, gestion du cycle de vie, persistance, notification...).

Pour les applications de médiation exécutées sur une plate-forme d'exécution, deux éléments sont récurrents :

7. En anglais, *Applications Objects*.
8. En anglais, *Domain Interface*.
9. En anglais, *Common Facilities*.
10. En anglais, *Common Object Services*.

- un **modèle de l'exécution** définissant l'application de médiation constituée de médiateurs. Généralement, l'application est décrite dans un langage de haut niveau, par exemple XML, ou bien sous forme graphique ;
- une **plate-forme d'exécution** assure la gestion du cycle de vie des composants qui sont des **médiateurs**. Chaque **médiateur** assure un traitement spécifique de l'information qu'il reçoit. Les médiateurs peuvent être chaînés. On parle alors de **chaîne de médiation**.

Le style plate-forme d'exécution appliqué à la médiation permet donc de séparer clairement le modèle de l'exécution de son exécution.

1.4.3 Bilan

Nous avons commencé, dans cette section, par décrire la médiation de données, son origine et ses principes. Nous avons continué en présentant un bilan des opérations de médiation largement utilisées. Nous avons décrit les deux patrons permettant la réalisation des opérations de médiation. Les deux ayant leurs avantages et inconvénients.

Le style *proxy* est généralement utilisé pour répondre à un besoin spécifique comme une opération de médiation. Un de ses principaux atouts est sa facilité de mise en œuvre. En contre partie, il est difficile de garantir l'évolution fonctionnelle de l'application globale.

Le style plate-forme, quant à lui, répond à des besoins plus complexes (plusieurs fonctions de médiation sont supportées). Il nécessite une plate-forme d'exécution pour gérer le cycle de vie des composants et un modèle de l'exécution. Plus les composants de la plate-forme d'exécution sont spécialisés dans le domaine de la médiation, plus ils seront faciles à développer, à mettre en œuvre, à assembler et finalement à réutiliser.

Nous avons vu précédemment que les services Web permettent la construction d'applications agiles et flexibles. Cependant, les évolutions fonctionnelles sont inévitables. Pour les services Web, cela consiste à faire évoluer la description du service (*i.e.* le WSDL). Dans cet écosystème des services Web, M. P. PAPAZOGLOU [Papazoglou, 2008] préconise d'utiliser un modèle d'intégration de services.Ce modèle, parmi d'autres propriétés, permet d'effectuer :

- une intégration entre services fournisseurs et consommateurs ;
- l'interception de messages ;
- la transformation des messages ;
- le routage des messages ;
- l'utilisation des patrons de messagerie.

Il y a donc clairement, de la part de M. P. PAPAZOGLOU, une proposition de médiation entre services Web basée sur le style plate-forme.

2 Enterprise Service Bus

Nous allons, dans cette section, présenter les *Enterprise Service Bus* (ESB) ; ils sont l'architecture d'intégration de services la plus évoluée à ce jour. Un ESB est le résultat de l'approche orientée service et des *middleware* de messagerie. Nous allons présenter les caractéristiques majeures de la génération précédente aux ESB, à savoir les *Enterprise Application Integration* (EAI) pour comprendre leurs avantages et défauts majeurs. Nous continuerons par la définition des ESB, en présentant les grands principes des *middleware* à messagerie. Nous avancerons dans cette section dans la description des architectures techniques mises en œuvre pour l'implantation des ESB. Ensuite, nous présenterons les patrons d'intégration des ESB utilisés pour intégrer des services à grande échelle.

2.1 Enterprise Application Integration

Un *Enterprise Application Integration* (EAI) est un environnement d'intégration permettant de faire inter-opérer des applications hétérogènes ; il repose sur une architecture *hub-and-spoke* [11] et intègre des protocoles de communication. La figure 3.7 « *Architecture hub-and-spoke* » illustre cette architecture centralisée. Toutes les communications entre les différentes unités (A/B/C/D/E) passent par le *hub* central.

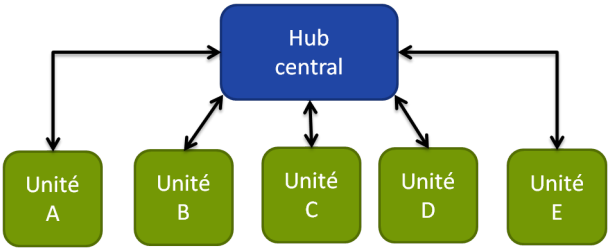


FIGURE 3.7 – Architecture *hub-and-spoke*.

Les EAI transforment des messages de communication dans des formats appropriés afin de pouvoir les traiter et assurer leurs routages vers des destinataires. Les EAI nécessitent trois fonctions :

1. **la transformation des messages** à la volée. Les applications communiquent selon leur protocole de communication et leur format de données. Cette fonction de transformation des messages est en charge de communiquer avec l'application et de changer les messages au format attendu par l'application EAI. Par contre, la signification du message n'est pas modifiée.
2. **le routage intelligent**, appelé aussi routage dynamique. Le destinataire est sélectionné dynamiquement selon le contenu du message.
3. **un moteur de règles**, qui décrit la logique métier d'intégration. Combiné avec le routage dynamique, ce moteur fournit la logique d'intégration des applications.

Les EAI ont des coûts de développement, d'intégration et d'évolution importants. Ils sont généralement des applications propriétaires et monolithiques.

11. Architecture logicielle permettant, à partir d'un point de connexion central, d'atteindre tous les points de connexion.

De l'utilisation de cette technique de transformation de données a émergé des patrons. Cette famille de patrons est appelée EIP [12]. Ils ont été recensés et, finalement, décrits dans le livre « *Enterprise Integration Patterns : Designing, Building, and Deploying Messaging Solutions* » de G. HOHPE et B. WOOLF [Hohpe and Woolf, 2003].

L'architecture des EAI s'applique facilement à l'organisation des entreprises ; car ces dernières sont constituées de départements ou d'unités [13] (voir la figure 3.7 « *Architecture hub-and-spoke* »). Chaque unité assure l'échange d'informations au travers de l'application, cœur des échanges.

Une vision unifiée de ces différentes unités passe nécessairement par une intégration des données. L'application réalisant l'application d'intégration dans une architecture *hub-and-spoke* peut être située à deux niveaux :

1. unité : une unité particulière est en charge d'assurer l'intégration des données de toutes les autres unités ;
2. *hub* central : il assure l'intégration en plus de sa fonction de centralisation des communications.

Prenons un exemple pour illustrer ces deux possibilités. Pour les grandes entreprises, les systèmes d'information sont délocalisés. Ils sont historiquement proches de la source des données. Une évolution locale du système d'information permet une adaptation à un besoin local. Chaque système d'information peut ainsi être optimisé selon les exigences locales (adapter un niveau de qualité du trafic des messages, dupliquer un serveur...), ou, plus structurelle, comme la prise en compte d'une évolution de la législation locale en vigueur. Il existe donc un besoin d'agilité et d'adaptation de ces systèmes d'information.

C'est ici que le rôle du *hub* central est crucial. En effet, si le *hub* central prend aussi en charge l'intégration des données, une modification sur une unité locale demande une évolution sur le *hub*. Le risque pris (régression, perte de communication) et le coût financier peuvent être un important frein à une demande d'évolution émanant d'une unité locale.

Les EAI ont été mis en place dans les entreprises pour intégrer de nombreuses applications. Ces systèmes étant lourds (à cause du nombre des applications et du nombre de protocoles afférents) et complexes (de par leurs propriétés non-fonctionnelles attendues), on a vu naître des départements spécialisés dans les entreprises pour les gérer. L'ajout de nouvelles fonctionnalités applicatives nécessite aujourd'hui une grande expertise et de nombreuses précautions, qui sont souvent la source de délais importants ; puisque l'EAI est central pour l'intégration des applications de l'entreprise. En réaction, on a vu apparaître les ESB, qui sont plus légers et spécifiques, en particulier, dédiés aux services.

2.2 Définitions

Pour définir les ESB, nous partons de deux constats établis dans les domaines des approches à services et des *middleware* :

– comme nous l'avons vu dans le section 5.3 « *Services Web* », page 46, les services Web fournissent des solutions d'intégration flexibles, légères et rapides. Ils sont indépendants de la plate-forme d'exécution. Le service est décrit dans un langage de haut niveau le WSDL [14] [Chinnici et al., 2007], le protocole de messagerie est SOAP [Gudgin et al., 2003].

12. Acronyme de *Enterprise Integration Pattern*.
13. En anglais, *business unit*.
14. Acronyme de *Web Services Description Language*

Et, pour terminer ce résumé des services Web, ils sont à ce jour une technologie largement utilisée et répandue.
- les *middleware* de messagerie ont montré toute leur utilité notamment par :
  - le faible couplage entre client et fournisseurs de messages ;
  - les styles de messageries (synchrones et asynchrones) ;
  - les modèles de messageries (point-à-point et publication et abonnement).

Les ESB sont issus de la pratique de l'architecture orientée service et des *middleware* de messagerie. L'architecture fonctionnelle d'un ESB repose sur deux socles technologiques :

1. il utilise les principes de conception de l'architecture orientée service ;
2. un *middleware* de messagerie assure les échanges de messages.

De ces deux points, nous pouvons énoncer une première définition :

Un ESB applique les principes de conception de l'architecture orientée services déclinés à un middleware d'intégration. Il a pour vocation de fournir une infrastructure de connectivité flexible pour l'intégration d'applications et de services.

L'utilisation des standards est le cœur du concept ESB. XML homogénéise le transfert et la communication des messages, il ouvre la possibilité à l'utilisation d'autres standards basés sur XML comme : XSLT [15] [Kay et al., 2007] pour la transformation de messages, XPATH [16] [Berglund et al., 2007] pour de la recherche et XQuery [Chamberlin, 2002] pour extraire et effectuer des modifications des informations d'un document XML et reconstruire un document XML.

De même, les protocoles de l'Internet - HTTP pour la couche transport, SOAP pour la partie communication par messagerie et WSDL comme langage de description des services - sont, eux aussi, largement implantés par les ESB et par presque tous les dispositifs et les applications développés de nos jours. C'est cette adoption des standard qui est à la base de l'interconnexion des dispositifs, des applications et des services hétérogènes.

Un ESB est une **infrastructure d'intégration** de services et d'applications, il fournit un **annuaire** pour enregistrer les **services** et un ensemble de **services techniques** (voir figure 3.8 « *Architecture ESB* »). Ces derniers sont des applications comme la gestion de la sécurité, la gestion de la qualité de services, la surveillance du trafic des messages, etc.

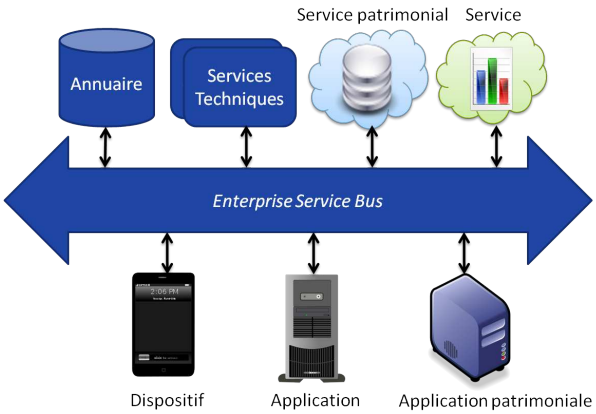


FIGURE 3.8 – Architecture ESB.

15. Acronyme de *eXtensible Stylesheet Language Transformations*.
16. Acronyme de *XML Path Language*.

L'infrastructure d'un ESB est un bus de communication d'échanges de données inter-applications, services et dispositifs. Généralement, le bus de communication offre des capacités de messagerie de style synchrone ou asynchrone. Le style synchrone est une messagerie unidirectionnelle ; c'est-à-dire un mode requête/réponse ou bien pair-à-pair. Le style asynchrone est un mécanisme de messagerie qui repose sur un mécanisme de files d'attente ou bien par un mécanisme de publication et abonnement [17]. Certains mécanismes additionnels peuvent être ajoutés tels des services de persistance. La persistance pour un ESB consiste, par exemple, à archiver un message dans une file d'attente jusqu'à ce que le destinataire soit prêt à le recevoir.

Les applications (services, dispositifs) ne sont pas directement liées au bus de communication. Ils sont découplés par une interface, elle est identique pour tous (le bus de communication n'est pas lié aux technologies et aux protocoles de communications des applications intégrées). Un ESB effectue des opérations de médiation sur les messages. Ces opérations de médiation sont souvent organisées en **chaînes de médiation**, elles transportent les messages du client au fournisseur et acheminent la réponse du fournisseur au client. Les chaînes de médiation sont composées de médiateurs chaînés les uns avec les autres. Et, finalement, un médiateur est l'entité qui réalise une fonction de traitement sur le message.

Terminons cette description générale d'un ESB par ses principales caractéristiques. Elles sont extraites de notre article [Morand et al., 2011a] :

- **leger** : l'application doit être facile à déployer et à maintenir : ne plus suivre le modèle d'application monolithique et coûteuses des EAI. Composer des services techniques suivant les besoins permet de réduire l'empreinte mémoire et la taille du code.
- **efficace** : la transformation de protocoles, la communication sur le bus, l'alignement syntaxique, l'exécution des propriétés non-fonctionnelles et toutes les tâches effectuées par l'ESB ne doivent pas dégrader le service global fourni par l'intégration.
- **facile à installer, à gérer et à utiliser** : la création de nouveaux services, leurs déploiements leurs intégrations nécessitent une configuration régulière des ESB. Les ESB doivent donc être faciles à installer et à faire évoluer.
- **flexible** : l'ESB doit supporter des ajouts, des retraits et des modifications des propriétés non-fonctionnelles ainsi que fonctionnelles. Une modification en ligne (à l'exécution) est toujours plus aisée pour un administrateur (celui qui effectue la modification) et pour les utilisateurs (il n'y a pas d'arrêt d'exploitation, juste une indisponibilité partielle du système intégré). Une opération hors-ligne (modification statique) nécessite trois phases : un arrêt de l'application en exécution, un déploiement de la nouvelle application et son redémarrage. Une application d'intégration à l'arrêt stoppe les échanges de toutes les applications intégrées, les utilisateurs auront le symptôme du trou noir, redouté par tous.
- **gestion des erreurs** : dès lors que des applications et services communiquent, il y a des risques de pertes de communications, qui peuvent être causées par un débordement de la capacité de traitement du bus de communication ; un message mal construit ; une réponse incorrecte. Un ESB se doit d'être robuste et d'appliquer une politique générale de gestion de ces erreurs. Centraliser la gestion des erreurs de communication est une composante essentielle dans la robustesse générale de l'application d'intégration.

Dans tous les cas, un ESB est une plate-forme à composants (logiciel) permettant l'implantation et l'exécution des EIP. La plate-forme est nécessaire pour atteindre des objectifs de modularité

17. En anglais, *publish/subscribe*.

et de réutilisation des composants (logiciel), d'ajout de propriétés non-fonctionnelles et de flexibilité. L'implantation et les exécutions des EIP sont issues de la bonne pratique des EAI dans le domaine de la médiation de messages.

2.3 Implantation

Nous avons présenté les concepts généraux d'un ESB et ses principales caractéristiques, attachons nous maintenant à décrire les principes généraux d'implantation. Les ESB sont généralement implantés selon l'un des trois principes :

1. soit développer une application au-dessus de **serveurs J2EE** [18] ;
2. soit développer une plate-forme spécialisée dans le domaine de la médiation ;
3. soit composer des applications spécialisées existantes.

La première approche d'implantation consiste à construire des applications au-dessus de la plate-forme J2EE. J2EE est un environnement pour développer, déployer et exécuter des applications réparties pour les entreprises. Le contexte d'exécution pour les entreprises nécessite d'assurer des niveaux de qualité, de performance et de services. Une plate-forme J2EE offre :

- une norme de spécification qui porte sur l'infrastructure d'exécution, la gestion des applications et des API ;
- un serveur d'applications dont le rôle est la gestion des sessions utilisateurs, la gestion des contextes clients. Citons deux exemples d'application de référence : Tomcat [19] de la fondation Apache et WebSphere [20] de IBM ;
- un ensemble de services extensibles, par exemple, les plus couramment utilisés sont : JMX [21] pour la supervision, JCA [22] pour les connexions aux systèmes d'information, JTA [23]/JTS [24] pour la gestion des transactions.

L'intérêt principal de cette approche est l'héritage du modèle de développement de l'infrastructure J2EE, de ses services applicatifs et de ses connectivités.

Les applications de médiation construites au-dessus de J2EE sont souvent des applications importantes en taille de code. Le modèle en architectures multi-niveaux apporté par J2EE ne convient pas pour le développement d'application de médiation (manque de performances pour des méthodes appelées à distance, architectures techniques très complexes, difficulté d'utilisation des services techniques).

Une seconde approche d'implantation consiste à écrire des plates-formes de médiation spécialisées. Cette approche nécessite une définition de l'architecture d'implantation, la définition des chaînes de médiation, des médiateurs, du bus de communication et d'une interface standard unique entre le bus et les applications (services et dispositifs) qui seront intégrés.

Cette approche est la plus longue en temps de développement, mais elle est celle qui répond le mieux aux besoins. Les besoins peuvent être de tout types :

18. Acronyme de *Java 2Enterprise Edition*.
19. `http://tomcat.apache.org/`
20. `http://www-01.ibm.com/software/fr/websphere/`
21. Acronyme de *Java Management Extensions*.
22. Acronyme de *Java Connector Architecture*.
23. Acronyme de *Java Transaction API*.
24. Acronyme de *Java Transaction Service*.

– optimiser la taille de l'ESB pour l'embarquer dans des dispositifs à faible ressource ;
– utiliser un bus de communication standard dans l'entreprise ;
– ...

La troisième approche d'implantation consiste à composer des applications. Chaque application apporte sa spécificité. Nous pouvons donner comme exemple de composition le suivant : Apache serviceMix[25] (voir figure 3.9 « *Approche par composition d'applications* ») pour faciliter l'intégration, Apache ActiveMQ[26] comme middleware de messagerie, Apache Camel[27] pour l'implantation des EIP et Apache CXF[28] pour offrir la connectivité avec des services Web.

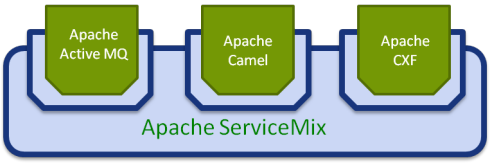


FIGURE 3.9 – Approche par composition d'applications.

Nous avons vu qu'avec un ESB les services intégrés sont faiblement couplés et ceci principalement pour deux raisons : tous les échanges passent par le bus de communication, l'ESB présente une interface unique pour tous les services, rendant le bus agnostique du protocole de communication. Les ESB apportent aussi une solution pour l'intégration des services et des applications à grande échelle. Ils peuvent être fédérés. La fédération permet de scinder la logique d'intégration globale en plusieurs logiques facilement gérables. L'intégration peut être réalisées par étapes incrémentales et par pas indépendants les uns avec les autres.

2.4 Patrons de fédération

L'intégration des services et des applications à grande échelle ne peut être réalisée qu'en fédérant des ESB. Le terme de distribution des ESB est également utilisé. Nous utilisons dans ce document le terme de fédération.

Intégrer des centaines voire des milliers d'applications est difficilement gérable. Plusieurs causes peuvent être citées :

1. **la fiabilité de l'applicatio**n réalisant l'intégration décroît avec le nombre d'applications à intégrer (beaucoup d'erreurs potentielles à prendre en compte).
2. les **performances du bus de messagerie** peuvent diminuer de façon dramatique et générer un dysfonctionnement majeur (panne générale). Par exemple, une surcharge de traitement du bus peut résulter d'une avalanche de messages d'erreurs. Le bus ne pourra plus garantir l'acheminement normal des messages le temps de cette surcharge d'événements.
3. certaines applications prennent en compte la **fraîcheur de la réponse** ; c'est-à-dire si le temps entre l'émission de la requête et la réponse dépasse un délai prédéfini (généralement le terme de promptitude est utilisé pour désigner une fraîcheur de réponse), l'émetteur traitera une erreur à la réception d'une réponse correcte. La fraîcheur de la

25. `http://servicemix.apache.org/`
26. `http://activemq.apache.org/`
27. `http://camel.apache.org/`
28. `http://cxf.apache.org/`

réponse est liée à la charge de l'ESB et à son implémentation (certains mécanismes de cache augmentent le débit des messages traités par l'ESB).

Pour assurer la qualité du service rendu par l'intégration à grande échelle, la fédération des ESB apporte une solution. Nous allons détailler deux patrons de fédération des ESB : direct et centralisé (chacun des deux patrons peut être amélioré selon les exigences).

2.4.1 Le style direct

Le style direct (voir figure 3.10 « *Fédération d'ESB, style direct* ») est un chaînage d'ESB. C'est la plus simple architecture de fédération. Elle apparaît naturellement lors des phases incrémentales d'intégration des services et des applications.

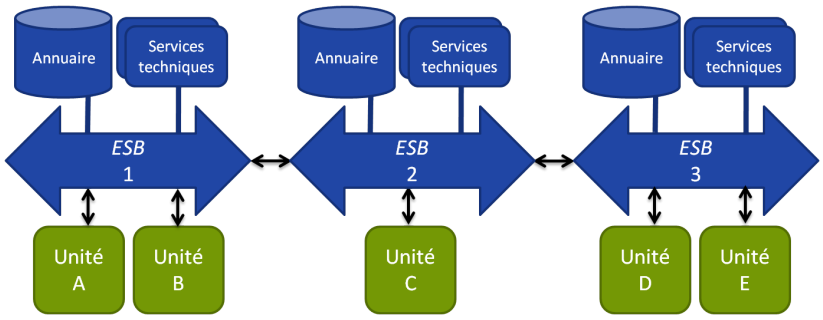


FIGURE 3.10 – Fédération d'ESB, style direct.

Ce style de fédération ne permet pas d'obtenir une vision globale des services. Chaque service est local à son ESB (l'annuaire des services est local), il est difficile de faire émerger une vision architecturale de haut niveau. L'aspect performance n'est pas traité globalement, tout comme la prise en compte des propriétés globales de type non-fonctionnelles.

2.4.2 Le style centralisé

C'est le plus riche et complexe à mettre en œuvre (voir la figure 3.11 « *Fédération d'ESB, style centralisé* » page 79). L'ESB central peut avoir plusieurs rôles :

- il peut être *broker* de service : son rôle est d'exposer de manière sélective les fournisseurs de services ou d'applications. Les interactions entre services sont facilitées par l'ESB central. Ce dernier met en œuvre des services communs comme la sécurité et la transformation des données.
- il peut prendre le rôle du patron d'intégration *hub-and-spoke*. Les demandes des services sont faites à l'ESB local, qui redirige ou non les demandes à l'ESB central ;
- il peut être du type *backbone* ; c'est-à-dire un mixte des deux précédentes solutions.

Plusieurs critères sont à prendre en compte pour définir le rôle de chaque ESB, on trouve généralement des exigences relatives à :

- la visibilité des services ;
- la qualité de service fourni par l'ensemble des ESB fédérés ;
- le degré d'autonomie des ESB locaux.

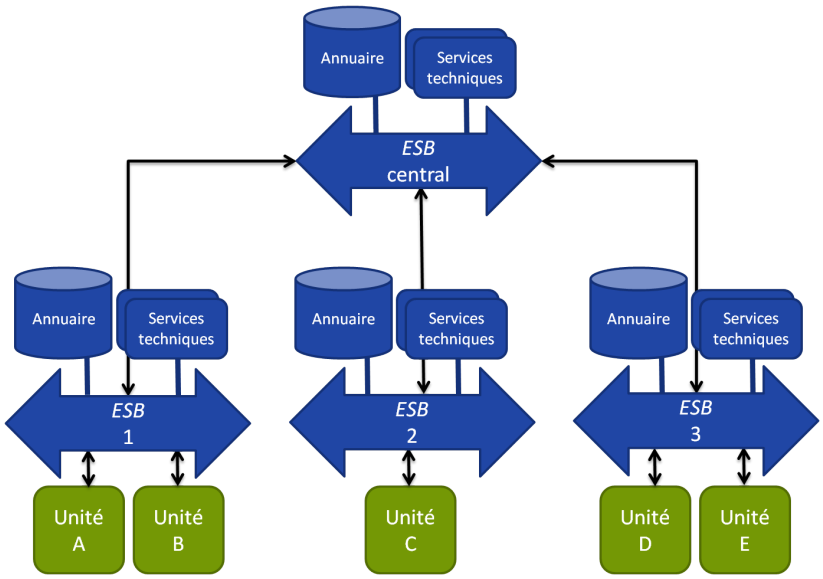


FIGURE 3.11 – Fédération d'ESB, style centralisé.

2.5 Synthèse

M. P. PAPAZOGLOU définit un ESB :

> « *The ESB is an open, standards-based message bus designed to enable the implementation, deployment, and management of SOA-based solutions.* » [Papazoglou et al., 2007]

Cette définition représente la vision historique des ESB lors de l'émergence du concept. Elle correspond à une forte demande des entreprises en réponse à la réduction des coûts des EAI (coût d'investissement et d'exploitation).

Un ESB reprend les bonnes pratiques des EAI : la communication est effectuée par messageries et implante des EIP. Utilisant massivement des technologies à services, il s'appuie naturellement sur des standard du domaine tels que XML, WS-* ... C'est l'approche *open source* [29] et, notamment, la modularité qui a popularisé l'utilisation des ESB.

Dans un autre domaine, l'informatique pervasive exige une forte demande d'intégration. Ces dispositifs sont hétérogènes, dynamiques et/ou mobiles et souvent à ressources logicielles et matérielles limitées. Pour l'informatique pervasive, les caractéristiques présentées dans le chapitre 3 « *Intégration des services* », section 2.2 « *Définitions* », page 73, restent valides. Cependant, l'aspect gestion dynamique de dispositifs hétérogènes devient une exigence supplémentaire pour un ESB.

Nous concluons avec la définition suivante des ESB de C. HÉRAULT *et al.* :

> « *Un ESB est une plate-forme à composant permettant d'implémenter des EIP et d'intégrer des applications, des services et des dispositifs, évoluant dans un environnement dynamique et hétérogène.* » [Hérault et al., 2005]

29. Libre de droits.

3 Solutions existantes

Nous avons vu précédemment dans ce chapitre trois principes d'implantation des ESB : au-dessus de J2EE, une plate-forme spécifique ou une plate-forme composable. Nous allons étudier et comparer quatre technologies implantant des ESB suivant ces principes. Les critères de comparaison, que nous prenons en compte, sont les suivants :

- quels sont les principes de fonctionnement ?
- quelles sont les entités mises en œuvre et leurs rôles ?
- quel est le type de routage des messages (statique ou dynamique) ?
- comment est résolue la communication avec les applications et les services fournisseurs de données et consommateurs de données ?
- comment est décrite l'architecture d'application de médiation (DSL [30], autres) ?
- comment est effectuée la phase de développement des composants de médiation ?
- comment est exécutée l'application de médiation : en *standalone*, sur une plate-forme, dans un conteneur d'application ?
- quel est le degré du dynamisme fourni ; c'est-à-dire quelle est la capacité à supporter des évolutions pendant l'exécution ?

3.1 Spring Integration

3.1.1 Principes et fonctionnement

Spring Integration [31] est un *middleware* d'intégration *open source* de SpringSource. Il repose sur deux bases : le *framework Spring* [32] et le livre *Enterprise Integration Pattern* [Hohpe and Woolf, 2003]. *Spring integration* implémente l'ensemble des patrons décrit dans ce livre.

Le *framework Spring* est un modèle de programmation reprenant le principe des POJO (les classes n'ont aucun lien avec le *framework Spring* et ne nécessitent pas l'implémentation d'interfaces pour être prises en compte par le *framework*). Il prend en charge la création d'objets ; la mise en relation des objets est effectuée par l'intermédiaire d'un fichier ADL. Ce fichier de configuration décrit les objets à créer et leurs relations. Spring apporte aussi une approche plus modulaire que J2EE.

Le livre *Enterprise Integration Pattern* a standardisé le vocabulaire ainsi que les patrons d'intégration de messagerie. Ce livre décrit des patrons d'échanges de messages entre applications. La spécificité de ces patrons est qu'ils sont constitués de trois éléments, qui sont : ***Message*** ; ***Message Channel*** et ***Message Endpoint***. Ce modèle de communication est repris par le *framework Spring* (voir figure 3.12 « *Eléments de Spring Integration* »).

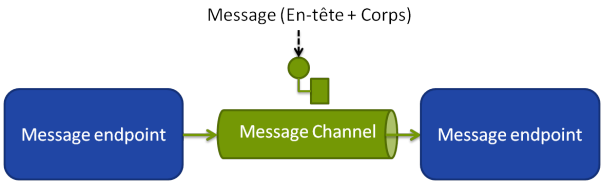


FIGURE 3.12 – Eléments de *Spring Integration*.

30. Acronyme de *Domain Specific Language*.
31. `http://www.springsource.org/spring-integration`
32. `http://www.springsource.org/`

- Le ***message*** est le contrat entre le fournisseur et le receveur. Il est l'unité d'échanges d'informations entre les différents ***messages endpoints***. Un message est constitué : d'un en-tête contenant les métadonnées nécessaires au système de messagerie et d'un corps contenant les informations utiles traitées par le *message endpoint* receveur.
- Le ***message channel*** est l'intermédiaire entre le fournisseur et le consommateur de messages. Le *message channel* gère la destination du message et son mode de livraison ; c'est-à-dire un transfert synchrone/asynchrone et une livraison en point-à-point ou mono-source/multi-consommateurs. Il ne modifie pas le corps du message.
- Le ***message endpoint*** réalise la manipulation de la partie corps du message et les fonctions sur les messages de type : synchronisation [33], filtrage [34], routage [35], séparation [36] et agrégation [37].

3.1.2 Assemblage des composants de médiation

La composition offerte par *Spring Integration* est relativement simple. Les *messages endpoints* sont liés par des *messages channels* selon le patron *pipe-and-filter*. Ce patron est composé de deux éléments : le ***pipe*** qui assure la connexion entre un **filtre** et son suivant ; le filtre qui reçoit les messages du *pipe* entrant, les traite et les émet dans le *pipe* de sortie. Tous les filtres ont la même interface technique pour recevoir des messages et les émettre. Cette interface commune permet la composition des filtres et simplifie la tâche de maintenance et d'évolution des applications. Les *pipes* sont réalisés par les *messages channels* et les *filters* sont réalisés par les *messages endpoints*.

Chaque message endpoint réalise une tâche de médiation (routage de messages, agrégation de données, traitement de messages, etc.). Dans *Spring*, les *messages endpoints* et les *messages channels* sont des *beans Spring*. La composition hérite de la composition des *beans* (par annotation Java et par API). *Spring Integration* ajoute la possibilité de configurer les *beans Spring* avec XML. Spring autorise la définition de plusieurs espaces de nommage XML. Cet ensemble d'espaces de nommage XML a pour objectif de pouvoir diviser une configuration XML en plusieurs fichiers indépendants.

3.1.3 Le cycle de vie

Une application d'intégration *Spring Integration* peut être déployée en *standalone*, sur une plate-forme OSGi ou bien dans un serveur d'applications. La version 2.5 implante le patron *Control Bus* décrit dans le livre *Enterprise Integration Pattern*. Cet élément permet d'effectuer des appels de méthodes à partir du bus de messages, une supervision peut être réalisée en utilisant ce composant *Control Bus.* La gestion de projet est assurée par MAVEN [38] et ANT [39], il n'y a pas d'environnement de développement intégré (IDE [40]) spécialisé dans la distribution de *Spring Integration*. Les composants sont développés en Java avec des fichiers XML de configurations et de paramétrages.

33. Dans la terminologie *Spring Integration*, ce sont les *Services Activator*.
34. En anglais, *filter*.
35. En anglais, *router*.
36. En anglais, *splitter*.
37. En anglais, *aggregator*.
38. `http://maven.apache.org/`
39. `http://ant.apache.org/`
40. Acronyme de *Integrated Development Environment*.

3.1.4 L'exploitation

L'exploitation est relativement simple, *Spring integration* permet de reconfigurer des composants *beans Spring*, qui sont les *messages endpoints* et *messages channels*. L'architecture de l'application en exécution n'est pas modifiable. Pour modifier une application, il est nécessaire d'en déployer une nouvelle. Trois phases sont nécessaires :

1. un arrêt de l'application en exécution ;
2. un déploiement de la nouvelle ;
3. un démarrage de l'application nouvellement déployée.

Dans la phase d'exploitation, le modèle de l'application en exécution n'est pas disponible.

3.2 ServiceMix

3.2.1 Principes et fonctionnement

ServiceMix[41] est un *framework* d'intégration *open source* basé sur OSGi développé par la fondation Apache. L'objectif de *ServiceMix* est de fournir un ESB qui implémente la spécification JBI pour *Java Busines Interface* JSR-208 du *Java Community Process(SM) Group*[42] avec des objectifs de connectivité et de flexibilité. Il permet l'intégration d'autres plates-formes issues de la fondation Apache telles Apache CXF[43] pour le développement de services Web, Camel[44] pour le routage et la médiation de données, Drools[45] qui fournit une plate-forme unifiée et intégrée pour le traitement des règles, de flux de données[46] et d'événements[47]. Pour la connectivité, *ServiceMix* fournit les protocoles largement utilisés tels que JMS, FTP, HTTP, etc.

L'architecture JBI repose sur deux concepts figure 3.13 « *Environnement JBI* » : les composants connectables et le *Normalized Message Router* (NMR). L'accueil des composants connectables est réalisé par les interfaces fournies par cet environnement technique JBI. Leurs communications sont réalisées par le NMR. La fonction conteneur de l'environnement JBI réalise l'intégration du NMR.

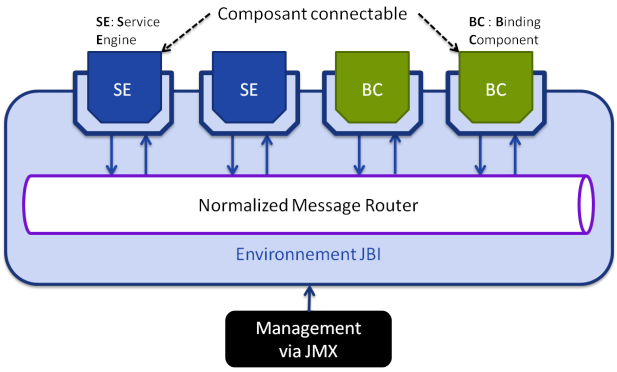


FIGURE 3.13 – Environnement JBI.

41. http://servicemix.apache.org/
42. http://www.jcp.org/en/home/index
43. http://cxf.apache.org/
44. http://camel.apache.org/
45. http://www.jboss.org/drools/
46. En anglais, *workflow*.
47. En anglais, *event processing*.

Cet environnement est administrable au travers des API *Java Management Extension* JMX [48] (intégré dans J2SE dès la version 5.0).

Les composants connectables sont de deux types :
- les ***service engines*** (SE) sont des composants dans l'environnement JBI qui fournissent des fonctionnalités à d'autres composants et consomment des services fournis par d'autres composants ;
- les ***binding components*** (BC) sont des composants JBI qui fournissent la connectivité à d'autres applications (et services) à l'extérieur de l'environnement JBI.

Un *binding component* contient la logique liée à l'infrastructure, le *service engine* contient, quant à lui, la fonction métier. Ces deux composants permettent une claire séparation entre les deux types de composants connectables d'un conteneur JBI.

Le ***Normalized Message Router*** (NMR) est l'intermédiaire entre les composants connectables type fournisseurs et consommateurs ; il fournit l'infrastructure pour la médiation de messages normalisés.Le NMR normalise quatre patrons d'échanges de messages :
- ***In-Only*** : le consommateur émet une requête au fournisseur, sans attendre de réponse de la part de ce dernier ;
- ***Robust-InOnly*** : le consommateur émet une requête au fournisseur, qui peut répondre avec une réponse de type faute s'il ne peut pas traiter la demande ;
- ***In-Out*** : le consommateur émet une requête au fournisseur. Ce dernier doit répondre avec une réponse de type faute s'il ne peut pas traiter la réponse ;
- ***In-Optional-Out*** : le consommateur émet une requête au fournisseur, qui peut renvoyer une réponse.

ServiceMix fournit une liste de composants JBI utilisables pour l'intégration d'applications et de services. Les composants sont de type *service engine* ou *binding components*. Les plus utilisés sont :
- **serviceMix bean**, qui est un **service engine** pour la gestion des composants POJO dans un conteneur JBI ;
- **serviceMix eip**, qui est un *service engine* qui fournit un ensemble (non complet) de patrons d'intégration issu du livre *Entreprise Integration Pattern* ;
- **serviceMix file**, qui est un *binding component* pour les accès fichiers ;
- **serviceMix http**, qui est un *binding component* pour offrir la communication par messagerie SOAP sur la couche transport HTTP.

3.2.2 Assemblage des composants de médiation

Les composants *serviceMix* et leurs compositions sont décrits dans un fichier XML xbean.xml. La configuration des composants est aussi décrite dans un fichier XML.

3.2.3 Le cycle de vie

L'unité de déploiement de *serviceMix* est un *service unit*. Le *service unit* n'est ni plus ni moins qu'un fichier .jar contenant un fichier de configuration xbean.xml. JBI ne spécifie pas le routage dynamique, ni la transformation des données. Les applications *serviceMix* peuvent être : *standalone*, exécutées dans un moteur de servlets comme Tomcat ou Jetty [49] ou exécutées dans un serveur d'applications. Il n'y a pas d'IDE dans la distribution de *ServiceMix*.

48. `http://www.oracle.com/technetwork/java/javase/tech/javamanagement-140525.html`
49. `http://jetty.codehaus.org/jetty/`

Les composants sont en Java avec des fichiers XML de configurations et de paramétrages. La gestion de projet est assurée par MAVEN et ANT.

3.2.4 L'exploitation

Java JMX est l'interface pour gérer la phase exploitation. Cette phase n'autorise pas la modification de l'architecture ni la disponibilité du modèle de l'exécution.

3.3 Mule

3.3.1 Principes et fonctionnement

Mule est une plate-forme d'intégration *open source* de MuleSoft [50]. A partir de la version 3 de Mule, deux **styles** d'architecture d'application sont disponibles : le style **service** et le style **flux de données** [51].

La figure 3.14 « *Architecture de Mule* », illustre le style **service** de Mule. Il est principalement constitué des cinq éléments suivants :

- le ***channel*** qui fournit un moyen pour une application externe de communiquer avec Mule ;
- l'***endpoint*** qui est en charge de la communication entre l'application et les services. Cet élément contient la configuration de la communication ; c'est-à-dire d'où viennent les messages et qui doit les recevoir. L'*endpoint* permet de décrire des services sans connaissance du protocole de communication et sans connaissance de l'émetteur et du récepteur de la donnée.
- ***inbound router*** qui effectue des pré-traitements sur le message ;
- ***outbound router*** qui effectue des post-traitements sur le message et détermine où le message doit être envoyé ;
- ***service component*** qui réalise la logique métier.

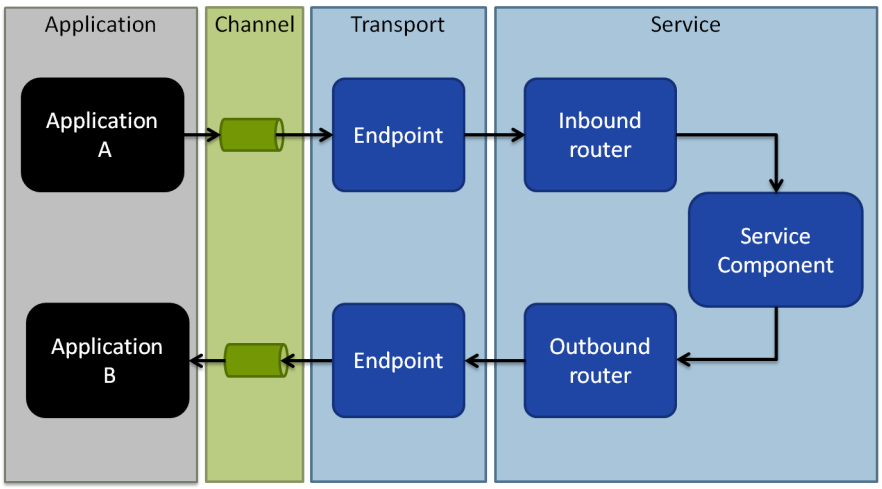


FIGURE 3.14 – Architecture de Mule.

50. `http://www.mulesoft.com/`
51. En anglais, *flow*.

Le style **flux de données** de Mule implémente le patron de messagerie *pipe-and-filter*. Mule n'implante pas l'élément *pipe*, chaque filtre est chaîné directement par référence directe de son successeur. Dans ce style, le *service component* n'est plus utilisé. Les tâches sont chaînées depuis un *endpoint* source jusqu'à un *endpoint* de destination.

Une grande partie des patrons d'intégration décrits dans le livre *Enterprise Integration Pattern* sont disponibles dans la distribution de Mule (le routeurs, les filtres, les transformations). Le routage des messages offert par Mule est dynamique.

La configuration de Mule est constituée de trois espaces de nommage appelés éléments dans la terminologie Mule :

- les **éléments spécifiques** : ce sont les éléments de base de Mule, on retrouve la configuration des éléments EIP (routeurs, transformateurs, filtres), la configuration de la stratégie des exceptions, la configuration des connecteurs http, des services REST, du gestionnaire de sécurité ;
- les **éléments personnalisés** : par exemple, la configuration de protocole spécifique ;
- les **éléments Spring** qui sont la configuration des éléments *Spring Beans* instanciés (point d'ouverture pour implanter des composants spécifiques).

3.3.2 Assemblage des composants de médiation

La composition offerte par Mule est identique à celle de *Spring*, c'est une composition statique basée sur XML.

3.3.3 Le cycle de vie

Mule supporte trois modèles de déploiement : il peut être exécuté en *standalone*, s'exécuter dans un moteur de *servlets* tel que Tomcat et Jetty ou dans un serveur d'applications. Mule supporte la fédération d'instances de Mule (voir section 2.4 « *Patrons de fédération* », page 77). L'interconnexion entre instances de Mule est généralement réalisée par JMS [52].

Mule utilise les mécanismes des exceptions Java pour gérer les erreurs d'exécution entre les divers composants : les *endpoints*, les *inbound* et les *outbound router*, ainsi que le *service component*. Il est possible d'implémenter sa propre stratégie des erreurs (par exemple, capturer les messages en erreur et les rediriger sur un *endpoint* particulier en fonction du type d'erreur).

Le déploiement de Mule est élaboré, c'est un déploiement collaboratif ; les applications sont sauvegardées dans un répertoire centralisé pour toutes les instances de Mule. Ce répertoire stocke les versions et l'historique de leurs modifications.

Les projets Mule sont gérés par MAVEN et ANT, un IDE spécialisé est disponible dans la distribution. Les applications peuvent être surveillées par une console d'administration.

La console d'administration permet d'arrêter et de démarrer les ressources et les services individuellement, de recevoir des notifications d'alertes lorsque les seuils associés à des métriques sont dépassés, d'auditer le trafic du flux de messages et autorise une surveillance des données système, de gérer le déploiement collaboratif de Mule selon le style DevOps [53].

52. Acronyme de *Java Message Service*.
53. Acronyme de *Development* (développement) et de l'abréviation usuelle (Ops) du mot anglais *operations* (exploitation). C'est un principe permettant de favoriser la collaboration entre les équipes d'exploitation et de développement.

Comme toutes les solutions précédentes, Mule supporte des MBeans qui peuvent exposer au travers de l'interface JMX des compteurs de performance, des états et des opérations de paramétrage.

3.3.4 L'exploitation

La modification de topologie de l'application en exécution n'est pas supportée. Il est nécessaire d'arrêter l'application, d'en déployer une nouvelle et de la démarrer. Le modèle de l'exécution n'est pas disponible dans cette phase.

3.4 Cilia

Cilia est un projet *open source* sous licence Apache, développé par l'équipe **Adèle** du **Laboratoire d'Informatique de Grenoble**. Ce projet résulte de la thèse de I. GARCIA [Garcia Garza, 2012].

Cilia est un modèle à composants spécialisés dans la médiation de données. Le défi de Cilia est d'être dynamique, léger, modulaire, en restant simple à utiliser et à déployer. Il apporte le support pour l'implantation des EIP. Il repose sur l'approche des modèles à composants. Cette approche renforce la réutilisabilité des composants et étaye la séparation des préoccupations.

Cilia offre le dynamisme au niveau chaîne de médiation. Il autorise l'ajout, la suppression et la modification des chaînes de médiation pendant leur exécution. Cette caractéristique accorde à Cilia son utilisation dans le domaine de l'informatique pervasive, qui est caractérisée par des dispositifs hétérogènes dynamiques et communicants.

3.4.1 Vision globale

Une application Cilia est un ensemble de composants appelés **médiateurs** faiblement couplés entre eux. Un médiateur réalise une unique opération de médiation. A partir de données reçues, il transmet le résultat aux médiateurs suivants. Cet assemblage de médiateurs forme une **chaîne de médiation**. La chaîne de médiation est la réalisation du patron de messagerie *pipe-and-filter*. Un médiateur est un consommateur mais aussi un producteur de nouveau(x) message(s). Les données entrantes et sortantes des chaînes de médiation sont prises en charge par des composants appelés **adaptateurs**.

Une application de médiation peut être constituée de plusieurs chaînes de médiation indépendantes les unes des autres. L'exécution de l'application de médiation est réalisée par une plate-forme spécialisée qui est en charge du cycle de vie des applications de médiation et du cycle de vie des composants médiateurs et adaptateurs ainsi que de leurs exécutions.

La figure 3.15 « *La plate-forme Cilia* » illustre la plate-forme d'exécution de Cilia et ses composants spécialisés qui sont les **adaptateurs** et les **médiateurs**. Les données entrantes de la chaîne de médiation sont accédées par les adaptateurs selon le protocole de communication de la source de données. Les adaptateurs transforment les données dans le format attendu du premier médiateur de la chaîne de médiation. Les données sont transformées, routées de médiateur en médiateur jusqu'à l'adaptateur de sortie. Ce dernier met à disposition la donnée selon le protocole de communication du consommateur de la donnée.

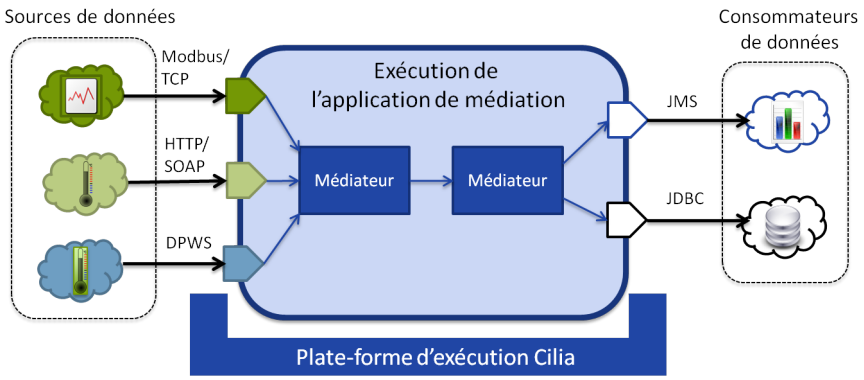


FIGURE 3.15 – La plate-forme Cilia.

Notons que les applications de médiation Cilia peuvent être distribuées. Le MOM [54] Joram [55] a servi de validation pour la fédération en style direct.

Cilia est à la fois un modèle de conception des composants, un gestionnaire du cycle de vie des applications et un modèle d'exécution, comme illustré par la figure 3.16 « *Cilia, vision générale* ».

