HAL
open science

Modèles de conception et d'exécution pour la médiation et l'intégration de services

Issac Noe Garcia Garza Garcia Garza

To cite this version:

Issac Noe Garcia Garza Garcia Garza. Modèles de conception et d'exécution pour la médiation et l'intégration de services. Autre [cs.OH]. Université de Grenoble, 2012. Français. NNT : 2012GRENM026 . tel-00767953

HAL Id: tel-00767953

https://theses.hal.science/tel-00767953v1

Submitted on 20 Dec 2012

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

Issac Noé García Garza

Thèse dirigée par **Philippe LALANDA**
et codirigée par **Catherine HAMON**

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

Modèles de conception et d'exécution pour la médiation et l'intégration de services

Thèse soutenue publiquement le **18 juin 2012**,
devant le jury composé de :

M. Patrick REIGNIER
Professeur à ENSIMAG, Grenoble INP, Examinateur
Mme. Elisabetta DI NITTO
Associate Professor à Politecnico di Milano, Rapporteur
M. Olivier PERRIN
Professeur à Nancy 2 Université, Rapporteur
M. Julien PONGE
Maitre de Conférences à l'INSA de Lyon, Examinateur
M. Philippe LALANDA
Professeur à Université de Grenoble, Directeur de thèse
Mme. Catherine HAMON
Ingénieur de recherche à France Télécom R&D, Co-Directeur de thèse

Remerciements

Je tiens tout d'abord à remercier tous les membres du jury qui m'ont fait l'honneur de participer à ma soutenance de thèse. Je remercie particulièrement Elisabetta DI NITTO et Olivier PERRIN d'avoir accepté de rapporter mon travail de thèse. Je remercie également Patrick REIGNIER et Julien PONGE pour avoir examiné mes travaux.

J'adresse mes sincères remerciements à Catherine HAMON et à Philippe LALANDA pour sa confiance, ses conseils, son soutiens et son aide tout au long de ce travail. Je souhaite aussi remercier Jacky ESTUBLIER et Philippe LALANDA qui m'ont acueilli au sein de l'equipe ADELE. Je remercie aussi l'ensemble des membres de l'equipe ADELE, particulièrement ceux qui ont contribué à l'amélioration de mon travail (Bassem, Gabriel, Mehdi, PA, Etienne), ceux qui ont échangé des connaissances à la Kfet (German, Walter, Kiev, Didier, Jo, el borracho, Johann), Walter en particulier qui m'a encouragé à débuter cette aventure. Un merci à tous les autres que je n'ai pas directement travaillé : Diana, Idi, Ozan, Eric, Marc, Joao, Stephanie, Vincent, Yoan, Ada.

D'autre part les amis (la plupart latinos) qui ont partagé des bons moments tout au long de cette période.

Je remercie énormement ma famille, ma grande famille qui m'a toujours soutenu. Tout spécialement mes parents Consuelo et Fernando, et aussi ma sœur.

Mes sentiments plus chaleureux sont pour mon épouse Elizabeth qui m'a soutenu, accompagné et supporté ces derniers années. Merci à elle. Finalement un grand merci a notre petite fille Ximena, qui a apporté le bonneur durant les derniers jours de ce travail.

Résumé

Les systèmes logiciels s'orientent vers des environnements de plus en plus hétérogènes et dynamiques. Cette évolution est induite par différents facteurs : explosion des dispositifs embarqués avec de fortes capacités de calcul, adoption rapide des services distants fournis par des tiers, mobilité des usagers et évolution du contexte associé, etc. Ces facteurs ouvrent de grandes possibilités pour la construction de nouveaux services numériques dans des domaines aussi divers que la santé, le divertissement, la domotique, ou encore le transport.

Ces nouveaux domaines d'applications demandent la mise en œuvre des opérations d'intégration dans des contextes dynamiques et hétérogènes. Il est aujourd'hui admis que les approches à services facilitent l'intégration logicielle par la définition de protocoles standard de découverte et de liaison. La problématique d'intégration, au sens médiation, reste néanmoins entière.