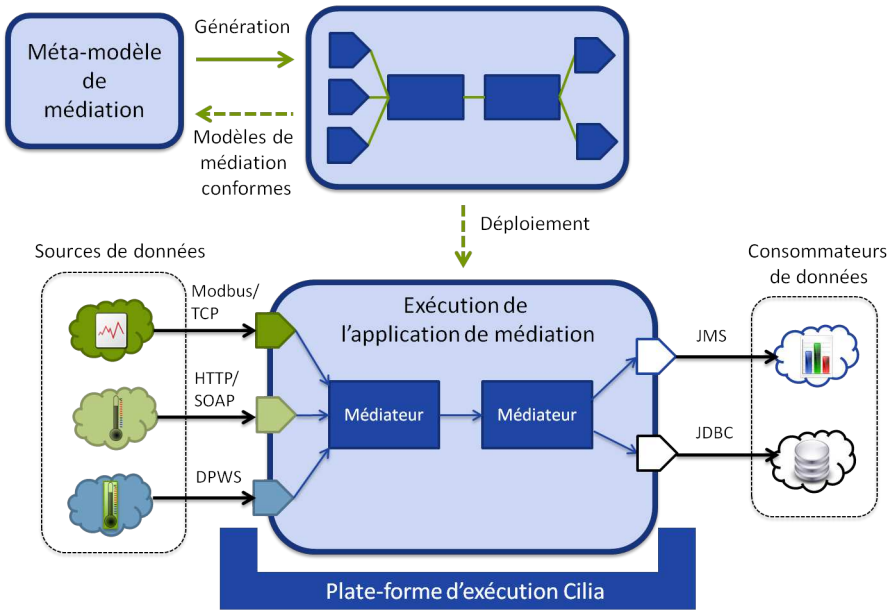


FIGURE 3.16 – Cilia, vision générale.

3.4.2 Modèle de conception

Le modèle de conception de Cilia est entièrement défini par les notions d'**adaptateur**, de **médiateur**, de **connecteur** et de **chaîne de médiation**. Nous allons présenter ces quatre entités.

54. Acronyme de *Message Oriented Middleware*.
55. Joram est une implantation de la spécification JMS 1.1 ; `http://joram.ow2.org/`

Adaptateurs. Les adaptateurs sont en charge de la communication entrante et sortante de la chaîne de médiation. Ils permettent la communication de la chaîne de médiation avec les sources de données et les consommateurs de données. Ces sources (consommatrices) de données sont hétérogènes, elles produisent (consomment) des données avec chacune une syntaxe et une sémantique particulières. Leur protocole de communication est spécifique ; il peut être bien évidemment être ad hoc ou standard.

Le style d'interactions entre les entités externes (sources, consommatrices) et les adaptateurs est soit en mode *push*, soit en mode *pull*. Si le mode d'interaction est en mode *push*, l'initiative de l'échange des données est à la charge de la source des données. En mode *pull*, cette initiative revient au consommateur des données.

Pour répondre à tous les besoins d'interactions avec les sources de données externes et les consommatrices de données externes, Cilia définit trois catégories d'adaptateurs :

- **les adaptateurs d'entrées** : ils assurent l'entrée des messages dans la chaîne de médiation ;
- **les adaptateurs de sorties** : ils assurent la sortie des messages de la chaîne de médiation vers les applications intégrées ;
- **Les adaptateurs d'entrées et de sorties** : ils permettent une communication asynchrone avec l'application. Les interactions sont de type requêtes et réponses. Le message entrant est traité par la chaîne de médiation, le message de sortie de la chaîne est renvoyé comme réponse.

Médiateurs Les médiateurs sont des unités élémentaires de traitement de médiation. I. GARCIA définit les médiateurs ainsi :

> *« Un composant Cilia, ou médiateur, est un module constitué de ports d'entrée et de ports de sortie contenant des données typées et d'une opération de médiation. L'opération de médiation est définie par une condition de déclenchement, une fonction de traitement des données présentes sur les ports d'entrée et par une décision de distribution des résultats vers les ports de sortie. Un médiateur possède une interface d'administration pour la gestion du cycle de vie. Enfin, il peut dépendre de code externe pour réaliser le traitement des données. »* [Garcia Garza, 2012]

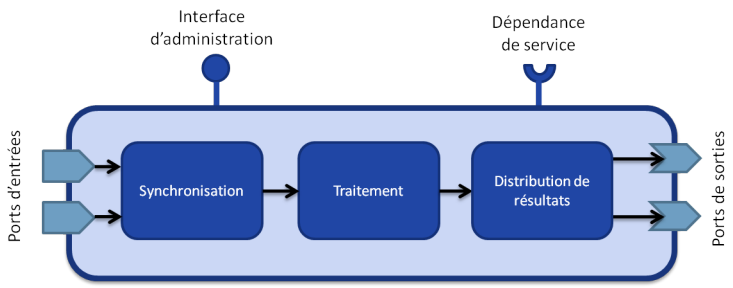


FIGURE 3.17 – Composant médiateur, selon [Garcia Garza, 2012].

Les ports sont des entités typées, ils ne reçoivent que des données de même type. Les ports d'entrées reçoivent les données d'un port de sortie précédent ou d'un **adaptateur** assurant la communication avec une application productrice de données. Les ports de sorties permettent la connexion à des ports d'entrées de **médiateurs** ou à des adaptateurs de sorties. Les adap-

tateurs de sorties assurent la distribution des données aux services intégrés qui sont consommateurs des données. (voir figure 3.17 « *Composant médiateur, selon [Garcia Garza, 2012]* »)

La dépendance de service permet des liaisons avec des services externes aux médiateurs (par exemple, la persistance de messages, des données).

L'interface d'administration assure la gestion du cycle de vie du composant médiateur. Cilia est une plate-forme d'exécution d'application dynamique de médiation ; cette interface permet l'ajout, la modification et le retrait des médiateurs en exécution.

Les trois tâches associées à un composant médiateur sont :
- **la tâche de synchronisation des données entrantes**. Cette fonction permet à un médiateur de récupérer et de synchroniser les données entrantes pour la fonction traitement des données. Elle est appelée *scheduler*.
- **la tâche de traitement** applique des traitements d'intégration aux données reçues. Cette fonction est appelée *processor*.
- **la tâche de distribution** transmet les données traitées au prochain médiateur ou à l'adaptateur de sortie. Elle assure la fonction de routage de distribution des données traitées dans les différents ports de sorties du médiateur. Elle est appelée *dispatcher*.

Un médiateur Cilia permet de regrouper trois logiques nécessaires à un composant de médiation : une logique d'ordonnancement et de synchronisation des données entrantes, une logique de traitement des données et une logique de routage des données. Un médiateur est une unité d'exécution d'une tâche de médiation réalisant ces trois logiques.

Le *scheduler* a deux tâches : récupérer les données des ports d'entrées et les fournir au *processor*. La récupération des données sur les ports d'entrées est faite lorsque la donnée est présente. La livraison des données à la fonction *processor* est dépendante du métier. Pour être générique et répondre à tous les besoins, le *scheduler* fournit une fonction importante : le stockage des données. Ce stockage est nécessaire lorsque les données ne doivent pas être transmises immédiatement au *processor*. Cette fonction de stockage récupére les données soit par ordre d'arrivée et par port d'entrée, soit par ordre d'arrivée indépendamment du port d'entrée. Un deuxième usage de cette fonction est la gestion du flux des données entrantes (*e.g.*, limiter par unité de temps le nombre de messages entrant dans la processeur, sans perte de message). Le *scheduler* réalise des fonctions plus complexes necessitant la synchronisation des données en provenance de différents ports d'entrées.

Le *processor* est la réalisation de la fonction de médiation. Seule cette opération effectue des traitements sur les données. Les opérations classiquement implantées relatives à la médiation de données sont : la transformation des données pour modifier la sémantique des données entrantes, le filtrage des données pour réduire la taille ou la quantité des données, l'enrichissement des données pour ajouter de l'information, la traduction pour modifier des types syntaxiques. Les données résultats sont transmises au *dispatcher*.

Le *dispatcher* traite le routage des données fournies par le *processor*. Il sélectionne le port de sortie du médiateur et délivre les données résultats.

Connecteurs. Les connecteurs sont les entités permettant de lier les médiateurs entre eux ou les adaptateurs aux médiateurs. Les connecteurs Cilia peuvent être de trois types :
- les liaisons **directes** : le port d'entrée du médiateur est le client du port d'entrée du médiateur aval. Ce mode est le classique style client/serveur.
- la liaison de type **message** : un MOM peut être utilisé pour réaliser les ports de sortie avec les ports d'entrée. La liaison hérite de toutes les propriétés fournies par le MOM : la

persistance des messages, l'acquittement des messages, la fiabilité de livraison, le mode synchrone ou asynchrone de livraison de messages, etc.
– la liaison par **mémoire partagée** (*buffer* mémoire).

Chaîne de médiation. Une chaîne de médiation est un assemblage d'adaptateurs, de médiateurs et de connecteurs. I. GARCIA définit une chaîne de médiation comme un modèle de calculs :

> « *Au sein d'une chaîne de médiation, un médiateur reçoit des données de façon asynchrone sur ses ports d'entrée. Lorsqu'une condition de déclenchement est satisfaite, une opération de médiation est appliquée sur les données sélectionnées. Les données résultantes sont déposées sur les ports de sortie et propagées vers les médiateurs connectés.* » [Garcia Garza, 2012]

La chaîne de médiation est une configuration de cet assemblage d'adaptateurs, de médiateurs et de connecteurs. Cilia fournit un ADL spécifique pemettant de décrire le graphe de liaisons des médiateurs par leurs ports. Seuls les ports de sortie sont liés avec des ports d'entrée. Et, les adaptateurs d'entrée peuvent être liés avec des ports d'entrée des mediateurs. Les ports de sortie des médiateurs peuvent être liés avec des adaptateurs de sorties.

3.4.3 Plate-forme d'exécution

La figure 3.18 « *Couches d'implantation de Cilia* », illustre les différentes couches d'implémentation de Cilia. OSGi apporte les mécanismes nécessaires pour la construction d'applications dynamiques et modulaires. La couche iPOJO facilite le développement d'applications à composants orientés services dynamiques. Cilia configure et exécute un ensemble de composants iPOJO. Ces derniers réalisent les fonctions du modèle de conception qui sont les *schedulers*, les *processors*, les *dispatchers*, les adaptateurs et les médiateurs.

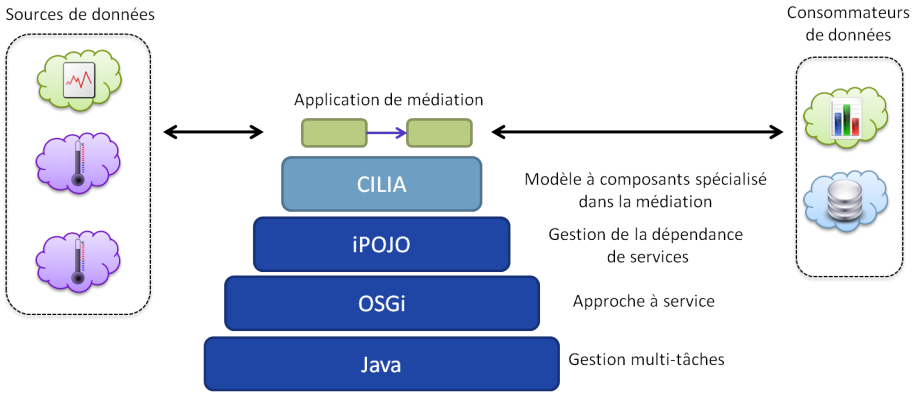


FIGURE 3.18 – Couches d'implantation de Cilia.

Cilia est un *middleware* d'intégration, qui fonctionne dans un environnement en constante évolution qui nécessite de grandes capacités d'adaptabilité. La capacité d'adaptabilité déclinée à un *middleware* peut être réalisée par une API réflective [Maes, 1987]. Une API réflective apporte aussi la propriété de séparation des préoccupations.

Une API réflective est organisée en deux niveaux : un niveau *base-level*, qui fournit les fonctions définies par les spécifications de l'application, et un niveau *meta-level*, qui utilise une représentation du niveau *base-level* pour observer et modifier le niveau *base-level*. Le mécanisme de représentation du niveau *base-level* utilisé par le *meta-level* est appelé réification. La réflexion est celui permettant d'exécuter les fonctions *base-level* au travers du niveau *meta-level*.

Pour réaliser la configuration et l'exécution des composants iPOJO, la plate-forme d'exécution Cilia implante une API réflective, comme illustré par la figure 3.19 « *Réalisation du système réflectif de Cilia* » et décrit dans notre article [Garcia et al., 2011]. L'**API réflective** permet à des chaînes de médiation de réaliser leurs modifications et leur suppression. Les opérations fournies par cette API opèrent sur trois entités :

- la **chaîne** de médiation pour créer, modifier et détruire une chaîne ;
- le **médiateur** pour créer détruire et modifier un médiateur ;
- l'**adaptateur** pour créer détruire et modifier un adaptateur.

Le ***meta-level*** est le conteneur de la description de l'application. Il contient la description et la configuration des médiateurs, des adaptateurs et des connexions entre les composants.

Le **gestionnaire de synchronisation** est l'observateur du *meta-level* ; il surveille les modifications du *meta-level* et réifie les modifications au niveau *base-level*. Ce gestionnaire est en charge de la gestion du cycle de vie des composants *base-level* (la création, la destruction, la paramétrisation des composants). Le *meta-level* et le *base-level* sont découplés par ce gestionnaire.

Le ***base-level*** est constitué de composants de médiation configurés.

Le **gestionnaire de données** a un rôle de persistance des données pour les *schedulers*. Il est externe au *base-level* pour permettre le remplacement à la volée des médiateurs et des adaptateurs.

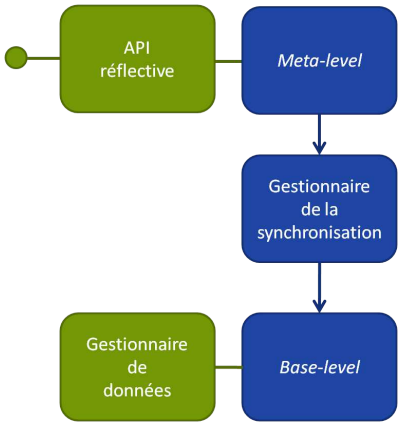


FIGURE 3.19 – Réalisation du système réflectif de Cilia.

L'API réflective est le seul point contact fourni par Cilia pour effectuer des créations, des modifications et des suppressions d'applications de médiation.

3.4.4 Cycle de vie

Le cycle de vie des applications Cilia (*cf.* figure 3.20 « *Cycle de vie d'une application Cilia* ») est séparé en trois grandes phases qui sont : la conception, le déploiement et l'administration.

FIGURE 3.20 – Cycle de vie d'une application Cilia.

La conception. L'objectif de la phase de conception est la production d'artefacts pour l'exécution de chaînes de médiation. Cette phase est organisée en trois sous-phases s'enchaînant les unes après les autres (voir la figure 3.21 « *Phases de la conception d'une application Cilia* »).

FIGURE 3.21 – Phases de la conception d'une application Cilia.

La spécification. Cilia peut exécuter plusieurs chaînes de médiation. La spécification consiste, dans un premier temps, à définir la ou les chaînes de médiation. Le second temps consiste, par chaîne de médiation, en la définition des enchaînements des médiateurs. Ces deux temps permettent de spécifier l'ensemble des médiateurs et des adaptateurs. Ceux déjà existants pourront être réutilisés et paramétrés, les autres devront être spécifiés puis développés.

Pour spécifier un médiateur, il est nécessaire de lui donner un nom (*i.e.*, le nom définit le type du médiateur dans l'application de médiation) et la spécification de ses trois constituants qui sont : le *scheduler*, le *processor* et le *dispatcher*.

L'adaptateur nécessite de définir son type. Il est soit un adaptateur d'entrée, soit de sortie soit d'entrée et sortie. Son nom et la spécification de ses propriétés fonctionnelles terminent sa spécification.

Le développement. Cette phase consiste à développer les algorithmes des constituants *scheduler*, *processor* et *dispatcher* ainsi que les adaptateurs. La séparation des constituants autorise le développement par des équipes séparées ; chacune ayant les compétences nécessaires.

La concrétisation. Cette étape correspond au paramétrage et à la configuration des fonctions communes et/ou non-fonctionnelles, par exemple ; le nombre de *threads* maximum disponibles pour l'application de médiation, les paramétrages des aspects d'authentification, de chiffrement/déchiffrement et d'autorisation d'accès.

Le déploiement. La phase de déploiement consiste à débuter par l'écriture du fichier de déploiement XML. Il définit les types des médiateurs et des adaptateurs qui seront à instancier et à configurer. Ce fichier contient aussi la topologie de la chaîne de médiation.

Le déploiement consiste en la mise à disposition sur la plate-forme d'exécution des artefacts issus de la phase de conception ainsi que de l'artefact en XML définissant les instances des médiateurs et adaptateurs à réaliser et à connecter

L'administration. Cilia est une plate-forme d'exécution d'application de médiation avec la possibilité d'administration des chaînes de médiation, des médiateurs et des adaptateurs.

Cette administration est disponible par l'intermédiaire d'un ensemble d'API qui permettent à la volée de :

- supprimer, déployer, modifier une chaîne. Modifier une chaîne consiste principalement à modifier les liens entre les médiateurs et/ou adaptateurs ;
- arrêter ou démarrer l'exécution d'une chaîne de médiation ;
- modifier les paramètres de configuration des médiateurs et des adaptateurs ;
- instancier un nouveau type adaptateur ou de médiateur ;
- supprimer l'instance d'un adaptateur ou d'un médiateur.

3.5 Comparaisons

3.5.1 Synthèses des technologies

Spring Integration est un conteneur d'applications Java similaire à un serveur d'applications J2EE. Les classes ne doivent pas implémenter une interface pour être prises en compte par le *framework Spring* (au contraire des EJB et de J2EE). Spring apporte :

- l'inversion de contrôle : c'est-à-dire la recherche de dépendances par interrogation du conteneur, la récupération des dépendances avec d'autres objets et l'injection des dépendances : par constructeur, par méthode spécialisée (*setter*) ou par une interface ;
- la programmation orientée aspects ;
- une couche d'abstraction ;
- une suite d'outils *Spring Tool Suite*, pour le développement d'application.

Spring Integration utilise cette couche d'abstraction. L'objectif est de fournir un *framework* pour la construction d'applications d'intégration basées sur la messagerie et les EIP tout en maintenant la séparation des préoccupations. *Spring integration* a plutôt une orientation d'intégration des applications existantes sans bus externe. Les adaptateurs permettent l'évolution vers une architecture distribuée avec un bus central MOM. Cependant, *Spring Integration* ne sépare pas clairement les préoccupations, par exemple, l'élément *endpoint* est en charge du traitement des données et de la communication avec les services externes. *Spring Integration* autorise la reconfiguration des composants pendant leur exécution. Le *control bus* offre un point d'entrée unique pour surveiller et modifier les paramètres des composants.

ServiceMix a pour objectif d'implanter la spécification JBI (*Java Busines Interface* JSR-208). Cette spécification définit les composants, leur rôle et les messages qui circulent sur le bus normalisé et les patrons de messagerie. La séparation des préoccupations offerte par *serviceMix* peut porter à confusion car le *service engin*e (SE) contient la logique métier [56] et la logique de routage. Plusieurs instances de *serviceMix* peuvent être mises en réseaux (Apache ActiveMQ), l'application cliente ne verra qu'un seul *Normalized Message Router*.

L'administration est fournie au travers de l'interface standard JMX. La version 5 de Java livre un outil graphique d'administration des *MBeans* utilisant cette interface. L'outil graphique autorise la surveillance et le paramétrage de ces composants.

Mule a pour l'objectif l'intégration des applications en les découplant. Mule est plutôt une solution d'intégration distribuée, en partie grâce à son mode avancé de déploiement d'applications multi-instances. Les instances de Mule découplent des grappes d'applications à intégrer.

56. Le code utilisateur.

Toutes ces instances communiquent par un bus externe MOM (généralement JMS). La séparation des préoccupations offerte par Mule n'est pas idéale ; l'élément *inbound* de Mule contient la logique métier et la logique de distribution des données.

Cilia est construit sur OSGi et iPOJO, c'est une plate-forme légère et dynamique spécialisée dans l'intégration des données et des services. Cilia est un modèle de conception orienté médiation de données et de services, il sépare les préoccupations. Les composants principaux sont les adaptateurs pour les connexions avec les sources/consommatrices de données, les médiateurs avec les trois logiques : de synchronisation des données, métier et de distribution des données. Les chaînes de médiations réalisent le chaînage des adaptateurs et des médiateurs.

Cilia offre un modèle d'exécution dynamique disponible en exécution. Les applications sont administrables et introspectables. L'instanciation, la modification et la destruction des composants sont réalisables pendant l'exécution de l'application. La structure de l'application d'intégration est, elle aussi, modifiable sans arrêt de l'application de médiation.

Cilia offre le support pour l'implantation des EIP et se distingue des ESB commerciaux principalement par :

1. sa capacité à être embarquable et à gérer le dynamisme, comme pour les applications pervasives ;
2. des solutions de déploiement moins riches que les ESB commerciaux ;
3. l'absence d'une pile WS-* dans sa distribution standard à la différence des ESB commerciaux.

3.5.2 Tableaux comparatifs

Les tableaux ci-après résument l'ensemble des *middleware* d'applications présentés dans la chapitre 3 « *Intégration des services* », section 3.5 « *Comparaisons* », page 93. La composition (table 3.1 « *Conception et déploiement* », page 95) regroupe l'ensemble des propriétés caractérisant l'assemblage des composants. Le cycle de vie (table 3.2 « *Assemblage des composants* », page 95) regroupe les phases conception, déploiement surveillance et reconfiguration des composants. L'exploitation (table 3.3 « *Exploitation* », page 96) est la phase où l'application est en service et peut nécessiter des évolutions. Deux propriétés importantes sont présentées, la capacité à faire évoluer l'application de médiation pendant l'exécution ainsi que la disponibilité du modèle de l'exécution.

4 Synthèse

Ce chapitre nous a permis de présenter les besoins d'une application d'intégration et les deux patrons d'intégration qui sont le style *proxy* et le style plate-forme d'exécution.

Le style *proxy* apporte des solutions pour les problèmes de médiation. Il exécute qu'une seule fonction de médiation. Ce style passe difficilement à l'échelle.

Le style plate-forme permet l'accueil de fonctions communes qu'elles soient fonctionnelles ou non-fonctionnelles, il offre le support technique pour l'exécution des chaînes de médiation, il permet de passer à l'échelle. Les médiateurs et les adaptateurs sont réutilisables pour d'autres applications de médiation.

CONCEPTION ET DEPLOIEMENT	SPRING INTEGRATION	SERVICEMIX	MULE	CILIA
Conception	IDE de *Spring*. Code Java, fichier XML (*Spring* XML)	Code Java et fichier XML	IDE spécialisé. Code Java et fichier XML (*Spring* XML)	Code Java avec fichier de configuration et paramétrage XML
Déploiement	Plate-forme OSGi, en *standalone*, dans un conteneur d'applications	Plate-forme OSGi, en *standalone*, dans un conteneur d'applications	Plate-forme OSGi, en *standalone*, dans un conteneur d'applications	Plate-forme OSGi, en *standalone*, dans un conteneur d'applications

TABLE 3.1 – Conception et déploiement.

ASSEMBLAGE	SPRING INTEGRATION	SERVICEMIX	MULE	CILIA
Communication avec l'externe	Adaptateurs spécialisés	Adaptateurs spécialisés	*Endpoints* liaison par URI ou par configuration XML	Adaptateurs spécialisés
Communication entre composants de médiation	Utilisation des *channels*	Au travers du *Normalized Message Router*	Implicite et directe	Liaisons configurées
Description de l'architecture	Extension de *Spring* basée sur XML	Basée sur XML	Basée sur les flux de communication	DSL Cilia basé sur XML
Taille de la plate-forme	*Spring*=45MB *Spring integration*=3,7MB	59MB	40MB	OSGi=570KB ; iPOJO=272KB ; Cilia=520KB
Style d'interactions	Point-à-point ; Publication/ Souscription	Point-à-point ; Publication/ Souscription	Requête/ Réponse ; *Event Message*	Tous le types de patron d'interactions

TABLE 3.2 – Assemblage des composants.

EXPLOITATION	SPRING INTEGRATION	SERVICEMIX	MULE	CILIA
Surveillance	Flux de données spécialisé disponible par un bus de contrôle	Console JMX pour les statistiques de messages, NMR, MBeans, environnement JBI, flux de messages	Console d'administration pour la surveillance des messages, compteurs de performances	Lecture des paramètres de configuration des médiateurs et adaptateurs par API
Reconfiguration	Reconfiguration des *endpoints* et des *channels* par l'interface JMX, en exécution	Reconfiguration des composants par l'interface JMX en exécution	Hors exécution	Reconfiguration des adaptateurs et médiateurs par API en exécution
Dynamisme	Pas de modification d'architecture en exécution	Pas de modification d'architecture en exécution	Pas de modification d'architecture en exécution	Modification l'architecture en exécution par API. Création, destruction des composants
Modèle de l'exécution	non	non	non	oui

TABLE 3.3 – Exploitation.

Nous avons vu que les EAI sont des solutions monolithiques, onéreuses par le coût des frais d'investissements initiaux et des frais d'exploitation. Les ESB sont des EAI plus évolués s'appuyant sur des standards et qui permettent une construction agile et flexible des applications d'intégration. La fédération des ESB permet la construction d'applications d'intégration par paliers successifs. Elle apporte aussi une solution d'intégration à grande voire à très grande échelle. Les coûts d'investissement initiaux et d'exploitation sont réduits par rapports aux solutions EAI.

Les ESB prennent en compte les exigences non-fonctionnelles comme : la sécurité, la performance, la traçabilité des échanges. Ils apportent aussi la rationalisation des services et permettent leur déploiement rapide.

Parmi les propriétés non-fonctionnelles, la traçabilité des échanges est importante. Elle peut être liée à une exigence légale ou utile à un administrateur. Ces traces aident l'administrateur dans l'optimisation des ressources (maintenir un fonctionnement nominal des capacités du *middleware* et des applications hébergées).

L'administrateur est aussi en charge de la partie disponibilité du service ou de l'application intégrée, par exemple, donner ou non l'accès à un service en fonction d'une plage horaire ou d'un besoin de mise en maintenance. Il doit avoir accès à des fonctions pour modifier le routage de messages, activer/désactiver la persistance des messages, etc.

Nous avons présenté quatre solutions existantes *open source* :

1. **Spring integration**, une solution basée au-dessus de Spring (une implantation de J2EE modulaire) ;

2. **ServiceMix**, une solution conteneur permettant d'unifier des services et des projets issus de la fondation Apache (ActiveMQ, Camel, CXF...) ;
3. **Mule**, une solution spécialisée dans la médiation ;
4. **Cilia**, une solution légère embarquable et dynamique qui est spécialisée dans la médiation avec prise en compte des exigences de l'informatique pervasive.

4

Informatique autonomique

Cette section est articulée en deux parties. La première définit l'informatique autonomique en énonçant ses buts et concepts et en présentant les théories qui sont sources d'influences. Nous verrons que l'informatique autonomique est un domaine ouvert en étant support pour l'implantation de théories autres que l'informatique. Son spectre de résolution de problèmes augmente avec la diversité de ses sources d'influences. La seconde partie présente la conception d'un système autonomique ; nous débuterons par le patron de conception tel que défini par IBM, nous continuerons par la présentation des architectures qui permettent le passage à très grande échelle. Pour terminer ce chapitre, nous exprimerons les besoins des systèmes pour qu'ils puissent être gérés de façon autonomique.

1 Définitions

1.1 Les prémisses de l'informatique autonomique

Le 27 avril 2001, IBM présente à la presse le projet eLiza [IBM Server group, 2002] **qui vise à réduire les coûts d'administration des serveurs** et à faire face à la pénurie (potentielle) des adminstrateurs système. Dans le cadre de ce projet, la solution proposée est de doter les serveurs de capacité d'**auto-administration** selon deux grands axes :

1. l'intervention humaine relative à la gestion des serveurs doit être réduite au maximum ;
2. les serveurs doivent être autonomes et être capable de prendre des décisions d'administration.

Ce projet a permis de définir quatre propriétés : l'auto-configuration, l'auto-réparation, l'auto-optimisation et l'auto-protection. Pour ne pas diminuer les performances des serveurs, eLiza a été développé de telle sorte que les couches les plus basses des systèmes (matériel, système d'exploitation) jusqu'au plus hautes implantent toutes les diverses technologies nécessaires. En outre, des interfaces de haut niveau ont été ajoutées dans eLiza de façon à simplifier la tâche des administrateurs.

Le projet eLiza a servi de substrat à la réflexion des concepts de l'**informatique autonomique**. Dans le prolongement du projet eLiza, IBM démarre un projet *IBM Global Technology Outlook* [Gassmann and Enkel, 2004, Morgan, 2012] qui vise à mettre au point l'informatique autonomique.

1.2 Le manifeste d'IBM

En octobre 2001, Paul HORN, vice-président recherche d'IBM, a diffusé à un ensemble d'industriels et d'académiques un manifeste [Horn, 2001] posant les concepts de l'informatique autonomique [1]. Selon Paul Horn, pour qu'un système informatique soit autonomique, il doit prendre en compte au moins les huit caractéristiques suivantes :

1. *« To be autonomic, a computing system needs to "know itself"- and comprise components that also possess a system identity. »* Un système autonomique doit avoir une connaissance approfondie de ses composants et de son environnement, il doit se **connaître lui-même** ; c'est-à-dire pour tous ses éléments, il doit avoir une connaissance des **états**, des **performances** et de leurs **capacités** à l'exécution.
2. *« An autonomic computing system must configure and reconfigure itself under varying and unpredictable conditions. »* Des **changements** de contexte externe ou interne peuvent induire le déploiement de **nouvelles configurations** ou nécessiter des **reconfigurations** sur les systèmes. Le nombre important des paramètres de ces systèmes modernes rend impossible à un administrateur de définir, dans des temps raisonnables, une nouvelle configuration ou une reconfiguration, puis de la déployer, le tout en fonction d'un contexte dynamique (changeant).
3. *« An autonomic computing system never settles for the status quo - it always looks for ways to optimize its working. »* L'optimisation permanente est une solution pour la gestion difficile des conflits entre les demandes (fonctionnelles ou non-fonctionnelles) et les besoins techniques. Cette optimisation nécessite une **boucle de contrôle** qui surveille des métriques et prend les décisions appropriées. L'optimisation permanente peut remettre

1. En anglais, *Autonomic Computing*.

en cause la stabilité générale du système. Une modification de paramètres peut avoir des effets immédiats ou tardifs sur son fonctionnement. De même, une optimisation peut être la recherche d'une meilleure voie de communication (réduction du temps de transfert des messages pour augmenter le nombre de messages émis), mais peut aussi dégrader le niveau de qualité d'un élément voisin, qui lui n'a pas les mêmes critères de performance (réduction de la consommation mémoire et des files d'attente de messages, par exemple).

4. *« An autonomic computing system must perform something akin to healing - it must be able to recover from routine and extraordianry events that might cause some of its parts to malfunction. »* Le système doit **découvrir** et **localiser** les problèmes existants et les problèmes potentiels. Il détermine une nouvelle configuration et/ou une nouvelle utilisation/réaffectation des ressources.
5. *« A virtual world is no less dangerous than the physical one, so an autonomic computing system must be an expert in self-protection. »* Cette caractéristique apporte les qualités nécessaires pour **résister** aux **attaques** et **intrusions**. Elle garantit le fonctionnement et l'intégrité générale du système.
6. *« An autonomic computing system knows its environment and the context surrounding its activity, and acts accordingly. »* Les systèmes sont communicants. Ils doivent être capables de se **décrire** et de **négocier** les uns avec les autres pour déterminer la **meilleure interaction** possible.
7. *« An autonomic computing system cannot exist in a hermetic environment. »* Les systèmes doivent évoluer dans un environnement hétérogène. Ils doivent se conformer à des standard ouverts. Ils ne peuvent pas être une solution fermée ou difficilement accessible.
8. *« Perhaps most critical for the user, an autonomic computing system will anticipate the optimized resources needed while keeping its complexity hidden. »* Un système autonomique délivre l'information essentielle. Le système doit anticiper les besoins, comme présenter l'information disponible selon un contexte d'une application en exécution. Il doit être capable de s'auto-gérer proactivement.

1.3 Définitions de l'informatique autonomique

Deux définitions de l'informatique autonomique nous semblent importantes. La première est celle de J. O. Kephart et de D. M. Chess, qui met en avant l'apport de l'informatique autonomique dans les phases d'exploitation et de maintenance du cycle de vie des systèmes.

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

Les tâches d'optimisation des systèmes ne seront plus effectuées par les administrateurs système. Ces systèmes, grâce à leur capacité d'auto-administration, seront en permanence à la recherche de leur exécution optimale. Les administrateurs seront ainsi soulagés de la connaissance fine de la dynamique et des réglages à appliquer aux systèmes. Ils pourront alors se consacrer à d'autres tâches.

La seconde définition est celle de A. G. GANEK et T. A. CORBI, qui ajoute la notion d'informatique distribuée. L'informatique autonomique est ainsi élargie aux systèmes informatiques **complexes**, **distribués** et **hétérogènes**.

> *« For decades system components and software have been evolving to deal with the increased complexity of system control, resource sharing, and operational management. [. . .] Autonomic computing is the next logicial evolution of these past trends to address the increasingly complex and distributed computing environments of today »* [Ganek and Corbi, 2003]

Dans cette thèse, on retiendra la définition suivante de l'informatique autonomique :

> *L'informatique autonomique consiste à automatiser la gestion des systèmes pour soulager les administrateurs ; ce dernier reste celui qui fixe les objets que doit atteindre le système géré.*

Une application autonomique s'auto-administre continuellement en fonction de l'évolution de son environnement et des **objectifs fixés par les utilisateurs**. Un gestionnaire autonomique prend en charge cette adaptation. Généralement, ce gestionnaire est externe à la ressource. Il reçoit les objectifs définis par les administrateurs et rend des informations sur l'état de la ressource gérée. Cet état rendu est présenté de sorte que leur tâche de compréhension du système (de la ressource gérée) soit aisée, par exemple, un tableau de bord contextuel regroupant des mesures et des consignes.

Les huit éléments définis dans le manifeste d'IBM sont essentiels dans la définition des systèmes autonomiques. Ils représentent une vision à long terme sur l'évolution d'un système : **quels sont les axes d'évolution à prendre en compte pour rendre des systèmes autonomiques** ? Conscient de la difficulté de la tâche pour arriver à la réalisation des huit éléments du manifeste, IBM a défini cinq degrés. Nous allons maintenant présenter ces cinq niveaux de maturité.

1.4 Niveaux de maturité d'un système autonomique

Les cinq niveaux de maturité, que l'on peut voir comme un degré de sophistication, définis par IBM débutent par le niveau 1 appelé basique et se terminent par le niveau 5 degré exprimant un système dit **autonomique** (voir figure 4.1 « *Augmentation des fonctionnalités autonomiques* ») :

1. **le niveau basique** : le système agrège plusieurs sources de données. L'analyse et la correction de problèmes sont manuelles. L'intervention des experts et des administrateurs est importante.
2. **le niveau géré** : des outils permettent la consolidation des mesures effectuées. Les informations collectées sont étudiées par des administrateurs et des experts ; les actions sont manuelles.
3. **le niveau prédictif** : le système collecte des données et les analyse. Le système propose des actions. Les experts et les administrateurs décident de leur mise en place ou non.
4. **le niveau adaptatif** : le système collecte des données et effectue des corrélations. Il est capable de s'adapter pour des décisions simples. Le système est plus résilient et agile. L'intervention des experts et des administrateurs est réduite.

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

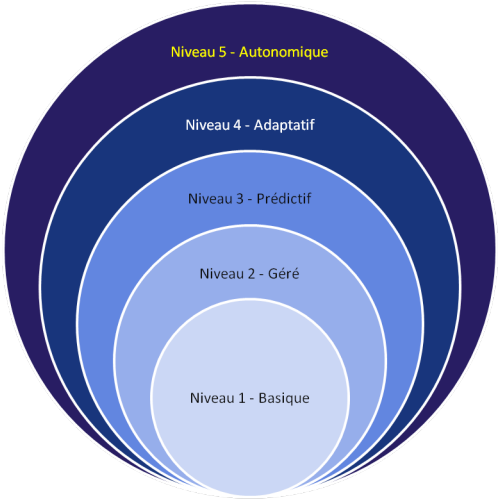


FIGURE 4.1 – Augmentation des fonctionnalités autonomiques.

Cette classification aide les constructeurs à introduire pas à pas l'informatique autonomique dans la définition et la réalisation de leurs systèmes. Ces cinq niveaux illustrent le champ d'application de la boucle autonomique. En effet, plus le périmètre d'application de la boucle autonomique s'agrandit (plus le degré augmente, voir figure 4.1 « *Augmentation des fonctionnalités autonomiques* »), plus la boucle autonomique augmente son périmètre d'application (par exemple, elle traite de la gestion d'un sous-système à la gestion de la fonction métier). A tout moment, les bonnes pratiques des experts, des utilisateurs et des administrateurs sont prises en compte pour aider à définir le niveau supérieur. C'est pour cela que **l'humain est et restera un acteur important de l'informatique autonomique**.

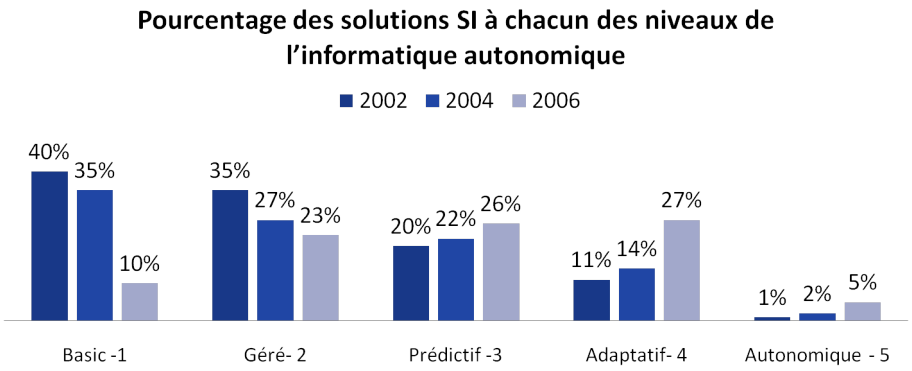


FIGURE 4.2 – Niveau de maturité de l'informatique autonomique dans les entreprises.

La figure 4.2 « *Niveau de maturité de l'informatique autonomique dans les entreprises* » extraite de [Murch, 2004] illustre le pourcentage des solutions SI [2] à chaque niveau de l'informatique autonomique. Ces mesures ont été effectuées par IBM. Pour l'année 2002, 40% des entreprises proposaient des solutions de niveau 5. D'années en années, la figure nous montre une décroissance des solutions de niveau 1 à 4, alors qu'en même temps le niveau 5 ne fait

2. Acronyme de Système d'Information.

que croître. Il faut par ailleurs savoir que le temps de mise sur le marché de solutions est largement supérieur à quatre ans. Nous avons ainsi, avec cette illustration, une mise en évidence que l'informatique autonomique est aussi adaptée aux systèmes patrimoniaux.

Dans les sections suivantes, nous allons détailler ce qu'est l'informatique autonomique. Pour montrer l'étendue du champs d'application de l'informatique autonomique, nous présenterons ses principales sources d'inspiration. Pour comprendre son fonctionnement, nous détaillerons et expliciterons, d'une part, les propriétés des systèmes auto-gérés [3] et, d'autre part, son architecture fonctionnelle telles que définie par IBM. Avant de conclure ce chapitre, nous étudierons les besoins des boucles de contrôle externe aux systèmes auto-gérés.

2 Inspirations

Nous allons, dans cette section, présenter les principales sources d'inspiration de l'informatique autonomique. H. A. MÜLLER nous explicite, dans la définition suivante, que l'informatique autonomique n'est pas une nouvelle discipline de l'informatique, mais une combinaison de bonnes pratiques, de théories, de techniques, etc.

> « *Autonomic computing is not a new field but rather an amalgamation of selected theories and practices from several existing areas including control theory, adaptive algorithms, software agents, robotics, fault-tolerant computing, distributed and real-time systems, machine learning, human-computer interaction (HCI), artificial intelligence, and many more.* » [Müller et al., 2006]

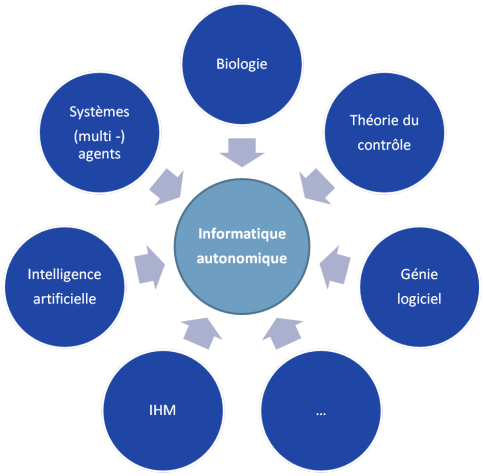


FIGURE 4.3 – Sources d'inspiration de l'informatique autonomique.

La figure 4.3 « *Sources d'inspiration de l'informatique autonomique* » illustre les théories et les pratiques qui sont utilisées par l'informatique autonomique. Plus la source d'inspiration est étendue plus le champs d'application à la résolution des problèmes est large. La biologie est la source originelle d'inspiration de l'informatique autonomique. Nous allons présenter le fonctionnement très général du système nerveux humain, pour appréhender les bases concep-

3. Usuellement regroupées sous le terme de propriétés auto-*.

tuelles sur lesquelles l'informatique autonomique repose. Nous allons, par la suite, présenter quelques théories et pratiques utilisées dans le cadre de l'informatique autonomique.

2.1 La biologie

L'informatique autonomique s'est inspirée du système nerveux autonome du corps humain. Ce système, nommé également végétatif ou involontaire, n'est, en général, pas soumis au contrôle de la volonté. Il assure, par exemple, le rythme des battements cardiaques, la régulation des glandes, la température du corps et coordonne toutes les fonctions vitales. D'un point de vue fonctionnel, le système nerveux autonome est constitué de deux sous-systèmes :

- **le système parasympathique** a pour rôle de ralentir l'activité des organes ;
- **le système sympathique** est la fonction opposée ; c'est-à-dire la mise en alerte de l'organisme et la préparation à l'activité physique et intellectuelle.

Il faut savoir que ces deux sous-systèmes agissent ensemble sur un organe quelconque du corps humain et de manière opposée.

Le système nerveux autonome permet de contrôler l'ensemble des fonctions vitales du corps humain (la concentration du sang, la pression sanguine, la respiration, etc.) **en fonction de l'évolution du contexte extérieur**. Ce contrôle est effectué sans concience.

Le système nerveux autonome a la particularité de s'auto-gérer en maintenant dans leur zone de bon fonctionnement l'ensemble des fonctions vitales. Cette capacité d'auto-gestion a été reprise et modélisée dans d'autres domaines. Chaque fonction vitale est autonome ; si elle ne peut plus réguler, et, par conséquent, garantir son bon fonctionnement (qui peut être vital pour l'organisme), le système nerveux central prend en charge de ramener la fonction vitale dans sa zone de survie. Sans cette délégation d'autonomie à chaque fonction vitale, il serait très difficile pour le système nerveux central de gérer directement le bon fonctionnement de l'ensemble de ces fonctions essentielles pour l'organisme.

2.2 Théorie du contrôle

Le principe d'auto-gestion a été également décrit et modélisé par W. R. Ashby dans le domaine de la théorie du contrôle [Ashby and Stein, 1954]. Cette modélisation a fortement inspiré l'évolution de la cybernétique [Wiener, 1961] et, en cascade, la théorie du contrôle, l'automatique, la robotique, etc. Ce modèle est connu sous le principe de système ultra-stable de W. R. Ashby (voir la figure 4.4 « *Système ultra-stable de W. R.* Ashby » page 106). Il a aussi été repris dans le domaine de l'informatique autonomique [Parashar and Hariri, 2005].

Le fonctionnement de ce système ultra-stable est le suivant : à un moment donné, le **contrôle fixe les objectifs** et sélectionne les paramètres du régulateur. Les variables essentielles représentent les objectifs à atteindre qu'ils soient qualitatifs ou quantitatifs. Le régulateur agit sur le système en fonction des perturbations extérieures. Tant que les variables essentielles sont dans leur plage de bon fonctionnement, le contrôle n'intervient pas. Dans le cas contraire, le contrôle est informé des déviations des variables essentielles. Il modifie les paramètres de réglage du régulateur (la dynamique du système est un facteur important).

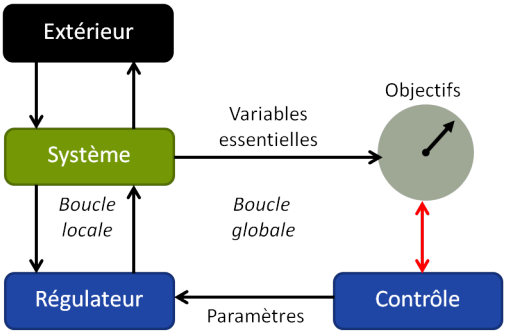


FIGURE 4.4 – Système ultra-stable de W. R. ASHBY.

Ce système est appelé ultra-stable ; car il possède deux boucles de stabilité et de réglage. Le régulateur suit le rythme et maintient la stabilité en fonction des perturbations extérieures (boucle locale). Le contrôle agit lentement, à un deuxième niveau de stabilité, en modifiant les réglages du régulateur (boucle globale). On peut imaginer des boucles de plus de deux niveaux où les niveaux les plus élevés se chargent des régulations les plus lentes qui sont, par conséquent, les effets les plus durables sur le système comme en robotique, par exemple [Hayes-Roth et al., 1995].

La réalisation de système ultra-stable est complexe et coûteuse. Généralement, une simple boucle est implantée (appelée contre-réaction [4]), elle est illustrée par la figure 4.5 « *Contre réaction* ». La boucle globale n'existe plus ; le régulateur ne reçoit plus d'ordre de l'extérieur. En fonction des écarts des variables essentielles, le régulateur agit alors sur le système. Le contrôle est optionnel ; il est utile pour fixer les paramètres du régulateur (mode manuel ou automatique en boucle ouverte).

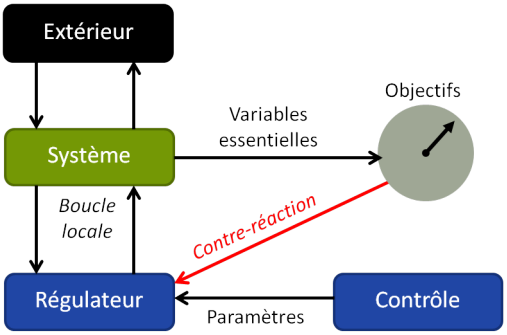


FIGURE 4.5 – Contre réaction.

La **contre-réaction fournit le squelette de chaque élément autonomique** [Ganek, 2003], elle permet de contrôler la stabilité du système en fonction des paramètres locaux.

Néanmoins, il faut souligner que : un système sera sous contrôle si l'on sait, d'une part, fixer des objectifs à atteindre et, d'autre part, atteindre ces objectifs.

4. En anglais, *feedback control* ou *closed loop*.

2.3 Autres sources d'inspiration

Nous avons décrit le fonctionnement d'une boucle ultra-stable de W. R. ASHBY et sa version simplifiée qui est la contre-réaction, intéressons-nous maintenant aux théories et aux technologies sources d'inspiration de l'informatique autonomique plus proche de l'informatique et des mathématiques, comme nous l'avons précédemment montré dans la figure 4.3 « *Sources d'inspiration de l'informatique autonomique* » page 104.

Tout d'abord, le domaine des multi-agents a influencé l'informatique autonomique, notamment les besoins de collaborations entre éléments. Chaque élément, nommé agent, est responsable, pendant l'exécution, de son comportement autonomique et de ses relations pour réaliser ses objectifs [Tesauro et al., 2004]. Historiquement, les premières propriétés décrites d'auto-gestion des systèmes sont issues des agents [Wooldridge et al., 1995]. M. WOOLDRIDGE et N. R. JENNINGS ont identifié les quatre propriétés suivantes :

- **l'autonomie** : les agents évoluent de manière indépendante et sans intervention humaine (ou machine). Ils ont un contrôle de leurs actions et de leurs états internes ;
- **la sociabilité** : les agents communiquent à l'aide d'un langage de communication avec un humain ou une machine ou encore avec un autre agent ;
- **la réactivité** : les agents sont sensibles à leur environnement d'exécution et doivent réagir aux changements ;
- **la pro-activité** : les agents présentent un comportement adapté en fonction des conditions externes. Pour atteindre un but, ils prennent des initiatives.