Le problème principal abordé par cette thèse est l'intégration de services dans des contextes hétérogènes et dynamiques. Plus précisément, nous avons conçu un modèle à composant spécifique à l'intégration logicielle, nommé CILIA. Ce modèle repose sur des composants, appelés médiateurs, et sur un langage d'assemblage de ces médiateurs. CILIA reprend les grands principes du Génie Logiciel tels que l'abstraction, la séparation de préoccupations et la modularité, et s'appuie sur des patrons d'intégration bien connus (*Enterprise Integration Patterns*).

CILIA est implanté sous la forme d'un framework dynamique qui permet la mise à jour à l'exécution des solutions d'intégration. Ce framework CILIA est pleinement opérationnel et disponible en open source. Il est utilisé dans plusieurs projets collaboratifs.

Mots-clés : modèle à composant, intégration de services, applications dynamiques.

Abstract

Software systems are moving toward highly dynamic and heterogeneous environments. This dynamism is derived by several factors : the massive arrival of embedded devices with computing capabilities, the rapid adoption of newer distributed services provided by third parties, the user mobility and the constantly changing context, etc. These factors open up great opportunities for the construction of new and innovative services on several application domains, such as health-care systems, entertainment systems, home automation systems, transportation or traceability systems.

These new application areas require the implementation of integration operations in dynamic and heterogeneous environments. It is well known that service-oriented computing eases the implementation of integrating systems by defining standard protocols to perform the discovery and the binding. However, some interoperability concerns, such as mediating, have been still unresolved.

In this work we address the integration concern of service integration in dynamic and highly evolving environments. Specifically, we have developed an integration-specific component model called CILIA. This model is based on components, called mediators, and its assembly language. CILIA respects software engineering principles such as abstraction, separation of concerns, modularity, and anticipation of change and also it is influenced by the well-known enterprise integration patterns (EIP).

CILIA has been developed as a dynamic framework that allows us to perform dynamic modifications at run-time on the integrated solutions. This framework is entirely developed and operational. It is available as an open source project and has been used by several collaborative projects.

Keywords : component models, service integration, dynamic applications.

Liste de publications

Les travaux discutés dans cette thèse ont été présentés précédemment dans de conférences et *workshops* internationales.

- Issac Noé Garcia Garza, Denis Morand, Bassem Debbabi, Philippe Lalanda and Pierre Bourret. ***A Reflective Framework for Mediation Applications***. In Proceedings of the 10th International Middleware Workshop on Adaptive and Reflective Middleware, 2011-12-12, Lisbon, Portugal.
- Denis Morand, Issac Noé Garcia Garza and Philippe Lalanda. ***Autonomic Enterprise Service Bus.*** In Proceedings of the Service Oriented Architectures in Converging Networked Environments (SOCNE), 2011-09-05, Toulouse, France.
- Denis Morand, Issac Noé Garcia Garza and Philippe Lalanda. ***Towards Autonomic Enterprise Service Bus.*** In Proceedings of the 1st Workshop on Middleware and Architectures for Autonomic and Sustainable Computing, 2011-05-12, Paris, France.
- Issac Noé Garcia Garza, Gabriel Pedraza, Bassem Debbabi, Philippe Lalanda and Catherine Hamon. ***Towards a service mediation framework for dynamic applications.*** In Proceedings of the IEEE 2010 Asia-Pacific Services Computing Conference, 2010-12-06, Hangzhou, China.
- Gabriel Pedraza, Issac Noé Garcia Garza and Bassem Debbabi. ***An RFID architecture based on an event-oriented component model.*** In Proceedings of the Fourth ACM International Conference on Distributed Event-Based Systems, 2010-07-12, .

Table des matières

1 Introduction **19**

1 Intégration de services 20
2 Intégration de services dans les entreprises 23
2.1 Problématique 23
2.2 Enterprise Application Integration « EAI » 23
2.3 Enterprise Service Bus « ESB » 24
3 Objectifs de cette thèse 26
4 Structure du document 27

I Etat de l'art **29**

2 Approches à services **31**

1 Services Logiciel 32
1.1 Généralités 32
1.2 Definition 33
2 Architecture 35
2.1 Principaux acteurs 35
2.2 Architectures à services 37
2.3 Dynamisme 38
3 Composition de services 39
3.1 Problématique 39
3.2 Composition par procédé 40
3.3 Composition structurelle 41
4 Technologie à services 42