Toujours dans le même domaine des systèmes multi-agents, S. BUSSMANN sépare différentes fonctions telles que les capteurs, les actionneurs, les protocoles de communication, l'interprétation des données, la décision pour réaliser un objectif et la planification des actions dans un but de gérer le contrôle de la fabrication dans les usines ([Bussmann, 1998]).

Le domaine des probabilités continue aussi à influencer l'informatique autonomique, notamment, avec les techniques des réseaux Bayésiens. Ces derniers permettent de déterminer le meilleur algorithme disponible [Guo, 2003]. Ils sont utilisés dans la réalisation de gestionnaires autonomique probabilistes.

L'optimisation (et la prédiction) permanente des systèmes auto-gérés emprunte diverses techniques comme celles issues de l'intelligence artificielle telles que la recherche tabou [5], l'apprentissage automatique [6]. Ces techniques permettent de contrôler les besoins d'optimisation des objectifs de haut niveau [Aiber et al., 2003].

Toujours dans un souci d'optimisation, une autre technique très largement utilisée, qui donne de remarquables résultats, est celle des fonctions d'utilité [7] [Kephart and Das, 2007]. Citons W. E. WALSH *et al.* [Walsh et al., 2004] qui les utilisent pour optimiser continuellement les ressources d'un système plongé dans un environnement dynamique et hétérogène. La théorie des fonctions d'utilité traite des méthodes pour attribuer une valeur d'**utilité** (une valeur scalaire) à chaque résultat possible. Elle permet de choisir le meilleur plan d'actions basé sur la maximisation de la valeur d'utilité [Keeney, 1993].

5. En anglais, *tabu search*, une méthode de recherche qui, à partir d'une position donnée, explore son voisinage.
6. En anglais, *machine learning*.
7. En anglais, *utility functions*.

L'Interface Homme Machine (IHM) est également une discipline importante de l'informatique autonomique :

> *« [. . .], system behavior must be understandable, or else users and systems cannot work together effectively. »* [Russell et al., 2003]

Etant donné le nombre de domaines d'inspiration, l'informatique autonomique n'est pas le premier grand projet traitant de l'auto-gestion, le tableau 4.1 retrace les projets importants selon [Huebscher and McCann, 2008].

PROJECT NAME	YEAR	INITIATOR	DESCRIPTION
SAS (Situation Awareness System)	1997	DARPA (Defense Advanced Research Projects Agency)	Decentralized self-adaptive (ad-hoc) wireless network of mobile nodes that adapt routing to the changing topology of nodes and adapt communication frequency and bandwidth to environmental and node topology conditions.
DASADA (Dynamic Assembly for Systems Adaptability, Dependability, and Assurance)	2000	DARPA	Introduction of gauges and probes in the architecture of software systems for monitoring the system. An adaptation engine then uses this monitored data to plan and trigger changes in the system, e.g., in order to optimise performance or counteract failure of a component.
AC (Autonomic Computing)	2001	IBM	Compares self-management to the human autonomic system, which autonomously performs unconscious biological tasks. Introduction of the four central self-management properties (self-configuring, self-optimising, self-healing and self-protecting).
SPS (Self-Regenerative Systems)	2003	DARPA	Self-healing (military) computing systems, that react to unitentional errors or attacks.
ANTS (Autonomous NanoTechnology Swarms)	2005	NASA	Architecture consisting of miniaturized, autonomous, reconfigurable components that form structures for deep-space and planetary exploration. Inspired by insect colonies.

TABLE 4.1 – Projets qui ont influencés l'auto-administration, selon [Huebscher and McCann, 2008].

L'informatique pervasive et ubiquitaire avec ces défis d'interconnexions (et de gestion) des réseaux de capteurs et de systèmes rendent la gestion d'un tel réseau difficile aux administrateurs. Des techniques inspirées de la biologie [8] apportent une solution [Suzuki and Suda, 2005] pour la réalisation d'une plate-forme d'exécution d'applications. Des principes issus de l'étude de l'évolution génétique peuvent servir de support pour la définition du cycle de vie des services (applications) [Linner et al., 2007] ; ils apportent des techniques inspirées du fonctionnement des phéromones pour résoudre des problèmes de coordination de services effectuant des tâches distribuées [Fusco et al., 2008].

Comme le signale IBM, il est important de garder à l'esprit qu'il existe une grande différence au niveau de la prise de décision entre l'activité autonomique du corps humain et celui des machines. Les décisions prises par les éléments autonomiques du corps humain sont codées dans nos gênes et nous n'avons pas de prise dessus alors que, pour l'informatique autonomique, ce sont les **professionnels qui choisissent** les tâches à déléguer aux différentes technologies. Ces dernières en auront la charge de l'exécution.

> *« However, there is an important distinction between autonomic activity in the human body and autonomic activities in IT systems. Many of the decisions made by autonomic capabilities in the body are involuntary. In contrast, self-managing autonomic capabilities in computer systems perform tasks that IT professionals choose to delegate to the technology according to policies. Adaptable policy - rather than hard-coded procedure - determines the types of decisions and actions that autonomic capabilities perform. »* [IBM, 2006]

Nous avons vu les principales sources d'inspiration de l'informatique autonomique, nous allons présenter les propriétés d'un système auto-administré.

3 Les propriétés auto-*

Les propriétés générales d'un système autonomique (d'auto-gestion) [9] sont généralement décrites par IBM selon quatre capacités [Kephart and Chess, 2003] et sont usuellement regroupées sous le terme générique auto-* [10] :

- **l'auto-configuration** [11] : les systèmes autonomiques doivent se configurer par eux-mêmes en fonction des objectifs de haut niveau. Ces derniers représentent des objectifs métier. Cette propriété d'auto-configuration n'exprime pas comment faire la configuration ; mais ce qu'il est attendu en terme d'objectifs métier. Le système autonomique s'adapte à son environnement ; il doit, par conséquent, s'auto-configurer en fonction de l'évolution de son contexte environnemental. La difficulté n'est pas uniquement la configuration d'une ressource physique ou logique, mais la configuration d'une infrastructure complète constituée de ressources distribuées et hétérogènes. Par exemple, un ajout d'un serveur dans une infrastructure nécessite deux étapes : sa découverte puis sa configuration. Sa configuration dépend des besoins utilisateurs et de ceux de l'infrastructure elle-même.

8. En anglais, *bio-inspired*.
9. En anglais, *self-managing*.
10. En anglais, *self-**.
11. En anglais, *self-configuring*.

- **l'auto-optimisation**[12] : les systèmes autonomiques doivent en permanence vérifier et améliorer leur performance d'utilisation. Cette amélioration passe par la surveillance du système et son optimisation. Cette dernière peut être effectuée de multiples manières, par exemple, en fonction des paramètres observés, il peut y avoir des modifications (ajustements) des valeurs des variables de configuration. Ces variables peuvent être de niveau métier, système ou de très bas niveau comme le nombre de threads dans un pool de threads, la taille des buffers mémoires. Dans le cas des systèmes distribués, il peut être très difficile de définir une configuration permettant l'exécution optimale du système dès sa mise en service. Il peut être nécessaire d'observer son comportement en exécution pour effectuer des ajustements de paramètres (l'optimisation de la configuration). Le choix d'augmenter ou de diminuer le nombre de threads est une optimisation qui nécessite d'observer le système en exécution. Prenons un exemple de communication par messagerie inter-applications, l'objectif de haut niveau est de garantir un temps de réaction du système global. Cet objectif ne peut être atteint que si chaque système lié aux échanges garantit ses temps de transfert des messages. On pourrait imaginer un principe d'optimisation consistant à découvrir des nouveaux chemins de communication et à augmenter et/ou diminuer la priorité du thread en charge d'émettre et/ou de recevoir les messages.
- **l'auto-réparation**[13] : c'est la capacité de découvrir une panne, de la localiser et de la réparer. Il faut noter que l'auto-optimisation et l'auto-configuration peuvent être la source de dysfonctionnements. Par exemple, le déploiement d'une version identifiée comme améliorant la performance d'une application peut générer des pannes intermittentes sur une autre application voisine ou communicante avec l'application optimisée. Dans ce cas, le système doit être capable de détecter l'origine du problème (déploiement d'une version logicielle), de mettre en place une action corrective (*e.g.*, de déployer une version antérieure, un patch, d'informer l'utilisateur, etc.), et de rendre disponible le type d'erreur avec sa cause. Ceci pour que l'auto-optimisation (dans notre exemple) ne déploie plus à nouveau la version logicielle source de dysfonctionnements. La tolérance aux fautes [Siewiorek and Swarz, 1998] est un aspect de l'auto-réparation.
- **l'auto-protection**[14] : comme défini dans [Kephart and Chess, 2003], c'est la capacité à protéger le système selon deux axes. Le premier est la protection de l'ensemble du système à des attaques et à des problèmes qui ne sont pas encore ou ne peuvent pas être corrigés par l'auto-réparation. La surveillance nécessaire à la propriété d'auto-optimisation peut être aussi utilisée dans un objectif d'anticipation du dysfonctionnement de l'exécution du système ; c'est le deuxième axe de protection.

Ce sont les quatre propriétés fondamentales des systèmes autonomiques définies par IBM. Dans la littérature, elles sont couramment appelées les propriétés CHOP[15]. Ces propriétés ont été complétées par R. Steritt et D. Bustard (voir figure 4.6 « *Arbre des propriétés autonomiques, selon [Sterritt and Bustard, 2003]* » page 111) avec les quatre attributs suivants :

- **l'auto-connaissance**[16] : le système doit posséder des capacités d'introspection qui lui permettent de remonter les informations de ses états, les comportements de l'ensemble de ses constituants ;
- **la connaissance de l'environnement**[17] : le système doit avoir une connaissance de son environnement d'exécution ;

12. En anglais, *self-optimizing*.
13. En anglais, *self-healing*.
14. En anglais, *self-protecting*.
15. Acronyme de *Configuring-Healing-Optimizing-Protecting*.
16. En anglais, *self-awareness*.
17. En anglais, *environment-awareness*.

– **l'auto-surveillance** [18] : c'est la fonction d'observation de la boucle autonomique ;
– **l'auto-réglage** [19] : c'est la fonction de contrôle de la boucle autonomique.

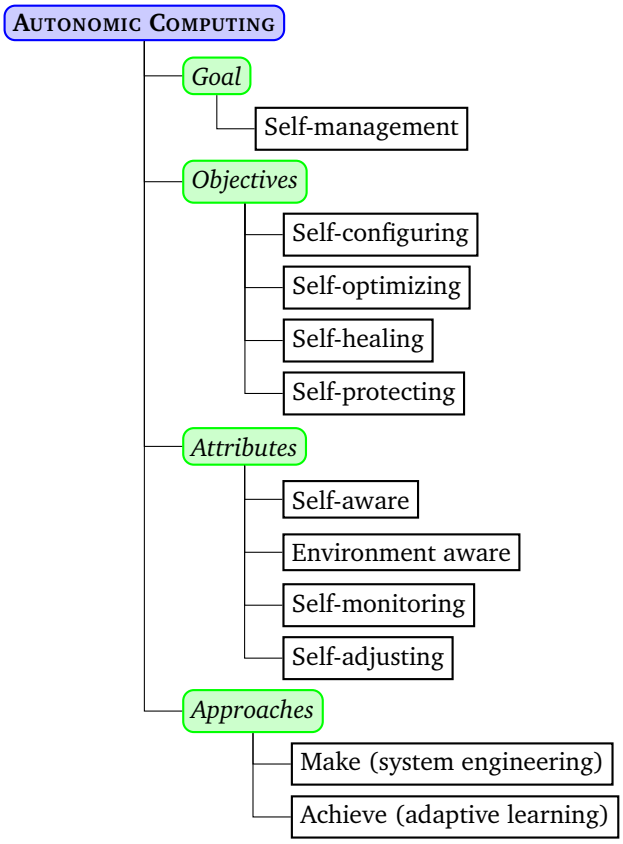


FIGURE 4.6 – Arbre des propriétés autonomiques, selon [Sterritt and Bustard, 2003].

Depuis l'origine du manifeste d'IBM en 2001, la liste des propriétés des systèmes autonomiques n'a cessé de croître en fonction des besoins des applications et des objectifs [Tianfield, 2003a, Tianfield, 2003b, Sterritt, 2005]. Il existe depuis plusieurs années des produits ayant des capacités de type auto-*, comme les ordinateurs et leurs propriétés d'auto-configuration. Ils peuvent recevoir depuis Internet des mises à jour logicielles. Dans le domaine des disques durs, la technologie RAID [20] [Buyya et al., 2009] permet de connecter plusieurs disques physiques pour offrir la vision d'un seul disque logique. Si un problème est détecté sur un accès en lecture ou en écriture, les données sont automatiquement copiées sur une autre partie du disque voire sur un autre disque physique.

Idéalement, la communauté de l'informatique autonomique [Ganek and Friedrich, 2006, Huebscher and McCann, 2008] voudrait qu'un système autonomique implante plus d'une des quatre propriétés auto-* d'IBM. Il reste donc encore un pas important avant que l'informatique autonomique soit présente dans notre environnement. R. STERITT définit deux approches pour faire de l'informatique autonomique une réalité :

> *« There are two perceived approaches to make Autonomic Computing a reality*

18. En anglais, *self-monitoring*.
19. En anglais, *self-adjusting*.
20. Acronyme de *Redundant Array of Independant Disks*.

– ***Making*** *autonomic computing.*
– ***Achieving*** *autonomic computing.* »
[Sterritt, 2002]

Nous allons dans la section suivante présenter l'architecture à mettre en place pour réaliser un système auto-géré.

4 Architecture

Selon J. O. KEPHART [Kephart and Chess, 2003], les éléments autonomiques gèrent leur comportement interne et leurs relations avec d'autres éléments autonomiques en accord avec des politiques que des humains ou d'autres éléments ont définis. Nous allons, dans cette section, présenter l'architecture interne des éléments autonomiques, les types de politiques qui serviront à exprimer des objectifs de haut niveau, les différentes architectures constituées de multiples éléments autonomiques et, pour terminer, les besoins de l'informatique autonomique qui en découlent.

4.1 Présentation

L'architecture d'un système est un ensemble d'éléments autonomiques en collaboration. Nous allons, à présent, expliciter ces éléments autonomiques.

La vue fonctionnelle d'un élément autonomique est illustrée par la figure 4.7 « *Boucle de contrôle* ». Un élément autonomique est constitué d'une boucle autonomique et d'une ressource à gérer. La boucle autonomique reçoit des objectifs de haut niveau fixés par des administrateurs et/ou des experts. La boucle de contrôle est de type contre-réaction. Sur la base de ses observations, elle prend des décisions et adapte la ressource.

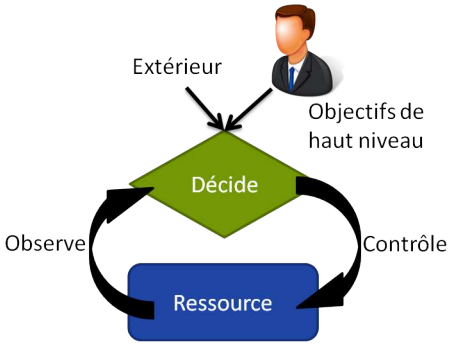


FIGURE 4.7 – Boucle de contrôle.

Une boucle de contrôle évoluée peut effectuer des opérations ou fournir des fonctionnalités d'auto-gestion comme, par exemple :
– **déployer** des nouvelles versions de logiciel ;
– **redémarrer** une ressource après la détection d'une panne ;
– **informer** ou arrêter une ressource après la détection d'une intrusion ou d'une cyber-attaque ;

- **détecter** l'apparition d'une ressource et la configurer ;
- **contrôler** la charge CPU d'une ressource en fonction de conditions internes ou externes (par exemple, pendant le déploiement d'une application, surveiller la charge CPU et agir pour ne pas perdre la communication entre la ressource cliente et le serveur).

La boucle autonomique reçoit des politiques de la part d'humains (administrateurs, experts) ou d'autres éléments prédéfinis. Elle peut même être en relation avec d'autres boucles autonomiques. Elle rend disponible (aux humains ou à d'autres éléments) les résultats de l'application de ces politiques définies.

Cette boucle de contrôle autonomique est externe à la ressource gérée. Il y a une séparation des préoccupations entre logique d'exécution et logique d'adaptation. La personne en charge de la logique d'adaptation n'a pas à connaître le fonctionnement interne. **L'externalisation de la boucle de contrôle au système administré apporte des propriétés importantes du génie logiciel : la réutilisation et l'évolution**. La logique du gestionnaire autonomique peut être réutilisée entièrement ou en partie ou bien encore utiliser des principes et des algorithmes venant d'autres domaines. Elle peut être changée/modifiée sans altérer l'élément administré. L'adaptation et l'évolution sont alors aisées.

La ressource gérée est en relation avec son environnement d'exécution. Elle peut être : une ressource physique (un routeur TCP/IP), une ressource logicielle (un serveur de requêtes HTTP, une base de données, un fichier...), une ressource virtuelle. Elle peut aussi être une ressource constituée d'un ensemble d'éléments (un *cluster* de serveurs), une infrastructure complexe à grande échelle (*cluster* de serveurs avec des routeurs et des applications intégrées par des EAI et/ou des ESB).

La réalisation de ce principe nécessite deux **éléments indépendants** et une **interface d'administration**. Le premier élément est la ressource à gérer, le second est la boucle autonomique. L'interface d'administration est l'interface entre la ressource à gérer et la boucle autonomique. La boucle autonomique effectue des observations et des adaptations de la ressource au travers de cette interface d'administration. Cette boucle autonomique offre, elle aussi, une **interface d'administration** pour recevoir des politiques de haut niveau et communiquer avec d'autres éléments autonomiques ou humains.

4.2 L'architecture générale

IBM a proposé [IBM, 2006] un **modèle** de référence pour réaliser des boucles de contrôle autonomiques. Ce modèle est un **patron de conception**. Son objectif est de définir l'ensemble des constituants et leurs relations. La figure 4.8 « *Architecture de référence selon IBM* » page 114 représente ce modèle défini par IBM.

Un élément autonomique gère des ressources et leurs états selon des politiques définies et donnent les effets à des humains (administrateurs, experts), ou à d'autres éléments autonomiques. Un élément autonomique est constitué d'un ou plusieurs **éléments administrés** (une ressource matérielle ou virtuelle) et d'un **gestionnaire autonomique** (la boucle autonomique). L'interface entre les deux entités est constituée de **capteurs** et **d'effecteurs**.

Le gestionnaire autonomique est la boucle de contrôle autonomique, couramment appelée boucle **MAPE-K**[21]. M. HUEBSCHER et J. MCCANN [Huebscher and McCann, 2008], pensent que la boucle MAPE-K proposée par IBM est inspirée des travaux de M. RUSSEL et P. NORVIG [Russell and Norvig, 2010] sur les agents génériques.

21. Acronyme de *Monitor*, *Analyse*, *Plan*, *Execute*, *Knowledge*.

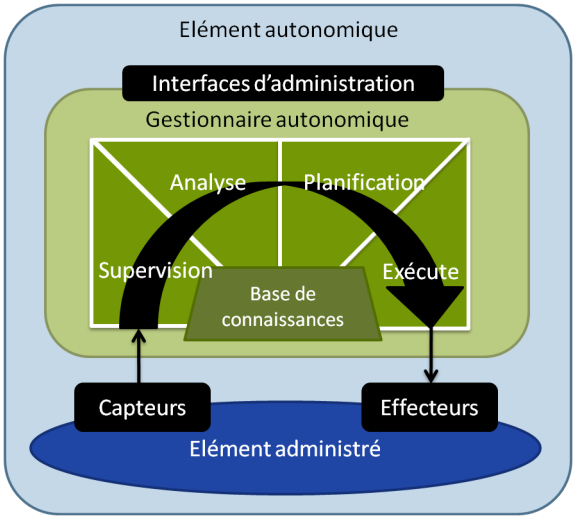


FIGURE 4.8 – Architecture de référence selon IBM.

4.2.1 L'élément administré

Dans l'architecture proposée par IBM, **l'élément administré** est la ressource gérée qui a été présentée au début de cette section. La granularité des éléments autonomiques n'est pas précisée. Il en résulte qu'un **élément administré** peut être une simple ressource (matérielle ou virtuelle) jusqu'à une très grande infrastructure de systèmes d'informations.

4.2.2 Les capteurs et les effecteurs

Les **capteurs** sont le premier point de contact entre le gestionnaire autonomique et l'élément administré. Les capteurs collectent les informations de l'élément administré. Ces données collectées peuvent être de haut niveau comme, par exemple, la surveillance de serveurs à grande échelle [Roblee et al., 2005] ou de données internes d'un système (taux d'occupation du processeur, consommation mémoire, etc.) [Sterritt and Bantz, 2004].

Les **effecteurs** constituent le second point de contact entre le gestionnaire autonomique et l'élément administré. Les effecteurs opèrent des modifications sur le système. Ces modifications peuvent être, elles aussi, de haut ou de bas niveau. Un changement de serveur [Schmerl and Garlan, 2002] est un exemple de modification de haut niveau. Sont considérées comme des opérations de bas niveau des modifications de paramètres de configuration [Sterritt et al., 2005].

4.2.3 Le gestionnaire autonomique

La structure interne d'un gestionnaire autonomique est composée de cinq éléments distincts ayant chacun une fonction clairement définie :

– **la base de connaissances (K)** est une composante essentielle de la boucle autonomique. Sans elle, la boucle autonomique serait une simple boucle de rétro-action. La base de connaissances contient l'ensemble des informations nécessaires pour la gestion de l'élément autonomique. Elle est commune entre la Surveillance, l'Analyse, la Planification et l'Exécution (MAPE). Elle peut contenir des mesures, des politiques, des états, des résultats des exécutions, des topologies, etc. Elle peut aussi être mise à jour

par la récupération d'une base de connaissances d'un ou plusieurs autres gestionnaires autonomiques.

- **la supervision (M)** collecte les données issues des capteurs. Ces données peuvent être agrégées, filtrées, corrélées, horodatées et stockées (ou non) dans la base de connaissances. La supervision peut prendre deux rôles. La surveillance décide de l'exécution de l'analyse lorsque, par exemple, une ou plusieurs données résultats sont hors zone de bon fonctionnement. Dans ce cas, la supervision a pour rôle de collecter des données et de détecter un problème potentiel (dysfonctionnement ou optimisation) sur les données résultats. La supervision peut aussi ne pas être en charge de décider si l'exécution de l'analyse doit démarrer ou non. Son rôle est alors de fournir des données résultats prêtes à être interprétées par l'Analyse.
- **l'analyse (A)** récupère les données résultats (et/ou la liste des problèmes potentiels selon le rôle pris par la supervision). Elle identifie les dysfonctionnements et les optimisations puis détermine la nécessité d'effectuer des modifications sur l'élément administré autonomique pour régler les problèmes identifiés. L'analyse utilise la base de connaissances pour effectuer des corrélations, des courbes de tendances, des statistiques, des probabilités, des séries de numériques horodatées, etc.
- **la planification (P)** prend en charge la définition des actions à effectuer sur l'élément administré pour résoudre les problèmes identifiés par l'analyse. Elle génère un plan d'actions.
- **l'exécution (E)** est en charge du calendrier des actions définies par la planification. Le calendrier consiste, par exemple, à définir une date ou un horaire pour l'exécution des actions qui doivent être exécutées. Elle est aussi le point de contact avec les effecteurs.

Le gestionnaire autonomique collecte les données de l'élément administré par les capteurs et les effecteurs. Son interface d'administration lui permet d'être configuré et de donner son état à des humains ou à d'autres éléments autonomiques. Il construit et maintient sa base de connaissances continuellement (rappelons que cette base de connaissances peut être initialisée par la récupération de celles d'autres éléments autonomiques).

La communication entre éléments autonomiques est une nécessité pour assurer une autogestion globale cohérente du système. Deux voies de communications sont possibles entre éléments autonomiques :

- **la voie directe** : les éléments autonomiques communiquent sans intermédiaire. Le mode de communication le plus approprié, dans ce cas, est le mode asynchrone. Il permet un découplage faible entre les participants, une gestion de la perte des messages. Chaque élément peut alors continuer sa gestion d'élément administré sans attente de résultat. Les systèmes autonomiques sont hétérogènes, chacun possède sa propre base de connaissances. Il en résulte que des problèmes d'interprétation entre les informations échangées peuvent être la source de dysfonctionnement. Une solution, parmi d'autres, consiste à modéliser les données échangées en définissant une ontologie [Uschold, 1998]. L'ontologie définit un vocabulaire, une spécification et la sémantique des données.
- **la voie indirecte** : les éléments autonomiques utilisent l'intermédiaire d'un ou plusieurs autres éléments, tels qu'une mémoire partagée.

Quelle soit directe ou indirecte, la communication entre éléments autonomiques est complexe ; car ce sont des informations de haut niveau qui sont échangées.

La boucle MAPE-K est une entité externe à l'élément administré. Il est possible de faire évoluer des systèmes patrimoniaux[22], qui sont gérés manuellement, en systèmes autonomiques. Le gestionnaire autonomique vient alors se connecter aux capteurs et effecteurs existants de ce système patrimonial [Kaiser et al., 2003].

Nous arrivons au terme de la présentation de la boucle autonomique, nous avons vu son architecture et les propriétés de ses constituants qui sont la Surveillance, l'Analyse, la Planification, l'Exécution et la Base de connaissances. Nous ne trouverons pas à ce jour, dans la littérature, un patron d'implantation général comme, par exemple, le **singleton pattern**. Concluons cette section sur la remarque suivante [Lalanda et al., 2013] :

La boucle MAPE-K est un modèle général, qui doit être adapté selon ses besoins. Elle sert de base pour la construction d'applications autonomiques. Elle n'apporte pas de solution technique d'implantation.

4.3 Les politiques

Les gestionnaires autonomiques utilisent des objectifs de haut niveau, aussi appelés politiques, pour gouverner les quatre éléments de la boucle MAPE. Dans le cadre de l'informatique autonomique, J. O. KEPHART et W. E. WALSH [Kephart and Walsh, 2004] ont classé les politiques de haut niveau en trois types :

- **les politiques d'actions** : à partir d'un état donné courant du système, l'action à effectuer est définie par la politique. Ces politiques sont exprimées sous forme de règles ECA[23] (*if condition then action*). Le système transite d'états en états par les actions effectuées sur lui.
- **les politiques de buts** : la différence consiste à définir non pas un nouvel état du système, mais un ensemble de critères qui permet de définir une liste d'états possibles. Le système doit définir les actions qui le feront passer d'un état à un autre état.
- **les fonctions d'objectifs**[24] : c'est une généralisation des politiques de buts. Une politique de fonction d'objectif exprime une valeur numérique de satisfaction pour chaque état possible. La liste des états possibles peut être ordonnée. L'ambiguïté du choix entre deux états est levée par leur ordonnancement (à chaque état est associé une valeur numérique).

Les politiques d'actions sont les plus faciles à implanter, même s'il faut noter comme limitation l'utilisation des règles ECA. Les politiques d'actions ne prennent pas en compte l'état du système. Le système transite d'un état à un autre par l'effet des actions sur lui, la conséquence est que l'état résultat est présumé connu par l'auteur de la règle. De même, un nombre important de règles peut aboutir à des conflits de politiques et à un comportement indéterminé du système. Les politiques de buts permettent de résoudre le cas des systèmes qui sont décrits par un nombre important d'états (*i.e.* un nombre important de règles ECA). Les politiques de fonctions d'objectifs permettent de supprimer les incertitudes et de classer les états possibles (ce qui n'est pas possible avec les politiques de buts).

22. En anglais, *legacy*.
23. Acronyme de *Event-Condition-Action*.
24. En anglais, *utility function policies*.

4.4 Architectures d'éléments autonomiques

Une application autonomique peut nécessiter la réalisation d'un ou plusieurs éléments autonomiques avec ou sans échanges d'informations entre eux. Dans le cas des échanges d'informations, il y a alors trois architectures possibles :

- une architecture distribuée ;
- une architecture centralisée ;
- une architecture hiérarchisée.

4.4.1 Architecture distribuée

L'architecture distribuée est illustrée par la figure 4.9. Chaque système autonomique a la charge de son élément administré et communique (si nécessaire) avec d'autres systèmes autonomiques.

Les données de surveillance et la boucle autonomique sont locales à chaque élément, les échanges d'informations entre le gestionnaire autonomique et l'élément administré sont, par conséquent, optimaux (il n'est pas possible de faire mieux). La complexité de la boucle autonomique ne dépend que de l'élément à administrer et des objectifs de haut niveau à atteindre. Cette architecture favorise l'approche agile ; c'est-à-dire une approche de déploiement incrémentale.

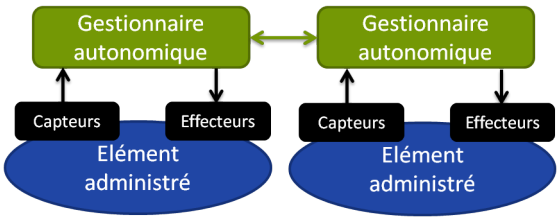


FIGURE 4.9 – Architecture distribuée.

La difficulté de cette architecture réside dans l'assurance du fonctionnement du système constitué de plusieurs éléments autonomiques ayant le même niveau. **Comment garantir qu'il n'y aura pas de conflits de politiques entre gestionnaires ?** Par exemple, une optimisation d'un élément administré peut déstabiliser l'élément voisin (géré par une boucle autonomique différente). La négociation et la collaboration entre gestionnaires autonomiques doivent être traitées pour la réalisation de cette architecture.

4.4.2 Architecture centralisée

L'architecture centralisée est illustrée par la figure 4.10. Le gestionnaire autonomique gère l'ensemble des éléments à administrer, récupère l'ensemble des données issues de tous les capteurs et opère des modifications par l'ensemble de tous les effecteurs.

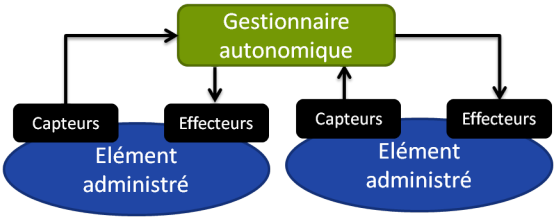


FIGURE 4.10 – Architecture centralisée.

Cette architecture permet la suppression des problèmes difficiles de communications entre les éléments autonomiques, il n'y a pas de conflit d'objectifs. Le gestionnaire possède un traitement centralisé, les décisions pourront être optimales pour l'ensemble des éléments administrés.

Cependant, la **surveillance** doit être capable d'identifier l'élément administré et les données issues des capteurs. De même pour l'**exécution**, les effecteurs doivent être identifiés et associés à leur élément administré. L'ajout d'éléments à administrer nécessite une évolution au niveau du gestionnaire autonomique, l'agilité d'évolution n'est pas garantie ni aisée. Une autre conséquence de cette architecture est que le gestionnaire autonomique doit être capable de trouver des solutions à tous les problèmes pour l'ensemble de ses éléments ; dans ce cas, les objectifs de haut niveau peuvent être difficiles à définir (la phase d'analyse est complexe par rapport à celle de l'architecture distribuée). Le passage à l'échelle peut être difficile à réaliser de par la complexité du gestionnaire autonomique, d'une part, et, d'autre part, en raison du flux important de données remontées des différents capteurs. Le gestionnaire autonomique peut avoir des difficultés pour trier les informations importantes de celles de moindre importance. Par exemple, l'ajout d'un serveur dans l'infrastructure ne nécessite pas une réaction immédiate (configuration du serveur). Par contre, c'est le cas pour la détection d'intrusion, elle nécessite une réactivité immédiate du gestionnaire autonomique pour déclencher des actions de protection.

4.4.3 Architecture hiérarchisée

L'architecture hiérarchisée est illustrée par la figure 4.11. Les gestionnaires autonomiques administrateur des éléments sont au plus bas dans la hiérarchie (les subalternes). Ils peuvent communiquer entre eux comme dans l'architecture distribuée.

Le gestionnaire autonomique de niveau supérieur est un gestionnaire d'autorité. Il traite des conflits et il a la vision globale du système constitué de plusieurs éléments autonomiques. La communication entre le gestionnaire d'autorité et les gestionnaires subalternes est une communication échangeant des concepts de haut niveau. Ce type d'échanges d'informations est difficile à modéliser et à implanter.

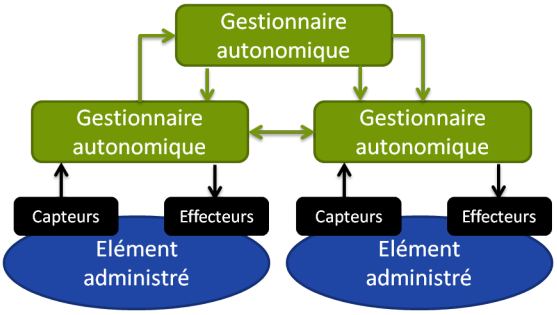


FIGURE 4.11 – Architecture hiérarchisée.

Nous avons présenté, dans cette section, les différents éléments constituant une boucle autonomique et les trois architectures théoriquement possibles. Nous allons, dans la section suivante, expliquer les besoins nécessaires à la réalisation d'un système auto-géré.

5 Besoins

Selon A. GANEK [Ganek, 2004], un système autonomique doit avoir les propriétés générales suivantes :

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

Ces trois caractéristiques sont à la base de la réalisation des quatre propriétés CHOP [25] définies par IBM. L'interface d'administration (les capteurs et les effecteurs) et l'élément administré ont également des besoins que nous allons détailler dans cette section. Par la suite, nous définirons la **surveillance** et l'**exécution** comme désignant respectivement le M (*Monitoring*) et le E (*Execute*) de la boucle MAPE-K. Nous prenons les définitions suivantes de l'observation et de l'adaptation :

*L'**observation** est l'action de la surveillance qui a pour finalité de collecter des informations des capteurs.*

*L'**adaptation**, quant à elle, désigne l'action de l'exécution qui modifie le système grâce aux effecteurs.*

Nous allons à présent nous intéresser aux besoins de la boucle autonomique : l'observation et l'adaptation.

25. Acronyme de *Configuring-Healing-Optimizing-Protecting*.

5.1 L'observation

5.1.1 Généralités

La surveillance collecte les données des capteurs et met à jour la base de connaissances. Le processus qui effectue des mesures, ne doit pas perturber l'exécution du système. La surveillance peut, par exemple, déclencher un événement par l'intermédiaire du capteur, sans perturber le traitement d'un événement système d'un point de vue fonctionnel et temporel. Les mesures fournies par les capteurs doivent avoir une précision suffisante pour caractériser le système observé. En cas de charge intensive, il sera peut-être nécessaire d'effectuer une mesure par seconde pour obtenir une vision précise et juste de la consommation mémoire. Dans d'autres circonstances, une fois par minute suffira.

On distingue deux types de systèmes : les patrimoniaux [26], qui ne peuvent être modifiés, et ceux qui peuvent l'être. Dans le cas où l'élément administré n'est pas un système patrimonial, comment peut-on définir le système pour qu'il puisse délivrer des informations pertinentes ? Sans observation pertinente, le gestionnaire autonomique ne pourra pas atteindre ou même maintenir ses objectifs. Prenons l'exemple de pannes non franches (dégradations lentes des performances du système). En pratique, elles demandent beaucoup de temps et d'énergie aux administrateurs et aux experts pour identifier les sources du dysfonctionnement. Sans information proche de la source d'erreur, la détermination du problème est extrêmement longue et difficile. C'est pour cela que pour gagner du temps de disponibilité du système, généralement le correctif consiste en une série de modifications de type contournement (le problème n'est pas corrigé, seule son apparition est supprimée). Seulement une connaissance approfondie et une expérience de la dynamique d'exécution de l'application permettent la définition des contournements. **En fonction de l'implantation du système, il ne faut donc pas hésiter à déterminer les données pertinentes qui peuvent être enfouies dans les couches les plus basses (matériel, système d'exploitation) jusqu'aux plus hautes.**

Les systèmes patrimoniaux sont différents ; toutes les informations ou les outils nécessaires à la compréhension de l'exécution du système ne sont peut-être pas disponibles. Il faut donc faire avec l'existant. Néanmoins, pour améliorer la capacité de résolution de la boucle autonomique, il est nécessaire d'utiliser l'ensemble des informations délivrées par le système et d'analyser les bonnes pratiques des experts et des administrateurs. **L'ensemble de ces bonnes pratiques servira à la définition de règles de la boucle MAPE-K.**

5.1.2 Les capteurs

Les capteurs délivrent généralement des données non corrélées [27], ils mesurent un système (ou un aspect du système). Généralement, on y retrouve des données qui permettent d'obtenir :

- des mesures de charge du système (ou sous-système) ;
- des temps de réponse du système (ou sous-système) ;
- des mesures de latence ;
- le nombre d'événements traités et/ou générés par unité de temps ;
- le nombre de fichiers ouverts ;
- le nombre de connexions TCP/IP ouvertes ;

26. En anglais, *legacies*.
27. La corrélation des données est généralement effectuée par la boucle autonomique.

- des mesures distribuées sur plusieurs systèmes nécessitant des mécanismes de synchronisation d'heure (une mesure de charge ou de consommation électrique exécutée au même moment par tous les systèmes et déclenchée par un top horaire) ;
- la topologie du système (par exemple, les chemins de communications disponibles) ;
- etc.

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

Le processus de génération des mesures consomme des ressources système (mémoire, processeur), beaucoup d'exécutions de ses processus dégradent les performances métier de l'application. La seconde conséquence est au niveau de la boucle autonomique. Un flux important et permanent de mesures rend difficile le fonctionnement nominal de la surveillance (sa capacité maximale de traitement pourrait être dépassée [28]). En offrant aux capteurs des fonctions d'activation et de désactivation, la surveillance peut piloter le flux et la charge d'exécution des processus de génération des mesures.

Attachons nous maintenant aux deux techniques générales d'obtention des mesures : une méthode directe et une méthode indirecte. Une mesure directe ne nécessite pas un processus spécifique pour la réaliser. Par exemple, la mesure de la température est le résultat d'un accès en lecture d'un registre (la mise à l'échelle est généralement effectuée juste après l'accès en lecture). Une interception de méthode [29] est une mesure directe.

Les mesures indirectes nécessitent un enchaînement d'opérations (un calcul de latence demande deux étapes : un horodatage au point de départ et un calcul de différence de temps au point d'arrivée). Plus généralement, certaines mesures demandent une injection de messages dans le système. Les (sous-)systèmes détectent ces messages et ajoutent une information (un horodatage, un identifiant). Au final le message est délivré dans un récepteur qui analyse le contenu du message et qui met à disposition un ou plusieurs résultats (statistiques, compteurs, fichier de traces). La détermination du moment où le message doit être injecté, est capitale ; ces messages ne doivent pas entraver la prise en compte d'événements (messages) applicatifs (relatifs au métier) et doit autant que se faire être effectué lorsque le système est capable d'absorber cette surcharge de traitements. Les mesures indirectes exigent une implantation spécifique (ajout de code non-fonctionnel).

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

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

- de notification : le capteur notifie la fraîcheur d'une information, la donnée peut être ou non incluse dans l'événement ;
- de surveillance périodique (ou non) : la tâche de surveillance est à l'initiative de la mise à jour des données issues des capteurs ;

28. En anglais, *over-run*.
29. En anglais, *hook*.

– de messageries avec la définition des patrons d'échanges de messagerie [30] entre la tâche de surveillance et les capteurs ;
– des appels de méthodes [31] ;
– etc.

Revenons sur le problème de dégradation des performances. Nous avons vu que les capteurs devraient offrir des fonctions d'activation et de désactivation des mesures. **Comment peut-on minimiser au mieux ces dégradations ?** Le seul point d'action reste la surveillance. **Elle doit être dynamique**. Une surveillance dynamique permet d'**arrêter**, de **démarrer**, de **configurer** des capteurs (mesures). On imagine aussi que la surveillance injecte du code dynamiquement et le supprime lorsqu'il n'est plus nécessaire. Plus simplement, c'est configurer de façon dynamique les capteurs (augmenter/diminuer la fréquence d'échantillonnage, injecter un message spécifique), surveiller des capteurs à la demande (en fonction du contexte d'exécution). L'exécution de l'analyse est étroitement liée à celle de la surveillance. En effet, l'analyse peut nécessiter la mise à jour d'informations supplémentaires (augmenter la période d'échantillonnage d'une mesure, mettre dans la base de connaissances des informations non encore mises à jour). La surveillance vient chercher un ensemble d'informations dans les capteurs sur demande de l'analyse. La propriété de surveillance dynamique doit se situer à tous les niveaux de la surveillance (le M) jusqu'aux capteurs (processus réalisant la mesure).

Nous avons jusqu'à présent traité le point 1 « **Avoir une base de connaissances** » des propriétés d'un système autonomique selon A. GANEK [Ganek, 2004]. Ce point est nécessaire à la détermination des états du système. Le point 2 « **Etre capable de comprendre et d'analyser les conditions environnementales** » élargit la connaissance à celle de l'environnement. La surveillance a en charge la découverte de la présence et de la disparition, ainsi que de l'identification des ressources externes qui peuvent influencer l'exécution ou l'optimisation du système administré. Les ressources sont de tous types : des dispositifs hétérogènes, des applications logicielles (une ressource logicielle), du matériel (une ressource physique), un *cluster* de serveurs, etc. La surveillance met à jour la base de connaissances avec ces informations.

La surveillance de l'environnement est une tâche complexe et dépendante de la technologie. La surveillance d'un dispositif hétérogène dépend de sa technologie et, s'il existe, de son protocole de communication. Les protocoles n'offrant pas la découverte, la surveillance des dispositifs doivent implémenter une surveillance active des dispositifs présents ainsi qu'un mécanisme de recherche de dispositifs. La complexité provient du nombre important de protocoles disponibles. Pour les systèmes distribués, la découverte du dispositif (système) est soit intégrée aux protocoles de communication soit ad hoc :

– intégrée dans les protocoles de communication : par exemple, DPWS inclut un protocole de découverte de dispositifs ;
– des mécanismes ad hoc : par exemple, Modbus/TCP ne fournit pas de protocole de découverte. Deux protocoles sont nécessaires : un premier pour découvrir le dispositif par émission de requêtes *ping* du protocole ICMP [32], puis une requête spécifique Modbus (43/14) pour identifier le dispositif.

Pour les systèmes locaux (OSGi), la plate-forme d'exécution apporte son mécanisme de découverte.

La connaissance de l'environnement nécessite aussi des méta-données complémentaires à celles d'identification et de présence. Par exemple, la localisation d'un dispositif peut néces-

30. En anglais, *Messages Exchange Pattern*.
31. En anglais, *callback*.
32. Acronyme de *Internet Control Message Protocol* (RFC 792).

siter des méta-données qui peuvent être configurées dans le dispositif, récupérées d'une base de données, des satellites GPS, etc.

Le troisième point de A. GANEK [Ganek, 2004] « **Etre capable de planifier et d'effectuer des modifications** » est l'aptitude à effectuer des modifications. La boucle autonomique définit un ensemble de modifications à apporter sur le système observé. Ces modifications font transiter l'état initial du système vers un nouvel état recherché. Nous allons présenter les besoins liés à l'adaptation des systèmes.

5.2 L'adaptation

5.2.1 Généralités

L'adaptation est un processus qui modifie un système (ou une application). Elle peut être soit statique, soit dynamique. Une adaptation statique nécessite un arrêt du système [33]. L'adaptation dynamique, quant à elle, autorise des modifications pendant l'exécution du système. Pour les utilisateurs et/ou les autres applications interconnectées, la disponibilité du système sera alors partielle (elle ne peut pas être totale pendant la modification).

L'objectif de l'informatique autonomique est de simplifier l'administration en exécution des systèmes informatiques. Bien que l'adaptation dynamique soit privilégiée, il existe des cas où seule l'adaptation statique est réalisable (déploiement d'une version d'un système d'exploitation). Dans cette section, nous nous intéressons aux besoins de l'adaptation dynamique.

Comme nous l'avons vu dans la section 5.1 « *L'observation* », page 120, il existe également le besoin d'identification du capteur avec le système administré. Nous allons détailler dans la section suivante les besoins des effecteurs liés à l'adaptation dynamique.

5.2.2 Les effecteurs

Les effecteurs peuvent, tout d'abord, opérer sur le système par le biais de la configuration. La configuration d'un système informatique comprend généralement deux types de paramètres de configuration ; ceux qui peuvent être modifiés à la volée (en exécution) et ceux qui ne le peuvent pas. Usuellement, on distingue ces deux types de paramètres par leur nom, les paramètres de réglage sont ceux qui sont modifiables à la volée (adaptations dynamiques) et les paramètres de configuration sont ceux que l'on ne peut pas modifier dynamiquement (adaptation statique). Dès la conception du système, il est important de privilégier le développement des paramètres de réglages. Ils sont plus difficiles à implanter ; car ils requièrent la prise en compte du dynamisme (disponibilité, refus si les contraintes ne sont pas respectées). Un deuxième type d'adaptation plus profonde est nécessaire pour répondre à des besoins de flexibilité (apporter des modifications, des évolutions incrémentales). Elle opère au niveau structurel du (sous-)système ; c'est-à-dire de sa composition et de ses relations.

J. KRAMER et J. MAGEE [Kramer and Magee, 1990] ont présenté les besoins pour des modifications de type dynamiques. L'approche utilise essentiellement la séparation des préoccupations à savoir l'implantation du métier, de l'isolation des processus de modification. Les propriétés sur les modifications définies par J. KRAMER et J. MAGEE sont :

- les modifications doivent minimiser les perturbations du système ;
- les modifications doivent laisser le système dans un état consistent ;

33. Trois étapes : arrêt du système, exécution des modifications, démarrage du système.

- les modifications doivent être indépendantes des algorithmes et des protocoles utilisés par le système ;
- les modifications doivent être déclaratives (séparation des préoccupations : implantation et spécification des modifications) ;
- les modifications doivent être décrites en termes de structure du système (il ne doit pas y avoir de modifications à grains fins, par exemple, pas de modifications de la structure interne d'un composant).

Il existe plusieurs patrons dont deux sont largement utilisés :

- le patron d'interposition dynamique implémenté par CORBA[34] [Baldoni et al., 2003]. Un intercepteur est inséré au niveau du serveur ou du client. L'intercepteur redirige les requêtes vers un *wrapper* spécifique.
- le remplacement dynamique de composant [Soules et al., 2003], qui est une évolution du patron précédent.

Ces deux patrons requièrent de mettre la partie du système à modifier dans un **état quiescent** (et de détecter cet état) [Kramer and Magee, 1990]. Ce besoin est principalement dicté par la nécessité de démarrer une modification lorsque l'état du (sous)-système est consistant, de le modifier et de rendre le (sous)-système consistant après la modification.

L'état de **quiescence** [Kramer and Magee, 1990, Vandewoude et al., 2006] est difficile à atteindre (et à détecter), des auteurs tels que les auteurs de [Vandewoude et al., 2006] proposent d'atteindre l'état dit de **tranquillité** pour effectuer des adaptations dynamiques.

Lorsque les systèmes autonomiques nécessitent l'assurance de la non perte des messages (événements), l'adaptation doit avoir une propriété additionnelle : la persistance. Cette propriété est utilisée pour :

- garantir la continuité des paramètres de configuration et de réglages (le système de version V_{n+1} aura les mêmes paramètres que celui de la version V_n ;
- garantir la non perte des messages (événements), pour conserver des données pendant l'adaptation et reprendre leurs traitements après les modifications sur le (sous-) système.

Les effecteurs sont le point de contact de l'exécution (E). Les demandes d'évolutions peuvent être classées en deux familles, des modifications fonctionnelles (le métier) et celles non-fonctionnelles (plate-forme d'exécution, chemins de communications). Les besoins de persistance sont généralement nécessaires pour des modifications fonctionnelles.

Pour terminer cette section sur l'adaptation, un point qui ne peut pas avoir de réponse générique, mais qui doit être pris en compte dans le cadre des systèmes autonomiques :

comment s'assurer que la qualité du service reste au niveau demandé suite à des adaptations (évolutions d'architecture, de versions logicielles, de composants) ?

34. L'OMG a défini le *Portable Request Interceptor*.

6 Conclusion

Dans ce chapitre, nous avons abordé la définition de l'**informatique autonomique** présentée par IBM en 2001, dont l'objectif est de rendre les systèmes informatiques gérables, ainsi que de réduire leurs coûts d'exploitation. Les administrateurs devaient voir leurs tâches simplifiées. Ils devaient également être aidés dans leur quotidien (grâce à des tableaux d'exploitation générés en fonction d'un contexte d'exécution).

Tout d'abord, l'architecture proposée par IBM est suffisamment explicite dans la définition des composants afin de pouvoir être utilisée comme modèle par les architectes et les développeurs. La granularité d'application de cette architecture n'est pas explicitée, elle peut être aussi bien appliquée pour des grands systèmes informatiques (systèmes d'information des entreprises à grande échelle, des *clusters* de serveurs) qu'à des systèmes informatiques (matériels ou virtuels) moins ambitieux.

L'optimisation des systèmes informatiques, pendant leur phase exploitation, est et deviendra un besoin grandissant dans un futur très proche. Les *data centers* vont devoir réguler leur consommation énergétique (plus finement qu'actuellement) en fonction du coût de l'électricité, du besoin en calculs, de la température et bien d'autres contraintes. La boucle MAPE-K permettra d'aborder le traitement de ces problèmes difficiles. La diversité des influences théoriques et pratiques (les utility functions, la théorie du contrôle) aide dans la recherche de solutions d'implantation des différents éléments de la boucle autonomique du M/A au P/E.

Le contenu et l'architecture technique de la base de connaissances (K) est une tâche difficile de par la diversité des informations à sauvegarder (connaissance des états du système, du contexte externe, des historiques). Il n'existe pas, à ce jour, de solution générique.

Les capteurs et les effecteurs ont des besoins de communications, d'élaboration de mesures pour être performant et simplifier la tâche de surveillance (M) et d'exécution (E). Ces besoins devront être pris en compte au plus tôt dans la réalisation du système (séparer les préoccupations du code fonctionnel du code réalisant les adaptations, implanter des techniques permettant de réduire le flux des données échangées). Ce que l'on doit retenir de la boucle MAPE-K, c'est qu'elle n'est pas une réponse technique (une implantation), mais un patron de conception des systèmes autonomiques avec les propriétés auto-*.

Nous avons également présenté les trois styles d'architectures mettant en œuvre plusieurs gestionnaires autonomiques. Nous retiendrons que la communication entre gestionnaires autonomiques échange des concepts de haut niveau et qu'elle est, par conséquent, difficile à modéliser et à implanter.

Pour terminer, l'informatique autonomique convient aux applications **patrimoniales** ; la boucle MAPE-K est fonctionnellement connectée au système par l'intermédiaire des capteurs et des effecteurs. Pour les systèmes qui doivent être développés, elle pourra atteindre les objectifs du manifeste d'IBM, à condition de prendre dès la phase conception les besoins des capteurs et des effecteurs.

Deuxième partie

Contribution

5

Proposition

Dans la première partie de cette thèse, nous avons présenté un état de l'art sur l'informatique orientée services, sur les *Entreprise Service Bus* (ESB) et sur une nouvelle approche permettant d'automatiser la gestion des systèmes informatiques, à savoir l'informatique autonomique.

L'informatique autonomique repose sur l'observation contrôlée d'un système informatique et sur la mise en œuvre d'actions de gestion sur ce même système de façon à améliorer sa pertinence et sa qualité de fonctionnement. Elle est basée sur un ensemble de techniques issues de divers domaines qui permettent de collecter des données, d'analyser la situation, de prendre des décisions et de réaliser les adaptations pour rendre les systèmes auto-gérés.

La seconde partie de cette thèse présente notre proposition, qui vise la conception et la mise en œuvre d'une version autonomique de l'ESB Cilia présenté précédemment. L'approche, que nous défendons, permet de construire des applications de médiation capables de s'auto-gérer. Plus précisément, la version autonomique de Cilia, que nous proposons, permet l'optimisation dynamique de l'utilisation des ressources de la plate-forme d'exécution et l'adaptation dynamique des chaînes de médiation (ajout, suppression, modification des médiateurs et des connecteurs). Elle permet également de présenter à tout moment un modèle compréhensible des phénomènes liés à l'exécution des chaînes et ainsi de faciliter le raisonnement et la prise de décision d'adaptation.

Le but de ce chapitre est de fournir une présentation de haut niveau de notre proposition, qui sera ensuite détaillée dans les chapitres suivants.

1 Problématique

Dans la première partie de ce manuscrit, nous avons d'abord présenté l'informatique orientée service. Cette nouvelle approche est basée sur les principes de faible couplage, de liaison retardée et de substituabilité de modules logiciels appelés service. L'informatique orientée service permet ainsi d'aborder de nouveaux domaines d'application caractérisés notamment par de fortes contraintes de dynamisme et par une faible prédictibilité. Nous avons également montré que, de façon non surprenante au vu de ses propriétés, l'informatique orientée service rencontre un réel succès pour la mise en place d'applications ambiantes (ou pervasives). Ces applications sont aujourd'hui de plus en plus répandues, et ce, même au niveau des usines. L'industrie manufacturière a en effet pris le virage des technologies de l'Internet et dispose actuellement de solutions qui permettent de relier les outils de production (ordres, gestion des stocks) et les systèmes d'information ; ce sont des progiciels de gestion intégrés tels que SAP ERP [1] ou PeopleSoft Entreprise [2] par exemple.

Dans une seconde partie, nous avons présenté un problème majeur soulevé par l'utilisation de services logiciels, à savoir l'intégration logicielle ; car, si l'informatique orientée service résout de nombreux problèmes liés à la technicité des protocoles (comment trouver une ressource ? comment l'appeler ? comment en changer ?), elle ne résout en aucune façon les problèmes d'interopérabilité syntaxique et d'alignement sémantique ni les problèmes liés aux aspects non-fonctionnels (sécurité, audit, qualité de service). Nous nous sommes alors intéressés aux solutions d'intégration de services. Nous avons naturellement étudié la notion d'*Enterprise Service Bus* (ESB). Un ESB est un bus d'intégration de services développant une approche assez similaire aux *Enterprise Application Integration* (EAI) mais en l'appliquant exclusivement aux services et en cassant l'architecture monolithique des EAI en plusieurs éléments d'intégration, plus facilement administrables et évolutifs. De façon générale, nous avons constaté un manque au niveau de la gestion du dynamisme et de l'adaptabilité des solutions actuelles.

Nous nous sommes intéressés en particulier au *framework* Cilia, développé par l'équipe Adèle [3]. Cilia est un ESB récent qui propose un modèle de programmation Java basé sur les notions de « *scheduler/processeur/dispatcher* » (voir section 3.4 « *Cilia* », page 86). Cilia apporte une réponse au problème difficile de l'intégration logicielle et permet une implantation naturelle des patrons d'intégration majeurs. L'utilisation de concepts spécifiques au domaine simplifie la création et le suivi des chaînes de médiation. Cependant, adapter des chaînes Cilia à de nouvelles conditions d'exécution demande toujours l'intervention d'administrateurs chevronnés et repose souvent sur l'arrêt des chaînes. Dans de nombreux domaines, les administrateurs ne sont pas disponibles ou les interruptions de service ne sont pas permises.

Enfin, dans une dernière partie, nous avons présenté la notion d'informatique autonomique. Nous pensons, en effet, que des propriétés autonomiques telles que l'auto-configuration, l'auto-réparation, l'auto-optimisation ou encore l'auto-protection peuvent permettre de faire face à la complexité croissante des besoins en intégration. Ces propriétés peuvent permettre de diminuer significativement les coûts d'exploitation et de maintenance, d'une part, et parer à l'absence d'administrateurs, d'autre part.

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

1. `http://www.sap.com/index.epx`
2. `http://www.oracle.com/us/lang/fr/applications/peoplesoft-enterprise.html`
3. `http://www-adele.imag.fr`

- **les services représentent un enjeu majeur en développement logiciel.** On rencontre, en effet, des services logiciels dans de nombreux domaines tels que la maison connectée, la ville intelligente ou l'industrie manufacturière ;
- **les ESB représentent une technologie complémentaire indispensable**. Il est, en effet, rapidement nécessaire d'intégrer des services hétérogènes afin de fournir des applications avancées. L'intégration ne peut se faire dans le code applicatif mais doit être clairement séparée et développée avec des outils spécifiques ;
- **la programmation des ESB est complexe**. Les modèles de programmation sont rarement adaptés à la mise en place aisée et sûre des patrons d'intégration majeurs. Le code d'intégration résultant est souvent compliqué et maîtrisé par un petit nombre d'experts ;
- **la maintenance des ESB est également 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 a contribué à diminuer la productivité et à augmenter les coûts de maintenance ;
- **l'ESB *open source* Cilia facilite le développement de solutions d'intégration.** Cilia est un *framework* à composants spécialisé pour la médiation de données, qui permet d'implanter de façon naturelle les patrons d'intégration majeurs. Il se caractérise également par une empreinte mémoire et une taille inférieures aux ESB du marché ; ce qui le rend plus facilement embarquable et utilisable pour l'informatique pervasive ;
- **dans certains domaines, l'environnement d'exécution est très dynamique**. Par exemple, les ateliers flexibles des usines intelligentes nécessitent l'intégration de dispositifs différents en fonction des ordres de fabrication. Il est dès lors nécessaire de mettre à jour les solutions d'intégration pour modifier le routage des messages, la transformation des données, etc. Pareillement, la gouvernance des entreprises exprime régulièrement de nouvelles exigences non-fonctionnelles (sécurité, traçabilité, persistance) nécessitant des mises à jour ;
- **l'arrêt des fonctions d'intégration est aujourd'hui mal accepté**. Le modèle économique des entreprises nécessite un fonctionnement en continu (99% de disponibilité correspond déjà à une indisponibilité de plus de trois jours et demi sur une année). Les activités, telles que le *e*-commerce ou le *e*-manufacturing, ne permettent pas les arrêts des fonctions d'intégration ;
- **l'informatique autonomique représente un axe d'évolution majeur pour les ESB**. La complexité des systèmes intégrés ne se réduira pas, tout du moins à moyen terme, et il devient impératif de trouver des solutions d'administration automatisées ne requérant pas d'arrêt. L'approche autonomique promeut en particulier l'auto-protection, l'auto-configuration, l'auto-réparation,l'auto-optimisation des solutions d'intégration. Cela permet d'envisager des administrations plus pérennes et moins coûteuses des développements effectués au-dessus des ESB ;
- **il n'existe pas aujourd'hui d'ESB autonomique**. Rendre un système autonomique est une entreprise très compliquée. L'informatique autonomique est au carrefour de plusieurs domaines : elle demande notamment des connaissances avancées en génie logiciel, en intelligence artificielle ainsi qu'en théorie du contrôle.

2 Objectifs

Le but de ce travail doctoral est de permettre la réalisation de solutions d'intégration autonomiques fondées sur l'ESB Cilia. Précisément, notre objectif est d'étendre Cilia afin de faciliter le développement de chaînes de médiation autonomiques ; c'est-à-dire capables de

s'auto-gérer. Il s'agit donc essentiellement de fournir aux développeurs de chaînes de médiation les moyens d'implanter ou de spécifier aisément des propriétés d'auto-gestion au niveau des chaînes de médiation.

Il faut bien comprendre que Cilia, comme tous les autres ESB, est un outil programmable. Il est bien sûr possible d'ajouter à Cilia un certain nombre de propriétés d'auto-gestion génériques, concernant essentiellement la configuration de la machine d'exécution. Cependant, la majeure partie des propriétés autonomiques attendues par les intégrateurs sont spécifiques aux chaînes de médiation exécutées et ne peuvent être implantées de façon générique au cœur de Cilia. Au contraire, elles donnent lieu à du code spécifique écrit par les programmeurs des chaînes. Aujourd'hui, ce travail est extrêmement complexe, faute de disposer des facilités nécessaires. **Le but de notre travail est dès lors de fournir des mécanismes et les interfaces associées pour permettre une programmation relativement aisée de capacités d'auto-gestion spécifiques.**

Il convient enfin de noter que Cilia possède un ensemble de propriétés techniques nécessaires à notre travail. En effet, Cilia est introspectable et repose sur une machine d'exécution dynamique (OSGi™ et Apache Felix iPOJO). Ce *framework* se caractérise également par une empreinte mémoire et une taille inférieures aux ESB du marché, ce qui le rend plus facilement embarquable et donc utilisable pour des applications pervasives ciblées par des entreprises telles que Schneider Electric.

De façon plus précise, nous pensons qu'une évolution au sein d'une chaîne de médiation peut se traduire des différentes façons suivantes :
- l'ajout ou le retrait d'une chaîne complète de médiation ;
- la modification des paramètres de configuration de la machine d'exécution ;
- la modification de paramètres de configuration au sein d'une chaîne ;
- l'ajout, le retrait ou le remplacement d'un médiateur au sein d'une chaîne ;
- la modification d'un *binding* entre médiateurs au sein d'une chaîne ;
- le changement des ressources utilisées par un adaptateur.

La gestion autonomique de ces différentes évolutions demande d'augmenter Cilia avec des capacités de perception de l'environnement d'exécution (externe et interne) et de modification dynamique des chaînes en cours d'exécution. Ces modifications doivent être contrôlées pour assurer la cohérence des chaînes à tout moment. Cela demande de revoir l'architecture interne de Cilia et de repenser les interfaces d'administration fournies.

Comme indiqué précédemment, certaines propriétés autonomiques peuvent être mises en œuvre de manière générique. Par exemple, la gestion de certaines propriétés de la machine d'exécution peut être réalisée grâce à des règles générales, valables dans tous les cas de figure. La plupart des propriétés néanmoins sont spécifiques aux applications traitées.

Le rôle du *framework* Cilia est dès lors de fournir toutes les informations nécessaires à une adaptation : quelle est la situation courante ? Quel est le niveau de performance des chaînes en cours d'exécution ? Quelles sont les tendances ? Quelles sont les possibilités de changement ? Où placer le code d'analyse et de prise de décision ? etc.

3 Proposition

3.1 Présentation générale

Afin d'atteindre les objectifs précédents, nous avons modifié le *framework* Cilia de façon à permettre aux développeurs d'insérer de multiples boucles de contrôle au sein des chaînes de médiation.

Chaque boucle de contrôle correspond à un gestionnaire autonomique tel que défini au chapitre 4 et est responsable de la gestion d'un aspect particulier d'une chaîne de médiation. Elle comprend une partie perception, une partie décision et une partie action, comme il se doit en informatique autonomique. Ces boucles de contrôle sont dirigées par des buts de haut niveau définis par des administrateurs.

Pour chaque chaîne de médiation, le programmeur peut ainsi créer une boucle de contrôle globale et des boucles de contrôle plus locales au niveau de chaque adaptateur. Au niveau global, le rôle de la boucle de contrôle est de maintenir la topologie et les configurations les plus appropriées de la chaîne de médiation en fonction des buts d'administration et du contexte d'exécution. Au niveau des adaptateurs, le rôle des boucles de contrôle est de garantir l'utilisation des meilleures ressources en fonction des buts d'administration et de l'environnement. Ces boucles sont locales et ne peuvent pas affecter d'autres éléments que les adaptateurs.

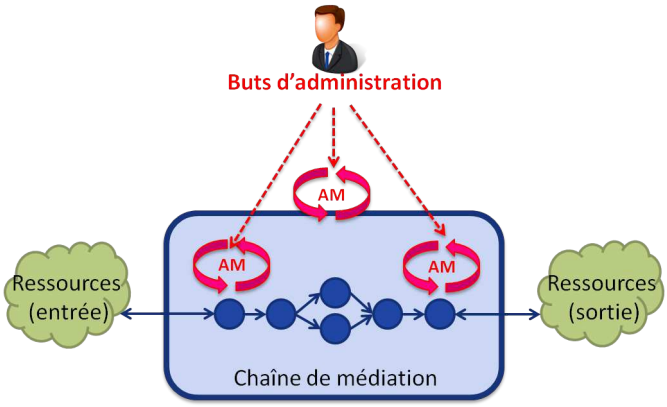


FIGURE 5.1 – Vision globale de Cilia autonomique.

Cette approche est illustrée par la figure 5.1 « *Vision globale de Cilia autonomique* ». Dans ce schéma, les boucles de contrôle sont symbolisées par l'abréviation AM [4] classiquement utilisé en informatique autonomique.

Au final, un code de médiation fondé sur Cilia comprend donc une partie spécifique aux opérations d'intégration ; c'est-à-dire une ou plusieurs chaînes de médiation, ainsi qu'une partie dédiée à l'auto-gestion de ces chaînes de médiation. Cette seconde partie est dirigée par les buts de l'administrateur qui peuvent évoluer au cours du temps et ainsi modifier les politiques de gestion. Comme nous le verrons, ces deux parties interagissent. Un enjeu important ici est de trouver le bon équilibre dans ces interactions de façon à ne pas impacter les performances tout en apportant les services d'auto-gestion attendus.

Chaque boucle de contrôle, ou gestionnaire autonomique, comprend une partie générique et une partie spécifique. La partie générique est directement implantée dans le code du *frame-*

4. Acronyme anglais de ***A**utonomic **M**anager*.

work Cilia. La partie spécifique est à la charge des programmeurs des chaînes de médiation. Les langages et les techniques utilisées pour les gestionnaires globaux et locaux sont différents. Cela est dû à la nature des informations manipulées (au niveau de la surveillance et des adaptations), à la complexité des stratégies d'auto-gestion à exprimer, mais aussi aux différentes échelles de temps de réaction attendue.

Pour les gestionnaires globaux (figure 5.2 « *Spécification des propriétés autonomiques de Cilia* »), le code générique est au cœur du *framework* Cilia et a demandé de nombreuses évolutions au niveau de l'architecture interne. Ce code apparaît sous la forme d'interfaces de surveillance et d'adaptation des chaînes. Ces interfaces sont contrôlables de façon fine et dynamique. La partie spécifique est apportée par les développeurs en utilisant le langage Java. Le code à développer est souvent complexe et très spécifique : le langage Java apporte toute latitude pour créer du code avancé. Au sein du *framework* Cilia, nous avons défini une classe Java destinée à abriter ce code. Cette classe peut accéder aux interfaces de surveillance et d'adaptation précédemment mentionnés.

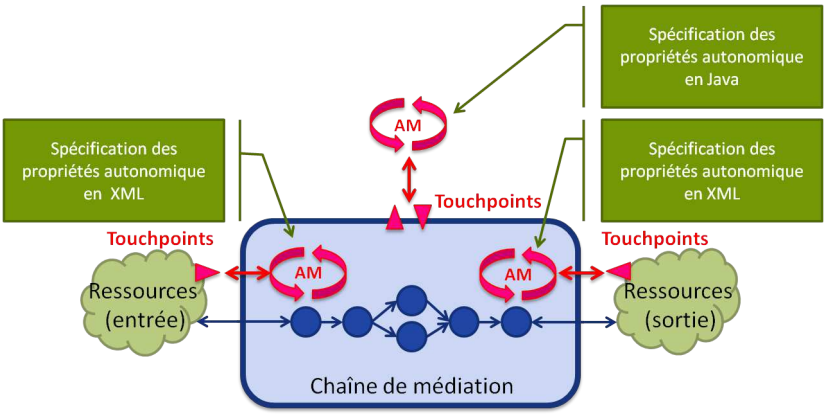


FIGURE 5.2 – Spécification des propriétés autonomiques de Cilia.

Pour les gestionnaires locaux (figure 5.2 « *Spécification des propriétés autonomiques de Cilia* »), le code générique se charge de la surveillance de l'environnement et de la notification de l'arrivée ou du départ de ressources (sous forme de services). L'implantation de ce code a également demandé de revoir en profondeur la façon de gérer les ressources au niveau des adaptateurs. La partie spécifique est apportée par les programmeurs à l'aide d'un langage dédié que nous avons défini. Ce langage permet d'exprimer les politiques de gestion des ressources selon une syntaxe XML. Il permet de spécifier les ressources attendues, le nombre de ressources utilisables, les conditions de changement, etc.

Nous insistons sur le fait que l'implantation des parties génériques est complexe. Il est important, par exemple, que la surveillance soit configurable pour correspondre au mieux à la situation courante et qu'elle n'impacte pas les performances au-delà des limites permises. De même, les adaptations doivent garantir la conservation des données et des états propres à l'exécution. Les différents types de boucles de contrôle demandent des techniques de surveillance, d'adaptation et de raisonnement différentes. Cela est dû à la nature des informations manipulées mais aussi aux différentes échelles de temps de réaction attendue.

3.2 Buts d'administration

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

Les buts prennent généralement la forme d'objectifs métier devant être réalisés par le système. La plupart du temps, ils sont exprimés comme des critères caractérisant l'état d'un système. La détermination des actions à mettre en œuvre pour satisfaire ces critères et leur réalisation sont laissées à la charge des gestionnaires autonomiques. Dans notre cas, les buts correspondent à des critères permettant de qualifier et d'orienter l'exécution des chaînes de médiation. Les buts peuvent être décomposés en sous-buts permettant d'exprimer des directives plus précises (voir figure 5.3 « *Interactions entre boucles de contrôle* »).

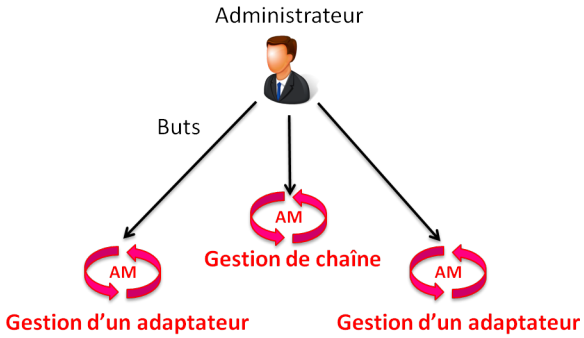


FIGURE 5.3 – Interactions entre boucles de contrôle.

3.3 Adaptation autonomique des chaînes de médiation

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

3.3.1 Monitoring

Les interfaces de surveillance permettent de donner, à tout moment, différents paramètres concernant la machine d'exécution (mémoire, taille du pool de thread, etc.) et la topologie de la chaîne de médiation. Elles permettent également de qualifier le niveau de performance d'une chaîne de médiation. Pour ce faire, nous avons défini un ensemble de variables numériques qui permettent de suivre les phénomènes dynamiques et statiques du système. Nous nous sommes pour cela inspirés des variables d'état de la théorie du contrôle [Ogata, 1997, Friedland, 2005]. Pour rappel, K. OGATA donne la définition suivante de l'état d'un système dynamique et des variables d'état :

Etat *: « L'état d'un système dynamique est le plus petit ensemble de variables tel que la connaissance de cet ensemble à un instant donné suffit à déterminer le comportement du système futur. »*

Variables d'état *: « Ce sont les variables qui constituent l'état du système. »*

[Ogata, 1997]

Les grandeurs collectées pour ces variables sont appelées mesures. Chaque mesure est horodatée à la source. L'horodatage généralisé de toutes les mesures permet de reconstruire les changements d'états de façon chronologique. Pour chaque variable , l'interface de surveillance permet de spécifier quatre seuils : « Très bas », « Bas », « Haut » et « Très haut » et d'orienter le fonctionnement de la collecte de données.

Dès lors qu'une variable franchit un des seuils (valeur au-dessous de « Bas » ou de « Très bas », au-dessus de « Haut » ou de « Très haut »), le *framework* Cilia émet un événement. Ces événements correspondent à des alertes ou à des alarmes en fonction du niveau de dépassement. Ils peuvent naturellement être utilisés par un gestionnaire autonomique. Les seuils permettent de simplifier le fonctionnement du gestionnaire autonomique.

En effet, si toutes les variables fournissent des mesures dans leur zone de bon fonctionnement, le système est stable et le gestionnaire n'intervient pas. Dans le cas contraire, le gestionnaire autonomique est informé par le *framework* Cilia qu'une mesure hors zone de bon fonctionnement a été prise (c'est-à-dire correspondant au dépassement du seuil d'alarme ou du seuil d'erreur).

Nous avons défini trois familles de variables d'état :

- des variables d'état permettant de qualifier la situation des flots de communication. Ces variables renseignent notamment sur le nombre d'appels, les temps d'exécution de chaque constituant des médiateurs (*Scheduler*, *Processor*, *Dispatcher*) et les temps de transfert des connecteurs ;
- des variables d'état permettant de qualifier l'environnement d'exécution. Ces variables notent et datent le départ et l'arrivée des *proxies* en charge de la communication avec les sources de données et les consommatrices de données ;
- des variables d'état spécifiques permettant d'auditer le code métier en exécution. Pour ce faire, les développeurs peuvent annoter des variables du code métier au niveau du *Scheduler*, du *Processor* et du *Dispatcher*. Les accès en lecture ou en écriture déclenchent une sauvegarde dans une variable d'état de la valeur de la variable annotée. Ces variables peuvent être utilisées pour sauvegarder des données suite à des événements. Par exemple, une exception Java est déclenchée sur un débordement de *buffer*, le code métier traite l'exception et notifie le médiateur. Le contenu de la notification est à la charge du développeur, mais il est sauvegardé et horodaté dans une variable d'état. Dans cet exemple, le gestionnaire autonomique peut être informé qu'une exception a été levée et traitée dans le code métier.

Ces différents types de variables d'état permettent de modéliser la dynamique d'exécution des chaînes de médiation, le code métier des médiateurs et adaptateurs ainsi que l'apparition et la disparition des sources de données et des consommateurs de données.

3.3.2 Adaptation

Les interfaces d'adaptation permettent la modification des chaînes de médiation (configuration et topologie) et de ses différents constituants (adapteurs, médiateurs, connecteurs). Les modifications sont effectuées dynamiquement et de manière très contrôlée. Nous avons, en effet, mis en place un protocole de quiescence permettant de mettre en veille les parties d'une chaîne de médiation devant être modifiées à chaud. Les parties mises en veille et modifiées sont ensuite redémarrées sans perte de message ni incohérence au sein du code. Nous étudierons en détail ce protocole de quiescence dans le chapitre suivant.

Le *framework* Cilia offre un ensemble de facilités pour identifier de façon unique les éléments à modifier. En particulier, des filtres LDAP peuvent être utilisés dans les interfaces d'adaptation pour accéder aux éléments logiciels à mettre à jour.

Il faut enfin noter que la séparation des interfaces de surveillance et d'adaptation apporte une simplification de maintenance et d'évolution du *framework* Cilia.

3.4 Base de connaissances

Cilia fournit une facilité supplémentaire aux développeurs sous la forme d'une base de connaissances (voir figure 5.4 « *Base de connaissances* ») automatiquement construite et mise à jour par le *framework*. Cette base de connaissances est une couche d'abstraction qui découple les interfaces de surveillance et d'adaptation du cœur des gestionnaires autonomiques globaux.

Précisément, la base de connaissances collecte des données (des variables d'état) de façon à construire une représentation de l'application actuelle en exécution. Elle offre ainsi la vision courante de l'exécution et de son environnement. La base de connaissance complète ses informations avec une sélection d'enregistrements configurables qui caractérisent l'application de médiation dans le temps et au fil de ses adaptations. La base de connaissances comprend ainsi une représentation synthétique de l'exécution de l'application de médiation avec une dimension historique. Cette base de connaissances constitue un modèle causal [Jackson, 2002, Muller et al., 2009]. En effet, une modification effectuée sur la représentation est reportée dans l'application de médiation en exécution, avec une faible latence.

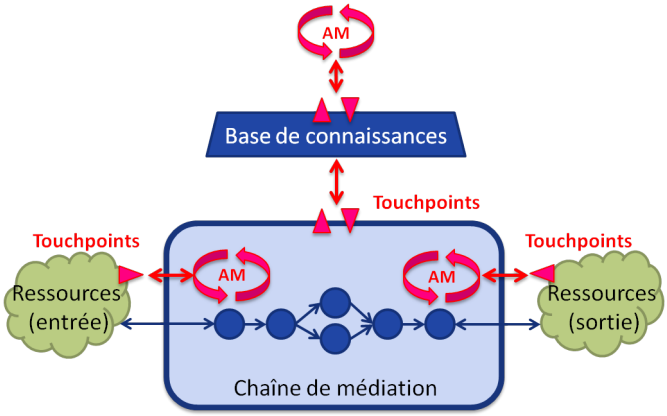


FIGURE 5.4 – Base de connaissances.

Les interfaces de surveillance et d'adaptation complétées par la base de connaissances primaire configurable correspond à l'architecture fonctionnelle décrite par IBM [IBM, 2006] pour réaliser des systèmes autonomiques (voir section 4 « *Architecture* », page 112).

Les boucles autonomiques peuvent au choix être développées directement avec les interfaces techniques offertes par les capteurs et les effecteurs, ou en utilisant la base de connaissances offerte par le *framework* Cilia. Bien évidemment, construire une boucle autonomique au-dessus de ce modèle causal est plus simple à réaliser ; une partie de la base de connaissances configurable est déjà implantée.

La collecte des données est elle aussi dynamique et flexible. En effet, la surveillance des données est configurable, activable, dé-activable à la volée et l'adaptation conserve les états.

4 Conclusion

La complexité des applications, l'hétérogénéité combinée avec un environnement extrêmement dynamique rend les applications d'intégration difficiles à faire évoluer et à garantir leur robustesse d'exécution.

Le but de notre travail est de permettre la réalisation d'applications de médiation autonomiques fondées sur l'ESB Cilia. Il s'agit d'étendre Cilia afin de faciliter le développement de chaînes de médiation autonomiques ; c'est-à-dire capables de s'auto-gérer. Nous cherchons ainsi à fournir aux développeurs de chaînes de médiation les moyens d'implanter ou de spécifier aisément des propriétés d'auto-gestion au niveau des chaînes de médiation.

Dans ce chapitre, nous avons présenté la vision générale de notre approche. Nous synthétisons les différents éléments de notre approche par trois points :

1. une architecture qui met en place des multi-boucles de contrôles ;
2. une adaptation autonomique des adaptateurs rend disponible les meilleures ressources en fonction des évolutions internes et externes. Le *framework* RoSe [5] fait réagir la boucle de contrôle à ces changements de ressources, elle sélectionne les ressources en fonction des objectifs définis par les administrateurs, les adaptateurs qui sont des composants orientés service dynamique autorise le changement d'architecture d'application à la volée.
3. une boucle de contrôle des chaînes de médiation qui effectue des adaptations topologiques et de configuration par l'intermédiaire d'une base de connaissances. Celle-ci est une représentation causale synthétique et historique de l'application de médiation en exécution.

La mise en place de « Cilia autonomique » a demandé de nombreuses évolutions au niveau de l'architecture interne de Cilia et l'ajout d'une somme importante de code nouveau. Nous avons, en particulier, mis en place un protocole de type « *meta-object protocol* » [Kon et al., 2002] pour permettre un monitoring dynamique. Egalement, un protocole de quiescence relativement complexe a du être mis en œuvre. Nous présentons ces éléments plus techniques dans le chapitre suivant.

5. RoSe est un *framework* disponible à `https://github.com/AdeleResearchGroup/RoSe`, il permet la gestion des dispositifs externes à la passerelle OSGi.

6

Implantation et réalisation

Dans le chapitre précédent, nous avons proposé de rendre autonomique le *framework* Cilia. Notre proposition consiste à donner la possibilité à des développeurs de mettre en place des boucles de contrôle lors du développement d'applications de médiation. Ces boucles de contrôle sont dirigées par des buts de haut niveau définis par des administrateurs. Rappelons que Cilia est un *framework* de médiation ; par conséquent, notre approche se concrétise par la mise à disposition d'interfaces de programmation rendant plus aisée la réalisation des propriétés d'auto-gestion de l'application exécutée par le *framework* Cilia.

Nous débuterons ce chapitre par la présentation de l'architecture générale d'auto-gestion du *framework* Cilia. Puis, nous présenterons le *framework* RoSe et son utilisation dans le cadre de l'ESB Cilia. Nous continuerons par l'adaptation autonomique des adaptateurs. A ce niveau de lecture, nous aurons ainsi apporté à l'ESB Cilia la sensibilité aux ressources d'entrée et de sortie. Nous poursuivrons par la seconde partie, qui permettra d'atteindre notre objectif ; c'est-à-dire de fournir un ensemble d'interfaces de programmation permettant de présenter, à tout moment, un modèle de l'application en exécution, mais aussi des interfaces programmatiques pour la modification de l'application en exécution.

1 Rappels

1.1 Approche

Comme introduit dans le chapitre précédent, notre proposition consiste à donner la possibilité aux développeurs d'applications de médiation Cilia de mettre en place des boucles de contrôle à différents niveaux et de façon relativement aisée. Ces boucles de contrôle sont dirigées par des buts de haut niveau définis par des administrateurs.

Comme introduit dans le chapitre précédent, notre proposition consiste à donner la possibilité aux développeurs d'applications de médiation Cilia de mettre en place des boucles de contrôle à différents niveaux et de façon relativement aisée. Ces boucles de contrôle sont dirigées par des buts de haut niveau définis par des administrateurs.

- **au niveau des adaptateurs** qui sont alors chargés de sélectionner et d'utiliser les meilleures sources ou destinations (ou ressources) possibles pour la chaîne de médiation. Ces boucles sont locales et ne peuvent pas affecter d'autres éléments que les adaptateurs.
- **au niveau global de la chaîne**. Là, la boucle de contrôle a pour rôle de gérer l'ensemble de la chaîne (structure, configuration et qualité de service) pour s'adapter au mieux aux évolutions de contexte et de stratégies spécifiées par les administrateurs.

Chaque boucle de contrôle correspond à un gestionnaire autonomique comprenant une partie perception, une partie décision et une partie action, comme proposé en informatique autonomique, notamment dans les architectures de référence d'IBM [IBM, 2006].

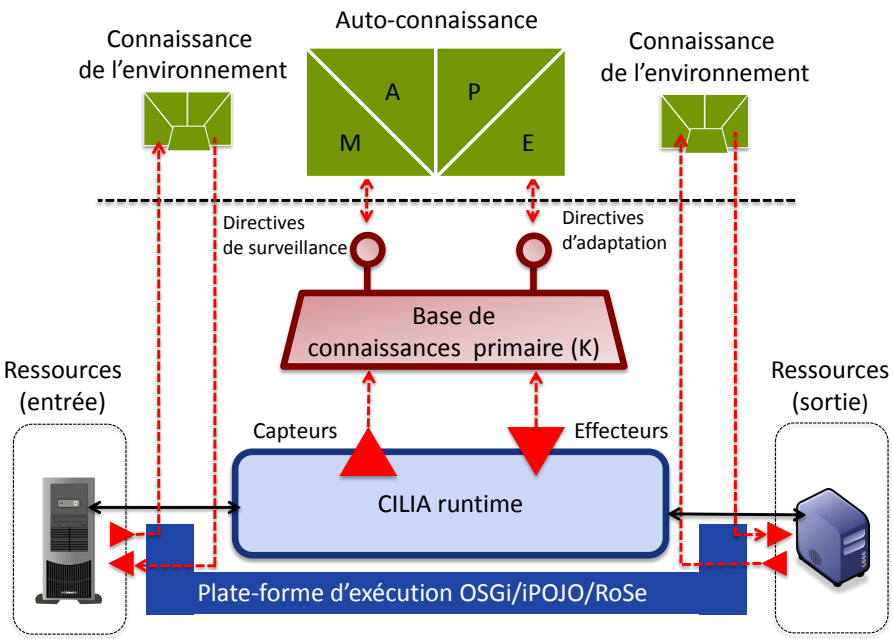


FIGURE 6.1 – Vision globale de Cilia autonomique.

Comme indiqué par la figure 6.1 « *Vision globale de Cilia autonomique* », même si les différentes boucles autonomiques implantent la même architecture MAPE-K, elles reposent sur des formalismes et des techniques de calcul différents. Les adaptateurs scrutent l'environnement pour suivre l'apparition et la disparition des équipements et mettre en œuvre des techniques de décision simples (*e.g.*, filtre LDAP). La boucle de contrôle générale, quant à elle, repose sur une base de connaissances causale complexe et peut effectuer des calculs d'analyse et de planification reposant sur du code complexe, éventuellement long.

1.2 Apache Felix iPOJO

Techniquement parlant, Cilia repose sur les technologies à service OSGi™et Apache Felix iPOJO. En effet, un médiateur constitué des classes Java pour les fonctions *scheduler*, *dispatcher* et *processor* est transformé en un certain nombre de composants iPOJO (nous verrons qu'il y en a 5 au minimum). Nous donnons ici quelques éléments de rappel sur le modèle iPOJO qui nous seront nécessaires dans la suite de ce chapitre.

Le *framework* iPOJO a pour objectifs la simplification de la construction d'applications à services dynamiques. Il cache la complexité de la gestion du cycle de vie des composants OSGi et de la composition structurelle des services, source d'erreur majeure en OSGi. Le modèle à composant orienté service iPOJO utilise la technologie des conteneurs pour apporter des propriétés non-fonctionnelles et effectuer un certain nombre de traitements. Un conteneur peut se voir adjoindre des *handlers* qui correspondent à du code appelé en interception sur un appel de méthode, une modification d'attributs, etc.

Comme illustré par la figure 6.2 « *Composant iPOJO* », un conteneur iPOJO est constitué par au moins quatre *handlers* techniques insérés par défaut.

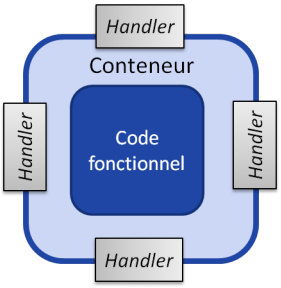


FIGURE 6.2 – Composant iPOJO.

Le *framework* iPOJO est extensible et permet au développeur d'écrire ses propres *handlers* techniques. Un *handler* iPOJO est lui même implanté sous la forme d'un composant iPOJO et peut utiliser un autre *handler* iPOJO. Comme nous le verrons dans ce chapitre, cette propriété remarquable a été utilisée pour l'implantation de la fonction *monitoring*.

Enfin, rappelons qu'un composant iPOJO peut être décrit en remplissant un fichier XML, en utilisant des annotations JAVA ou bien en manipulant directement une API fournie à cet effet. Et finalement, le code du composant iPOJO peut être intercepté et manipulé ; ce qui permet l'injection de méthodes et de variables.

2 Adaptateurs autonomiques

Le but de cette section est de décrire la conception et l'implantation des boucles de contrôle au niveau des adaptateurs. Cette boucle repose sur l'utilisation du *framework* RoSe pour l'import et l'export de ressources et sur un ensemble de composants que nous avons ajoutés au *framework* Cilia pour faire le lien entre les spécifications Cilia et les ressources.

Nous allons détailler l'architecture générale permettant de lier le *framework* RoSe à celui de Cilia. Puis, nous expliquerons les mécanismes qui permettent de réaliser une adaptation autonomique des adaptateurs.

2.1 Principes

Rappelons que le *framework* Cilia [Garcia Garza, 2012] définit deux types de connecteurs : les connecteurs internes et les connecteurs externes. Les connecteurs internes réalisent la connexion de deux composants Cilia. Les connecteurs externes permettent de communiquer vers des ressources externes comme des services (Twitter, par exemple), des applications (une base de données, par exemple) et des dispositifs (capteurs de température, mesure de consommation électrique, contacteurs électriques, etc.). Dans la terminologie de Cilia, l'entité connecteur externe est appelée **adaptateur**. Un adaptateur est défini comme étant un médiateur particulier, avec un code de connexion vers l'extérieur. Ce médiateur est constitué d'un *scheduler* de type *immediate* (le *scheduler* transmet le message au *processor* sans attente), d'un *processor* vide (le message est transmis tel quel au *dispatcher*) et d'un *dispatcher* de type *multicast* (le message est transmis sur tous les connecteurs de type interne connectés à l'adaptateur).

Le DSL Cilia permet d'exprimer la chaîne de médiation. Précisément, c'est le langage de description des instances des adaptateurs, des médiateurs et des liens qui permettent de relier des éléments (médiateurs et adaptateurs). Le code source 6.1 « *Structure du Cilia DSL (extrait)* » illustre la structure XML générale du DSL Cilia.

```
1  <?xml version="1.0" encoding="UTF-8" standalone="no" ?>
   <chain>
3    <mediators>
       <!— mediators instances —>
5    </mediators>
     <adapters>
7      <!—  adapteurs instances—>
     </adapters>
9    <binding>
       <!— link between adapters/mediators or mediators/mediators  —>
11   </bindings>
   </chain>
```

Code source 6.1 – Structure du Cilia DSL (extrait).

Un adaptateur est décrit en XML à l'aide du langage de spécification DSCilia :

- `<adapter> :=` décrit un adaptateur, il contient un élément appelé *collector* et deux attributs qui sont le *name* (type du médiateur) et le *pattern* (connecteur de messages entrant, de messages sortant ou de messages entrant et sortant) ;
- `<collector> :=` décrit la référence vers l'entité qui réalise la connexion. Le *collector* n'a qu'un seul attribut : le type (le nom du *collector*).

Le langage de description des chaînes de médiation Cilia est un DSL statique. Il décrit des médiateurs de transformation de données, des médiateurs d'entrée et de sortie, ainsi que la topologie de ces médiateurs. Pour rendre les adaptateurs auto-gérés, nous avons donc dû :

– étendre le DSL Cilia de façon à introduire une certaine forme d'abstraction (ou de variabilité). Ces éléments de langage, détaillés dans la section 2.2 « *Extension du langage* », page 143, permettent de configurer le comportement dynamique des adaptateurs.
– modifier la machine d'exécution Cilia afin que les adaptateurs puissent prendre des décisions locales (en accord avec leur spécification) et donc être capables de s'adapter aux conditions externes. Techniquement parlant, nous avons créé un nouvel *handler* iPOJO pour la liaison aux ressources au sein de la machine Cilia.

Nous présentons ces deux extensions dans les deux sections suivantes.

2.2 Extension du langage

Comme indiqué ci-dessus, nous avons étendu le langage DSCilia pour apporter de la variabilité et donc de la flexibilité à l'exécution. Précisément, nous avons ajouté dans les langages des adaptateurs les aspects suivants :

`<filter>` `:=` *cet attribut permet de spécifier un filtre sur les ressources recherchées ;*

`<cardinality>` `:=` *cet attribut spécifie les nombres maximal et minimal de ressources que peut traiter un adaptateur ;*

`<ranking>` `:=` *cet attribut est un booléen. Lorsqu'il est à vrai, les ressources sont classées à l'aide d'une fonction de coût ;*

`<immediate>` `:=` *cet attribut est un booléen. Lorsqu'il est positionné à vrai, le classement des ressources est modifiable à la volée.*

Le code source 6.2 « *Configuration du comportement dynamique d'une instance d'adaptateur (extrait)* » présente un exemple de la configuration d'une instance d'un adaptateur CILIA avec le langage étendu.

```
<adapter-instance type="sample-slave-adapter" id="SampleEntry" >
    <dependency filter ="(domain.id=carros-06-building-#B2-room-#1)"
                cardinality="1..2" ranking="true" immediate="true" />
</adapter-instance>
```

Code source 6.2 – Configuration du comportement dynamique d'une instance d'adaptateur (extrait).

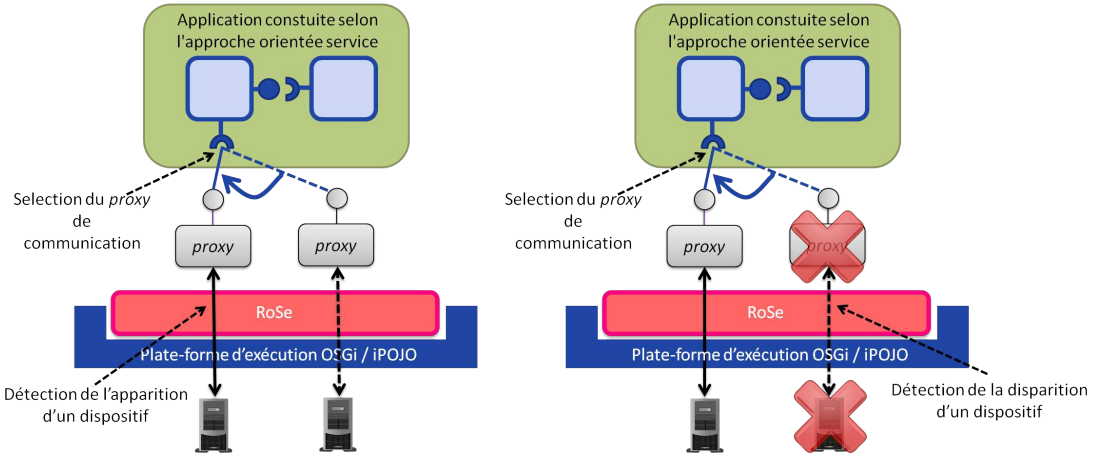
2.3 Modification de la machine d'exécution de Cilia

2.3.1 RoSe

Pour rendre Cilia sensible au contexte externe, nous utilisons le *framework* RoSe. L'approche de RoSe repose sur la réification des ressources découvertes sous forme de services OSGi ou iPOJO. Cette réification n'est pas liée au protocole de découverte. Le mécanisme est, en effet, le même pour découvrir des ressources utilisant, par exemple, le protocole WS-Discovery [OASIS, 2009] que pour des ressources basées sur Ethernet (*e.g.*, requête ICMP [1]).

1. Acronyme de *Internet Control Message Protocol* (RFC 792). ICMP est un protocole qui contrôle les erreurs de transmission. Il est couramment utilisé par une machine émettrice pour détecter la présence ou non d'une machine distante sur un réseau IP.

Les figures 6.3(a) et 6.3(b) illustrent le principe de la réification des services. Le *framework* RoSe est notifié de la découverte d'une ressource distante. Il détermine si la ressource doit être importée en utilisant des règles de configuration spécifiée par l'administrateur. Dans le cas où la ressource doit être importée, le *framework* RoSe instancie un *proxy* sous la forme d'un service. Ce service représente la ressource distante. Avant l'instanciation, RoSe peut appeler un service dédié pour ajouter des méta-données dans le *proxy* réifié. Le *proxy* est ajouté au registre de services approprié. Ainsi, une application construite selon l'approche orientée service peut « naturellement » utiliser ce service *proxy*.



(a) Apparition d'une ressource.

(b) Disparition d'une ressource

FIGURE 6.3 – RoSe : apparition et disparition d'un proxy.

La 6.3(b) illustre la disparition d'un service. Lorsqu'une ressource distante n'est plus disponible, le *proxy* la représentant doit être détruit (retiré du registre). C'est le *framework* RoSe prend en charge la suppression de ce *proxy* mais, pour cela, il doit être prévenu au préalable de la disparition de la ressource. Généralement, la disparition de la ressource est notifiée par le protocole de découverte. Dans le cas contraire, il faut ajouter des mécanismes de surveillance des ressources utilisées.

RoSe est extensible. Cela signifie que des API sont disponibles pour ajouter ou retirer un protocole de découverte. Comme nous le verrons ci-après, nous avons utilisé cette propriété pour ajouter le protocole industriel Modbus, fréquemment mis en œuvre chez Schneider-Electric.

L'intégration du *framework* RoSe est la base pour rendre Cilia sensible à son contexte. L'adaptateur, tel que nous l'avons précédemment rappelé, est un médiateur minimal qui contient un objet de type *collector*, chargé de la communication et de la collecte de données. La figure 6.4 « *Adaptateur avec le framework RoSe* » illustre l'architecture que nous avons mise en place et validée au cours de notre travail doctoral. On voit apparaître deux composants importants : « *factory* » et « Découverte », que nous détaillons dans les sections à venir.