4.1 Services Web 42
4.2 OSGi 45
4.3 iPOJO 48
5 Integration de services 51
5.1 Introduction 51
5.2 Secure FOCAS 52
5.3 TACC Proxies 53
5.4 SCENE 54
5.5 Travaux avec ontologies 55
5.6 Synthèse 56
6 Synthèse 57

3 Architectures d'intégration 59

1 Intégration 60
1.1 Introduction 60
1.2 Opérations d'intégration 61
1.3 Besoin en adaptation 62
2 Architectures d'intégration 63
2.1 Principes 63
2.2 *Middlewares* de communication 65
2.3 Enterprise Application Integration 68
2.4 Enterprise Service Bus 70
2.5 Intégration de données 72
2.6 Synthèse 74
3 Patrons d'intégration 75
3.1 Réception de messages 76
3.2 Transformation de messages 77
3.3 Routage de messages 78
3.4 Gestion de systèmes 79
3.5 Patrons composés 80
3.6 Synthèse 81
4 Solutions existantes 82
4.1 Introduction 82
4.2 Spring Integration 83
4.3 Camel 85
4.4 Mule ESB 87
4.5 Comparaisons 89
5 Synthèse 92

II Proposition **93**

4 Proposition **95**

1 Problématique 96
2 Objectifs 98
3 Introduction sur les composants logiciels 99
3.1 Modèle à composant 99
3.2 Framework d'exécution 101
4 Notre approche : CILIA 102
4.1 Vision globale 102
4.2 Composants CILIA 103
4.3 Configuration d'architectures en CILIA 105
4.4 Framework d'exécution 106
4.5 Cycle de vie 108
5 Synthèse 109

5 Le modèle à composant CILIA **111**

1 Vers des composants d'intégration 112
2 Composants CILIA 113
2.1 Notion de médiateur 113
2.2 Rôle du *scheduler* 115
2.3 Rôle du *processor* 116
2.4 Rôle du *dispatcher* 117
3 Assemblage CILIA 118
3.1 Notion de connecteur 118
3.2 Connecteurs internes 119
3.3 Connecteurs externes 120
4 Configuration CILIA 121
5 Framework d'exécution CILIA 123
6 Cycle de vie CILIA 125
6.1 Conception 125
6.2 Déploiement 128
6.3 Administration 128
7 Synthèse 129

6 Réalisation **131**

1 Technologies de base 132
1.1 OSGi 132
1.2 iPOJO 132

2 Exemple de support 134
3 Langage de spécification 136
3.1 Définition d'un « médiateur » 136
3.2 Définition d'un « *scheduler* » 137
3.3 Définition d'un « *processor* » 138
3.4 Définition d'un « *dispatcher* » 139
3.5 Définition d'un connecteur externe 140
3.6 Définition d'un connecteur interne 143
3.7 Définition d'une chaîne de médiation 145
4 Framework de développement 147
4.1 Implantation des types de données 147
4.2 Implantation des « *schedulers* » 148
4.3 Implantation des « *processors* » 150
4.4 Implantation des « *dispatchers* » 150
4.5 Implantation des connecteurs externes 152
4.6 Implantation des connecteurs internes 156
5 Framework d'exécution 157
6 Cycle de vie 161
6.1 Outils à la conception 161
6.2 Outils de déploiement 163
6.3 Outils à l'exécution 163
7 Synthèse 165

7 Validation **167**

1 Introduction 168
2 Projet ASPIRE 169
2.1 Besoins du cas d'usage 169
2.2 Solution 170
2.3 Evaluation 173
3 Projet MEDICAL 174
3.1 Besoins du cas d'usage 175
3.2 Solution 176
3.3 Evaluation 178
4 Intégration de systèmes d'information patrimoniaux (Orange) 179
4.1 Besoins du cas d'usage 179
4.2 Solution 180
4.3 Evaluation 184
5 Evaluation de l'environnement d'exécution 185
5.1 Qualité de code produit 185
5.2 Mesures de l'impact de reconfiguration 187

5.3 Mesures de l'impact de l'exécution 187
5.4 Résultats 189
6 Synthèse 190

8 Conclusions & Perspectives 193

1 Conclusions 194
1.1 Contexte 194
1.2 Exigences 194
1.3 Contribution 195
2 Perspectives de recherche 197
2.1 Langage de haut niveau d'abstraction 197
2.2 Validation à l'exécution 198
2.3 Reconfiguration autonomique 198
3 Perspectives de travail 199