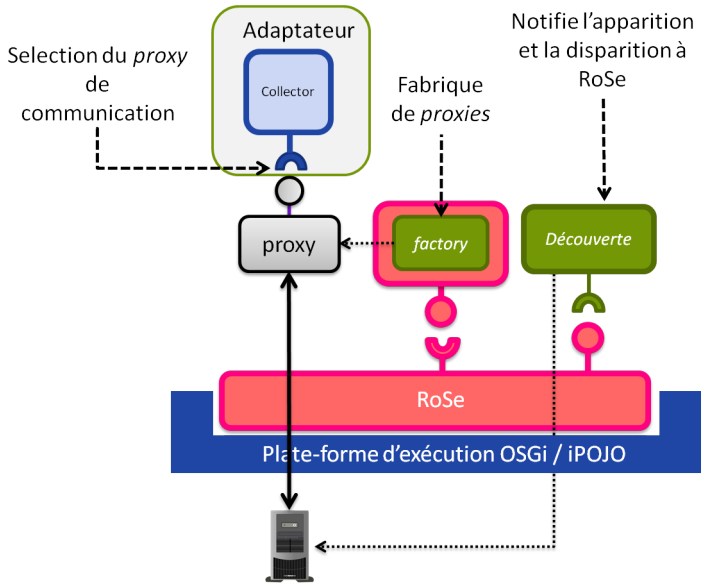


FIGURE 6.4 – Adaptateur avec le *framework* RoSe.

2.3.2 Principes

La solution technique, que nous avons mise en place, est conforme à l'architecture et aux API définies par RoSe. Elle repose essentiellement sur deux composants iPOJO :

- un composant pour la **gestion de disponibilité dynamique** des équipements ;
- un composant pour la **gestion des *factories*** qui permettent la création des *proxies*.

Précisément, nous avons développé le composant iPOJO traitant le protocole Modbus. Lors de l'apparition d'une ressource externe utilisant ce protocole, le composant *Découverte* génère un *endpoint* et le transmet à RoSe. Un *endpoint* est un objet avec des méta-données qui renseigne la ressource découverte. Inversement, lors de la disparition de la ressource, le composant de découverte notifie RoSe de la disparition du *endpoint*.

RoSe se base ensuite sur les règles de configuration définies par l'administrateur pour décider s'il faut ou non créer un *proxy* via la fabrique prévue à cet effet. Techniquement, la fabrique de *proxy* est la réalisation d'une interface technique du *framework* RoSe. Grâce à la technique d'injection de dépendances fournie par RoSE et iPOJO, le développeur n'a pas à spécifier les dépendances du composant qui implémente la fabrique du proxy.

La configuration de RoSe s'effectue directement via l'API de RoSe ou bien via un fichier de configuration [Bardin, 2012]. Le code source 6.3 « *Configuration du framework RoSe (extrait)* », page 145 montre un exemple de configuration par fichier. Les connexions décrivent les services qui doivent être importés ou exportés par RoSe. Une *connection in* définit une ressource qui doit être importée dans une application. Inversement, une *connection out* permet d'exporter une ressource suivant un protocole donné (REST par exemple).

```
"machine" :
{
  "id" : "modbus-A7D468EA-1BE7-4E1A-B8AC-E246B14F3033",
  "host" : "localhost",
  "connection" :[
                  { "in": { "endpoint_filter" : "(service.imported=*)",
                            "protocol" : ["Modbus/TCP"]
                          },
                  }
                ],
  "instances" : [
                  { "factory" : "ModbusTCP.importer" },
                  { "factory" : "ModbusTCP.discovery" ,
```

```
                    "properties" : { "modbus.port.number" : "502",
                                     "scan.period" : "1000" ,
                                     "scan.delay" : "1000" ,
                                     "ping.time.out" : "1000",
                                     "start.ip.address" : "10.10.7.20",
                                     "end.ip.address" :"10.10.7.80"
                                   }
                  },
                ]
}
```

Code source 6.3 – Configuration du *framework* RoSe (extrait).

Pour traiter le protocole Modbus, nous avons créé deux composants :

- `ModbusTCP.importer` : le conteneur de la fabrique de *proxies* ;
- `ModbusTCP.discovery` : le service de découverte avec ses paramètres de configuration (`start.ip.address`, `end.ip.address`, etc.).

L'adaptateur est un conteneur de *collector*. Ce dernier a une dépendance fonctionnelle vers les proxies (voir figure 6.4 « *Adaptateur avec le framework RoSe* » page 145). Ainsi, un *collector* est capable de :

- **obtenir** les *proxies* instanciés par une fabrique spécifique ;
- **classer** les *proxies* selon une fonction de coût fournie par le développeur ;
- **sélectionner** un nombre minimal et maximal dans cet ensemble ordonné de *proxies* ;
- **modifier** l'ordonnancement des *proxies*.

Les points 2 à 4 sont optionnels. Nous avons implanté ces propriétés sous la forme d'un *handler* iPOJO. Précisément, nous avons étendu le *handler* de dépendances de iPOJO de façon à réaliser les quatre propriétés décrites précédemment. Nous décrivons dans la section suivante la façon dont nous avons implanté ces quatre propriétés.

2.3.3 Implantation du nouvel *handler* iPOJO pour l'adaptateur

Point 1 : retrouver un proxy en fonction de la fabrique de *proxy*. Le mécanisme est identique à celui d'iPOJO. Le *proxy* expose des propriétés de niveau service. L'adaptateur sélectionne le(s) *proxy(ies)* en définissant un filtre selon la syntaxe LDAP. Ce filtre est rendu modifiable et disponible dans le DSL Cilia.

Le code source 6.4 « *Configuration d'un adapteur-filtre (extrait)* » illustre un collecteur prenant la forme d'un tableau `m_proxies` de classes et contenant des références de *proxies*. Ces *proxies* représentent des ressources distantes qui satisfont le filtre LDAP (mot clé *filter*).

```
<collector
  classname="fr.liglab.adele.cilia.sample.SampleCollector"
                name="sample-in-collector" namespace="fr.liglab.adele.cilia">
  <cilia:dependency  field="m_proxies"
                     filter="(&(domain.id=carros-building)(model.name=TempMeter-*))" >
  </cilia:dependency>
</collector>

<adapter name="sample-in-adapter" pattern="in-only" >
   <collector type="sample-in-collector" />
   <ports>
<out-port name="unique" type="*" />
   </ports>
</adapter>
```

Code source 6.4 – Configuration d'un adapteur-filtre (extrait).

Point 2 : classer les *proxies* selon une fonction de coût. Une réalisation simple et native consiste à utiliser le classement des services effectué par OSGi. OSGi ordonne les services par ordre croissant selon la valeur numérique de la propriété `service.ranking`. Appliqué à notre exemple, la fabrique de *proxies* (voir figure 6.4 « *Adaptateur avec le framework RoSe* »

page 145) ajoute la propriété `service.ranking` avec la valeur issue d'un calcul, d'un algorithme, d'une valeur prédéfinie dans un fichier, etc. Le mot clé *ranking* (*c.f.* code source 6.5 « *Configuration d'un adapteur-cardinalités (extrait)* ») indique si la fonction de coût est prise en compte ou non (si le classement est effectué ou non avant l'injection de référence de services).

Point 3 : sélectionner un nombre minimal/maximal de *proxies*. Cette fonctionnalité nous permet de rendre un adaptateur invalide (au sens iPOJO) si le nombre minimal de services injectés n'est pas atteint. Il permet aussi de maximiser le nombre d'injections de références de services. Dans notre exemple, l'adaptateur ne pourra être valide avant que deux services correspondent au filtre LDAP. De plus, quatre références au maximum seront injectées (*c.f* code source 6.5 « *Configuration d'un adapteur-cardinalités (extrait)* »).

```
<collector
  classname="fr.liglab.adele.cilia.sample.SampleCollector"
                name="sample-in-collector" namespace="fr.liglab.adele.cilia">
  <cilia:dependency  field="m_proxies"
                     filter="(&(domain.id=carros-building)(model.name=TempMeter-*))"
                     ranking="true"
                     cardinality="2..4"
                     immediate="true">
  </cilia:dependency>
</collector>
<adapter name="sample-in-adapter" pattern="in-only" >
   <collector type="sample-in-collector" />
   <ports>
<out-port name="unique" type="*" />
   </ports>
</adapter>
```

Code source 6.5 – Configuration d'un adapteur-cardinalités (extrait).

Point 4 : modifier l'ordonnancement des *proxies*. Cette fonctionnalité permet de figer ou non l'ensemble des *proxies*. Le mot clé `immediate` permet de déclencher ou non une remise à jour de la dépendance de services. Cette propriété sert à maintenir (si elle est positionnée à `true`) un nombre borné de services injectés avec les meilleures qualités de service (`service.ranking`) disponibles.

En effet, si la cardinalité maximale est définie et atteinte (il y a déjà le nombre maximal de services injectés) alors, celui ayant la plus faible valeur de coût (`service.ranking`), est remplacé par celui qui vient d'apparaître. Si la propriété `immediate` est positionnée à `false`, il n'y a pas de réévaluation des services injectés (voir code source 6.6 « *Configuration d'un adapteur (extrait)* », page 147).

```
<collector
  classname="fr.liglab.adele.cilia.sample.SampleCollector"
                name="sample-in-collector" namespace="fr.liglab.adele.cilia" >
  <cilia:dependency  field="m_proxies"
                     filter="(&(domain.id=carros-building)(model.name=TempMeter-*))"
                     ranking="true"
                     cardinality="2..4"
                     immediate="true">

  </cilia:dependency>
</collector>
<adapter name="sample-in-adapter" pattern="in-only" >
   <collector type="sample-in-collector" />
   <ports>
      <out-port name="unique" type="*" />
   </ports>
</adapter>
```

Code source 6.6 – Configuration d'un adapteur (extrait).

3 Boucle de contrôle globale

Une boucle de contrôle globale au *framework* Cilia a pour rôle de gérer les chaînes dans leur ensemble et de les adapter de façon autonomique en fonction des évolutions internes et externes. Pour faciliter la mise en œuvre d'une telle boucle, nous avons conçu et développé les éléments suivants :

- des **capteurs** permettant d'acquérir des informations sur les chaînes en exécution et des effecteurs permettant de modifier ces mêmes chaînes ;
- un ***monitoring*** dynamique permettant de suivre les chaînes de médiation à l'exécution ;
- des **capacités de modification dynamique** sur des chaînes de médiation sans arrêt de l'exécution.

Pour cela, nous avons mis en place une architecture réflexive permettant de maintenir en permanence le lien entre code de bas niveau et concepts métier de plus haut niveau d'abstraction. Une telle architecture est reconnue comme étant une solution efficace et élégante. Il est également communément accepté qu'une telle architecture est difficile à mettre en œuvre, notamment pour des raisons de synchronisation. Nous examinons en détails ces différents éléments techniques dans les sections suivantes.

3.1 Les *touchpoints* : capteurs et effecteurs

Le *framework* Cilia fournit deux points de contrôle (capteurs et effecteurs) permettant de surveiller les applications de médiation en exécution et de les adapter à la volée. En particulier, il est possible via ces interfaces de connaître et de modifier les topologies des chaînes, les médiateurs (leur configuration), les connecteurs et les ressources du *framework* Cilia (taille de *pool* de *Threads* par exemple). Ces deux points de contrôle, nommés « *Builder* » et « *ApplicationRuntime* », sont illustrés par la figure 6.5 « *L'interface CiliaContext* » ci-dessous. Ils sont regroupés dans une interface Java, nommée `CiliaContext` (voir code source 6.7 « *Interface CiliaContext (extrait)* »).

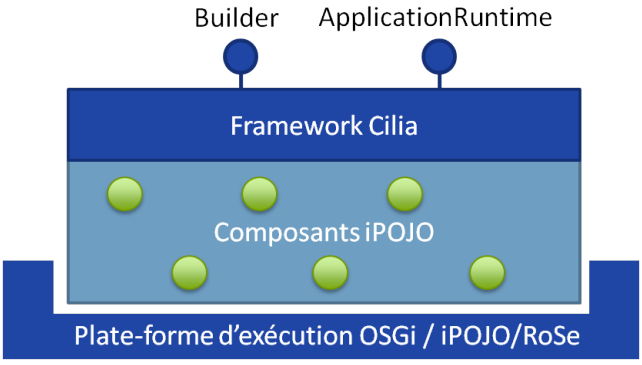


FIGURE 6.5 – L'interface *CiliaContext*.

L'interface `CiliaContext` est en fait une façade, si l'on reprend le terme introduit par les patrons de conception logicielle [Vlissides et al., 1995]. Une façade est communément utilisée pour présenter d'une manière abstraite un ensemble d'objets hétérogènes en relation. Ce patron est particulièrement bien adapté à notre cas pour implanter les points de contrôle qui représentent l'unique point de contact pour la base de connaissances et/ou des composants d'administration.

Les points de contrôle permettent donc l'accès à deux objets fondamentaux :

– l'objet **`Builder`** dont la référence est obtenue par la méthode `getBuilder()` ;
– l'objet **`ApplicationRuntime`** dont la référence est obtenue par la méthode `getApplicationRuntime()`.

L'objet `Builder` est implanté, lui aussi, selon le patron de conception de même nom [Vlissides et al., 1995] et permet la construction des différents objets apparaissant dans une chaîne de médiation. L'objet `ApplicationRuntime`, quant à lui, fournit un ensemble d'informations caractérisant l'exécution des chaînes de médiation (topologie, configuration liaison, etc.). Des API permettent de définir dynamiquement les éléments à surveiller. Le code source 6.7 « *Interface CiliaContext (extrait)* » présente l'interface `CiliaContext`.

```
1  public interface CiliaContext {
   /**
3   * Get the version of the executing Cilia.
    *
5   * @return the version as a String.
    */
7   String getVersion();
    /**
9    * @return the date of start up of the Executing Cilia
     */
11  Date getDateStartUp() ;
    /**
13   * Retrieve a builder instance to modify/create a mediation chain and its
     * components. This method always return a new Builder object.
15   * And this objects have an internal state.
     *
17   * @return the new Builder object.
     */
19   Builder getBuilder();
     /**
21    * Retrieve the ApplicationRuntime instance which allows to inspect
      * the runtime information of mediation chains.
23    *
      * @return the ApplicationSpecification instance.
25    */
      ApplicationRuntime getApplicationRuntime();
27 }
```

Code source 6.7 – Interface *CiliaContext* (extrait).

Il est à noter que l'interface `CiliaContext` est exportée par le *framework* RoSe sous la forme d'une API REST rendant ainsi disponible, par exemple pour un navigateur, les objets `Builder` et `ApplicationRuntime`. Les `Builder` et `ApplicationRuntime` permettent de manipuler les concepts de haut niveau propres à Cilia ; c'est-à-dire les concepts de médiateur, d'adaptateur, de chaîne, etc. Cela permet aux administrateurs, humains et autonomiques, de conserver la terminologie utilisée lors du développement. C'est un point fondamental pour la rapidité et la qualité de l'administration. Cette propriété est assurée grâce à la mise en place d'une architecture réflexive que nous détaillons dans la section suivante.

3.2 Architecture réflexive

3.2.1 Principes

Nous avons vu que le modèle de Cilia est constitué de quatre concepts qui sont le médiateur, l'adaptateur, la liaison interne et la chaîne. Ces éléments sont transformés en composants iPOJO pour être exécutés. Ainsi, deux niveaux apparaissent : celui de la spécification de la médiation de données décrit par le DSL Cilia et celui des composants en exécution. Nous pensons qu'un lien technique et permanent doit être maintenu entre ces deux niveaux pour faciliter le suivi et la manipulation des chaînes à l'exécution. Pour cela, nous avons mis en place une approche architecturale fondée sur la réflexion. Le principe de cette approche est de scinder l'implantation d'une application en deux parties distinctes : une partie dite *meta-level* et une partie dite *base-level*. Le *meta-level* est exprimé en termes métier ; il comprend une description métier des entités en cours d'exécution. Le *base-level*, quant à lui, contient les objets informa-

tiques en exécution. Typiquement, le *meta-level* comprend les notions de médiateurs alors que le *base-level* contient des composants iPOJO.

A tous moments, il est nécessaire de synchroniser les niveaux *meta-* et *base-level*. Ceci demande du code très précis si l'on veut éviter un coût trop important pour la synchronisation. Nous allons détailler cette architecture réflexive, illustrée par la figure 6.6 « *Architecture réflexive de Cilia* », dans les sections à venir.

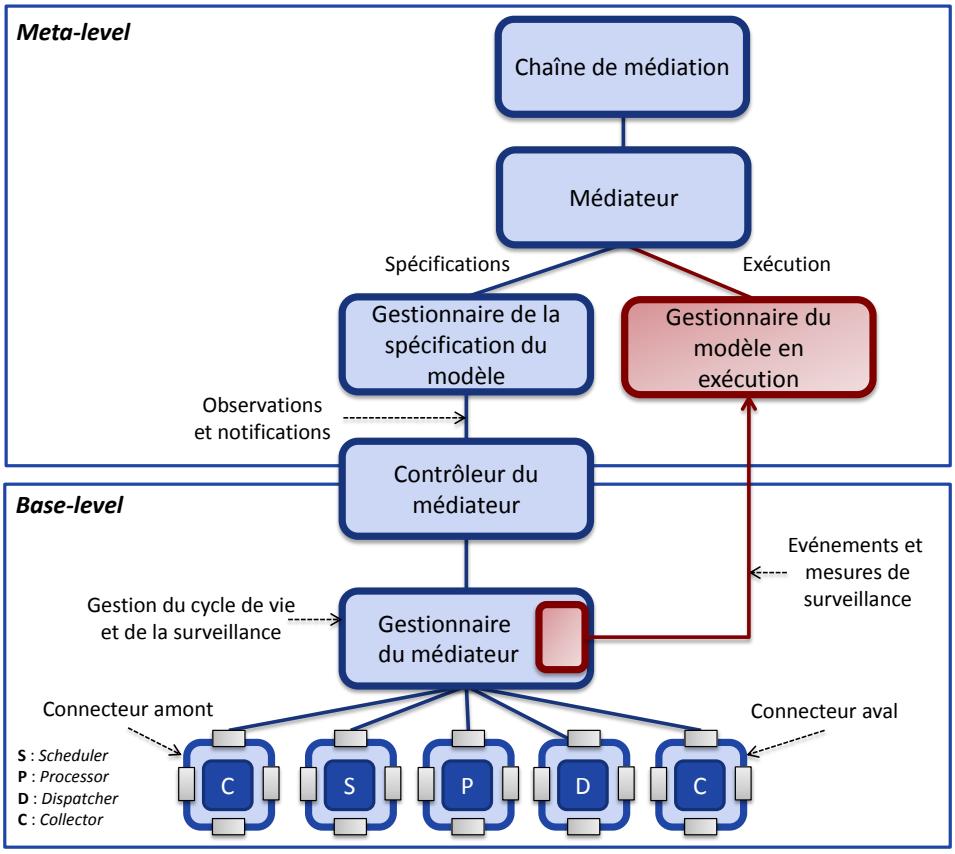


FIGURE 6.6 – Architecture réflexive de Cilia.

3.2.2 Le *meta-level*

Le *meta-level* présente des informations de haut niveau d'abstraction. Cela permet de faciliter la mise en œuvre des opérations d'administration qui ne comprennent que les concepts métier et non pas les détails d'implantation. Le *meta-level* maintient une représentation des chaînes de médiation en exécution en termes de médiateurs, d'adaptateurs, de connecteurs, etc. Le *meta-level* fournit des interfaces de haut niveau permettant la surveillance dynamique et l'adaptation dynamique de ces éléments.

Le code source 6.8 « *Modèle d'une chaîne de médiation (extrait)* » présente l'interface de plus haut niveau qui permet d'accéder et de modifier les éléments médiateur, adaptateur et connexions. Cette interface `Chain` est une extension de l'interface `Component` ; cette dernière est l'interface pour configurer, modifier et rechercher des propriétés de type clé/valeur de manière standard pour tous les éléments appartenant au niveau *meta-level*.

```
public interface Chain extends Component{
        /**
         * Obtain the mediator model which has the given identificator.
         * @param mediatorId MediatorImpl identificator.
         * @return the mediator which has the given identificator.
         */
        Mediator getMediator(String mediatorId);
        /**
         * Get all the mediator models added to the chain model.
         * @return
         */
        Set<Mediator> getMediators();
        /**
         * Obtain the adaptor model which has the given identificator.
         * @param mediatorId MediatorImpl identificator.
         * @return the mediator which has the given identificator.
         */
        Adapter getAdapter(String adapterId);
        /**
         * Get all the mediators models added to the chain model.
         * @return
         */
        Set<Adapter> getAdapters();
        /**
         * Get all the bindings added to the chain model.
         * @return the added bindings.
         */
        Set<Binding> getBindings();
        /**
         * Obtain an array of all the bindings asociated to the given mediators.
         * @param source source mediator which contains the searched bindings.
         * @param target target mediator which contains the searched bindings.
         * @return the array of bindings.
         */
        Binding[] getBindings(MediatorComponent source, MediatorComponent target);
...
}
```

Code source 6.8 – Modèle d'une chaîne de médiation (extrait).

Tous les concepts métier sont maintenus au niveau *meta*. Pour l'exemple, focalisons nous sur le concept de médiateur. L'interface de la classe médiateur s'articule autour de trois ensembles de méthodes :

- **des méthodes pour gérer le modèle et la configuration d'un médiateur**. On trouve par exemple les méthodes `getInPort()` et `getBindings()` ;
- **des méthodes pour gérer une liste d'extension de modèle** telles que, par exemple, les méthodes `extendedModelName()`, `getModel()` et `addModel()` ;
- **des méthodes pour introspecter dynamiquement un médiateur en exécution** comme, par exemple, `getState()` ou `isRunning()`.

Le code source 6.9 « *Modèle d'un médiateur (extrait)* » présente l'interface Java pour les médiateurs au niveau du *meta-level*.

```
public interface MediatorComponent extends Node,Component {
    /**
     * @return chain model hosting the mediator
     */
     Chain getChain();
    /**
     * Obtains the object Port  given the  name
     * @param name port name
     * @return the Class Port with the given port Name.
     */
     Port getInPort(String name);
     Port getOutPort(String name);
    /**
     * Obtains an array of all theoming indings to the mediator.
     * @return Arrays of Class binding In
     */
     Binding[] getInBindings();
     Binding[] getOutBindings();
    /**
     * Obtains all binding ( in or out) given the name
     * @param outPort
     * @return
     */
     Binding[] getBinding(Port name);
    /**
     * Obtains the list of object extended model name attached that curren tobject specification model
     * @return list of extended model
     */
     String[] extendedModelName() ;
    /**
     * Return the object ModelExtension given the model name
```

```
     * @param modelName
33   * @return Model extended or null if modelName doesn't exist
     */
35   ModelExtension getModel(String modelName) ;
    /**
37   * add the object extensionModel to current object specification model
     * @param modelName id of the extensionModel
39   * @param modelExtension object to attache to the current model
     * @return Model extended or null if modelName doesn't exist
41   */
     void addModel(String modelName,ModelExtension modelExtension) ;
43  /**
     * remove  the object extensionModel class to current object specification model
45   * @param modelName id of the extensionModel
     */
47   void removeModel(String modelName) ;
    /**
49   * return the current state of the mediator
     * @return mediator state
51   */
     int getState();
53  /**
     * @return true if the mediator is running
55   */
     boolean isRunning();
57 ...
}
```

Code source 6.9 – Modèle d'un médiateur (extrait).

3.2.3 *Base-level* et synchronisation

Le *base-level* est constitué des composants iPOJO en exécution. Ces composants sont instrumentés de façon à remonter des valeurs caractérisant leur exécution vers le *meta-level*. Ces valeurs permettent de valuer des variables d'état au niveau du *meta-level* ; ces variables caractérisent ainsi l'exécution des concepts métier. Nous revenons sur ces notions dans la section traitant du *monitoring* dynamique. Deux composants sont au cœur du *base-level* et de la synchronisation :

- le **Gestionnaire** de médiateur ;
- le **Contrôleur** de médiateur.

Le gestionnaire de médiateur est le composant technique qui gère le cycle de vie des composants iPOJO, leur configuration, la politique de *monitoring* et, finalement, la publication de mesures vers le *meta-level*. Le code source 6.10 « *Contrôleur de médiateur (extrait)* » est un extrait du code réalisant le contrôleur de médiateur.

Le gestionnaire de spécification du modèle fournit ainsi toutes les informations de topologie et de configuration (exprimées dans le DSCilia). Le gestionnaire du modèle en exécution, quant à lui, renseigne sur l'exécution. Il est alimenté par des valeurs en provenance des composants iPOJO en exécution.

Pour recevoir les données des composants iPOJO, nous avons utilisé le service OSGi `EventAdmin` qui est l'implantation d'un modèle *publish/subscribe* basé sur les *topics*. Pour émettre un événement, le *base-level* émet un *topic* (une chaîne de caractères structurée) avec un objet Java de type `Map`. Techniquement, c'est un *handler* iPOJO que nous appellons `MonitorHandlerStateVar`. Il est en charge de l'émission des événements. Il effectue des mesures pour quantifier l'aspect dynamique du médiateur. Le contenu des événements (l'objet `Map` associé au *topic*) est une mesure de variable d'état.

Pour recevoir ces événements, il est nécessaire d'enregistrer un *handler* d'événements (une classe qui implémente l'interface fournie par OSGi et qui s'enregistre auprès du service `EventAdmin`). Sur réception des événements, la classe enregistrée appelle la méthode `addMesure()` de l'instance correspondante du `MediatorMonitoring`. Le modèle du médiateur en exécution est ainsi mis à jour avec les mesures issues du *base-level*.

Le flux de communication est unidirectionnel et remontant du *base-level* au *meta-level*. La seule connaissance que doit avoir le *meta-level*, est la structure de l'événement reçu (les noms des propriétés et la sémantique des valeurs). Pour garantir l'aspect embarquabilité de Cilia, une seule classe s'enregistre auprès du service `EventAdmin` de OSGi et démultiplexe le contenu de l'événement reçu.

```
public class MediatorControllerImpl implements Observer {
. . .
        public MediatorControllerImpl(BundleContext context,
                        MediatorComponent model, CreatorThread creat, FirerEvents notifier) {
                bcontext = context;
                creator = creat;
                mediatorModel = (MediatorComponentImpl) model;
                mediatorModel.addObserver(this);
                . . .
        }
        /**
         * Method called when some event happened in the mediator model.
         *
         * @param mediator
         *            Mediator model observed.
         * @param arg
         *            Observer method parameters.
         */
        public void update(Observable mediator, Object arg) {
                if (mediator instanceof MediatorComponent) {
                        MediatorComponent md = ((MediatorComponent) mediator);
                        UpdateEvent event = (UpdateEvent) arg;
                        if (event != null) {
                                int action = event.getUpdateAction();
                                switch (action) {
                                case UpdateActions.UPDATE_PROPERTIES: {
                                        logger.debug("[{}] updating Properties: \n\t {}", md.getId(), md.getProperties());
                                        updateInstanceProperties(md.getProperties());
                                }
                                }
                        }
                }
                ...
        }
. . .
}
```

Code source 6.10 – Contrôleur de médiateur (extrait).

3.2.4 Synthèse

La nouvelle version de Cilia repose sur une architecture réflexive et sur des points de contrôle permettant le suivi et la modification des chaînes de médiation à l'exécution. Les points de contrôle fournissent une API fondée sur les concepts métier utilisés par le DSCilia. Ainsi, les langages de développement et d'administration sont les mêmes. Il n'y a pas de rupture entre ces deux phases du cycle de vie logiciel.

Une architecture réflexive est particulièrement difficile à mettre en œuvre. Elle demande une synchronisation précise et maîtrisée des niveaux *meta* et *base*. Dans notre cas, le niveau *meta* s'articule autour de deux composants complémentaires :

- **un gestionnaire de la spécification** qui contient, à tous moments, la spécification courante de la chaîne en exécution et qui répercute les éventuelles modifications vers le niveau *base* ;
- **un gestionnaire du modèle en exécution** qui gère de façon dynamique la remontée d'informations concernant l'exécution.

Ces deux composants sont directement liés aux API de suivi (gestionnaire du modèle en exécution) et de modification (gestionnaire de la spécification).

Dans les sections suivantes, nous expliquons comment les actions de surveillance et de modification sont conçues et implantées.

4 La surveillance dynamique

4.1 Principes

La surveillance dynamique est implantée au niveau *base-level* ; elle est configurée au niveau *meta-level*. Elle est fondée sur la notion de variable d'état qui est une mesure horodatée pouvant résulter d'un calcul au niveau *base*. Un ensemble de variables d'état permet d'évaluer la dynamique d'un système. Les surveillances mises en place sont configurables durant l'exécution, ce qui apporte beaucoup de flexibilité et la capacité de réagir rapidement à des situations imprévues.

Nous avons défini un ensemble de variables d'état génériques propres au *framework* Cilia. En particulier, les temps d'exécution et la taille des messages traversant les *scheduler*, *processor* et *dispatcher* des médiateurs peuvent être surveillés, consolidés et remontés au niveau *meta*. Mais, il est également possible de suivre des variables spécifiques. Les variables suivantes (*c.f.* table 6.1 « *Variables d'état pour les appels système* ») permettent de quantifier le flot de communication entre les médiateurs et au sein des médiateurs.

NOM	CARACTÉRISATION
scheduler.count	Nombre d'appels au *scheduler*
scheduler.data	Message traité par le *scheduler*
process.entry.count	Nombre d'activation de la phase *processor*
process.entry.data	Message traité par le *processor*
process.exit.count	Nombre de messages en sortie du *processor*
process.exit.data	Message en sortie du *processor*
process.err.count	Nombre d'erreur système dans le *processor*
process.err.data	Message d'erreur système dans le *processor*
process.msg.treated	Nombre de messages traités par le *processor*
processing.delay	Temps d'execution de la phase *processor* (la précision de la mesure dépend du *Java Runtime*)
process.dispatch.count	Nombre d'appel au *dispatcher*
process.dispatch.data	Message en sortie du *processor*
message.history	Liste des médiateurs qui ont traité le message
transmission.delay	Temps entre le *dispatcher* aval et le *scheduleur* courant (la précision de la mesure dépend du *Java Runtime*)

TABLE 6.1 – Variables d'état pour les appels système.

Comme nous l'avons dit, toutes les mesures sont horodatées. Cette caractéristique permet au consommateur de reconstruire la chronologie de l'exécution fine du médiateur.

Une deuxième famille de variables d'état permet de décrire l'environnement d'exécution. A chaque arrivée ou départ de *proxies* requis par un adaptateur, une mesure est effectuée (*c.f.* table 6.2 « *Variables d'état de l'environnement d'exécution* »).

NOM	CARACTÉRISATION
service.arrival	Nouvelle dépendance de service
scheduler.count	Nombre d'injections de référence de service
service.departure	Une référence de service a été supprimée de l'injection
service.departure.count	Nombre de références de services supprimées

TABLE 6.2 – Variables d'état de l'environnement d'exécution.

Une troisième famille correspond à l'audit du code métier en exécution (*c.f.* table 6.3 « *Variables d'états pour le code métier* »)). Dans ce cas, la configuration ne peut pas être issue du *meta-level*, elle provient du code métier lui-même.

NOM	CARACTÉRISATION
fire.event	Evénement émis par le code métier du *scheduler* et/ou du *processor* et/ou *dispatcher*
fire.count	Nombre d'événements reçus
field.set	Nom et valeur de la variable écrite
fire.set.count	Nombre d'accès en écriture
field.get	Nom et valeur de la variable lue
fire.get.count	Nombre d'accès en lecture

TABLE 6.3 – Variables d'états pour le code métier.

Comme nous l'avons vu avec l'architecture réflexive, c'est le composant `mediatorManager` qui est en charge de la gestion du cycle de vie de l'ensemble des composants iPOJO qui réalisent un médiateur. Nous avons défini un *handler* particulier, appelé `monitor-handler`, qui intercepte les démarrages et terminaisons des *schedulers*, *processors* et *dispatchers*. Une instance de cet *handler* est créée par instance de `mediatorManager` (la *factory* de cet *handler* est exécutée avant celle du `mediatorManager`).

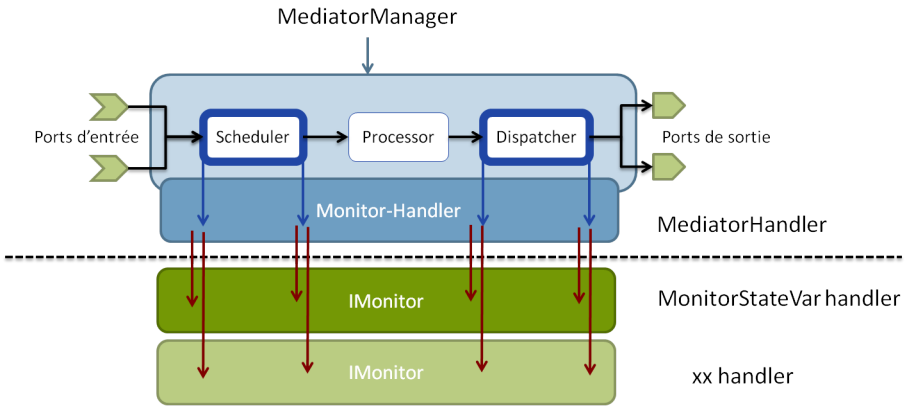


FIGURE 6.7 – Bloc fonctionnel du `mediatorManager` et du *monitoring*.

La figure 6.7 « *Bloc fonctionnel du `mediatorManager` et du monitoring* » illustre le fonctionnement de la surveillance dynamique. Des événements sont générés par les *handlers* techniques du *scheduler* et du *dispatcher*. Le `monitor-handler` intercepte ces événements et délègue le traitement à des *handlers* spécifiques.

4.2 Réalisation

Notre conception repose sur un *handler* nommé `monitor-handler`. Le rôle de ce *handler* est de fournir une interface de remontée de données et d'appeler des objets effectuant les opérations de *monitoring*. Ces objets doivent implanter l'interface `IMonitor` (voir code source 6.12 « *Contrôleur de médiateur (extrait)* », page 156 et le code source 6.11 « *Interface `IMonitor`* », page 156).

```
public interface IMonitor extends IProcessorMonitor, IServiceMonitor, IFieldMonitor {
        void onCollect(Data data);
        void onProcessEntry(List data);
        void onProcessExit(List data);
        void onProcessError(List data, Exception e);
        void onDispatch(List data);
}
```

Code source 6.11 – Interface `IMonitor`.

```
public class MonitorHandler extends PrimitiveHandler implements IProcessorMonitor,
                IServiceMonitor,IFieldMonitor {
        /* Makes a fresch copy of underlying arrays */
        CopyOnWriteArrayList listeners = new CopyOnWriteArrayList();
        . . .
        /* called by scheduler handler when processing phase is starting
           and calls all handleurs listener on onProcessEntry
        */
        public void notifyOnProcessEntry(List<Data> data) {
                Iterator it= listeners.listIterator() ;
                while (it.hasNext())
                ((IMonitor )it.next()).onProcessEntry(data);
        }
        /* called by scheduler handler when processing phase is finishing
           and calls all handleurs listener on onProcessEntry
        */
        public void notifyOnProcessExit(List<Data> data) {
                Iterator it= listeners.listIterator() ;
                while (it.hasNext())
                        ((IMonitor )it.next()).onProcessExit(data);
        }
}
```

Code source 6.12 – Contrôleur de médiateur (extrait).

Comme illustré par la figure 6.8 « *Handlers iPOJO du* `mediatorManager` », un médiateur Cilia comprend des *handlers* au niveau *scheduler* et *dispatcher*. Ces deux *handlers* apportent le code non-fonctionnel nécessaire pour enchaîner les exécutions du *scheduler*, du *processor* et du *dispatcher*. Le `monitor-handler` permet l'ajout de fonctions de surveillance.

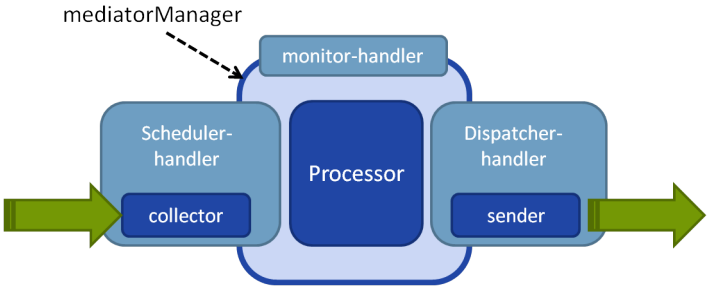


FIGURE 6.8 – Handlers iPOJO du `mediatorManager`.

Précisément, nous avons défini quatre événements permettant de reconstruire la dynamique d'exécution du médiateur :

- **démarrage** du *scheduler* ;
- **démarrage** de l'exécution du *processor* ;
- **fin d'exécution** du *processor* ;
- **démarrage** du *dispatcher*.

Ces quatre événements sont générés par appels de méthode. Précisément, les éléments `scheduler-handler` et `dispatcher-handler` appellent les méthodes du `monitor-handler` prévues à cet effet, soit :

- `notifyOnCollect(...)` : pour le démarrage du *scheduler* ;
- `notifyOnProcessEntry(...)` : pour le démarrage du *processor* ;
- `notifyOnProcessExit (...)` : pour la fin de la phase *processor* ;
- `notifyOnDispatch(...)` : pour le démarrage du *dispatcher*.

Le code permettant la configuration du *monitoring* est présenté dans le code source 6.13 « *MediatorManager configuration (extrait)* », page 157. On voit que le `mediatorManager` doit d'abord obtenir la référence du `monitor-handler`, du `scheduler-handler` et du `dispatcher-handler`. Puis, il doit obtenir l'instance de tous les *handlers* qui implémentent l'interface `IMonitor`. Et, enfin, il doit donner la référence de tous les *handlers* de *monitoring* trouvés à l'instance du `monitor-handler`.

```
public class MediatorManager extends MediatorComponentManager implements ComponentInstance, InstanceStateListener {
. . .
        // Configures technical handler
        public void configure(Element metadata, Dictionary config) throws ConfigurationException {
                this.configuration = config;
                // Add the name
                m_name = (String) config.get("instance.name");
                m_name = m_name + "-Mediator";
                // Get the standard handlers and add these handlers to the list
                SchedulerHandler sch = (SchedulerHandler) ((InstanceManager)pinstance).getHandler(Const.ciliaQualifiedName(
                      "scheduler"));
                MonitorHandler monitor = (MonitorHandler) ((InstanceManager)pinstance).getHandler(Const.ciliaQualifiedName(
                      "monitor-handler"));
                DispatcherHandler dsp = (DispatcherHandler) ((InstanceManager)pinstance).getHandler(Const.
                      ciliaQualifiedName("dispatcher"));
                for (int i = 0; i < m_handlers.length; i++) {
                        m_handlers[i].init(this, metadata, config);
                        //Add subscription IMonitor Handler
                        Handler handler = m_handlers[i].getHandler();
                        // add the monitor, to listen the scheduler/dispatcher events.
                        if (handler instanceof IMonitor) {
                                if (monitor != null) {
                                        monitor.addListener((IMonitor)handler);
                                }
                        }
                        //Add the scheduler/dispatcher references to the monitor-handler
                        if (handler instanceof MediatorHandler) {
                                MediatorHandler mh = (MediatorHandler)handler;
                                if (dsp != null) {
                                        mh.setDispatcher(dsp);
                                }
                                if (sch != null) {
                                        mh.setScheduler(sch);
                                }
                        }
                }
        }
. . .
}
```

Code source 6.13 – MediatorManager configuration (extrait).

4.3 Remontée des données

Pour configurer une variable d'état, il faut :

1. une information booléenne qui indique si une mesure doit être effectuée ou non (état *enable* ou *disable*) ;
2. un filtre LDAP pour contrôler le flux de données publiées.

Ces informations servent à configurer les mécanismes de remontée de données qui sont fondées sur le patron de communication publication/abonnement. Plus précisément, nous utilisons le service `EventAdmin`, fourni par OSGi qui permet d'envoyer des *topics* du niveau *base* vers le niveau *meta*. Un *topic* comprend toujours le nom de la chaîne concernée. Le corps du message (un objet Java de type `Map`) est construit avec le nom du médiateur, le nom de la variable d'état et la valeur de la mesure.

Pour garder une maîtrise sur la fréquence de publication des variables, nous ajoutons une condition sur l'émission de l'événement. OSGi fournit les mécanismes de gestion de filtres avec la syntaxe LDAP, il nous reste à définir les mots clés et leur sémantique. Dans le cas de publication de mesures de variables d'états, les filtres doivent permettre de définir le temps entre deux publications et définir des conditions sur des mesures obtenues. Les filtres basés sur le temps sont les suivants (*c.f* table 6.4 « *Mots clés pour un filtrage du temps* ») :

MOT CLÉ	SIGNIFICATION
time.elapsed	Nombre de millisecondes écoulées depuis la dernière publication
time.current	Temps courant exprimé en ms
time.previous	Temps de la dernière publication exprimé en ms

TABLE 6.4 – Mots clés pour un filtrage du temps.

Les filtres basés sur les valeurs sont les suivants (*c.f* table 6.5 « *Mots clés pour un filtrage de la valeur* ») :

MOT CLÉ	SIGNIFICATION
value.current	Valeur courante de la mesure (Java Double)
value.previous	Valeur de la précédente mesure (Java Double)
value.relative	Variation en valeur absolue : $\lvert value.current - value.previous \rvert$
delta.absolue	Variation relative (pour définir des bandes mortes) : $\lvert value.relative/value.current \rvert$

TABLE 6.5 – Mots clés pour un filtrage de la valeur.

Nous terminons avec un extrait du code du *handler* `MonitorHandlerStateVar` qui effectue des mesures en fonction des méthodes appelées par le `monitor-handler`. Ce code est instancié par défaut pour chaque médiateur et adaptateur.

```
public class MonitorHandlerStateVar extends AbstractMonitor {
. . .
        /* TAG for storing message history */
        private static final String PROPERTY_MSG_HISTORY = "cilia.message.history";
        private static final String PROPERTY_BINDING_TIME = "cilia.message.time.bind";
        private String topic;
. . .
        /* Handler configuration */
        public void configure(Element metadata, Dictionary configuration) throws ConfigurationException {
                chainId = (String) configuration.get(Const.PROPERTY_CHAIN_ID);
                componentId = (String) configuration.get(Const.PROPERTY_COMPONENT_ID);
                uuid = (String) configuration.get(Const.PROPERTY_UUID);
                topic = "cilia/runtime/statevar/" + chainId;
                . . .
        }
        /* Retreives OSGi public/subscribe based-topic messagerie pattern */
        private ServiceReference retreiveEventAdmin() {
                ServiceReference[] refs = null;
                ServiceReference refEventAdmin;
                try {
                        refs = m_bundleContext.getServiceReferences(EventAdmin.class.getName(), null);
                } catch (InvalidSyntaxException e) {
                        logger.error("Event Admin  service lookup unrecoverable error");
                        throw new RuntimeException("Event Adminservice lookup unrecoverable error");
                }
                if (refs != null)
                        refEventAdmin = refs[0];
                else
                        refEventAdmin = null;
                return refEventAdmin;
        }
        /* Publish a new value if the control flow is matching the LDAP filter */
        private void publish(String stateVarId, Object data, long ticksCount) {
                long last_ticksCount;
                Condition cond;
                boolean fire;
                StateVarItem item = (StateVarItem) m_statevar.get(stateVarId);
                if (item != null) {
                        fire = true;
                } else {
                        cond = item.condition;
                        if (cond != null) {
                                last_ticksCount = item.lastpublish.longValue();
                                /* Check LDAP condition */
                                fire = cond.match(ticksCount,(Watch.fromTicksToMs(ticksCount) - Watch.fromTicksToMs(
                                        last_ticksCount)));
                        } else {
                                fire = true;
                                item.lastpublish = new Long(ticksCount);
                        }

```

```
                }
52              /* Fire data to the base-level */
                if (fire) firer(stateVarId, data, ticksCount);
54      }
        /* Build the message ( sate var name  + value + timestamp )and publish with the topic */
56      private void firer(String stateVarId, Object value, long ticksCount) {
                EventAdmin m_eventAdmin;
58              ServiceReference refEventAdmin = retreiveEventAdmin();
                if (refEventAdmin == null) {
60                      logger.error("Unable to retrieve Event Admin");
                } else {
62                      /* Gather data to be published */
                        Map data = new HashMap(5);
64                      data.put(ConstRuntime.EVENT_TYPE, ConstRuntime.TYPE_DATA);
                        data.put(ConstRuntime.UUID, uuid);
66                      data.put(ConstRuntime.VARIABLE_ID, stateVarId);
                        data.put(ConstRuntime.VALUE, value);
68                      data.put(ConstRuntime.TIMESTAMP, new Long(ticksCount));
                        StateVarItem item = (StateVarItem) m_statevar.get(stateVarId);
70                      if (item != null) item.lastpublish = new Long(ticksCount);
                        m_eventAdmin = (EventAdmin) m_bundleContext.getService(refEventAdmin);
72                      m_eventAdmin.postEvent(new Event(topic, data));
                        m_bundleContext.ungetService(refEventAdmin);
74              }
        }
76      /* Called by the monitor-handler when the scheduler start running */
        public void onCollect(Data data) {
78              if (listStateVarEnabled.isEmpty())
                        return;
80              . . .
                if (isEnabled("scheduler.count")) {
82                      m_counters[0]++;
            /* Get a thread in a thread pool and publish asynchrouly the result */
84                      m_systemQueue.execute(new AsynchronousExec("scheduler.count", new Long(m_counters[0])));
                }
86              if (isEnabled("scheduler.data"))
                        m_systemQueue.execute(new AsynchronousExec("scheduler.data", new Data(data)));
88      }
        /**
90       * Asynchronous execution
         */
92      private class AsynchronousExec implements Runnable {
                private final String stateVar;
94              private final Object data;
                private final long tickCount = Watch.getCurrentTicks();
96
                AsynchronousExec(String stateVar, Object data) {
98                      this.stateVar = stateVar;
                        this.data = data;
100             }
                public void run() {
102                     publish(stateVar, data, tickCount);
                }
104     }
 . . .
106 }
```

Code source 6.14 – `MonitorHandlerStateVar` (extrait).

4.4 Audit et événements

Pour augmenter les informations utilisables par un gestionnaire autonomique, nous avons conçu un mécanisme permettant de remonter des variables d'état spécifique à une application (et non pas générique au *framework* Cilia). Le *handler* `MonitorHandlerStateVar` (le *handler* de surveillance instancié par défaut pour chaque médiateur) est capable de capter des événements en provenance du code métier et de les publier sous forme de mesures pour le niveau *meta-level*. Ce sont les variables d'état `field.get` et `field.set`.

Le développeur doit poser des annotations `@Audit` sur des variables de son code métier. Les accès en lecture et/ou écriture sont alors sauvegardés sous forme de mesures et génèrent des événements si la condition de publication est satisfaite. Le code source 6.15 « *Audit de variables du code métier (extrait)* » nous montre la simplicité de configuration permettant l'audit des variables du code métier du *processor*.

```
import fr.liglab.adele.cilia.handler.Audit ;
2 import fr.liglab.adele.cilia.Data;
public class MyProcessor  {
4       /* Store the name  processor.m_RW and the value on read and write access */
        @Audit (access="rw" namespace= "processor")
```

```
6         private int m_RW =0;
          /* Store the name  processor.m_R and the value on read and write access */
8         @Audit (access="r" namespace ="processor")
          private int m_R =0 ;
10        /* Store the name  processor.m_W and the value on read and write access */
          @Audit (access="w" namespace ="processor")
12        private  int m_W =0 ;
          public Data process(Data data) {
14        . . .
          }
16 }
```

Code source 6.15 – Audit de variables du code métier (extrait).

Il existe un autre mécanisme qui permet de générer des événements à partir du code métier. Le développeur peut générer des événements sous forme de variables d'état sans pour autant connaître les services du *Cilia runtime*. La référence de l'objet `event` est automatiquement injectée par le *framework* Cilia. Le `MonitorHandleurStateVar` récupère l'événement et le publie sous forme de mesure, la variable d'état associée est `fire.event` (voir code source 6.16 « *Génération d'événements à partir du code métier (extrait)* », page 160).

```
  import fr.liglab.adele.cilia.Data ;
2 import fr.liglab.adele.cilia.framework.monitor.ProcessorNotifier;
  public class MyProcessor  {
4         ProcessorNotifier event; /* Reference injected by the Cilia Framework */
          public Data process(Data data) {
6                 . . .
                  if (buffer.length > 10) {
8                         event.fireEvent(new HashMap("The buffer is in overrun",new Integer(buffer.length));
                  }
10                . . .
          }
12 }
```

Code source 6.16 – Génération d'événements à partir du code métier (extrait).