Bibliographie 201

TABLE DES FIGURES

1.1 Services intégrés aux usagers 20

2.1 Object, composants et services 32
2.2 Acteurs fondamentaux de l'approche à service. 35
2.3 Architecture à services (SOA) 37
2.4 Composition par procédés 40
2.5 Composition structurelle 41
2.6 Exemple d'un fichier WSDL 44
2.7 Framework OSGi 45
2.8 Cycle de vie du bundle 46
2.9 SOA sur OSGi 47
2.10 Architecture d'un composant iPOJO 48
2.11 Intégration à base de proxies 51
2.12 Approche de Secure FOCAS 52
2.13 TACC Proxies 53
2.14 13 Architecture de la plate-forme SCENE 54
2.15 Architecture d'intégration en utilisant ontologies 55

3.1 Intégration point à point 63
3.2 Intégration centralisé 64
3.3 Architecture publication/souscription 66
3.4 EAI pour faire interagir applications 68
3.5 Architecture ESB 70
3.6 Architecture de médiation 72
3.7 Base de données fédérées 73
3.8 Patron ordonnanceur (du livre d'EIP) 76

3.9 Aggregateur (du livre d'EIP) 77
3.10 Content-Based Router (du livre de EIP) 78
3.11 Wire Tap (du livre EIP) 79
3.12 Split/Aggregate - patron composé 80
3.13 Architecture d'un application Spring Integration 83
3.14 Architecture d'un application Camel 85
3.15 Architecture de Mule 2 87

4.1 Vision globale de CILIA. 102
4.2 Structure des composants « médiateur ». 103
4.3 Adaptateur d'entrée, de sortie et d'entrée/sortie (de gauche à droite) 104
4.4 Exemple de chaîne de médiation. 105
4.5 Framework d'exécution 106
4.6 Cycle de vie 108

5.1 Médiateur CILIA 113
5.2 Modèle d'un médiateur CILIA 114
5.3 Scheduler CILIA 115
5.4 Processor CILIA 116
5.5 dispatcher CILIA 117
5.6 Exemple de connecteur entre deux médiateurs. 118
5.7 Liaisons possibles entre médiateurs. 119
5.8 Sens de la connexion pour les connecteurs d'entrée. 120
5.9 Exemple de chaîne de médiation. 121
5.10 Framework d'exécution CILIA 123
5.11 Etapes pour la production d'une application CILIA. 125

6.1 Architecture d'un composant iPOJO 133
6.2 Exemple d'intégration utilisé dans ce chapitre. 134
6.3 Solution Cilia pour notre exemple d'intégration. 135
6.4 Méta-modèle de spécifications de médiateurs 136
6.5 Méta-modèle de spécification de scheduler 137
6.6 Méta-modèle de spécification de *processor* 138
6.7 Méta-modèle de spécification de dispatcher 139
6.8 Méta-modèle d'adaptateur d'entrée et de collector. 140
6.9 Méta-modèle d'adaptateur de sortie et de sender 141
6.10 Méta-modèle d'adaptateur d'entre/sortie 143
6.11 Méta-modèle d'un linker 144
6.12 API de la classe fr.liglab.adele.cilia.Data 147
6.13 Classes abstraites de schedulers 148
6.14 Classes abstraites de collectors. 152

6.15 Exemple de l'utilisation de l'API réflexif 157
6.16 API de CiliaContext et Chain 158
6.17 Exemple de la création d'un liaison entre médiateurs 159
6.18 Editeur de chaîne de médiation 162
6.19 Shell d'administration de chaînes de médiation 163
6.20 API du gestionnaire 164

7.1 Chaîne de médiation de Middleware RFID 172
7.2 Suivi des habitudes 175
7.3 Chaîne de médiation de suivi des habitudes 177
7.4 Architecture d'interaction de systèmes intégrés 180
7.5 Chaîne de médiation d'intégration de service de téléphonie fixe, de téléphonie mobile et d'internet 182
7.6 Relation de bundles de CILIA 185
7.7 Temps d'exécution d'une application de médiation 188
7.8 Fonctionnement de banc d'essai 189
7.9 Overhead de Camel, Spring Integration et CILIA 190