5 Adaptation dynamique

Tout comme pour la surveillance, l'adaptation dynamique est implantée au niveau *base-level* et est décidée au niveau *meta-level*. Selon notre approche, il est possible de modifier dynamiquement les quatre éléments constituant le modèle d'exécution de Cilia, à savoir les chaînes, les médiateurs, les adaptateurs et les liaisons. Bien sûr, une adaptation dynamique n'est acceptable que si le flux des messages entrants et sortants n'est pas rompu et si les messages en transit sont correctement traités. Dans cette section, nous allons examiner trois types d'opérations :

- des opérations pour **créer** et détruire des chaînes de médiation ;
- des opérations pour **modifier** la topologie des chaînes de médiation ;
- des opérations pour **garantir la conservation des états** et la non perte des messages.

5.1 Création et destruction de composants

Le service `CiliaContext` fournit un objet `Builder` permettant d'effectuer des modifications qui opèrent uniquement au niveau *meta-level* (les seules références connues par ses API sont sur des objets appartenant au niveau *meta-level*). Le contrôleur de médiateur observe et réifie les modifications au niveau *base-level*.

Le code source 6.17 « *Création de la chaîne simple (extrait)* » est un extrait de code exécutant des appels à l'objet `builder` pour :

1. **construire** entièrement une chaîne de médiation à la volée. La figure 6.9 « *Topologie de la chaîne simple* » est la visualisation de la chaîne ainsi construite.

2. **modifier** un paramètre (« `period` ») d'une instance particulière d'un médiateur (« a »).

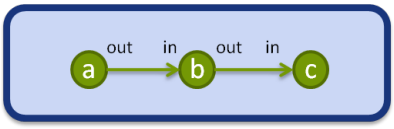


FIGURE 6.9 – Topologie de la chaîne simple.

Les fonctions `create()`, `get()` et `delete()` du `builder` permettent respectivement de créer une chaîne de médiation, d'accéder à une chaîne et, finalement, de détruire une chaîne de médiation et l'ensemble de ces constituants.

```
import org.apache.felix.ipojo.annotations.Requires;
import fr.liglab.adele.cilia.CiliaContext;
. . .
@Requires
CiliaContext ccontext; /*IPOJO injecte la reference du service CiliaContext */
public void creationChaineSimple() {
        Builder builder = ccontext.getBuilder();
        // Cree une chaine "ChaineSimple" vide
        Architecture chaine = builder.create("ChaineSimple");
        // Instanciation des mediateurs
        chaine.create().mediator().type("Mediator_Type_A").id("a");
        chaine.create().mediator().type("Mediator_Type_B").id("b");
        chaine.create().mediator().type("Mediator_Type_C").id("c");
        // Definition de la topologie
        chaine.bind().from("a:out").to("b:in");
        chaine.bind().from("b:out").to("c:in");
        // Creation de la chaine
        builder.done();
        // Modification de la valeur du parametre period du mediateur "a"
        Builder newBuilder = ccontext.getBuilder();
        // Recupere le modele de la "ChaineSimple"
        Architecture arch = builder.get("ChaineSimple");
        arch.configure().mediator().id("a").key("period").value("100");
        // Re Configuration du mediateur avec la nouvelle valeur de la propriete "period"
        builder.done();
}
```

Code source 6.17 – Création de la chaîne simple (extrait).

5.2 Modification topologique

La deuxième famille de méthodes autorise la modification topologique de chaîne de médiation en exécution. Le code source 6.18 « *Modification de topologie de la chaîne simple (extrait)* » est un exemple d'appels de méthodes de l'objet `builder` permettant de modifier la structure de la chaîne « ChaineSimple » (voir figure 6.9 « *Topologie de la chaîne simple* ») en une topologie illustrée par la figure 6.10 « *Topologie de la chaîne simple modifiée* » page 161.

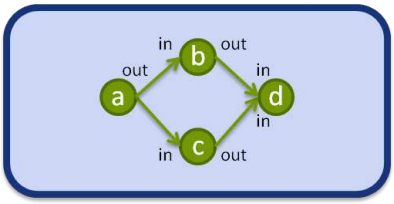


FIGURE 6.10 – Topologie de la chaîne simple modifiée.

```
import org.apache.felix.ipojo.annotations.Requires;
import fr.liglab.adele.cilia.CiliaContext;
. . .
@Requires
CiliaContext ccontext; /* IPOJO injecte la reference du service CiliaContext */
public void changeTopology() {
        Builder builder = ccontext.getBuilder();
        // Recupere la chaine "ChaineSimple"
        Architecture chaine = builder.get("ChaineSimple");
```

```
10          // Instancie le mediator D , avec l'ID =d
            chaine.create().mediator().type("Mediator_Type_D").id("d");
12          // Reconstruit la nouvelle topologie en memoire
            chaine.unbind().from("b:out").to("c:in");
14          chaine.bind().from("b:out").to("d:in");
            chaine.bind().from("c:out").to("d:in");
16          chaine.bind().from("a:out").to("c:in");
            // Reconfiguration de la chaine
18          builder.done();
    }
```

Code source 6.18 – Modification de topologie de la chaîne simple (extrait).

5.3 Gestion de la conservation des états

Un aspect fondamental lié à la modification des chaînes de médiation est la conservation des états. Notre objectif est de pouvoir remplacer un adaptateur et/ou un médiateur à la volée en assurant le traitement de l'ensemble des messages en transit dans la chaîne de médiation. Pour atteindre cet objectif, nous avons besoin des deux fonctionnalités suivantes :

- **cloner** un adaptateur/médiateur. Le but est de pouvoir instancier un adaptateur/médiateur avec l'ensemble de ses paramètres courants ;
- **remplacer** une instance d'un adaptateur/médiateur par un autre sans la perte de message.

Pour traiter le premier point, nous avons ajouté une méthode `clone()` à la classe `Builder`. Cette méthode ne pose pas de problème d'implantation particulier. Elle enchaîne simplement la création d'une instance de médiateur ou d'adaptateur et la recopie de l'ensemble des paramètres du médiateur/adaptateur initial. Le second point est plus délicat à traiter. Nous avons ajouté deux méthodes `replaceMediator()` et `replaceAdapter()` au `Builder` pour assurer le remplacement à la volée d'un médiateur/adaptateur par un autre. Précisément, nous utilisons un mécanisme de sauvegarde des messages entrants d'un médiateur/adaptateur vers un service de persistance externe, appelé `DataPersistency`. Le rôle de ce service est de maintenir un espace mémoire privé réservé à la persistance des messages par adaptateur/médiateur. Ce service fournit l'interface Java suivante :

```
1   public interface AdminData {
            /**
3            * retreive data buffer map related to the mediator.
             * @param mediatorId mediator reference id
5            * @param isRegular true, selects regular message flow, false selects messages bufferized
             * @return Map containing buffer Data
7            */
            public Map getData(String mediatorId, boolean isRegular);
9           /**
             * clear all messages (regular and bufferized messages)
11           * @param mediatorId, mediator unique id
             */
13          public void clearData(String mediatorId);
            /**
15           * copy messages ( regular and stored)
             * @param mediatorfrom_Id
17           * @param mediatorTo_Id
             */
19          public void copyData(String mediatorFrom_Id, String mediatorTo_Id) ;
    }
```

Code source 6.19 – Interface fournie par le service `DataPersistency`.

Cette interface est faite pour :
- **sauvegarder** des messages ;
- **récupérer** l'ensemble des messages sauvegardés ;
- **copier** les références d'un espace privé à un autre espace privé ;
- **supprimer** les références des messages.

Le `scheduleur-handler` sauvegarde systématiquement, dans le service `DataPersistency`, la référence du message entrant. La sauvegarde est effectuée dans l'espace mémoire correspondant à l'instance de son adaptateur/médiateur. Le `scheduler-handleur` a deux propriétés configurables à partir de la spécification du médiateur :
- **la propriété *lock*** : le `scheduler-handler` informe le collecteur d'un message à traiter ;
- **la propriété *unlock*** : le `scheduler-handler` n'est pas informé de l'arrivée d'un nouveau message.

Le service `DataPersistency` crée et gère dynamiquement un objet Java de type `Map`. Si la propriété *unlock* est active alors le `scheduler-handler` appelle le *collector* qui obtient l'ensemble des références des messages sauvegardés. Si la propriété `lock` est active, alors le `scheduler-handler` n'appelle pas le *collector*. Sur transition de `lock` à *unlock* , le `scheduler-handler` appelle le *collector* (qui reprend tous les messages le concernant).

Pour assurer la non perte des messages, il faut enchaîner les opérations de *lock* et *unlock* au niveau du `scheduler-handler`, ce qui revient à copier [2] les messages dans l'espace privé au adaptateur/médiateur. La figure 6.11 « *Gestion des messages dans le `scheduler` Cilia* » illustre notre approche.

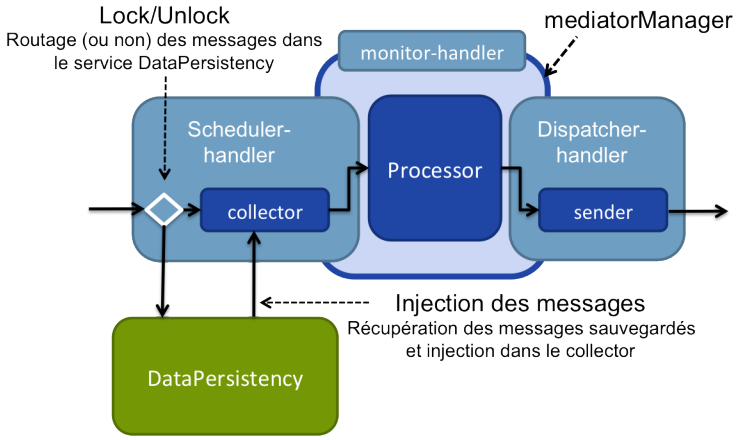


FIGURE 6.11 – Gestion des messages dans le `scheduler` Cilia.

Le code source 6.20 « *Interface fournie par le service `DataPersistency`* » illustre l'enchaînement des opérations.

```
public interface AdminData {
        /**
         * retreive data buffer map related to the mediator.
         * @param mediatorId mediator reference id
         * @param isRegular true, selects regular message flow, false selects messages bufferized
         * @return Map containing buffer Data
         */
        public Map getData(String mediatorId, boolean isRegular);
        /**
         * clear all messages (regular and bufferized messages)
         * @param mediatorId, mediator unique id
         */
        public void clearData(String mediatorId);
        /**
```

2. Techniquement ce n'est pas une copie mais une modification de référence dans la partie value de la `Map` générale. La performance n'est pas liée au nombre de messages sauvegardés ; elle est constante.

```
         * copy messages ( regular and stored)
         * @param mediatorfrom_Id
         * @param mediatorTo_Id
         */
        public void copyData(String mediatorFrom_Id, String mediatorTo_Id) ;
}
```

Code source 6.20 – Interface fournie par le service `DataPersistency`.

L'enchaînement des actions est assuré par les méthodes `replaceMediator()` (voir code source 6.21 « *`Builder` - Remplacement d'un médiateur (extrait)* ») et `replaceAdapter()` de l'objet `Builder`. Ces deux méthodes enchaînent quatre étapes :

- ***lock*** des médiateurs ;
- **remplacement** des liaisons amont et aval ;
- **copies** des messages de l'espace mémoire de l'ancien médiateur dans celui du nouveau ;
- ***unlock*** des médiateurs (traitement dans l'ordre des messages sauvegardés puis des nouveaux arrivants).

```
private void replaceMediator(ReplacerImpl rep) throws BuilderPerformerException{
    MediatorComponent from = getMediatorComponent(rep.id);
    MediatorComponent to = getMediatorComponent(rep.to);
    //we lock mediator execution.
   ((MediatorComponentImpl)from).lockRuntime();
   ((MediatorComponentImpl)to).lockRuntime();
   //replace out bindings.
   Binding outbindings[] = from.getOutBindings();
  for (int i = 0; outbindings != null &&  i < outbindings.length ; i ++) {
    replaceOutBinding(to, rep, outbindings[i]);
  }
  //replace in bindings.
  Binding inbindings[] = from.getInBindings();
  for (int i = 0; inbindings != null &&  i < inbindings.length ; i ++) {
    replaceInBinding(to, rep, inbindings[i]);
  }
  // Now data is injected
  try {
   ccontext.getApplicationRuntime().copyData(from, to);
  } catch (CiliaIllegalParameterException e) {
     throw new BuilderPerformerException(e.getMessage());
  }
  //we unlock mediation execution
  finally {
   ((MediatorComponentImpl)from).unLockRuntime();
   ((MediatorComponentImpl)to).unLockRuntime();
  }
}
```

Code source 6.21 – `Builder` - Remplacement d'un médiateur (extrait).

6 La base de connaissances

Notre proposition consiste à fournir aux développeurs une base de connaissances construite et mise à jour par le *framework*. Cette base de connaissances contient la topologie courante des chaînes de médiation en exécution, les mesures des variables d'états avec pour chacune une profondeur d'archivage donnée, des événements système et des méthodes de type passerelles pour récupérer les gestionnaires en exécution et de spécification (l'ensemble des objets du `meta-level`). La base de connaissances contient également une historisation de ces différents éléments.

Rappelons que cette base de connaissances est causale. Une modification de la base de connaissances est répercutée sur les chaînes de médiation en exécution. De même, une modification au niveau chaîne de médiation est répercutée dans la base de connaissances.

6.1 Découverte des chaînes en éxécution

Nous modélisons une chaîne de médiation comme un graphe orienté avec des points d'entrée et des points de sortie (*c.f.* code source 6.22 « *Interface `Topology` (extrait)* »). Cette modélisation est le cœur de la structuration de la base de connaissances que nous présentons ci-après.

```
public interface Topology {
        /**
         * Retreives all nodes matching the filter
         * @param ldapFilter, keywords = chain, node, uuid
         * @return array of node matching the filter, array size 0 if no node matching the filter
         * @throws CiliaInvalidSyntaxException, ldap syntax error
         */
        Node[] findNodeByFilter(String ldapFilter) throws CiliaIllegalParameterException,
                        CiliaInvalidSyntaxException;
        /**
         * @param ldapFilter,ldap filters keywords (chain,node,uuid) uuid is relevant only for dynamic components
                 ( SetUp,RawData,Threshold)
         * @return list of adapter In
         * @throws CiliaInvalidSyntaxException, LDAP syntax error
         * @throws CiliaIllegalParameterException, null parameter
         */
        Node[] endpointIn(String ldapFilter) throws CiliaIllegalParameterException,
                        CiliaInvalidSyntaxException;
        /**
         * @param ldapFilter,ldap filters keywords (chain,node,uuid) uuid is relevant only for dynamic components
                 ( SetUp,RawData,Threshold)
         * @return list of adapter Out
         * @throws CiliaInvalidSyntaxException, LDAP syntax error
         * @throws CiliaIllegalParameterException, null parameter
         */
        Node[] endpointOut(String ldapFilter) throws CiliaIllegalParameterException,
                        CiliaInvalidSyntaxException;
        /**
         * @param node node reference
         * @return array of successors , size = 0 if no successor
         * @throws CiliaIllegalStateException the node doesn't exist
         */
        Node[] connectedTo(Node node) throws CiliaIllegalParameterException,CiliaIllegalStateException;
        /**
         * @param ldapFilterldap filters keywords (chain,node,uuid) uuid is relevant only for dynamic components
                 (SetUp,RawData,Threshold)
         * @return array of successors, size=0 if no node matching the filter
         * @throws CiliaIllegalParameterException , null parameter
         * @throws CiliaInvalidSyntaxException, LDAP syntax error
         */
        Node[] connectedTo(String ldapFilter) throws CiliaIllegalParameterException,
                        CiliaInvalidSyntaxException;
...
}
```

Code source 6.22 – Interface `Topology` (extrait).

L'interface `Topology` propose trois familles de méthodes permettant :

1. **la recherche d'ensembles de nœuds vérifiant une condition LDAP**. Les mots clés du filtre sont l'identificateur du médiateur/adaptateur, son nom d'instance, le nom de sa chaîne d'appartenance ;
2. **l'obtention des successeurs d'un nœud** (un adaptateur ou un médiateur) pour le parcours d'un graphe orienté ;
3. **l'obtention des points d'entrée et de sortie** (adaptateurs).

Un nœud (`Node`) est un objet Java qui modélise à gros grain un nœud de la chaîne de médiation. Il peut représenter un adaptateur ou un médiateur. Cet objet fournit des informations qui permettent de l'identifier de manière unique (nom de la chaîne, nom de l'instance du médiateur ou de l'adaptateur, date et heure de son instanciation) et est utilisé en paramètre de toutes les méthodes proposées par la base de connaissances.

Par défaut, l'objet rendu par toutes ces méthodes est un *proxy* (un *proxy* dynamique Java) sur l'objet médiateur ou adaptateur. En effet, si un médiateur est détruit, le *proxy* générera une exception Java prédéfinie plutôt qu'un `NullPointeurException`. L'utilisateur de l'objet `Node` sera ainsi informé par une exception Java signée, si l'objet est détruit.

6.2 Evénements

Les événements configurables et générés par le *framework* Cilia permettent de caractériser le comportement dynamique de l'application de médiation. Nous avons trois familles d'**événements** :

- événements de niveau **chaîne de médiation** ;
- événements de niveau **nœuds** de médiation ;
- événements de niveau **variables d'états**.

*Ce mécanisme d'événements permet à un observateur de l'exécution de l'application d'être informé sur un changement de type **structurel, fonctionnel** ou **comportemental**.*

Toutes les interfaces Java de configuration des événements sont basées sur le même principe. Un filtre LDAP qui permet de sélectionner une ou plusieurs chaînes, un ou plusieurs nœuds (médiateur/adaptateur) et une méthode *callback* qui sera appelée si le filtre est satisfait.

Les événements de niveau chaîne générés par le *framework* sont :

- **nouvelle chaîne déployée** : `onAdded(String chain)` ;
- **chaîne supprimée** : `onRemoved(String chain)` ;
- **nouvel état de la chaîne** (démarrée ou arrêtée) : `onStateChange(String chain, boolean state)`.

Les événements de niveau noeud générés par le *framework* sont :

- **nouvelle instance d'un nœud** : `onArrival(Node node)` ;
- **nœud supprimé** : `onDepartureNode(Node)` ;
- **nœud modifié (paramètres)** :`onModified(Node node)` ;
- **nouvelle liaison entre deux nœuds** :`onBind(Node From, Node to)` ;
- **suppression de la liaison entre deux nœuds** : `onUnbind(Node from, Node to)` ;
- **changement d'état d'un nœud** : `onStateChange(Node node,boolean isValid)`.

Les événements de niveau variable d'état sont de deux types : un premier pour informer l'arrivée dans la base de connaissances d'une nouvelle mesure et le second pour informer de la mise à disposition d'une mesure. Les événements liés aux mesures sont les suivants :

- **nouvelle mesure sauvegardée dans la base de connaissances** : `onUpdate(Node node, String variable, Measure m)` ;
- **changement d'état d'une variable d'état** (*enable-disable*) : `onStateChange(Node node, String variable , boolean enable)` ;
- **seuil mesure** : `onThreshold(Node node, String variable, Measure m, int thresholdType)`. Les quatre niveaux de seuils pour une variable sont configurables par une API Java de la base de connaissances.

6.3 Configuration de la base de connaissances

La configuration de la base de connaissances se fait par l'intermédiaire des objets suivants :

- l'objet `SetUp` agit au niveau `Node`, il permet de configurer :
  - la profondeur de l'historisation des variables sauvegardées,
  - l'état des variables d'état (*enable* ou *disable*).
- l'object `RawData` permet de récupérer les mesures historisées pour un objet `Node` ;
- l'object `Threshold` permet de configurer les seuils de bon fonctionnement d'une mesure pour une variable d'état d'un objet `Node`.

Chaque objet `SetUp`, `RawData` et `Threshold` peut être retrouvé par l'objet `Node` ou plus généralement par un filtre LDAP, si le nœud n'est pas connu. L'extrait de code suivant montre l'interface de configuration de la base de connaissances.

```
public interface ApplicationRuntime extends Topology, EventsConfiguration, ModelComponents {
        /**
         * @throws CiliaInvalidSyntaxException
         * @param ldapFilter, ldap filters, keywords {uuid,chain,node}
         * @return array of Setup , size is 0 if no node found matching the filter
         * @throws CiliaIllegalStateException
         */
        SetUp[] nodeSetup(String ldapFilter) throws CiliaIllegalParameterException,
                        CiliaInvalidSyntaxException ;
        /**
         * fast access using the node reference
         * @param node
         * @return object Setup
         * @throws IllegalStateException, if the node is no more existing
         * @throws CiliaIllegalParameterException, if the node is null
         * @throws CiliaIllegalStateException
         */
        SetUp nodeSetup(Node node) throws CiliaIllegalParameterException,
                        CiliaIllegalStateException;
        /**
         * @param ldapFilter
         *            ldap filters, keywords {uuid,chain,node}
         * @return array of object raw data matching the ldap filter
         * @throws InvalidSyntaxException, if the ldap syntax is not valid
         * @throws CiliaIllegalParameterException
         * @throws CiliaInvalidSyntaxException
         */
        RawData[] nodeRawData(String ldapFilter) throws CiliaIllegalParameterException,
                        CiliaInvalidSyntaxException;
        /**
         * fast access using the node reference
         * @param node
         * @return object providing raw data
         * @throws InvalidSyntaxException
         *             if ldap syntax is not valid
         * @throws CiliaIllegalParameterException, if the node is null
         * @throws CiliaIllegalStateException
         */
        RawData nodeRawData(Node node) throws CiliaIllegalParameterException,
                        CiliaIllegalStateException;
        /**
         * @param ldapFilter
         *            ldap filters, keywords {uuid,chain,node}
         * @return array of object type Diagnostic matching the ldap filter
         * @throws InvalidSyntaxException, if ldap syntax is not valid
         * @throws CiliaIllegalParameterException
         * @throws CiliaInvalidSyntaxException
         */
        Thresholds[] nodeMonitoring(String ldapFilter)
                        throws CiliaIllegalParameterException, CiliaInvalidSyntaxException;
        /**
         * fast access using the node reference
         * @param node
         * @return object Diagnostics
         * @throws CiliaIllegalParameterException, if the node is null
         * @throws CiliaIllegalStateException
         */
        Thresholds nodeMonitoring(Node node) throws CiliaIllegalParameterException,
                        CiliaIllegalStateException;
. . .
}
```

Code source 6.23 – Base de connaissances (extrait).

6.4 Résumé

La figure 6.12 illustre la base de connaissances. Les événements émis sont configurables et permettent de reconstruire la dynamique des chaînes de médiation en exécution. Les mesures sont sauvegardées dans des *buffers* circulaires et bornés. Des variables d'état permettent de reconstruire l'aspect comportement interne du médiateur (ou bien adaptateur). Les intervalles de bon fonctionnement des mesures permettent d'être informé sur une instabilité du système.

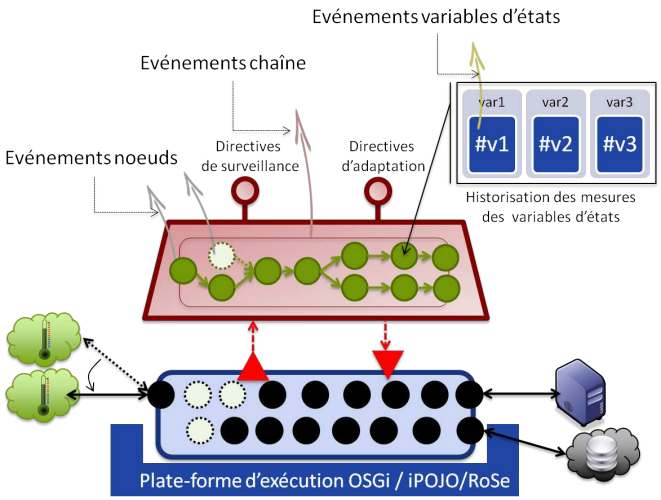


FIGURE 6.12 – Vue générale de la base de connaissances.

La base de connaissance permet à un gestionnaire autonomique de détecter deux points importants dans l'adaptation dynamique d'un système :

- **l'instant du démarrage** des modifications ;
- **l'instant où le système est dans un état stable** ou dans un état attendu.

7 Conclusion

Dans ce chapitre, nous avons présenté l'implantation de notre proposition. Précisément, nous avons montré comment l'architecture à multiple boucles de contrôle que nous défendons a été conçue et entièrement implantée. Architecturalement parlant, deux points nous semblent très structurants :

- la mise en place d'une architecture réflective, avec un niveau *base* et un niveau *meta*. Une telle architecture permet de maintenir un lien causal entre les objets en exécution et les concepts spécifiés pour définir une chaîne de médiation. Cette architecture offre un excellent découplage mais est difficile à mettre en place, notamment au niveau de la synchronisation.
- la mise en place d'une base de connaissances permettant d'abstraire le niveau de raisonnement des gestionnaires autonomiques. Cette base de connaissances offre également un historique des évolutions des chaînes de médiation qui peut être utile aux raisonnements autonomiques.

Ces deux éléments concourent à la réalisation des deux mécanismes fondamentaux à Cilia autonomique, à savoir la surveillance dynamique et l'adaptation à chaud sans perte d'information. Dans la suite, nous montrons comment cette implantation a été validée.

7

Validation et évaluation

Le travail de cette thèse a été validé à l'aide de cas d'utilisation issus du projet Medical [1]. Dans la première partie de ce chapitre, nous allons présenter en détail ce projet collaboratif. Nous commençons par expliciter les objectifs du projet, puis continuons par la présentation du cas d'utilisation retenu pour notre validation. Plus loin dans cette première partie, nous approfondissons et détaillons le fonctionnement de la chaîne de médiation telle qu'elle a été mise en place initialement dans ce projet.

Dans la seconde partie de ce chapitre, nous détaillons la nouvelle solution que nous proposons pour ce cas d'usage. Précisément, nous montrons l'architecture mise en place avec notamment un gestionnaire autonomique avec deux objectifs de haut niveau et un élément administré.

Avant de conclure cette section, nous détaillons l'expérimentation telle qu'elle a été mise en œuvre avec les commentaires explicatifs des résultats obtenus.

1. http://medical.imag.fr

1 Projet MEDICAL

Le projet MEDICAL [2] est un projet collaboratif soutenu par le pôle de compétitivité Minalogic [3], dans le cadre du FUI (Fonds Unique Interministériel). L'objectif principal de ce projet est la production d'un *middleware* permettant l'intégration d'équipements hétérogènes et dynamiques au sein de passerelles domestiques. L'objectif est de fournir des services numériques à domicile. Ce projet regroupe Orange Labs [4] , la société ScalAgent D.T. [5], l'université Joseph Fourier [6] de Grenoble et Telecom ParisTech [7].

Les domaines de la santé et, notamment, ceux liés au maintien à domicile des personnes représentent un enjeu sociétal majeur. L'utilisation de solutions numériques ouvre des perspectives majeures en termes de fonctionnalités apportées et de maîtrise des coûts. Néanmoins, la mise en place de telles solutions repose sur l'usage d'infrastructures de calculs sophistiquées et sur l'intégration dynamique des nombreux équipements numériques hétérogènes que l'on trouve au sein des habitations.

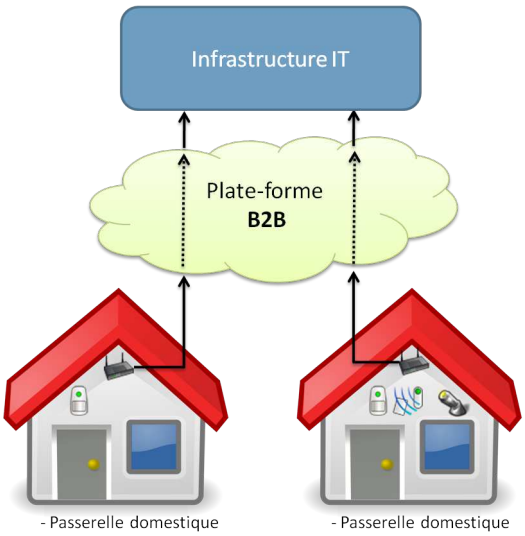


FIGURE 7.1 – Application *Actimétrie*.

Plus précisément, la mise en place de services numériques au sein des habitats soulève de nombreux défis aux différents niveaux suivants :

- **l'interconnexion** de plates-formes hétérogènes ;
- **l'intégration** dans les plates-formes de dispositifs hétérogènes et dynamiques ;
- **la limitation maximale des arrêts** des applications ;
- **le faible coût** de mises en œuvre et d'exploitation permis ;
- **l'absence d'administrateurs** au sein des habitations.

2. Acronyme de *Middleware* Embarqué D'Intégration de Capteurs et d'Application pour les services à L'habitat, `http://medical.imag.fr`
3. `http://www.minalogic.org`
4. `http://www.orange.fr`
5. `http://www.scalagent.com/fr`
6. `http://www-adele.imag.fr`
7. `http://www.telecom-paristech.fr/nc/formation-et-innovation-dans-le-numerique.html`

Dans ce contexte, le projet MEDICAL reprend une architecture technique aujourd'hui classique pour la mise en place de services pervasifs. Cette architecture repose sur trois plates-formes d'exécution hétérogènes et connectées, c'est-à-dire :

1. une passerelle domestique qui héberge une partie du code applicatif et se charge de l'intégration des équipements ;
2. une plate-forme B2B qui se charge de l'interconnexion entre les passerelles domestiques et l'infrastructure IT décrite ci-après. Elle doit répondre à des contraintes de passage à l'échelle et de sécurité, notamment ;
3. une infrastructure IT distante qui héberge le reste du code applicatif ainsi qu'un ensemble de services techniques déployés sur une ou plusieurs machines.

L'utilisation, au sens large, d'une telle infrastructure reste très complexe. Les exigences du projet sont dès lors de simplifier l'usage de ces infrastructures et surtout de réduire l'ensemble des coûts de développement, de mise en place et, finalement, d'exploitation et d'évolution. En ce sens, le projet vise la fourniture des éléments suivants :

- l'extension du *middleware* Cilia (au cœur de cette thèse) pour l'intégration dynamique d'équipements dans le cadre d'applications pervasives ;
- un simulateur, nommé iCasa, permettant de tester et valider les applications à installer au sein d'une maison ;
- un ensemble de cas d'utilisation permettant de valider les propositions architecturales et techniques du projet.

Techniquement parlant, le projet MEDICAL repose sur les technologies OSGi, Apache Felix iPOJO, RoSe *framework* permettant l'import et l'export de services disponibles au sein d'OW2 et JORAM [8] *middleware* de messagerie d'OW2. Comme nous l'avons vu précédemment, Cilia repose sur iPOJO. Il en est de même pour le simulateur iCasa. iCasa est en fait un écosystème constitué d'une plate-forme pour le développement [9], l'exécution [10] et la simulation [11] d'applications domotiques.

La simulation de iCasa permet de définir une habitation avec des pièces, des capteurs et des actionneurs de divers types (exemple de capteurs : détection de mouvements, mesure de luminosité, mesure de température ambiante, mesure de consommation électrique ainsi que des actionneurs comme, par exemple : interrupteurs de lumière, commandes de chauffage et de ventilateur). Ces capteurs sont positionnés dynamiquement dans les pièces de l'habitation et fournissent des mesures. Les mesures peuvent être spécifiées de façon manuelle par l'utilisateur via une fenêtre d'interaction ou de façon programmatique via l'utilisation d'un *script* qui enchaîne des commandes du simulateur. L'exécution est cadencée par une horloge virtuelle. Des personnes peuvent également être simulées. Le *script* apporte la dimension dynamique en permettant le déplacement des personnes dans chaque pièce de l'habitation.

2 Cas d'utilisation *Actimétrie*

2.1 Description

Le projet MEDICAL nous a donné accès à l'un de ses cas d'utilisation pour tester et illustrer les extensions autonomiques apportées à Cilia. Plus précisément, nous avons utilisé un service

8. Acronyme de *Java Open Reliable Asynchronous Messaging*, `http://joram.ow2.org`
9. `http://adeleresearchgroup.github.io/iCasa-IDE/`
10. `http://adeleresearchgroup.github.io/iCasa-Platform/`
11. `http://adeleresearchgroup.github.io/iCasa-Simulator/`

de suivi des habitudes nommé « Actimétrie ». Le but de ce service est de suivre l'activité d'une personne chez elle et de détecter d'éventuelles déviations. Pour cela, il faut suivre toutes les activités des personnes grâce aux capteurs disponibles, construire des modèles d'activité à l'aide des données récoltées et d'analyser ces modèles en temps réel.

Ce service d'actimétrie est implanté de façon distribuée sur une passerelle domestique et sur une infrastructure IT. La passerelle domestique est une passerelle OSGi/iPOJO. Au-dessus s'exécute le *framework* RoSe qui permet la gestion des dispositifs externes à la passerelle OSGi. RoSe est le composant qui donne l'accès à la plate-forme OSGi des données des différents capteurs distants. La passerelle abrite également une chaîne de médiation qui permet de remonter des données depuis les capteurs jusqu'à l'infrastructure IT où sont construits des modèles d'activité. Le *framework* Cilia est ainsi utilisé pour la conception, le développement et, finalement, l'exécution de la chaîne de médiation. La figure 7.2 « *Déploiement de l'application Actimétrie* » illustre le déploiement de ces différents modules sur une passerelle domestique.

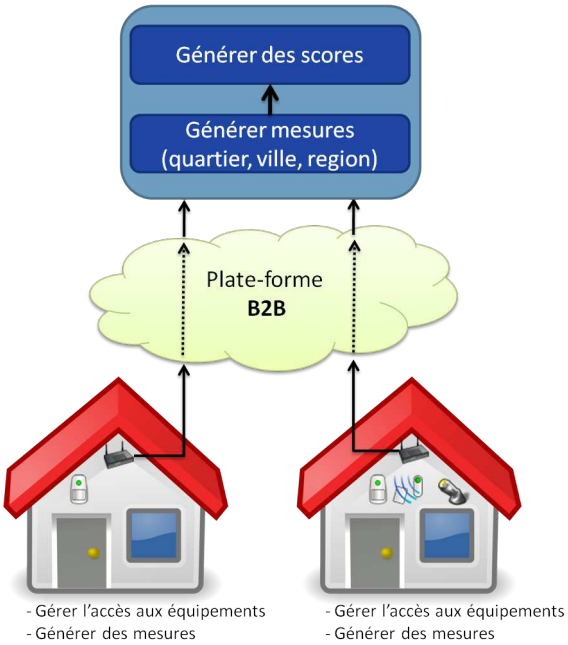


FIGURE 7.2 – Déploiement de l'application *Actimétrie*.

Dans son implantation initiale, ce service est constitué des trois modules (ou sous-systèmes) suivants :

1. **le module de génération des scores** : ce module produit des statistiques (un modèle d'activité) sur le taux d'occupation d'une personne dans les différentes pièces de l'habitation à partir des données collectées. Il est appelé « actimétrie ».
2. **le module de génération des mesures** : ce module produit des mesures exploitables pour le service d'actimétrie. Il fournit des mesures sur l'occupation des pièces en fonction des données collectées par les différents capteurs.
3. **le module d'accès aux équipements** : ce module donne l'accès à une plate-forme et à des capteurs hétérogènes, dynamiques et distribués. Ces capteurs hétérogènes sont les sources de données pour le module de génération de mesures.

L'architecture fonctionnelle du service est illustrée par la figure 7.3 « *Architecture fonctionnelle du service Actimétrie* » où nous retrouvons les trois modules mentionés précédemment. Le module « Générer des scores » ne sera pas détaillé ; il est le consommateur des données

produites par le module « Générer des mesures ». Ce dernier module est le pivot de notre validation. Nous allons, dans les sections suivantes, détailler son fonctionnement ainsi que celui du module « Gérer l'accès aux équipements » en charge de la communication avec les dispositifs physiques.

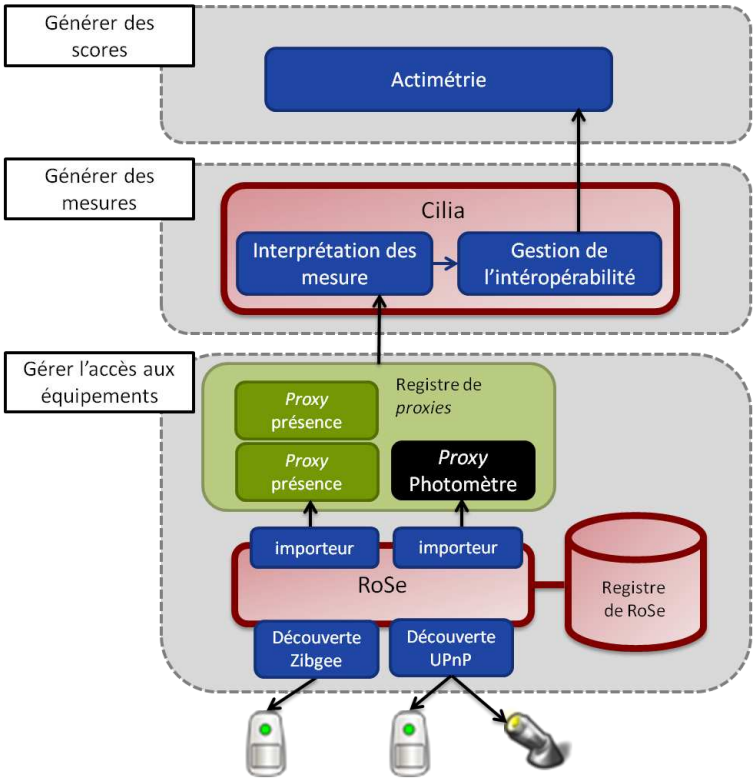


FIGURE 7.3 – Architecture fonctionnelle du service *Actimétrie*.

2.2 Module d'accès aux équipements

Le module « Gérer l'accès aux équipements » a la charge de découvrir les dispositifs physiques hétérogènes et dynamiques. Ces dispositifs sont dynamiques pour l'application de suivi des habitudes. A tout moment, ils peuvent être ajoutés, retirés, remplacés, en panne transitoire ou définitive. Les dispositifs sont hétérogènes de par leur protocole de découverte (DPWS, UPnP, Zigbee), leur protocole de communication et, finalement, leur modèle de données. Le pivot architectural de ce module est le *framework* RoSe qui, comme nous l'avons vu dans la section précédente, permet l'import de ressources distantes dans une plate-forme à composants orientés services (OSGi) et l'export de ressources locales à la plate-forme. L'import et l'export des ressources sont effectués de façon automatisée et de manière dynamique. Rappelons que, techniquement, le *framework* RoSe est basé sur la plate-forme OSGi. Il permet :

- de découvrir des services disponibles dans l'environnement d'exécution ;
- d'instancier automatiquement des *proxies* métier ;
- de gérer le cycle de vie des différents *proxies*, d'enregistrer les différents *proxies* dans la plate-forme OSGi ou bien de les retirer.

Les différents dispositifs découverts, suivant le protocole de découverte approprié, sont réifiés dans la passerelle en tant que service OSGi. Ce sont en fait des *proxies* métier permettant l'accès aux fonctionnalités des dispositifs physiques. Le cycle de vie des *proxies* géré par le *framework* RoSe est celui des dispositifs physiques : apparition, disparition et modification.

2.3 Module de génération des mesures

Le module « Générer des mesures » est en charge de la génération et de la transmission de mesures vers le module d'actimétrie. Ce module est entièrement implanté à l'aide de Cilia. La figure 7.4 « *Chaîne de médiation Actimétrie* » illustre les deux blocs fonctionnels réalisés par la chaîne de médiation Cilia qui sont : l'interprétation des mesures et la gestion de l'interopérabilité.

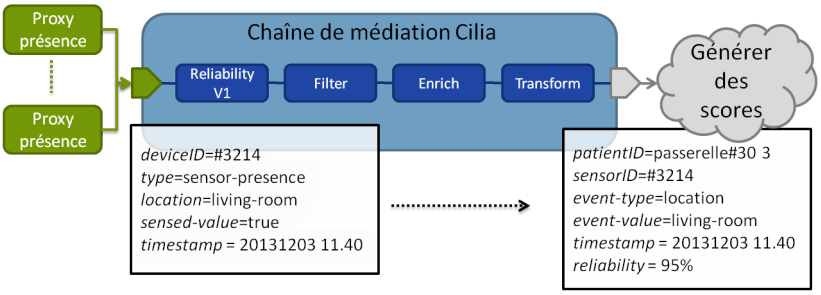


FIGURE 7.4 – Chaîne de médiation *Actimétrie*.

Cette séparation fonctionnelle détermine deux groupes de fonctions indépendantes. Le premier groupe traite du formatage et de l'enrichissement des mesures issues des *proxies* (qui, rappelons-le, assurent la communication avec le dispositif physique). Le second groupe est en charge de l'alignement syntaxique et sémantique assurant ainsi l'interopérabilité avec le module d'actimétrie.

La source des données entrantes pour le composant « Générer des mesures » est donc les *proxies* métiers. Ces derniers fournissent des informations sur l'équipement lui-même (identification physique, état de bon fonctionnement ou panne, etc.), sa localisation dans la pièce, l'heure et la date de l'élaboration de la mesure. Ces mesures issues des dispositifs doivent être filtrées, enrichies, transformées pour fournir en sortie des données de niveau métier exploitable par le module d'actimétrie (génération des scores).

D'après la figure 7.4 « *Chaîne de médiation Actimétrie* », la chaîne de médiation Cilia est constituée de deux types d'adaptateurs (un d'entrée et un de sortie) et de quatre médiateurs. Nous allons détailler le rôle de chacun des composants Cilia.

L'adaptateur d'entrée récupère les données fournies par les capteurs de présence via les *proxies* métier. Le premier médiateur nommé *Reliability* est un médiateur en charge de l'évaluation de la fiabilité de la mesure prise. Il enrichit la mesure d'une fiabilité comprise entre 0 et 100% qui est fonction, entre autres, du nombre de personnes dans l'habitat. Le message est ensuite traité par le médiateur *Filter*, dont le rôle est de supprimer le message si la mesure est obtenue avec une fiabilité inférieure à un seuil prédéfini par l'administrateur système. Le médiateur *Enrich* ajoute une information permettant d'identifier la personne détectée nommée *patientID* (la valeur est configurée lors du déploiement de la chaîne de médiation). Le dernier traitement consiste à effectuer des alignements sémantiques pour l'adaptateur de sortie en renommant des propriétés. L'adaptateur de sortie est en charge de communiquer les données résultats au service Web « Générer des scores ».

Nous avons précédemment vu que l'application de médiation Cilia est entièrement décrite par un DSL. Ce dernier déclare les types des médiateurs/adaptateurs qui doivent être instanciés et les liaisons entre les médiateurs. Le code source 7.1 « *DSL Cilia Actimétrie* » est la

chaîne de médiation de référence de l'application *Actimétrie*. Notre cas d'usage s'appuie sur cette application de médiation existante.

```
<?xml version="1.0" encoding="UTF-8"?>
<cilia>
        <chain id="generator-mesures" type="generator-mesures">
                <adapters>
                        <adapter-instance id="presence-collector" type="PresenceDetectorAdapter" />
                        <adapter-instance id="event-webservice" type="EventServiceAdapter" />
                </adapters>
                <mediators>
                        <mediator-instance id="reliability" type="MeasureReliabilityMediator" />
                        <mediator-instance id="filter" type="MeasureFilterMediator" >
                                <processor> <property name="filter.threshold"  value="70" /> </processor>
                        </mediator-instance>
                        <mediator-instance id="enricher" type="MeasureEnricherMediator" >
                                <processor> <property name="gateway.id" value="310086a0" /> </processor>
                        </mediator-instance>
                        <mediator-instance id="transformer" type="MeasureTransformerMediator" >
                                <processor> <property name="user" value="Paul"/> </processor>
                        </mediator-instance>
                </mediators>
                <bindings>
                        <binding from="presence-collector:out" to="reliability:in" />
                        <binding from="reliability:out" to="filter:in" />
                        <binding from="filter:out" to="enricher:in" />
                        <binding from="enricher:out" to="transformer:in" />
                        <binding from="transformer:out" to="event-webservice:in" />
                </bindings>
        </chain>
</cilia>
```

Code source 7.1 – DSL Cilia *Actimétrie*

3 Expérimentation

3.1 Plate-forme de validation

Nous avons étendu le cas d'utilisation de l'actimétrie de façon à mettre en évidence les évolutions autonomiques apportées par notre approche. Précisément, nous allons montrer dans cette section comment mettre en œuvre un gestionnaire autonomique fondé sur une base de connaissances construite à partir des *sensors* de Cilia. Le rôle de ce gestionnaire autonomique est d'assurer la réalisation et le maintien des objectifs de haut niveau définis par l'administrateur. Il doit aussi pouvoir fournir, en permanence, à des administrateurs l'état courant du système. A cet effet, nous avons mis en place une plate-forme de validation spécifique. Cette plate-forme, illustrée par la figure 7.5 « *Plate-forme d'Actimétrie* », est composée des différents éléments fonctionnels suivant :

- de la plate-forme d'exécution OSGi augmentée des *frameworks* Apache Felix iPOJO et RoSe ;
- du *framework* Cilia avec son application *Actimétrie*, tel que décrit dans la section précédente. Le *framework* Cilia fournit un *proxy* pour rendre accessible par REST les *touchpoints Builder* et *Runtime* (comme indiqué dans le chapitre précédent). Les composants de la chaîne de médiation (adaptateurs, médiateurs) sont implantés dans le bundle OSGi *Composants Cilia applicatifs* ;
- du module de simulation d'une maison avec son IHM Web issue de l'écosystème iCasa ;
- d'un gestionnaire autonomique qui, en fonction d'observations et de buts de haut niveau, adapte l'application Cilia en exécution. Il installe si besoin un bundle OSGi complémentaire *Composants Cilia Compendium* ;
- d'un visualisateur qui permet à un administrateur d'observer les effets des gestionnaires autonomiques associés aux applications. Ce visualisateur peut prendre deux formes : soit celle d'un outil graphique qui introspecte l'application en exécution, soit celle d'un visualisateur de fichiers de traces ;

– un répertoire externe à la passerelle. Il sert à un administrateur pour déposer des *bundles* OSGi complémentaires, des fichiers de configuration pour configurer le service autonomique.

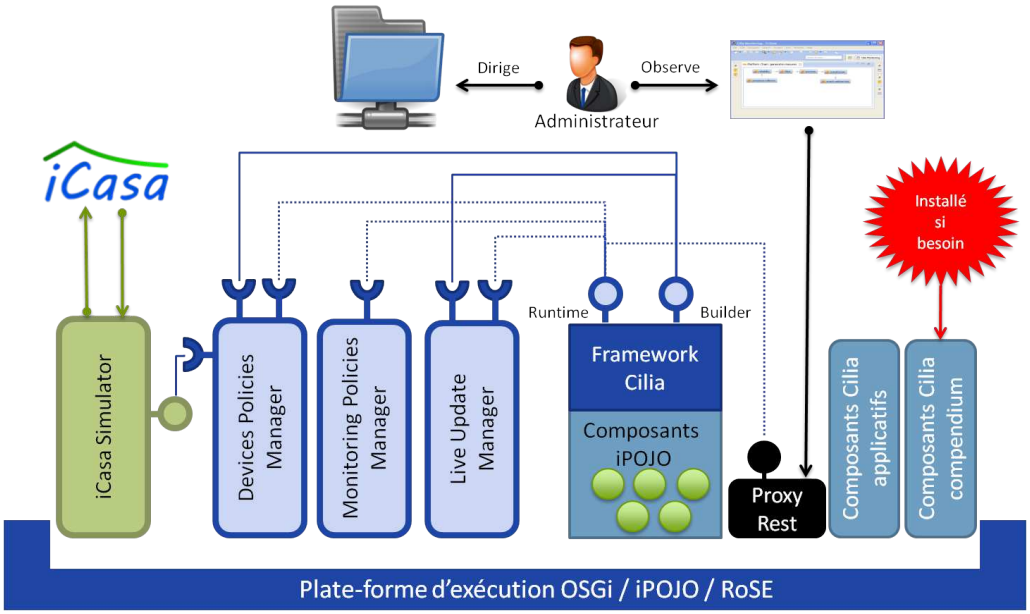


FIGURE 7.5 – Plate-forme d'*Actimétrie*.