Liste des tableaux

3.1 Liste de patrons d'intégration 81
3.2 Résumé de middlewares d'intégration basées sur des EIP 91

7.1 Composants identifiés pour le middleware RFID 171
7.2 Composants identifiés pour suivre des habitudes 176
7.3 composants de médiation de système intégré 181
7.4 Résultats de l'analyse de code en utilisant SONAR 186
7.5 Temps par reconfiguration 187

1

Introduction

Sommaire

1 **Intégration de services** 20
2 **Intégration de services dans les entreprises** 23
3 **Objectifs de cette thèse** 26
4 **Structure du document** 27

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

En particulier, nous montrons que les développements informatiques ont considérablement évolué ces dernières années qu'ils abordent de plus en plus le domaine des services à la personne. Ce domaine présente des exigences de dynamisme et d'intégration structurantes et très difficile à atteindre.

1 Intégration de services

Nous sommes entrés dans un monde de services numériques. De plus en plus de services aux personnes sont proposés via une multitude d'équipements électroniques tels que des ordinateurs, des téléphones portables, des tablettes. Certains services reposent également sur divers capteurs intégrés aux environnements de vie et permettant d'acquérir des informations riches et variées de manière automatique et transparente.

Malgré un manque évident de maturité, et donc de fiabilité et de sécurité par exemple, ces services suscitent dès aujourd'hui de grandes attentes dans des domaines aussi variés que la santé, la domotique, l'énergie, le transport, etc. La figure 1.1 illustre la diversité des domaines abordés mais aussi la nécessité de relier ces domaines dès lors que la mobilité des usagers et des équipements est prise en compte.

Les projets de services numériques innovants et à forte valeur ajoutée se multiplient, avec des résultats parfois extraordinaires, souvent très décevants. 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.

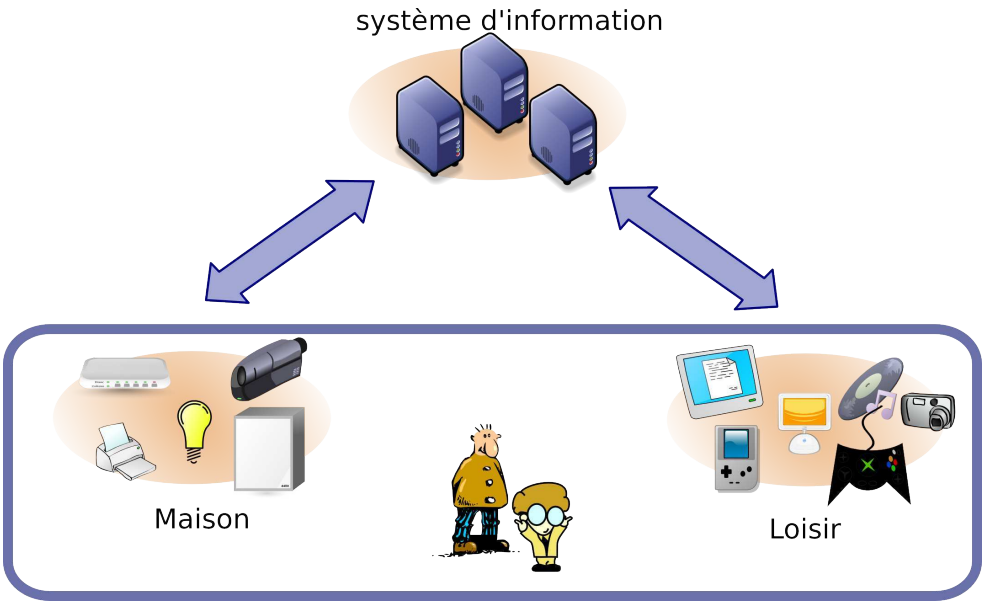

FIGURE 1.1 – Services intégrés aux usagers

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

Les fournisseurs de services intégrés font face à trois défis majeurs qui sont au cœur des enjeux techniques d'aujourd'hui :

1. intégrer rapidement des services nouveaux dans les Systèmes d'Information (SI), au sein de Plates-Formes de Service (PFS),
2. permettre aux utilisateurs finaux d'accéder à des services ou à des contenus de plus en plus variés, dans leur environnement de vie ou en situation de mobilité,
3. simplifier le déploiement et l'administration de ces services.