Le gestionnaire autonomique et le visualisateur ont été développés par nos soins. Le gestionnaire autonomique comprend les services *Devices Policies Manager* et *Monitoring Policies Manager* chargés respectivement de gérer les équipements utilisés par l'actimétrie et de gérer la qualité logicielle attendue pour les modules s'exécutant sur la passerelle. On peut néanmoins noter que ces modules peuvent également s'exécuter de façon distante. En effet, les *touchpoints* (le *Runtime* et le *Builder*) du *framework* Cilia sont accessibles à distance par l'intermédiaire d'un *proxy* REST (HTTP/JSON) créé par le *framework* RoSe.

Le gestionnaire autonomique, sur la base de ses observations, adapte le système pour maintenir l'ensemble des objectifs d'administration de haut niveau définis par l'administrateur système. Pour notre cas d'usage, nous avons défini les objectifs d'administration suivants :

1. **Maintien de la qualité du service rendu par l'application**. Dans des environnements hétérogènes et dynamiques, il est difficile de définir l'ensemble des paramètres de l'application. On retrouve généralement deux familles de paramètres. Une première famille sert à configurer le *framework* et à atteindre des objectifs, par exemple, de performances (taille de *pool* des *threads*), ou de taille des empreintes mémoire (nombre et taille des *buffers* de stockage). La deuxième famille est liée à l'application et ses contraintes (période de synchronisation des médiateurs, etc.). La phase de mise au point ne permet généralement pas de tester avec l'ensemble des événements qui peuvent survenir. Généralement, la configuration obtenue n'est pas la plus optimale. Il peut être demandé à un administrateur de vérifier (périodiquement ou non) et de modifier les paramètres du *framework* et/ou de l'application en exécution.
2. **Modification de fonctionnalité de l'application en fonction du contexte**. Cette tâche peut sembler simple. Mais, une application en production ne doit pas s'arrêter. Il est généralement demandé à un administrateur (ou un groupe de personnes) d'effectuer les tâches suivantes : récupérer l'ensemble des modifications à déployer, définir un scénario de modification ainsi que les date et heure de déploiement les plus appropriées, puis

déterminer un scénario de validation du déploiement et un cas de mise en défaut de l'application sur échec du déploiement et/ou de l'exécution de la nouvelle application.

Au travers de ces deux cas d'usage, nous voulons démontrer que l'ensemble des propriétés ajoutées au *framework* Cilia ; c'est-à-dire la surveillance, l'adaptation dynamique et la base de connaissances causale permettent de développer des applications Cilia autonomiques de manière aisée.

Nous ne cherchons pas ici à évaluer le formalisme utilisé pour décrire les différents objectifs de haut niveau.

3.2 Conditions initiales

Dans les deux cas d'usage implantés, nous partons d'une application dans un état stable, puis effectuons des modifications structurelles et/ou fonctionnelles pour arriver à une nouvelle configuration (qui, elle aussi, sera au bout d'un temps fini dans un état stable) réalisant des objectifs de haut niveau spécifiés par un administrateur. Les résultats seront disponibles sous forme de fichiers de traces et de copies d'écran.

Rappelons que lorsqu'un système physique (matériel, logiciel) est **modifié**, il passe d'un **état stable** à **un état instable**, pour atteindre (si tout se passe bien) **un nouvel état stable**. Un tel système peut avoir plusieurs avenirs/états possibles suite à une modification. **Le gestionnaire autonomique a également pour rôle de suivre le comportement dynamique du système modifié et d'assurer son retour dans un état stable**.

Dans les différentes expériences, l'état initial et stable est le suivant. Il s'agit d'une habitation (voir figure 7.6 « *État initial de l'application simulée* » page 178) constituée d'une salle de bain, d'une chambre, d'un couloir, d'un salon et d'une cuisine. Toutes les pièces sont équipées d'un dispositif *détecteur de présence*. L'habitant est dans le salon ; le détecteur de présence du salon produit des mesures en ce sens.

Cet état est spécifié au travers du simulateur iCasa à l'aide du langage de *script* associé. Ce même langage va nous permettre ensuite de faire évoluer dynamiquement cet environnement d'exécution. Par exemple, il va nous permettre de déplacer une ou plusieurs personnes dans l'appartement et d'ajouter de nouveaux dispositifs dans les pièces de l'habitation.

FIGURE 7.6 – État initial de l'application simulée.

Détaillons à présent les deux scénarios que sont le *Maintien de la qualité de service* et *l'amélioration de l'application*.

3.3 Cas d'utilisation 1 : Maintien de la qualité de service

Ce service est en charge du maintien de la qualité du service rendu par le module *Générer des mesures*. Pour cela, il s'agit d'ajuster les paramètres de ce module de façon dynamique en fonction des évolutions de la qualité logicielle observée.

La qualité de service attendue est spécifiée par un but administratif constitué d'une règle que l'application doit continuellement respecter, d'une action corrective pour amener le système dans l'état voulu et d'un comportement si la règle n'est pas respectée. Précisément, le but considéré est le suivant :

Règle *: une mesure au plus est filtrée par heure.*
Correction *: le seuil de filtrage qui définit le rejet d'une mesure peut être augmenté une fois de 10 points.*
Comportement en cas d'échec de l'action corrective *: une alarme horodatée permet à un administrateur d'être notifié et d'identifier la passerelle.*

Nous avons implanté cette règle dans le gestionnaire autonomique sous la forme d'un service iPOJO. Dans un premier temps, nous allons détailler les éléments fonctionnels de ce service, puis dans un second temps, nous visualiserons les effets du gestionnaire autonomique lorsque la règle précédente n'est plus respectée par le système.

Le premier besoin fonctionnel consiste à fournir une méthode à exécution périodique. Nous avons utilisé pour cela la classe `Timer` de Java. Cette classe est configurée pour exécuter une fonction une fois par heure. A chaque exécution de cette fonction, la valeur du compteur de messages filtrés (qui est une variable de classe) est contrôlée. Si sa valeur est supérieure ou égale à deux alors un message est ajouté à un fichier de traces. De plus, si le médiateur *Filter* n'a pas encore été modifié, sa propriété `filter.threshold` (voir le code source 7.1 « *DSL Cilia Actimétrie* », page 175 est incrémentée de 10 points. Cette modification de propriété passe par le modèle du médiateur et, plus précisément, les méthodes `getModel()` qui à partir d'un nœud retourne le méta-modèle du médiateur/adaptateur en exécution et `get/setProperty` qui offrent les opérations de lecture et de modification de propriété du méta-modèle d'un médiateur et/ou adaptateur. L'exécution de cette méthode se termine par une remise à 0 du compteur des messages filtrés. Le code source 7.2 « *Fonction périodique (extrait)* » fournit le code de la fonction exécutée périodiquement par la classe `Timer` de Java.

```
. . .
@Override
public void run() {
        logger.info("Checking the number of measurement filtered (current value ={})",counterFilterMsg);
        synchronized (_lock) {
                if (counterFilterMsg >= m_threshold) {
                        logger.error("QoS Fault : too much messages filtered");
                        /* Try one corrective action */
                        actionCorrective();
                }
                counterFilterMsg = 0;
        }
}
private void actionCorrective() {
        /* Modifications */
        if (!done) {
                done = true;
                try {
                        /* Retreive the modele for the relevant node */
                        MediatorComponent model = ciliaContext.getApplicationRuntime().getModel(nodeFilter);
                        /* Increase thresold level to 10 points */
                        try {
                                int i = Integer.parseInt((String) model.getProperty("filter.threshold")) + 10;
                                model.setProperty("filter.threshold", Integer.toString(i));
                                logger.info("Node [{}],the thresolhd has been set to [{}]",model.getId(), model.getProperty
                                        ("filter.threshold"));
                        } catch (NumberFormatException e) {
                                logger.error("Node [{}] has no property [filter.threshold] ",nodeFilter.nodeId());
                        }
                }
                catch (CiliaIllegalParameterException e) {}
                catch (CiliaIllegalStateException e) {}
        }
}
```

Code source 7.2 – Fonction périodique (extrait).

Nous avons avec cette fonction mis en œuvre l'API réflexive de Cilia, principalement pour lire et modifier le paramètre `threshold.value`. Cette propriété est définie dans la description du médiateur et elle est configurable dans le DSL Cilia. Le *touchpoint* nous a permis de récupérer l'instance du médiateur nécessitant des ajustements du paramètre.

Attachons nous maintenant à la description de la détection d'un message filtré. Le code source des médiateurs/adaptateurs ne peut pas être recompilé. Il est, par conséquent, nécessaire de déterminer si un message a été filtré ou non. Cette opération doit être effectuée sans intrusion dans le code fonctionnel des médiateurs et des adaptateurs. Pour ce cas d'usage nous avons utilisé la chaîne de médiation du projet Médical.

Détaillons le rôle des différents médiateurs : le médiateur [12] *Filter* ne propage pas le message au médiateur suivant *Enrich* (voir la figure 7.4 « *Chaîne de médiation Actimétrie* » page 174) en générant un message Java `null` en sortie de la phase *Processor*. Ce principe de filtre utilise la spécification du *framework* Cilia qui définit que les messages Java `null` sont ignorés. Le *framework* Cilia, avant chaque exécution des entités *Scheduler/Processor/Dispatcher*, le

12. Rappelons qu'un médiateur est constitué des trois entités qui sont le *scheduler*, le *processor* et le *dispatcher*.

message est testé et s'il est `null` l'entité ne sera pas démarrée. Détecter un message `null` à la fin de la phase *processor* est l'événement qui va être utilisé pour incrémenter la variable de classe représentant le nombre de messages filtrés.

Revenons sur les variables d'états, elles permettent de reconstruire la dynamique d'exécution des médiateurs. Notamment la variable `process.exit.data` fournit la valeur du message horodaté en sortie de l'entité *processor*, sa mesure est tout simplement le message qui va être en entrée de l'entité *dispatch*. Détecter un message filtré revient à lire la mesure de la variable d'état `process.exit.data` et de vérifier si elle est égale à `null` avec une date (une mesure est un objet horodaté).

Par défaut et pour des raisons de performance d'une part et d'autre part dans un souci de diminution de l'empreinte mémoire du *framework* Cilia, le *monitoring* n'est pas actif. Pour chaque adaptateur/médiateur, toutes les variables d'états sont dans l'etat *disable* (aucune mesure n'est générée) et la profondeur d'historisation dans la base de connaissances est configuré à 0. Il reste deux étapes :

1. **configurer** le *monitoring* spécifique pour le médiateur *Filter* ;
2. **implanter** une fonction pour déterminer si un message est filtré.

La configuration du *monitoring* du médiateur est apportée par l'objet Java `SetUp`. C'est un objet de niveau *meta-level* qui est en charge de configurer les variables d'état actives et la profondeur d'historisation pour l'instance du médiateur. Techniquement, l'objet `SetUp` est un *proxy* Java d'un nœud de médiation (`Setup setup = nodeSetUp(node)`). Cet objet permet de configurer individuellement les variables d'état du médiateur ainsi que la profondeur d'historisation dans la base de connaissances.

Le service *QoSManager* est un service local à la plate-forme OSGi. Nous avons configuré la base de connaissances pour qu'elle génère des événements de niveau nœud [13] et de niveau variables d'état. En effet, les événements de niveau nœud permettent de notifier leurs apparitions, leurs disparitions et, finalement, leurs modifications. Les événements de niveau variables permettent de notifier le changement d'état d'une variable (passage de *enable* à *disable* et vice-versa), mais aussi lorsque le *framework* sauvegarde une mesure dans la base de connaissances.

La configuration des événements de type nœud, nécessite un filtre LDAP. Ce filtre permet de sélectionner les nœuds pour lesquels il est intéressant d'être notifié. Le code source 7.3 « *Enregistrement des événements nœuds (extrait)* » est un extrait de la méthode `start()` appelée au démarrage du *bundle* OSGi (`start`), tous les événements (arrivée, départ, modification) de niveau nœud de toutes les chaînes de médiation en exécution seront notifiés au service *QoSManager*.

```
1  public void start() {
       try {
3              // Register all events level node ( arrival, departure ...)
               ciliaContext.getApplicationRuntime().addListener("(&(chain=*)(node=*))",(NodeCallback) this);
5              // Start the countdown Timer
               timer.schedule(this, m_period, m_period);
7              logger.debug("Service [QoS Policies Manager] started");
       } catch (CiliaIllegalParameterException e) {
9              logger.error("LDAP filter error  [{}]", e.getMessage());
       } catch (CiliaInvalidSyntaxException e) {
11             logger.error("LDAP syntax error [{}]", e.getMessage());
       }
13 }
```

Code source 7.3 – Enregistrement des événements nœuds (extrait).

13. Un nœud est une même représentation d'un médiateur et d'un adaptateur en exécution.

Pour ce cas d'usage, la profondeur de l'historisation n'a aucune importance, nous n'avons pas, d'une part, un besoin de reconstruction de la dynamique du passé et, d'autre part, nous avons configuré la base de connaissances pour être notifié de chaque nouvelle mesure disponible. Par conséquent, seule la dernière mesure est sauvegardée (profondeur d'historisation = 1). Le code source 7.4 « *Méthode callback d'un nœud (extrait)* » est un extrait de la méthode *callback* appelée à l'arrivée d'un nœud (la création d'une instance qu'elle soit pour un médiateur/adaptateur). La méthode enregistre les événements de niveau variable et configure le *monitoring* dynamique (la profondeur d'historisation), l'état est positionné à *enable* (`true` dans le code source) pour autoriser la remontée des mesures dans le méta-modèle en exécution (voir section 3.2 « *Architecture réflexive* », page 149).

```
. . .
private boolean isFilterMediator(Node node) throws CiliaIllegalParameterException, CiliaIllegalStateException {
        MediatorComponent model = ciliaContext.getApplicationRuntime().getModel(node);
        return (model.getType().compareTo(FILTER_MEDIATOR_TYPE) == 0);
}
@Override
/**
 * On node arrival , set the monitoring policies
 */
public void onArrival(Node node) {
        /* set the monitoring for all node */
        Setup setup = ciliaContext.getApplicationRuntime().nodeSetup(node);
        try {
                /* if the node is an instance of Filter */
                if (isFilterMediator(node)) {
                        nodeFilter = node;
                        /* install the callback on event new measure stored */
                        /* set the state var dispatcher.data */
                        setup.setMonitoring("process.exit.data", 1, null, true);
                        StringBuffer sb = new StringBuffer("(&(chain=*)(node=");
                        sb.append(node.nodeId()).append("))");
                        ciliaContext.getApplicationRuntime().addListener(sb.toString(),(VariableCallback) this);
                        logger.info("Node [{}], state Var [process.exit.data] successfully configured",node.nodeId());
                }
        } catch (CiliaIllegalParameterException e) {
                logger.error(e.getMessage());
        } catch (CiliaIllegalStateException e) {
                logger.error(e.getMessage());
        } catch (CiliaInvalidSyntaxException e) {
                logger.error(e.getMessage());
        }
}
```

Code source 7.4 – Méthode *callback* d'un nœud (extrait).

Nous avons mis en œuvre dans cette implantation (voir le code source 7.3 « *Enregistrement des événements nœuds (extrait)* » et code source 7.4 « *Méthode callback d'un nœud (extrait)* ») la configuration de la base de connaissances pour, d'une part, la configurer pour générer des événements sur des événements issus des nœuds de médiation (adaptateurs/médiateurs) et des variables d'état (états de la variable et génération d'une mesure). et, d'autre part, la base de connaissances a été configurée pour définir la profondeur d'historisation des mesures. Nous avons aussi utilisé le *monitoring* dynamique ; c'est-à-dire que nous avons configuré une seule variable d'état du seul médiateur *Filter*.

Avant de décrire le scénario dynamique et les effets du gestionnaire autonomique sur l'application en exécution, nous allons présenter rapidement le principe du médiateur *Reliability*. Ce médiateur est en charge de définir une valeur entre 0 et 100% qui donne la fiabilité de la mesure. La fiabilité diminue de 10% par nombre de détecteurs de présence actifs. Lorsqu'une seule personne est dans l'habitation, la mesure a une fiabilité de 90%. Trois personnes dans l'habitation la fiabilité baisse à 70%. Ici encore, l'algorithme du calcul de la fiabilité n'a pas d'importance pour notre cas d'usage.

Nous allons avec le *script* de simulation iCasa, ajouter des personnes dans différentes pièces. La personne entrante dans la pièce sera l'événement qui fera changer l'état du détecteur de présence de non actif à actif. Sur cette transition, l'adaptateur Cilia de présence injectera une nouvelle mesure. Le *script* va positionner deux personnes dans des pièces dif-

férentes, au total trois capteurs de présence seront actifs et deux nouvelles mesures seront injectées. La figure 7.7 « *Etat final du cas d'utilisation Maintien de la QoS* » page 182 illustre le résultat de la simulation. Rappelons que l'état initial est celui de la figure 7.6 « *État initial de l'application simulée* » page 178.

FIGURE 7.7 – Etat final du cas d'utilisation *Maintien de la QoS*.

La première mesure injectée aura une fiabilité calculée 80%. Elle ne sera pas filtrée. La seconde mesure aura une fiabilité calculée de 70%, elle sera filtrée (voir le code source 7.5 « *Trace de l'activité du cas d'usage Maintien de la QoS (extrait)* » les messages du médiateur *MeasureFilter*, `**** measure filtered ****`). Et, pour terminer, la troisième personne entrante dans une pièce (dans la cuisine voir le code source 7.5 « *Trace de l'activité du cas d'usage Maintien de la QoS (extrait)* ») va déclencher une mesure avec une fiabilité calculée de 60%, qui sera elle aussi filtrée.

```
...
2013-03-16 16:17:31,841 [INFO]  (MeasureCollector) : Adapter presence --> location=[salon], sensor value=[true]
2013-03-16 16:17:31,843 [INFO]  (MeasureReliability) : Mediator reliability --> reliability set to 90.0%
2013-03-16 16:17:31,859 [INFO]  (MeasureEnricher) : Mediator enricher --> the gatewayId set to [310086a0]
2013-03-16 16:17:31,859 [INFO]  (MeasureTransformer) : Mediator transformer --> Patient Id set Paul
2013-03-16 16:17:31,875 [INFO]  (MeasureSender) : Adapter Web Service --> message processed
2013-03-16 16:17:31,875 [INFO]  (MeasureCollector) : Adapter presence --> location=[chambre], sensor value=[true]
2013-03-16 16:17:31,906 [INFO]  (MeasureReliability) : Mediator reliability --> reliability set to 80.0%
2013-03-16 16:17:31,906 [INFO]  (MeasureEnricher) : Mediator enricher --> the gatewayId set to [310086a0]
2013-03-16 16:17:32,031 [INFO]  (MeasureTransformer) : Mediator transformer --> Patient Id set Paul
2013-03-16 16:17:32,031 [INFO]  (MeasureSender) : Adapter Web Service --> message processed
2013-03-16 16:17:32,046 [INFO]  (MeasureCollector) : Adapter presence --> location=[salle-de-bain], sensor value=[true]
2013-03-16 16:17:52,031 [INFO]  (MeasureReliability) : Mediator reliability --> reliability set to 70.0%
2013-03-16 16:17:52,031 [INFO]  (MeasureFilter) : Mediator Filter --> reliability 70.0%, threshold=70.0% ,**** measure
      filtered ****
2013-03-16 16:18:06,562 [INFO]  (MeasureCollector) : Adapter presence --> location=[cuisine], sensor value=[true]
2013-03-16 16:18:14,984 [INFO]  (MeasureReliability) : Mediator reliability --> reliability set to 60.0%
2013-03-16 16:18:14,984 [INFO]  (MeasureFilter) : Mediator Filter --> reliability 60.0%, threshold=70.0% ,**** measure
      filtered ****
2013-03-16 16:18:15,000 [INFO]  (MeasureCollector) : Adapter presence --> location=[salon], sensor value=[true]
2013-03-16 16:18:16,703 [INFO]  (QoSPolicyManager) : Checking the number of measurements filtered (current value =2)
2013-03-16 16:18:16,718 [ERROR] (QoSPolicyManager) : QoS Fault : too much messages filtered
2013-03-16 16:18:16,718 [INFO]  (QoSPolicyManager) : Node [filter] threshold set to [80%]
...
```

Code source 7.5 – Trace de l'activité du cas d'usage *Maintien de la QoS* (extrait).

A l'exécution de la fonction périodique, le nombre de messages filtrés est supérieur à un (voir le code source 7.5 « *Trace de l'activité du cas d'usage Maintien de la QoS (extrait)* », le message `QosPolicyManager checking the number of measurements filtered`), la boucle autonomique devient active et effectue son action corrective avec la mise à jour de la propriété `filter.threshold` du médiateur *Filter* en augmentation son seuil de filtrage de 10%.

3.4 Cas d'utilisation 2 : Modification de fonctionnalité

Ce second cas d'usage met en œuvre un gestionnaire autonomique pour modifier les fonctionnalités de l'application en exécution. Ce cas d'usage est l'implantation du second besoin tel que décrit plus tôt dans cette section. Sa réalisation implique la recherche de code manquant, le déploiement sur la passerelle, la modification de l'application en exécution et la vérification du bon fonctionnement de l'application.

Le cas d'usage, que nous allons détailler, permet de mettre en œuvre un scénario lié à l'évolution d'une application pendant sa phase d'exploitation. Dans notre contexte, les mises à jour peuvent être dues à :

1. des corrections de défauts : un médiateur/adaptateur en exécution est remplacé par un autre de même type ;
2. des évolutions fonctionnelles : ce sont des modifications topologiques qui ont pour effet de modifier la structure de la chaîne de médiation. Il peut s'agir de la modification, du remplacement ou de l'ajout de médiateurs ;
3. des évolutions de contexte : ce sont, généralement, les adapteurs qui sont modifiés, ajoutés ou remplacés. Mais, il peut également s'agir de médiateurs internes.

Le cadre de ce cas d'utilisation est similaire au précédent ; c'est-à-dire le service *Actimétrie* et le module de génération de mesures et l'état initial sont tels que décrit dans la section 3.2 « *Conditions initiales* », page 177. L'objectif de haut niveau que nous avons défini est le suivant :

Les photomètres améliorent la fiabilité d'une mesure de présence d'un patient. Ils doivent être déployés dans les pièces où les capteurs de présence sont placés face à une source de lumière directe. La mesure de fiabilité est augmentée de 10% pour les mesures issues d'une pièce avec deux capteurs (présence et photomètre).

Détaillons à présent notre scénario. Alors que l'application de médiation est en exécution, nous ajoutons un capteur de type photomètre dans la cuisine. Le gestionnaire autonomique doit adapter l'application en conséquence. Cela signifie qu'il doit tout d'abord rechercher le code de l'adaptateur photomètre et celui du médiateur *Reliability* prenant un compte un photomètre pour calculer une fiabilité plus précise. Ensuite, il doit instancier l'adaptateur en charge de traiter les données du photomètre, remplacer le médiateur *Reliability* par sa nouvelle version, connecter l'adaptateur présence au médiateur et, finalement, détruire l'ancienne version du médiateur.

Le visualisateur est utilisé pour présenter à tout moment l'état de la chaîne de médiation. Il communique avec la passerelle OSGi par le *proxy* REST (voir la figure 7.5 « *Plate-forme d'Actimétrie* » page 176) et découvre ainsi la topologie de la chaîne de médiation en exécution. Pour chaque nœud découvert, il présente les valeurs des variables d'état actives. Comme nous

l'avons dit précédemment, ce visualisateur repose **uniquement** sur les API du *touchpoint Runtime*. La figure 7.8 « *Etat initial de la chaîne de médiation obtenue dans le visualisateur* » est une copie d'écran de ce visualisateur.

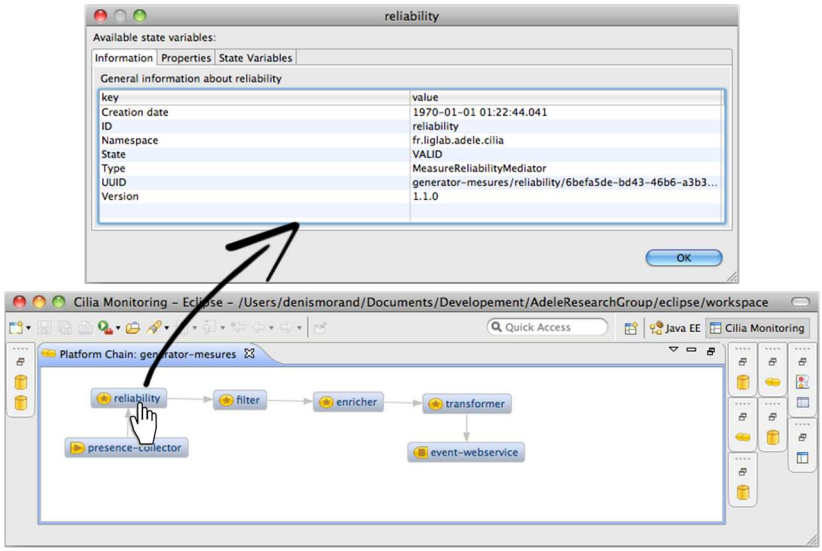


FIGURE 7.8 – Etat initial de la chaîne de médiation obtenue dans le visualisateur.

Nous avons implanté ce gestionnaire autonomique sous la forme d'un service OSGi nommé *DevicesPoliciesManager* (voir figure 7.5 « *Plate-forme d'Actimétrie* »). La mise en place de ce gestionnaire demande de pouvoir :

1. réagir à l'arrivée d'un dispositif et de l'identifier ;
2. vérifier la présence d'un *bundle* installé et opérationnel dans la plate-forme, installer un bundle OSGi manquant ;
3. découvrir la topologie de la chaîne de médiation en exécution ;
4. modifier la topologie de la chaîne de médiation en exécution et présenter les résultats.

Comme indiqué sur la figure 7.9 « *Phase d'activités du gestionnaire DevicesManagerPolicies* » page 184, nous avons ainsi défini trois phases d'activités pour le gestionnaire autonomique. Ces différentes phases sont présentées ci-après.

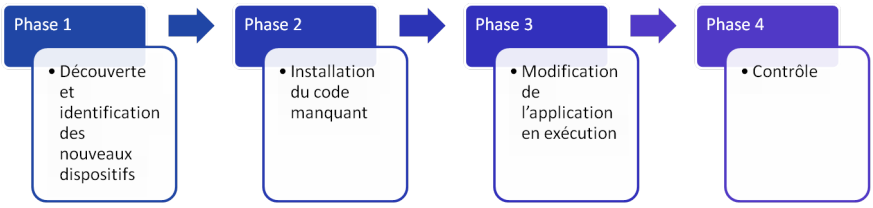


FIGURE 7.9 – Phase d'activités du gestionnaire *DevicesManagerPolicies*.

3.4.1 Phase 1 : Découverte

Cette phase permet de rendre sensible le gestionnaire autonomique à son environnement d'exécution. A cet effet, le simulateur iCasa fournit les API pour s'abonner aux changements relatifs aux dispositifs simulés (apparition, disparition, modification). Pour s'abonner à ces

événements, il suffit d'implanter les méthodes de l'interface `LocatedDeviceListener` et enregistrer la classe dans le simulateur.

Le code source 7.6 « *Phase 1 (extrait)* » est un extrait de la méthode `start()` exécutée au démarrage du *bundle* OSGi (démarrage du service). Le fichier de propriété lu dans la méthode `start()` permet de faire le lien entre un type de dispositif physique et l'adaptateur à instancier.

```
. . .
public void start() {
        /* Installing callback on simulated devices */
        iCasaContext.addListener(this);
        /* Loading configuration file : devices allowed to be configured */
        if (m_devicesProperties != null) {
                try {
                        configDevices.load(new URL(m_devicesProperties).openStream());
                } catch (MalformedURLException e) {
                        logger.error("Invalid URL");
                } catch (IOException e) {
                        logger.error("url {} is not existing", m_devicesProperties);
                }
        }
        logger.info("Service {} started ", this.getClass().getSimpleName());
}
@Override
        /**
         * @param device created by the simulator
         */
public void deviceAdded(LocatedDevice device) {
        final String adapterType;
        final String deviceType = device.getType();
        logger.info ("*** Running phase 1 ***") ;
        logger.info("Device discovered type [{}], serial number [{}] ", deviceType,.getSerialNumber());
        /* Retreive the name (or null) of the Cilia adapter */
        adapterType = configDevices.getProperty(deviceType);
        if (adapterType == null) {
                logger.info("The type [{}] is not allowed to be configured ",device.getType());
                logger.info("*** End phase 1 ***") ;
        } else {
                /* Spwan an asynchronous job : Phase two */
                Runnable task = new Runnable() {
                @Override
                /* Check if the code is available in the gateway */
                public void run() { phaseTwo(adapterType); }
                };
                new Thread(task).start();
        }
}
```

Code source 7.6 – Phase 1 (extrait).

La méthode `deviceAdded` est appelée par le simulateur iCasa lors d'événements de niveau dispositifs (apparition/disparition/modification). Pour notre implantation, cette méthode vérifie s'il existe un adaptateur réalisant la communication avec le dispositif nouvellement instancié. Dans le cas d'une réponse positive, la phase 2 est lancée de manière asynchrone.

3.4.2 Phase 2 : Installation

Le rôle de cette seconde phase est d'installer le code manquant, en l'occurrence celui lié aux adaptateurs et aux nouvelles versions de médiateur. Précisément, cette phase consiste à rechercher si le code des adaptateurs/médiateurs est déjà déployé dans la passerelle. Si ce n'est pas le cas, alors les *bundles* manquants sont recherchés et installés dans la passerelle OSGi.

Rappelons que les composants Cilia sont tous instanciés sous forme de composants iPOJO . Le *framework* Cilia peut instancier un composant si et seulement si sa *factory* est présente. Si cette *factory* n'existe pas (voir la méthode `isbundleMissing()` du code source 7.7 « *Phase 2 (extrait)* » ci-après), le bundle est recherché dans un dépôt (voir figure 7.5 « *Plate-forme d'Actimétrie* ») et installé en utilisant l'API OSGi `installBundle`.

```
. . .
```

```
2  @Override
   //OSGi callback : bundle events
4  public void bundleChanged(BundleEvent event) {
        if (event.getType() == BundleEvent.INSTALLED) {
6               logger.info("Bundle {} installed successfully", event.getBundle().getSymbolicName());
        try {
8               event.getBundle().start();
        } catch (BundleException e) {
10              logger.error("Error while starting the bundle {} : message :{}", event.getBundle().getSymbolicName(), e.
                     getMessage());
                bundleContext.removeBundleListener(this);
12              /* Remove the phase 2 pending */
                phaseThreePending.remove(event.getBundle().getLocation());
14              e.printStackTrace();
                }
16      }
        if (event.getType() == Bundle.ACTIVE) {
18              logger.info("Bundle {} started successfully", event.getBundle().getSymbolicName());
                /* Remove the callback */
20              bundleContext.removeBundleListener(this);
                /* Spwan an Phase 2 asynchronous Job Cilia modification topology */
22              final String adapterType = phaseThreePending.get(event.getBundle().getLocation());
                /* Spwan asynchronous job : Running Phase Three */
24              Runnable task = new Runnable() {
                        @Override
26                      /* Check if the code is available in the gateway */
                        public void run() { phaseThree(adapterType); }
28              };
                new Thread(task).start();
30      }
   }
32 /* Ckeck if the factory for the Cilia component is existing */
   private boolean isBundleMissing(String type) throws InvalidSyntaxException {
34      ServiceReference[] sr = null;
        return (bundleContext.getServiceReferences(Factory.class.getName(),("(factory.name=" + type + ")")) == null);
36 }
   /* Installing the missing code */
38 private void phaseTwo(String adapterType) {
        try {
40              logger.info("*** End phase 1 ***") ;
                logger.info("*** Running phase 2 *** ");
42              if (!isBundleMissing(adapterType)) {
                        logger.info("Code for Cilia component [{}] is already installed in the gateway",adapterType);
44                      /* Running immediately the phase 2 : Cilia component installation */
                         phaseThree(adapterType);
46              } else {
                        /* The bundle name to install */
48                      logger.info("Code for Cilia component [{}] is not installed in the gateway",adapterType);
                        String location = configDevices.getProperty(adapterType);
50                      if (location != null) {
                                logger.info("Code found for the Cilia component [{}] in bundle [{}]",adapterType, location)
                                        ;
52                              /* Store relation bundle/adapter, for asynchronous run Phase */
                                phaseThreePending.put(location, adapterType);
54                              /* Install the callback : Phase two will run when bundle
                                 * State is ACTIVE */
56                              bundleContext.addBundleListener(this);
                                /* Install the bundle in the gateway */
58                              bundleContext.installBundle(location);
                        } else {
60                              logger.info("No code found for the Cilia component [{}]", adapterType);
                        }
62              }
        } catch (BundleException e) {
64              bundleContext.removeBundleListener(this);
                logger.error("Error while installing bundle " + e.getMessage());
66      } catch (InvalidSyntaxException e) {
                logger.error("Syntax error " + e.getMessage());
68      }
   }
```

Code source 7.7 – Phase 2 (extrait).

La couche cycle de vie de OSGi est en charge de démarrer le *bundle*. Sur réception de l'événement `start` du *bundle* précédemment installé (voir la méthode `bundleChanged()` du code source 7.7 « *Phase 2 (extrait)* »), la phase trois est exécutée dans un thread indépendant.

3.4.3 Phase 3 : Modification

La phase de modification s'appuie uniquement sur les *touchpoints* du *framework* Cilia. Son rôle est de modifier effectivement l'application de médiation en exécution. Au niveau de notre cas d'utilisation, nous passons d'une architecture de médiation contenant un adaptateur de capteur de présence et un médiateur *Reliability* en version V1 à une architecture contenant

un adaptateur de présence (non modifié), un adaptateur pour le capteur photomètre et un médiateur de calcul de fiabilité V2. A noter, que dans sa version V1, le médiateur *Reliability* contient un seul port d'entrée alors qu'il en a deux dans sa version V2. Soit, un port inchangé pour traiter les messages en provenance de l'adaptateur de présence et un second pour traiter les messages en provenance de l'adaptateur photomètre. Il implante aussi le nouvel algorithme de calcul de fiabilité. La figure 7.10 « *Chaîne de médiation Actimétrie* » *avec photomètre* » illustre la nouvelle topologie qui devra être réalisée.

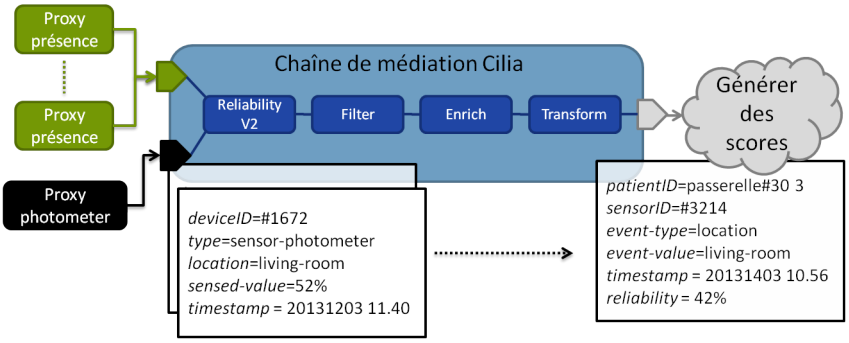


FIGURE 7.10 – Chaîne de médiation *Actimétrie* » avec photomètre.

Le code source 7.8 « *Phase 3 (extrait)* » présente un extrait de code où les actions déroulées en séquences sont :

- l'**instanciation** de l'adaptateur photomètre ;
- le **remplacement** du médiateur *Reliability* V1 par la version V2 (la liaison avec l'adaptateur de présence est conservée par l'action remplace) ;
- la **liaison** entre l'adaptateur photomètre et le médiateur *Reliability* V2 ;
- la **destruction** de l'instance du médiateur *Reliability* V1.

```
. . .
private void phaseThree(String adapterType) {
        logger.info("*** End phase 2 *** ");
        try {
                Node[] node = ciliaContext.getApplicationRuntime().nodeByType(adapterType);
                if (node.length == 0) {
                        /* Load the configuration file list of mediators to change , replace , update */
                        if (m_mediatorsProperties != null) {
                                try {
                                        configMediators.load(new URL(m_mediatorsProperties).openStream());
                                } catch (MalformedURLException e) {
                                        logger.error("Invalid URL");
                                } catch (IOException e) {
                                        logger.error("url {} is not existing", m_mediatorsProperties);
                                }
                        }
                        /* No type instancied */
                        try {
                                logger.info("*** Running phase 3 *** ");
                                String chainId[] = ciliaContext.getApplicationRuntime().getChainId();
                                /* Get the builder */
                                Builder builder = ciliaContext.getBuilder();
                                /* Instanciate an adapter type */
                                logger.info("Create an instance of adapter type [{}]", adapterType);

                                builder.get(chainId[0]).create().adapter().type(adapterType)
                                                .category("medical").id(buildInstanceName(adapterType));
                                /* Instanciate a mediator reliability V2 */
                                logger.info("Create an instance of Mediator type [{}]",
                                                MEDIATOR_RELIABILITY_V2);
                                builder.get(chainId[0]).create().mediator()
                                                .type(MEDIATOR_RELIABILITY_V2).category("medical")
                                                .id(buildInstanceName(MEDIATOR_RELIABILITY_V2));
                                builder.done();
                                /* Retreive the two nodes mediators reliability V1 and V2 */
                                Node[] newNode = ciliaContext.getApplicationRuntime().nodeByType(
                                                MEDIATOR_RELIABILITY_V2);
                                Node[] old = ciliaContext.getApplicationRuntime().nodeByType(
                                                MEDIATOR_RELIABILITY);
                                /* Retreiving adapter instance previously created */
                                Node[] adapterNode = ciliaContext.getApplicationRuntime().nodeByType(
                                                adapterType);
```

```
43                                  builder = ciliaContext.getBuilder();
                                    /* Replace on the fly the mediator V1 by V2 */
45                                  logger.info("Replace on the fly mediator instance [{}] by [{}]",
                                                old[0].nodeId(), newNode[0].nodeId());
47                                  builder.get(chainId[0]).replace().id(old[0].nodeId())
                                                .to(newNode[0].nodeId());
49                                  builder.done();
                                    logger.info("Add binding from [{}] to [{}]", adapterNode[0].nodeId(),
51                                              newNode[0].nodeId());
                                    /* Binding from photometer to mediator */
53                                  logger.info("Add binding from [{}] to [{}]", adapterNode[0].nodeId(),
                                                newNode[0].nodeId());
55                                  builder = ciliaContext.getBuilder();
                                            builder.get(chainId[0]).bind().from(adapterNode[0].nodeId() + ":in")
57                                                      .to(newNode[0].nodeId() + ":in-photometer");
                                    /* Remove unused mediator instance */
59                                  logger.info("Remove mediator instance [{}]", old[0].nodeId(),newNode[0].nodeId());
                                    builder.get(chainId[0]).remove().mediator().id(old[0].nodeId());
61                                  builder.done();
                                    logger.info("*** End phase 3 *** ");
63                              } catch (BuilderConfigurationException e) {
                                    logger.error("Builder configuration Exception {}", e.getMessage());
65                                  e.printStackTrace();
                                } catch (BuilderException e) {
67                                  logger.error("Builder Exception {}", e.getMessage());
                                    e.printStackTrace();
69                              } catch (BuilderPerformerException e) {
                                    logger.error("BuilderPerformerException {}", e.getMessage());
71                                  e.printStackTrace();
                                }
73                          } else {
                                logger.info("Component type [{}] already instancied [{}]", adapterType,
75                                          node[0].nodeId());
                            }
77                      }
                        catch (CiliaIllegalParameterException e) {
79                      }
                }
```

Code source 7.8 – Phase 3 (extrait).

3.4.4 Phase 4 : Contrôle

Nous avons à présent un gestionnaire autonomique capable, à la volée, de détecter l'arrivée du capteur, d'installer le code d'intégration manquant et modifier la topologie de la chaîne de médiation, selon une règle définie par l'utilisateur. Pour illustrer ce cas d'usage, nous avons avec iCasa Simulator instancié de façon dynamique un capteur photomètre dans la cuisine, voir figure 7.11 « *Etat initial, cas d'usage amélioration de l'application* ».

FIGURE 7.11 – Etat initial, cas d'usage amélioration de l'application.

Comme nous l'avons précédemment dit, le contrôle est visuel, l'outil de visualisation utilise les même API (via un *proxy* REST) qu'un service en local à la plate-forme OSGi. Ce contrôle peut s'effectuer de façon programmatique et sans difficulté, notre choix pour la validation a

été de fournir un contrôle visuel. La figure figure 7.12 « *Contrôle de la modification* » est la copie d'écran obtenue après les modifications effectuées sur la chaîne de médiation.

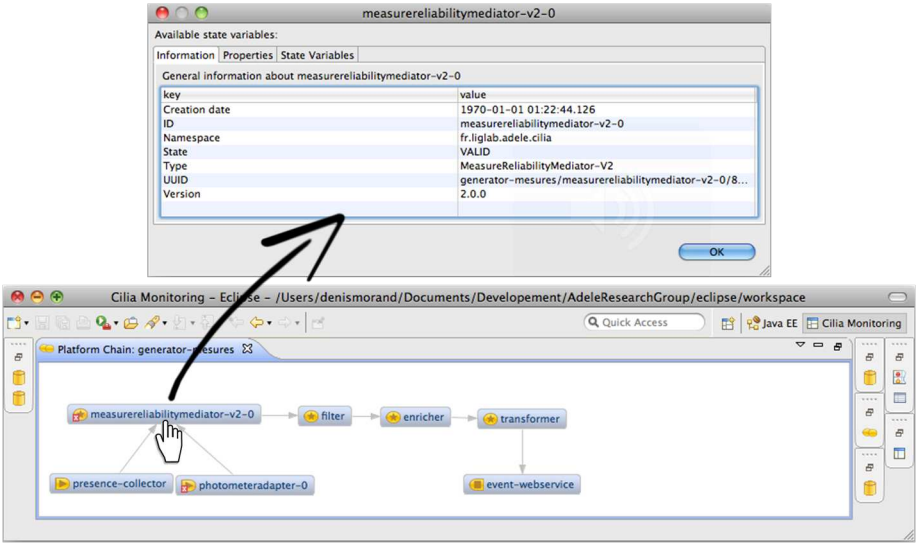


FIGURE 7.12 – Contrôle de la modification.

Le gestionnaire autonomique composé des services iPOJO QoSPoliciesManager et DevicesPoliciesManager a été écrit en ***396 lignes de codes Java*** *mesurés par l'outil SONAR (`http://www.sonarsource.org`).*

4 Evaluation

Dans la section précédente, nous avons montré la relative facilité de développement d'applications auto-administrées avec le *framework* Cilia. Il est ainsi possible de modifier le comportement d'une application de médiation durant son exécution, en fonction d'événements externes et/ou interne au *framework* et à l'application. Cette facilité est apportée par :

- **les *touchpoints*** fournis par le *framework* Cilia (seuls points de contact avec le framework) ;
- **le *monitoring* dynamique** (configuration dynamique des variables d'états) ;
- **l'adaptation dynamique** (remplacement de médiateurs, adaptateurs, etc.) ;
- **la base de connaissances causale** qui fournit une représentation courante de l'application de médiation en exécution. Elle collecte les informations depuis les variables d'états (mesures historisées) et émet des événements pour signaler toutes évolutions importantes.

Nous allons présenter dans cette section notre approche pour évaluer l'implantation.

4.1 Description de l'évaluation

Dans cette partie, nous décrivons l'évaluation de notre implantation ainsi que les résultats. Ces éléments sont extraits de notre article [Garcia et al., 2011]. Nous avions pour objectif de

quantifier l'impact du monitoring et de la modification dynamique d'une application Cilia pendant son exécution. Nous avons effectué deux expérimentations. La première pour mesurer les impacts du monitoring et la seconde pour mesurer les impacts de la reconfiguration.

4.1.1 Banc de tests

Une application Cilia est une chaîne de médiation constituée de médiateurs connectés entre eux. Chaque médiateur réalise une unique opération de médiation qui, à partir du message reçu, le transforme et sélectionne le médiateur suivant. Sur cette propriété fondamentale du *framework* Cilia, nous avons considéré comme important le temps de latence des messages dans la chaîne de médiation. Cela correspond au temps qui s'écoule entre l'arrivée d'un message dans le premier médiateur et la sortie de ce message du dernier médiateur, autrement dit, le temps de transit du message dans la chaîne. Pour quantifier le coût en performances, deux séries de mesures sont nécessaires :

- une première qui permet d'obtenir des mesures dites de référence. Les fonctions de *monitoring* ne sont pas activées et aucune opération relative aux modifications topologiques de chaînes n'est exécutée ;
- la seconde mesure fournit le temps de latence, lorsque toutes les variables d'états de tous les médiateurs sont activées.

Nous avons retenu trois facteurs influençant le temps de latence :

1. **le nombre de médiateurs** (tâches de médiation) ;
2. **le temps de traitement** de chaque tâche de médiation ;
3. **l'*overhead* du *framework*** qui caractérise le temps de transfert du message du médiateur à son suivant.

L'application de référence voir figure 7.13 « *Application de mesure du temps de latence* » est constituée :

- d'un injecteur de messages qui, toutes les 100ms, injecte un message dans l'adaptateur d'entrée de manière asynchrone ;
- d'un objet message constitué de trois champs : un champ `uuid` contenant un numéro unique qui sert d'identifiant au message, un champ `Ti` qui contient le nombre de nanosecondes du système Java pris au moment où le message va être envoyé au premier médiateur et, finalement, d'un champ `Tf` qui est aussi le nombre de nanosecondes du système Java mais enregistré au moment de la sortie de la chaîne ;
- d'un adaptateur d'entrée qui reçoit le message de l'injecteur et génère un `uuid` (un compteur avec un pas de 1) puis le sauvegarde dans le message et, finalement, initialise la valeur du compteur `Ti` (origine de la mesure) avec la valeur fournie par Java du nombre de nanosecondes ;
- de médiateurs tous identiques constitués d'un *immediate-scheduler*, d'un *simple-processor* (exécution vide) et d'un *multicast-dispatcher* ;
- d'un adaptateur de sortie qui, dans l'ordre, enregistre dans le champ `Tf` le nombre de nanosecondes fourni par le *runtime* Java et stocke le message dans un fichier selon un format permettant un traitement automatique ultérieur (fichier de logs dont chaque ligne contient les trois champs `uuid`, `Ti`, `Tf`).

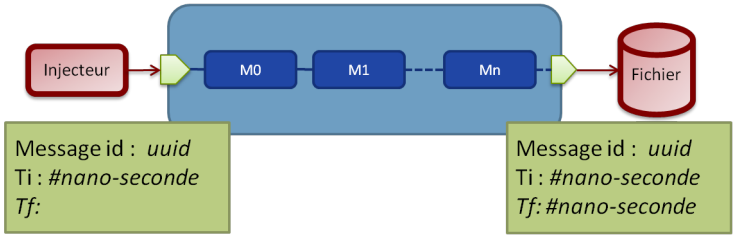


FIGURE 7.13 – Application de mesure du temps de latence.

La chaîne de médiation de référence est constituée de cinquante médiateurs. Le temps de transit du message dans la chaîne est donné pour chaque messages par `Tf-Ti`.

4.2 Mesure de l'impact du *monitoring*

Nous avons effectué deux mesures sur le banc de tests : une première série de mesures où le *monitoring* des cinquante médiateurs n'est pas activé ; puis, une seconde série où les variables d'état des médiateurs sont toutes activées. L'axe des ordonnées de la figure 7.14 « *Chaîne monitorée versus chaîne non monitorée* » représente le temps de latence (`Tf-Ti`) pour chaque message alors que l'axe des abscisses représente le nombre de messages traités par la chaîne de médiation.