La première problématique liée à l'intégration d'applications, notamment au sein d'une entreprise, est déjà ancienne. L'intérêt d'une approche « middleware » vis-à-vis de cette problématique est aujourd'hui un fait bien établi. C'est en effet autour de briques de type middleware que se structure désormais l'architecture globale des systèmes d'information. Cette approche permet à la fois de répondre aux besoins des entreprises mais aussi de faire face plus aisément aux évolutions d'organisation des entreprises, que ce soit pour des aspects techniques ou commerciaux [Cha99]. Dans les grands systèmes (SI, PFS), l'émergence de l'informatique orientée service (SoC, *Service-Oriented Computing*) a fortement influencé les architectures d'intégration actuelles [Gle05, Pap03]. En particulier, les Web Services, basés sur la pile standardisée WS-*, apparaissent désormais en tant que connecteurs dans les EAI (*Enterprise Application Integration*), solutions historiques d'intégration et dans les ESB (*Enterprise Service Buses*) [Cha04], solutions d'intégration récentes, plus légères. EAI et ESB fournissent une interface unique aux applications et éliminent ainsi tout contact direct entre applications. Cela permet une meilleure séparation des préoccupations, entre code applicatif et code d'intégration, et ainsi des évolutions plus aisées.

Il est probable que les EAI, de par leur complexité et leur coût, soient amenés dans les années à venir à se marginaliser ou tout du moins à se focaliser sur des besoins en intégration lourds. Les ESB, quant à eux, ne se substitueront pas massivement aux EAI dans les grandes entreprises. Ils seront plutôt utilisés au cas par cas de manière raisonnée pour des intégrations bien spécifiques. Dans les deux cas, néanmoins, des progrès significatifs sont nécessaires en matière de qualité logicielle. En effet, les éditeurs d'EAI et d'ESB ont davantage cherché à élaborer une offre fonctionnelle riche sans réellement se positionner sur des problèmes de fond de type Génie Logiciel (ouverture, facilité d'utilisation et de gestion, gestion des erreurs, capacité à intégrer dynamiquement des sources distantes, etc.). Il est du reste regrettable que certains ESB récents prennent le même chemin que les EAI et n'aient pas tiré les leçons du passé !

Le deuxième défi auquel les fournisseurs de services doivent faire face est associé à l'explosion du marché des téléphones portables, des assistants personnels et plus généralement des « box » de toutes sortes. L'intégration de services apparaît en force dans ce nouveau domaine de l'informatique. Les applications potentielles dans ce domaine sont nombreuses et présentent de réels enjeux économiques et de société. Elles varient de l'aide au maintien à domicile des personnes âgées et des personnes convalescentes aux applications de cinéma à la maison en passant par l'automatisation de la gestion de la maison et la maîtrise de la consommation d'énergie. D'autres applications ont pour but d'aider les personnes dans leur vie de tous les jours. Les premières

applications résidentielles existantes étaient dédiées à un équipement physique en particulier. Les nouvelles applications du domaine tendent à utiliser conjointement les capacités de plusieurs équipements. Malgré les récents progrès du matériel, les applications effectives restent relativement peu nombreuses et surtout bien loin des fonctionnalités auxquelles on pourrait s'attendre. Ceci est principalement dû au fait que l'essentiel des efforts de recherche dans ce domaine a porté jusqu'à présent sur les capacités matérielles des équipements utilisés au détriment des logiciels.

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

- la distribution des équipements. La distribution est liée à la multiplicité des équipements mis en réseau. Des techniques intergicielles permettant de masquer l'aspect distribué de fonctions distantes sont encore aujourd'hui nécessaires ;
- l'hétérogénéité des équipements. Les équipements viennent avec des protocoles de communication variés, des langages de programmation distincts et des modèles de données sémantiquement différents qu'il convient d'assembler au sein de mêmes applications. Les services de demain seront amenés à inter-opérer avec un environnement de plus en plus riche de 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 disponibilité 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 besoins en intégration. De nouveaux besoins émergent, allant de pair avec de nouveaux modèles économiques. Aujourd'hui, un service est généralement une application téléchargée sur un téléphone, une tablette ou un équipement domestique. Demain, les nouvelles applications seront plus riches et devront inter-opérer avec des équipements hétérogènes ; elles seront réparties entre des serveurs IT, des passerelles domestiques et des équipements terminaux. Nous voyons clairement que la problématique de l'intégration se diversifie et se complexifie ;

Le troisième et dernier défi mentionné ici porte sur le déploiement et l'administration de services aussi bien au niveau des infrastructures IT qu'au niveau des domiciles des utilisateurs finaux. L'exploitation de services intégrés en milieu hétérogène, des capteurs jusqu'aux serveurs IT, est incontestablement un des freins majeurs à l'émergence de nouveaux services à valeur ajoutée.