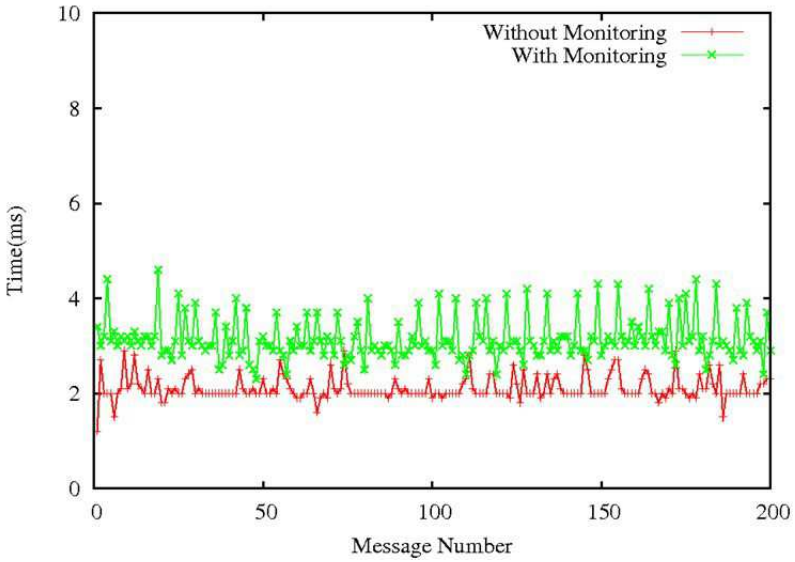


FIGURE 7.14 – Chaîne monitorée versus chaîne non monitorée.

Nous pouvons constater deux points importants :

- la courbe *Without Monitoring* n'augmente pas dans le temps alors que la courbe *With Monitoring* montre un coût de traitement supérieur mais sans divergence. Il n'y a pas de dégradation de performance dans le temps lorsque le *monitoring* est activé. Ce test a été exécuté sans interruption pendant plusieurs heures, confirmant ainsi la stabilité de l'exécution du *framework* Cilia.
- la moyenne de la latence sans *monitoring* est de 2,108ms alors que celle avec le *monitoring* est de 3,142. Soit environ une augmentation d'une milliseconde pour cinquante médiateurs. On peut raisonnablement dire que le coût du *monitoring* pour un médiateur est de 0,02ms. Cette mesure par médiateur (déduite) met en évidence le traitement

asynchrone de la fonction *monitoring*. C'est en fait le temps de traitement des données résultats des fonctions *monitoring* et non pas de l'élaboration de la mesure dynamique. En effet, les fonctions de *monitoring* sont exécutées dans des *Threads* Java de faible priorité (de manière asynchrone et lorsque la charge du système le permet).

4.3 Mesure de l'impact de la reconfiguration

En utilisant le même banc de tests, c'est-à-dire la même chaîne de médiation constituée de cinquante médiateurs et d'un producteur de messages toutes les 100ms, nous avons mesuré l'impact de la reconfiguration à chaud de l'application. Le point significatif est que chaque médiateur n'a qu'une seule liaison amont et aval (un seul port d'entrée et un seul port de sortie). Le temps de reconfiguration augmente avec le nombre de liaisons entre médiateurs. Notons toutefois que l'augmentation n'est pas significative tant que l'on reste sur un nombre raisonnable de liaisons (de l'ordre de la dizaine). C'est pour cela que nous n'avons effectué aucune mesure en relation avec l'augmentation du nombre de liaisons entre médiateurs. Le table 7.1 « *Temps de reconfiguration par opération* » présente les opérations de modifications de chaînes de médiation les plus couramment utilisées.

Opération	Temps (ms)
Création d'un médiateur	130
Création des liaisons	60
Destruction des liaisons	50
Destruction d'un médiateur	115
Remplacement d'un médiateur	200
Reconfiguration d'un médiateur	7

Table 7.1 – Temps de reconfiguration par opération.

Les temps mesurés sont relativement importants par rapport aux mesures précédentes. Notre objectif de la ***modification à chaud*** *est* ***la non perte de messages en cours de traitement****, ce qui nous autorise à accepter des temps de reconfiguration unitaire assez importants par rapport à la dizaine de millisecondes.*

5 Synthèse

Dans ce chapitre nous avons proposé une validation des évolutions autonomiques apportées à Cilia. Pour cela, nous avons emprunté le cas d'utilisation d'*Actimétrie* au projet FUI MEDICAL. Nous avons également défini deux scenarios permettant de mettre en évidence des types d'évolutions fréquemment rencontrées.

Pour ces deux scenarios, nous avons montré la faisabilité et présenté le code mis en œuvre. Dans les deux cas, l'utilisation de la base de connaissances et des capacités de *monitoring* et d'adaptation ont permis de réaliser les adaptations autonomiques attendues. En terme de code, les gestionnaires autonomiques sont :

– **courts** : par exemple, le second gestionnaire représente moins de 400 lignes de code Java ;

– **simples** : le code ne nécessite pas de connaître les détails d'implantation de Cilia. Il n'utilise que les API de haut niveau fournies par Cilia. Ainsi, les gestionnaires autonomiques ne manipulent que des concepts métier propres à la médiation.

Nous avons terminé ce chapitre avec des mesures de performances en s'attachant à mesurer les effets du *monitoring* et de l'adaptation dynamique. Nous avons vu que l'impact du *monitoring* est négligeable et constant dans le temps.

On peut imaginer qu'un développement ad hoc de fonctions de *monitoring* et d'adaptation, c'est-à-dire propres à une chaîne de médiation donnée, serait plus efficace en terme de temps d'exécution. Mais, en revanche, le développement serait plus long, sujet à plus d'erreurs et nécessiterait une **connaissance profonde du fonctionnement interne** du *framework* Cilia. En outre, le code dépasserait très largement les 400 lignes de code Java.

8

Conclusion et perspectives

Dans les chapitres précédents, nous avons présenté *Cilia autonomique*, un ESB *open source* permettant de définir des solutions d'intégration adaptables à l'exécution. Cet ESB repose sur la mise en place d'une architecture possédant de multiples boucles de contrôle, inspirée des travaux en informatique autonomique.

Nous sommes conscients qu'il reste des questions à résoudre. Dans ce chapitre, nous détaillons des perspectives de recherche et des apports de fonctionnalités à l'ESB Cilia. Plus spécifiquement, nous avons identifié quatre perspectives à court et moyen fermes. Elles concernent la définition plus fine du modèle de l'exécution, la fédération d'ESB Cilia autonomique, le couplage du framework Cilia avec un répertoire dépôt et une exposition de la base de connaissances sous forme de service.

1 Conclusion

1.1 Contexte

Nous nous sommes intéressés, dans cette thèse, à l'approche orientée service et à la problématique d'intégration. L'orientation service a fortement impacté les développements informatiques ces dernières années en apportant notamment plus de flexibilité à l'exécution. Un client utilisateur de service n'a pas à prendre en charge le cycle de vie du service, ni son environnement d'exécution ; il doit simplement l'utiliser lorsqu'il est disponible dès lors qu'il satisfait les propriétés requises. Nous avons également étudié en détail les ESB (*Enterprise Service Bus*) qui permettent l'intégration de données et de services. Les ESB sont de nos jours largement utilisés dans l'industrie. Ils répondent à une forte demande du marché, principalement sur les deux points suivants :

1. ils forment une infrastructure d'intégration beaucoup moins onéreuse que les EAI (*Enterprise Application Integration*) ;
2. ils apportent une facilité d'évolution plus forte que les EAI.

Nous avons également montré que, dans de nombreuses situations, les ESB restent difficiles à administrer. Cela est dû à la complexité même des domaines d'application et au fait que, dans certains cas, les administrateurs ne sont pas assez formés ou tout simplement absents. Nous avons alors étudié la notion d'informatique autonomique qui vise à faciliter l'administration de systèmes informatiques et à décroître le coût associé. Précisément, l'informatique autonomique prend en charge la gestion des applications pendant leur exécution pour aider les administrateurs dans les phases de déploiement, d'exploitation et de maintenance. Les bénéfices de l'informatique autonomique sont multiples. Ils incluent notamment l'optimisation des ressources matérielles et logicielles et la diminution des coûts des frais d'exploitation de maintenance des systèmes logiciels. La solution technique proposée par l'informatique autonomique consiste à administrer le système par l'intermédiaire d'une boucle externe qui, à partir des observations de l'état du système, décide et planifie des adaptations. Les objectifs à atteindre ou à maintenir par cette boucle de contrôle sont définis par un administrateur (et/ou des experts du domaine) de façon dynamique. La boucle de décision d'administration du système s'appuie sur de nombreux travaux et théories (la biologie, les probabilités, l'intelligence artificielle, les systèmes multi-agents, l'interface homme machine...).

Nous avons souhaité, tout au long de cette thèse, appliquer la philosophie de l'informatique autonomique à l'ESB *open source* Cilia. Nous allons, dans cette section, commencer par rappeler certaines notions concernant Cilia ainsi que les exigences de ce travail doctoral. Nous continuons par un rappel de nos contributions. D'une manière générale, cette section résumera les différents points abordés tout au long de cette thèse.

1.2 Besoins

L'ESB Cilia est un *framework* de médiation fondé sur un DSL pour la partie développement et sur un modèle à composant orienté service (iPOJO) pour la partie exécution. Le langage de spécification repose entièrement sur quatre entités spécifiques au métier de l'intégration : l'adaptateur, le médiateur, le connecteur et la chaîne de médiation. Ces entités sont facilement appréhendées par les spécialistes de l'intégration et permettent de mettre en place les patrons d'intégration les plus connus, les EIP [1]. La machine d'exécution est modulaire, dynamique et

1. Acronyme de *Enterprise Application Integration*.

relativement légère au niveau de son empreinte mémoire. Cilia se différencie des autres ESB par sa capacité à pouvoir être embarqué dans des dispositifs à fortes contraintes de ressources (mémoire, puissance CPU). Cette caractéristique doit être conservée dans la définition et l'implantation des évolutions de l'ESB Cilia.

L'embarquabilité de Cilia a amené des entreprises comme Orange à l'utiliser en informatique pervasive pour l'intégration de capteurs/actionneurs, présentés sous forme de services. Ces dispositifs, incluant par exemple des RTU [2], sont très hétérogènes au niveau protocole de communication et modèle de données. Nous retrouvons par exemple DPWS [3] pour les services (les photocopieuses par exemple), le standard KNX [4] pour la domotique (des détecteurs de luminosités par exemple), Modbus pour les dispositifs industriels (des dispositifs de mesure de consommation électrique). Bien évidemment tous ces équipements communicants sont dynamiques.

Dans ce domaine, des *frameworks* comme Cilia doivent faire face à des évolutions permanentes, notamment dans les domaines des bâtiments intelligents ou des usines intelligentes par exemple. Ces évolutions peuvent provenir d'une nouvelle législation nécessitant l'ajout d'un dispositif et donc de l'intégration de nouvelles données, l'apparition de nouveaux équipements plus performants ou moins chers, des évolutions fonctionnelles ou encore des corrections liées à des dysfonctionnements. La reconfiguration des applications d'intégrations de données et services est ainsi un point important et récurrent.

Même si Cilia est remarquable pas son dynamisme, son administration ne pouvait être faite par des non experts. Pour être applicable à des environnements pervasifs, l'administration de Cilia devait devenir simple et accessible à tous. Le but de cette thèse était de répondre à ce problème grâce à l'informatique autonomique et, en ce sens, de rendre Cilia administrable via des buts de haut niveau. Plus précisément, le but est de fournir les API et les mécanismes sous-jacents permettant de rendre autogérée chaque application développée avec Cilia. Notons que cet objectif est ambitieux : fournir des interfaces et une infrastructure permettant de l'écriture de gestionnaire autonomique de façon simple est un réel défi.

1.3 Contribution

Pour atteindre nos objectifs, nous avons proposé une architecture fondée sur plusieurs boucles de contrôle utilisant différents formalismes et différents mécanismes.

Nous avons défini deux types de boucles de contrôle. Chaque boucle de contrôle gère un scope et un aspect bien précis et il n'y a pas d'échange d'information entre les boucles. Le premier type de boucle autorise l'adaptation autonomique des adaptateurs ; le second permet l'adaptation autonomique des chaînes de médiation dans leur ensemble. L'architecture mise en place est fondée sur un protocole de type *meta-object protocol* qui nous a permis d'implanter la surveillance et l'adaptation dynamiques. Précisément, notre proposition offre les caractéristiques uniques suivantes :

- **un monitoring dynamique** : permettant à tout moment de surveiller à la demande de nombreux aspects liés à l'exécution et, ainsi, de quantifier le niveau de performance d'une chaîne de médiation à l'aide de variables d'état. L'administrateur peut également définir des seuils sur les variables d'état et définir des plages numériques caractérisant

2. Acronyme de *Remote Terminal Unit*.
3. Acronyme de *Device Profile for Web Service*.
4. KNX est à la fois un bus de terrain et un protocole de communication pour le bâtiment, voir `http://www.knx.org/`

un fonctionnement nominal. Le monitoring dynamique n'impacte que faiblement les performances générales du système.

- **une adaptation dynamique** : permettant la modification à chaud des constituants d'une chaîne de médiation ou l'ajout de nouveaux constituants. Les modifications sont traitées de manière à assurer la non perte des messages en cours de traitements dans la chaîne de médiation. Cette fonctionnalité utilise un mécanisme de quiescence que nous avons conçu et implanté.
- **une adaptation autonomique des adaptateurs** : permettant de sélectionner, ordonner, limiter et même changer les ressources utilisées par les adaptateurs. Les décisions prises par cette boucle autonomique sont fonction des objectifs de haut niveau définis par l'administrateur.
- **une base de connaissances** : permettant d'abstraire les capteurs et les effecteurs des gestionnaires autonomiques. La base de connaissances contient les données issues du *monitoring* dynamique et offre ainsi une modélisation des chaînes de médiation en exécution. Elle permet la configuration de la profondeur des enregistrements ainsi que l'historisation des modifications. La reconstruction des modifications et du comportement dynamique du passé est une tâche qui devient aisée pour un développeur de gestionnaire autonomique. Les événements générés par la base de connaissances sont configurables, ils permettent de construire des gestionnaires autonomiques en mode *push* ou *pull*. Finalement, cette base de connaissances est un modèle causal. La définition et l'implantation de la base de connaissances permettent de réaliser les opérations d'introspection et d'adaptation avec un faible temps de latence.

A partir de cette nouvelle version de Cilia, les développeurs peuvent réaliser des applications de médiation autonomiques. Ils peuvent aussi n'utiliser que les propriétés d'introspection et des mécanismes de la base de connaissances pour surveiller par exemple la qualité du service rendu par la chaîne de médiation. Ni la connaissance des détails d'implantation du *framework* Cilia, ni la modification du code métier des adaptateurs et médiateurs ne sont nécessaires pour implanter des applications de médiations autonomiques (*i.e.* avec une boucle de contrôle d'adaptation).

2 Perspectives

Nous considérons que ce travail peut être enrichi suivant plusieurs axes de recherches et un axe de travail. Nous présentons ici ces axes que nous considérons comme intéressant et important.

2.1 Extension des éléments administrés

La figure 8.1 présente un système constitué d'un gestionnaire autonomique et de l'élément qu'il administre. Le système contient des éléments qui ne sont pas administrés par le gestionnaire autonomique. La tendance naturelle et normale en informatique autonomique est d'augmenter la surface de l'élément administré au détriment de celle de l'élément non administré par le gestionnaire autonomique.

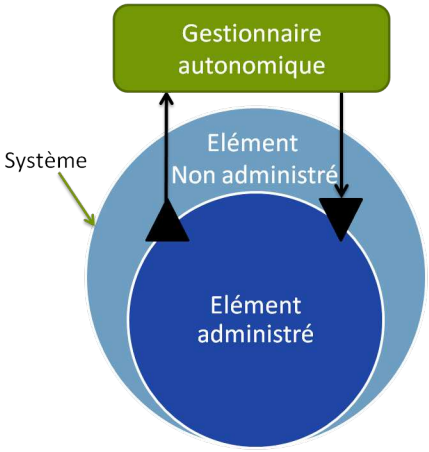


FIGURE 8.1 – Système autonomique.

Dans le cas de Cilia autonomique, il serait intéressant d'augmenter le nombre de ressources considérées pour une adaptation et, surtout, de les considérer au niveau de la boucle de contrôle globale. Une faiblesse actuelle est que le *monitoring* du *framework* Cilia ne remonte pas d'information sur les dispositifs externes notamment leurs qualités de service, leur état, et leurs propriétés.

Actuellement, les ressources externes sont en effet traitées localement à l'aide du *framework* RoSe au niveau des adaptateurs autonomiques. RoSe permet de mettre en œuvre des boucles autonomiques réactives au contexte externe. La solution technique du *framework* RoSe repose sur l'importation et l'exportation des ressources sous la forme de *proxies* métier. Le cycle de vie de ces *proxies* est géré automatiquement par le *framework* RoSe.

Un premier axe de recherche serait de définir la notion de dispositif externe pour une application de médiation Cilia. Actuellement la plate-forme OSGi a la connaissance de la présence ou non d'un dispositif. Cette connaissance est déduite par la présence ou non du *proxy* métier (réifié par le *framework* RoSe). Etendre le modèle de spécification de Cilia est un axe que nous considérons important. Le modèle en exécution nécessitera bien évidemment une extension lui aussi. Ce travail permettra de pouvoir élargir la connaissance des boucles autonomiques des adaptateurs. La base de connaissances construite par le *framework* Cilia sera augmentée avec des informations judicieuses sur les dispositifs utilisés par les chaînes de médiation. Les boucles de contrôle au-dessus de cette base de connaissances seront implantées plus aisément qu'actuellement.

2.2 Auto-gestion de fédération de *framework* Cilia

Une des caractéristiques des ESB est de pouvoir être fédéré, c'est-à-dire exécuté de manière individuelle sur plusieurs machines (voir chapitre 3 « *Intégration des services* », section 2.4 « *Patrons de fédération* », page 77). Nous proposons comme perspective de recherche, la fédération des ESB autonomiques. Cette tâche est intéressante mais certainement très ambitieuse. Elle soulève des problèmes importants tels que :

- la communication entre gestionnaires autonomiques ;
- l'alignement des sémantiques utilisées par les gestionnaires autonomiques ;
- la synchronisation des connaissances évolutives des gestionnaires autonomiques ;
- l'émergence d'un fonctionnement cohérent et contrôlé ;

– l'atteinte de propriétés non-fonctionnelles au niveau de la fédération, comme décrit dans [Cheng et al., 2008] ;
– etc.

Une voie intéressante à explorer pour ces types de problématique est celle des Systèmes Multi-Agents (SMA[5]). En effet, le domaine des SMA s'intéresse depuis longtemps à l'intégration et l'interopérabilité d'applications hétérogènes. L'approche de résolution consiste à modéliser et à résoudre les problèmes par coopérations entre agents. Cette approche est difficile tant au niveau de la modélisation des agents que de la communication entre les agents. Cependant, il existe des plates-formes de développement à agents qui seraient certainement utilisables. Bien évidemment, chaque plate-forme est dédiée à une architecture particulière. Citons JADE[6] un *framework* qui permet la simplification des systèmes multi-agents. Il implante les standard FIPA[7]. Les spécifications FIPA pour la communication au travers de ACL[8] répondent à une famille de besoins pour les interactions entre agents. Elles pourraient être une voie d'étude pour ainsi apporter à des éléments autonomiques des caractéristiques de coopération issues des agents.

La communication n'est cependant pas suffisante. Les éléments autonomiques devront être enrichis par des mécanismes d'échanges des connaissances qui pourraient aussi être inspirés par des techniques et des standard issus des SMA.

2.3 Couplage avec un répertoire de dépôt

Nous avons dans le chapitre validation mis en évidence le besoin de déploiement de code à chaud. Ce code a deux finalités possibles : remplacer du code existant (*patch* de correction de dysfonctionnement par exemple), ou ajouter des fonctionnalités à l'application en exécution.

Dans le cadre de l'ESB Cilia, l'auto-administration prend en compte l'évolution du contexte externe. Nous avons mis en évidence dans notre validation, la nécessité d'ajouter la *factory* d'un *proxy* métier dans la passerelle OSGi. Ce *proxy* est en charge de la communication avec le nouveau capteur (dans notre validation, le capteur de luminosité a été ajouté et son *proxy* de communication instancié). Nous avons également été amené à remplacer le code d'un médiateur par un autre (dans notre validation le capteur de remplacement traite en plus des données issues du capteur de luminosité). Une gestion dynamique des composants de médiation (adaptateurs, médiateurs) mais aussi des *proxies* (de communication avec les dispositifs) consiste au minium à vérifier que le code n'est pas déjà déployé dans la passerelle, récupérer le code manquant (le ou les *bundle(s)* OSGi), déployer le ou les *bundles* OSGi dans la passerelle, vérifier que l'ensemble des *bundles* est correctement démarré.

Nous avons intégré cette gestion du code manquant au niveau du gestionnaire autonomique. Déployer automatiquement du code est une caractéristique récurrente des systèmes autogérés. Nous pensons qu'un axe de travail pourrait être de factoriser cette gestion dans le framework Cilia.

Une solution technique serait d'utiliser le service OBR[9] de OSGi. C'est le moteur permettant d'installer à partir d'un *repository* un *bundle* OSGi et toutes ses dépendances techniques.

5. Acronyme de *System Multi-Agent*.
6. Acronyme de *Java Agent DEvelopment Framework* : `http://jade.tilab.com/`
7. Acronyme de *Foundation for Intelligent Physical Agent* : `http://www.fipa.org/`
8. Acronyme de *Agent Communication Language*.
9. Acronyme de *OSGi Bundle Repository* :
`http://felix.apache.org/site/apache-felix-osgi-bundle-repository.html`

Les mécanismes offerts par l'OBR sont de très bas niveau, et difficilement exploitables pour un développeur d'applications de médiation. Le framework Cilia pourrait offrir des API de plus haut niveau et plus spécialisées : dans l'expression des besoins des gestionnaires autonomiques, dans la création de la passerelle, dans la désinstallation des *bundles* inutilisés. Une étude des protocoles de communication utilisés dans le Cloud permettrait d'offrir (sous condition de faisabilité) des boîtes de dépôts dans le Cloud. La prochaine étape pourrait consister à définir et implanter une *Application Store like*, c'est-à-dire une boîte de dépôt dans le Cloud.

2.4 La base de connaissances exposée sous la forme de service

Notre proposition s'appuie sur une base de connaissances causale. Elle permet la simplification de l'écriture des gestionnaires autonomiques. Techniquement, cette base de connaissances est disponible sous forme d'API Java pour la plate-forme OSGi et sous forme d'URL REST/JSON pour un accès distant.

Nous constatons qu'il existe de nos jours une multitude de plates-formes pour le développement d'applications orientées services. En *open source*, deux références bien connues et largement utilisées sont Eclipse SOA Platform Projet [10] et Apache CXF [11]. Le *framework Eclipse SOA platform* réunit tous les acteurs d'une architecture à service, des architectes aux développeurs. Il permet la définition, la configuration, le déploiement, la surveillance et, finalement, l'administration d'applications orientées services. Le projet Apache CXF, quant-à-lui, permet de développer des services avec une grande variété de protocoles et de couches transports.

Ces *frameworks* permettent de ne pas écrire entièrement des clients et des fournisseurs de services. Nous constatons aussi que, dans le domaine de l'informatique autonomique, il n'est pas rare de trouver des éléments administrés exposés sous la forme de services logiciels.

Ces deux constats nous amènent à penser qu'il serait intéressant de fournir la base de connaissances sous la forme d'un ou plusieurs services logiciels. Cette implantation autoriserait l'utilisation des techniques issues des services Web sémantique [McIlraith et al., 2001], permettant ainsi d'aller encore plus loin dans les tâches d'administration automatisées.

10. `http://www.eclipse.org/soa/`
11. `http://cxf.apache.org/`

BIBLIOGRAPHIE

[Aiber et al., 2003] Aiber, S., Etzion, O., and Wasserkrug, S. (2003). The utilization of AI techniques in the autonomic monitoring and optimization of business objectives. In *IJCAI workshop on AI and autonomic computing : developing a research agenda for self-managing computer systems, Acapulco, Mexico, 10th August*.

[Aiello et al., 2005] Aiello, M., Frankova, G., and Malfatti, D. (2005). What's in an Agreement ? An Analysis and an Extension of WS-Agreement. In *Third International Conference on Service-Oriented Computing (ICSOC 2005)*, volume 3826 of *Lecture Notes in Computer Science*, pages 424–436. Springer.

[Apache Software Fundation, 2008] Apache Software Fundation (2008). The Apache Tuscany Project. `http://incubator.apache.org/tuscany/`.

[Apache Software Fundation, 2013] Apache Software Fundation (2013). Web Services - Axis. `http://axis.apache.org/axis/java/index.html`.

[Arsanjani, 2004] Arsanjani, A. (2004). Service-oriented Modeling and Architecture. `http://www.ibm.com/developerworks/webservices/library/ws-soa-design1/`.

[Ashby and Stein, 1954] Ashby, R. and Stein, P. R. (1954). Design for a brain. *Physics Today*, 7(4) :24–26.

[Austin et al., 2004] Austin, D., Barbir, A., Peters, E., and Ross-Talbot, S. (2004). Web Services Choreography Requirements 1.0. `http://www.w3.org/TR/2003/WD-ws-chor-reqs-20030812/`.

[Bajaj et al., 2006] Bajaj, S., Box, D., Chappell, D., Curbera, F., Daniels, G., Hallam-Baker, P., Hondo, M., Kaler, C., Langworthy, D., Nadalin, A., et al. (2006). Web Services policy 1.2-framework (WS-policy). *W3C Member Submission*, 25 :12.

[Baldoni et al., 2003] Baldoni, R., Marchetti, C., and Verde, L. (2003). CORBA request portable interceptors : analysis and applications. *Concurrency and Computation : Practice and Experience*, 15(6) :551–579.

[Ballinger et al., 2001] Ballinger, K., Brittenham, P., Malhotra, A., Nagy, W. A., and Pharies, S. (2001). Web Services Inspection language (WS-Inspection) 1.0. *IBM, Microsoft*.

[Bardin, 2012] Bardin, J. M. (2012). *RoSe : Un framework pour la conception et l'exécution d'applications distribuées dynamiques et hétérogènes*. PhD thesis, Université Joseph Fourier.

[Berglund et al., 2007] Berglund, A., Boag, S., Chamberlin, D., Fernandez, M. F., Kay, M., Robie, J., and Siméon, J. (2007). Xml Path language (XPath) 2.0. *W3C recommendation*, 23.

[Beugnard et al., 1999] Beugnard, A., Jézéquel, J.-M., Plouzeau, N., and Watkins, D. (1999). Making Components Contract Aware. *Computer*, 32(7) :38–45.

[Box et al., 2004] Box, D., Christensen, E., Curbera, F., Ferguson, D., Frey, J., Hadley, M., Kaler, C., Langworthy, D., Leymann, F., Lovering, B., et al. (2004). Web Services Addressing (WS-Addressing).

[Bray et al., 1997] Bray, T., Paoli, J., Sperberg-McQueen, C. M., Maler, E., and Yergeau, F. (1997). Extensible markup language (XML). *World Wide Web Journal*, 2(4) :27–66.

[Bussmann, 1998] Bussmann, S. (1998). Agent-oriented programming of manufacturing control tasks. In *Multi Agent Systems, 1998. Proceedings. International Conference on*, pages 57–63. IEEE.

[Buyya et al., 2009] Buyya, R., Cortes, T., and Jin, H. (2009). A Case for Redundant Arrays of Inexpensive Disks (RAID).

[Campbell and Grady, 1999] Campbell, J. and Grady, H. (1999). Adaptable components. In *Proceedings of the 21st International Conference on Software Engineering*, ICSE '99, pages 685–686, New York, NY, USA. ACM.

[Cervantes, 2004] Cervantes, H. (2004). *Vers un modèle à composants orienté services pour supporter la disponibilité dynamique*. PhD thesis, Université Joseph Fourier.

[Cervantes and Hall, 2004] Cervantes, H. and Hall, R. S. (2004). Autonomous adaptation to dynamic availability using a service-oriented component model. In *Proceedings of the 26th International Conference on Software Engineering*, pages 614–623. IEEE Computer Society.

[Chamberlin, 2002] Chamberlin, D. (2002). XQuery : An XML Query language. *IBM systems journal*, 41(4) :597–615.

[Cheng et al., 2008] Cheng, Y., Leon-Garcia, A., and Foster, I. (2008). Toward an autonomic service management framework : A holistic vision of SOA, AON, and autonomic computing. *Communications Magazine, IEEE*, 46(5) :138–146.

[Chinnici et al., 2007] Chinnici, R., Moreau, J.-J., Ryman, A., and Weerawarana, S. (2007). Web Services Description Language (wsdl) version 2.0 part 1 : Core language. *W3C Recommendation*, 26.

[Chollet, 2009] Chollet, S. (2009). *Orchestration de services hétérogènes et sécurisés*. PhD thesis, Université Joseph Fourier.

[Escoffier, 2008] Escoffier, C. (2008). *iPOJO : Un modèle à composant à service flexible pour les systèmes dynamiques*. PhD thesis, Université Joseph Fourier.

[Escoffier et al., 2007] Escoffier, C., Hall, R., and Lalanda, P. (2007). iPOJO : an Extensible Service-Oriented Component Framework. In *Services Computing, 2007. SCC 2007. IEEE International Conference on*, pages 474–481.

[Friedland, 2005] Friedland, B. (2005). *Control system design : an introduction to state-space methods*. DoverPublications. com.

[Fusco et al., 2008] Fusco, A., Manzalini, A., Moiso, C., Blazquez, H., Solé-Pareta, J., and Spadaro, S. (2008). Autonomic wireless communications in digital cities : an experimental use case. *Proceedings of the Mobilware. Austria*.

[Ganek, 2003] Ganek, A. (2003). Autonomic computing : implementing the vision. In *Autonomic Computing Workshop. 2003. Proceedings of the*, pages 1–1. IEEE.

[Ganek, 2004] Ganek, A. (2004). Overview of Autonomic Computing : Origins, Evolution, Direction. *Autonomic Computing : Concepts, Infrastructure, and Applications., by Salim Hariri Manish Parashar*, pages 3–18.

[Ganek and Friedrich, 2006] Ganek, A. and Friedrich, R. (2006). The road ahead achieving wide-scale deployment of autonomic technologies. In *Proceedings of the 3rd IEEE International Conference on Autonomic Computing*.

[Ganek and Corbi, 2003] Ganek, A. G. and Corbi, T. A. (2003). The dawning of the autonomic computing era. *IBM Systems Journal*, 42(1) :5–18.

[Garcia et al., 2011] Garcia, I., Morand, D., Debbabi, B., Lalanda, P., and Bourret, P. (2011). A reflective framework for mediation applications. In *Adaptive and Reflective Middleware on Proceedings of the International Workshop*, pages 22–28. ACM.

[Garcia Garza, 2012] Garcia Garza, I. N. (2012). *Modèle de conception et d'exécution pour la médiation et l'intégration de services*. PhD thesis, Université Joseph Fourier.

[Garlan et al., 2001] Garlan, D., Schmerl, B., and Chang, J. (2001). Using gauges for architecture-based monitoring and adaptation.

[Gassmann and Enkel, 2004] Gassmann, O. and Enkel, E. (2004). Towards a theory of open innovation : three core process archetypes. In *R&D management conference*, pages 1–18.

[Gudgin et al., 2003] Gudgin, M., Hadley, M., Mendelsohn, N., Moreau, J.-J., Nielsen, H. F., Karmarkar, A., and Lafon, Y. (2003). SOAP Version 1.2. *W3C recommendation*, 24.

[Gudgin et al., 2006] Gudgin, M., Hadley, M., and Rogers, T. (2006). Web services addressing 1.0-core. *W3C, W3C Recommendation, May*.

[Guo, 2003] Guo, H. (2003). A bayesian approach for automatic algorithm selection. In *Proceedings of the International Joint Conference on Artificial Intelligence (IJCAI03), Workshop on AI and Autonomic Computing, Acapulco, Mexico*, pages 1–5. Citeseer.

[Hayes-Roth et al., 1995] Hayes-Roth, B., Pfleger, K., Lalanda, P., Morignot, P., and Balabanovic, M. (1995). A domain-specific software architecture for adaptive intelligent systems. *Software Engineering, IEEE Transactions on*, 21(4) :288–301.

[Hérault et al., 2005] Hérault, C., Thomas, G., and Lalanda, P. (2005). Mediation and enterprise service bus : A position paper. In *Proceedings of the First International Workshop on Mediation in Semantic Web Services (MEDIATE 2005)*, pages 67–80.

[Hohpe and Woolf, 2003] Hohpe, G. and Woolf, B. (2003). *Enterprise Integration Patterns : Designing, Building, and Deploying Messaging Solutions*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA.

[Horn, 2001] Horn, P. (2001). Autonomic computing : IBM's Perspective on the State of Information Technology.

[Huebscher and McCann, 2008] Huebscher, M. C. and McCann, J. A. (2008). A survey of autonomic computing–degrees, models, and applications. *ACM Computing Surveys (CSUR)*, 40(3) :7.

[IBM, 2006] IBM (2006). An architectural blueprint for autonomic computing. *Autonomic Computing - IBM White Paper*.

[IBM Server group, 2002] IBM Server group (2002). eLiza : Building an Intelligent Infrastucture for E-business – Technology for self-Managing Server Environment.

[Jackson, 2002] Jackson, M. (2002). Some basic tenets of description. *Software and Systems Modeling*, 1(1) :5–9.

[Jordan et al., 2007] Jordan, D., Evdemon, J., Alves, A., Arkin, A., Askary, S., Barreto, C., Bloch, B., Curbera, F., Ford, M., Goland, Y., et al. (2007). Web services business process execution language version 2.0.

[Kaiser et al., 2003] Kaiser, G., Parekh, J., Gross, P., and Valetto, G. (2003). Kinesthetics extreme : An external infrastructure for monitoring distributed legacy systems. In *Autonomic Computing Workshop. 2003. Proceedings of the*, pages 22–30. IEEE.

[Kavantzas et al., 2005] Kavantzas, N., Burdett, D., Ritzinger, G., Fletcher, T., Lafon, Y., and Barreto, C. (2005). Web services choreography description language version 1.0. *W3C candidate recommendation*, 9.

[Kay et al., 2007] Kay, M. et al. (2007). Xsl transformations (xslt) version 2.0. *W3C Recommendation*, 23.

[Keeney, 1993] Keeney, R. L. (1993). *Decisions with multiple objectives : preferences and value trade-offs*. Cambridge University Press.

[Kephart and Das, 2007] Kephart, J. and Das, R. (2007). Achieving self-management via utility functions. *Internet Computing, IEEE*, 11(1) :40–48.

[Kephart and Walsh, 2004] Kephart, J. and Walsh, W. (2004). An artificial intelligence perspective on autonomic computing policies. In *Policies for Distributed Systems and Networks, 2004. POLICY 2004. Proceedings. Fifth IEEE International Workshop on*, pages 3–12. IEEE.

[Kephart and Chess, 2003] Kephart, J. O. and Chess, D. M. (2003). The Vision of Autonomic Computing. *Computer*, 36(1) :41–50.

[Kon et al., 2002] Kon, F., Costa, F., Blair, G., and Campbell, R. H. (2002). The case for reflective middleware. *Communications of the ACM*, 45(6) :33–38.

[Krakowiak, 2009] Krakowiak, S. (2009). Middleware Architecture with Patterns and Frameworks. `http://sardes.inrialpes.fr/~krakowia/MW-Book/`.

[Kramer and Magee, 1990] Kramer, J. and Magee, J. (1990). The evolving philosophers problem : Dynamic change management. *Software Engineering, IEEE Transactions on*, 16(11) :1293–1306.

[Lalanda et al., 2013] Lalanda, P., McCann, J. A., and Diaconescu, A. (2013). *Autonomic Computing - Principles, Design and Implementation*. Springer–Verlag.

[Linner et al., 2007] Linner, D., Pfeffer, H., and Steglich, S. (2007). A genetic algorithm for the adaptation of service compositions. In *Bio-Inspired Models of Network, Information and Computing Systems, 2007. Bionetics 2007. 2nd*, pages 277–281. IEEE.

[Maes, 1987] Maes, P. (1987). Concepts and experiments in computational reflection. In *ACM Sigplan Notices*, volume 22, pages 147–155. ACM.

[McIlraith et al., 2001] McIlraith, S. A., Son, T. C., and Zeng, H. (2001). Semantic web services. *Intelligent Systems, IEEE*, 16(2) :46–53.

[Morand et al., 2011a] Morand, D., Garcia Garza, I. N., and Lalanda, P. (2011a). Autonomic Enterprise Service Bus. In *Proceedings of 6th IEEE International Workshop on Service Oriented Architectures in Converging Networked Environments*.

[Morand et al., 2011b] Morand, D., Garcia Garza, I. N., and Lalanda, P. (2011b). Towards Autonomic Enterprise Service Bus. In *Proceedings of the 1st Workshop on Middleware and Architectures for Autonomic and Sustainable Computing*, pages 19–23, New York, NY, USA. ACM.

[Morgan, 2012] Morgan, T. (2012). IBM Global Technology Outlook 2012. *Technology Innovation Exchange, IBM Warwick*.

[Morrison et al., 2004] Morrison, R., Kirby, G., Balasubramaniam, D., Mickan, K., Oquendo, F., Cîmpan, S., Warboys, B., Snowdon, B., and Greenwood, R. M. (2004). Support for evolving software architectures in the ArchWare ADL. In *Software Architecture, 2004. WICSA 2004. Proceedings. Fourth Working IEEE/IFIP Conference on*, pages 69–78. IEEE.

[Müller et al., 2006] Müller, H., O'Brien, L., Klein, M., and Wood, B. (2006). Autonomic computing. Technical report, DTIC Document.

[Muller et al., 2009] Muller, P., Fondement, F., and Baudry, B. (2009). Modeling modeling. *Model Driven Engineering Languages and Systems*, pages 2–16.

[Murch, 2004] Murch, R. (2004). *Autonomic computing*. IBM Press.

[OASIS, 2004] OASIS (2004). UDDI Version 3.0.2. `https://www.oasis-open.org/committees/uddi-spec/doc/spec/v3/uddi-v3.0.2-20041019.htm`.

[OASIS, 2006a] OASIS (2006a). Reference Model for Service Oriented Architecture. `http://docs.oasis-open.org/soa-rm/v1.0/soa-rm.pdf`.

[OASIS, 2006b] OASIS (2006b). Web Services Security : SOAP Message Security 1.1. `http://docs.oasis-open.org/wss/v1.1/wss-v1.1-spec-pr-SOAPMessageSecurity-01.pdf`.

[OASIS, 2007] OASIS (2007). Web Services Business Process Execution Language Version 2.0. `http://docs.oasis-open.org/wsbpel/2.0/OS/wsbpel-v2.0-OS.html`.

[OASIS, 2009] OASIS (2009). Devices Profile for Web Services Version 1.1. `http://docs.oasis-open.org/ws-dd/dpws/1.1/pr-01/wsdd-dpws-1.1-spec-pr-01.html`.

[OASIS, 2009] OASIS (2009). Web Services Dynamic Discovery (WS-Discovery) Version 1.1. `http://docs.oasis-open.org/ws-dd/discovery/1.1/os/wsdd-discovery-1.1-spec-os.html`.

[OASIS, 2013] OASIS (2013). Web Services Interoperability Organization. `http://www.ws-i.org/`.

[Object Management Group (OMG™), 2008] Object Management Group (OMG™) (2008). CORBA 3.1. `http://www.omg.org/spec/CORBA/3.1/`.

[Ogata, 1997] Ogata, K. (1997). *Modern control engineering*, volume 4. Prentice Hall, USA, ISBN : 0-13-261389-1.

[Oreizy, 1998] Oreizy, P. (1998). Issues in modeling and analyzing dynamic software architectures. In *Proceedings of the International Workshop on the Role of Software Architecture in Testing and Analysis*, pages 54–57.

[OSGi Alliance, 2007] OSGi Alliance (2007). OSGi™ service platform, core specification, release 4, version 4.1. `http://www.osgi.org/download/r4v41/r4.core.pdf`.

[Papazoglou, 2003] Papazoglou, M. P. (2003). Service-Oriented Computing : Concepts, Characteristics and Directions. In *Proceedings of the fourth International Conference on Web Information Systems Engineering*, pages 3–12, Los Alamitos, CA, USA.

[Papazoglou, 2008] Papazoglou, M. P. (2008). The Challenges of Service Evolution. In *Proceedings of the 20th international conference on Advanced Information Systems Engineering*, CAiSE '08, pages 1–15, Berlin, Heidelberg. Springer-Verlag.

[Papazoglou and Georgakopoulos, 2003] Papazoglou, M. P. and Georgakopoulos, D. (2003). Introduction : Service-oriented computing. *Communications of the ACM - Service-oriented computing*, 46(10) :24–28.

[Papazoglou and Heuvel, 2007] Papazoglou, M. P. and Heuvel, W.-J. (2007). Service oriented architectures : approaches, technologies and research issues. *The VLDB Journal*, 16(3) :389–415.

[Papazoglou et al., 2007] Papazoglou, M. P., Traverso, P., Dustdar, S., and Leymann, F. (2007). Service-Oriented Computing : State of the Art and Research Challenges. *IEEE Computer*, 40(11) :38–45.

[Parashar and Hariri, 2005] Parashar, M. and Hariri, S. (2005). Autonomic computing : An overview. *Unconventional Programming Paradigms*, pages 97–97.

[Peltz, 2003] Peltz, C. (2003). Web Services Orchestration and Choreography. *Computer*, 36(10) :46–52.

[Roblee et al., 2005] Roblee, C., Berk, V., and Cybenko, G. (2005). Implementing large-scale autonomic server monitoring using process query systems. In *Autonomic Computing, 2005. ICAC 2005. Proceedings. Second International Conference on*, pages 123–133. IEEE.

[Russell et al., 2003] Russell, D., Maglio, P., Dordick, R., and Neti, C. (2003). Dealing with ghosts : Managing the user experience of autonomic computing. *IBM Systems Journal*, 42(1) :177–188.

[Russell and Norvig, 2010] Russell, S. and Norvig, P. (2010). *Artificial intelligence : a modern approach*. Prentice hall Upper Saddle River, NJ.

[Schmerl and Garlan, 2002] Schmerl, B. and Garlan, D. (2002). Exploiting architectural design knowledge to support self-repairing systems. In *Proceedings of the 14th international conference on Software engineering and knowledge engineering*, pages 241–248. ACM.

[Shapiro, 1986] Shapiro, M. (1986). Structure and encapsulation in distributed systems : the proxy principle. In *icdcs*, pages 198–204.

[Shvaiko and Euzenat, 2013] Shvaiko, P. and Euzenat, J. (2013). Ontology Matching : State of the Art and Future Challenges. *IEEE Trans. Knowl. Data Eng.*, 25(1) :158–176.

[Siewiorek and Swarz, 1998] Siewiorek, D. and Swarz, R. (1998). *Reliable computer systems : design and evaluation*, volume 3. AK Peters MA.

[Soules et al., 2003] Soules, C., Appavoo, J., Hui, K., Wisniewski, R., Da Silva, D., Ganger, G., Krieger, O., Stumm, M., Auslander, M., Ostrowski, M., et al. (2003). System support for online reconfiguration. In *Proceedings of the 2003 USENIX Technical Conference*, pages 141–154.

[Sterritt, 2002] Sterritt, R. (2002). Towards autonomic computing : effective event management. In *Software Engineering Workshop, 2002. Proceedings. 27th Annual NASA Goddard/IEEE*, pages 40–47. IEEE.

[Sterritt, 2005] Sterritt, R. (2005). Autonomic computing. *Innovations in systems and software engineering*, 1(1) :79–88.

[Sterritt and Bantz, 2004] Sterritt, R. and Bantz, D. (2004). Pac-men : Personal autonomic computing monitoring environment. In *Database and Expert Systems Applications, 2004. Proceedings. 15th International Workshop on*, pages 737–741. IEEE.

[Sterritt and Bustard, 2003] Sterritt, R. and Bustard, D. (2003). Autonomic Computing-a means of achieving dependability ? In *Engineering of Computer-Based Systems, 2003. Proceedings. 10th IEEE International Conference and Workshop on the*, pages 247–251. IEEE.

[Sterritt et al., 2005] Sterritt, R., Smyth, B., and Bradley, M. (2005). Pact : personal autonomic computing tools. In *Engineering of Computer-Based Systems, 2005. ECBS'05. 12th IEEE International Conference and Workshops on the*, pages 519–527. IEEE.

[Suzuki and Suda, 2005] Suzuki, J. and Suda, T. (2005). A middleware platform for a biologically inspired network architecture supporting autonomous and adaptive applications. *Selected Areas in Communications, IEEE Journal on*, 23(2) :249–260.

[Szyperski, 2002] Szyperski, C. (2002). *Component Software : Beyond Object-Oriented Programming, 2nd edition*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA.

[Tesauro et al., 2004] Tesauro, G., Chess, D., Walsh, W., Das, R., Segal, A., Whalley, I., Kephart, J., and White, S. (2004). A multi-agent systems approach to autonomic computing. In *Proceedings of the Third International Joint Conference on Autonomous Agents and Multiagent Systems-Volume 1*, pages 464–471. IEEE Computer Society.

[Tianfield, 2003a] Tianfield, H. (2003a). Multi-agent autonomic architecture and its application in e-medicine. In *Intelligent Agent Technology, 2003. IAT 2003. IEEE/WIC International Conference on*, pages 601–604. IEEE.

[Tianfield, 2003b] Tianfield, H. (2003b). Multi-agent based autonomic architecture for network management. In *Industrial Informatics, 2003. INDIN 2003. Proceedings. IEEE International Conference on*, pages 462–469. IEEE.

[Touseau, 2010] Touseau, L. (2010). *Politique de liaison aux services intermittents dirigée par les accords de niveau de service*. PhD thesis, Université Joseph Fourier.

[UPnP Forum, 2008a] UPnP Forum (2008a). UPnP™ Forum. `http://www.upnp.org`.

[UPnP Forum, 2008b] UPnP Forum (2008b). UPnP™ Device Architecture 1.1. `http://upnp.org/specs/arch/UPnP-arch-DeviceArchitecture-v1.1.pdf`.

[Uschold, 1998] Uschold, M. (1998). Knowledge level modelling : concepts and terminology. *The knowledge engineering review*, 13(01) :5–29.

[Van Engelen and Gallivan, 2002] Van Engelen, R. A. and Gallivan, K. A. (2002). The gSOAP toolkit for web services and peer-to-peer computing networks. In *Cluster Computing and the Grid, 2002. 2nd IEEE/ACM International Symposium on*, pages 128–128. IEEE.

[Vandewoude et al., 2006] Vandewoude, Y., Ebraert, P., Berbers, Y., and D'Hondt, T. (2006). An alternative to quiescence : Tranquility. In *Software Maintenance, 2006. ICSM'06. 22nd IEEE International Conference on*, pages 73–82. IEEE.

[Vedamuthu et al., 2007] Vedamuthu, A. S., Orchard, D., Hirsch, F., Hondo, M., Yendluri, P., Boubez, T., and Yalcinalp, U. (2007). Web services policy 1.5-attachment. *W3C Recommendation*, 4.

[Vlissides et al., 1995] Vlissides, J., Helm, R., Johnson, R., and Gamma, E. (1995). Design patterns : Elements of reusable object-oriented software. *Reading : Addison-Wesley*, 49 :120.

[Walsh et al., 2004] Walsh, W., Tesauro, G., Kephart, J., and Das, R. (2004). Utility functions in autonomic systems. In *Autonomic Computing, 2004. Proceedings. International Conference on*, pages 70–77. IEEE.

[Weiser, 1991] Weiser, M. (1991). The computer for the 21st century. *Scientific american*, 265(3) :94–104.

[Wiederhold, 1992] Wiederhold, G. (1992). Mediators in the Architecture of Future Information Systems. *IEEE Computer*, 25(3) :38–49.

[Wiener, 1961] Wiener, N. (1961). *Cybernetics : or the Control and Communication in the Animal and the Machine*, volume 25. The MIT press.

[Wooldridge et al., 1995] Wooldridge, M., Jennings, N., et al. (1995). Intelligent agents : Theory and practice. *Knowledge engineering review*, 10(2) :115–152.

[Yang and Papazoglou, 2004] Yang, J. and Papazoglou, M. P. (2004). Service components for managing the life-cycle of service compositions. *Information Systems*, 29(2) :97–125.