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.

2 Intégration de services dans les entreprises

2.1 Problématique

Avec la prolifération des applications patrimoniales dans les Systèmes d'Information et la diversité des protocoles de communication, l'intégration des applications dans les Systèmes d'Information devient un enjeu majeur. C'est néanmoins une activité particulièrement complexe qui doit notamment gommer les différences syntaxiques, sémantiques et techniques séparant les multiples applications d'un même système d'information.

De façon plus générale, l'intégration d'applications demande de réaliser des opérations de médiation assez diverses. Certaines que l'on peut qualifier de fonctionnelles se concentrent sur des alignements syntaxiques et sémantiques alors que d'autres abordent plutôt des aspects non fonctionnels comme la sécurité ou la persistance de certaines informations.

La création de nouvelles applications reposant de plus en plus sur l'assemblage d'éléments logiciels préexistants, l'intégration a posteriori s'est imposé comme une approche systématique dans le développement de nouvelles applications. Dans ce contexte, les Systèmes Informatiques des grandes entreprises se sont progressivement structurés autour des EAI qui ont amené un modèle de gestion centralisé des échanges entre des applications hétérogènes.

Cette section dresse un premier constat sur l'utilisation des EAI, leur complexité et aussi leur impact sur les processus d'entreprise. Ce constat permet de mieux appréhender les attentes actuelles des entreprises vis-à-vis des ESB et d'expliquer des utilisations finalement limitées et très ciblées de ces nouvelles solutions, alors que dans le même temps les besoins en intégration s'amplifient et se diversifient, notamment avec l'ouverture des infrastructures d'entreprise à des partenaires tiers.

2.2 Enterprise Application Integration « EAI »

Les EAI ont largement influencé l'urbanisation des Systèmes d'Information à la fin des années 1990. En apportant un modèle centralisé, véritable plaque tournante des échanges entre applications, ils ont contribué à structurer les différents processus de l'entreprise autour d'un modèle de données global « standardisé », dessinant les objets métier de l'entreprise. Ainsi, dans la logique « EAI », un processus se connecte à l'EAI qui prend en charge la transformation des données du processus vers les données du modèle global et achemine les données adaptées vers un ou plusieurs processus destinataires.

Les EAI sont représentatifs de solutions d'intégration de type « boîte noire » s'accompagnant d'un modèle de gestion centralisé du code d'intégration. Force est de constater qu'ils ont eu un impact fort sur l'organisation et le pilotage des équipes projet en charge du développement de services par intégration d'applications patrimoniales. Le modèle EAI a, en effet, favorisé la mise en place d'un guichet unique chargé du traitement des besoins projet et des évolutions par pallier

du schéma de données global. Dans cette organisation, les équipes projet n'ont plus la main sur les problèmes d'intégration qui sont gérés de façon centralisée.

Ce type d'organisation a clairement ses limites dans un contexte où le développement de services intégrés s'accélère et les besoins en intégration deviennent de plus en plus variés et complexes. Ce constat a débouché sur l'expression de nouveaux besoins chez les fournisseurs de services intègres, à savoir :

- Gérer séparément les fonctions de gestion communes à toutes les applications[TOHS99] (sécurité, gestion des SLA, reporting/log, transformations, etc.) et les fonctions d'intégration spécifiques à chaque flux applicatif (traductions sémantiques, filtrage sur le contenu, enrichissements, routages conditionnels, etc.). La première forme d'intégration est désignée par le terme « médiation technique », la seconde par le terme « médiation métier » ou « médiation fonctionnelle ».
- Mutualiser les fonctions de médiation technique au niveau d'un modèle d'intégration centralisé "léger".
- Reporter sur les équipes projet la responsabilité du développement et de l'administration des fonctions de médiation fonctionnelles. Ces équipes doivent répondre rapidement et de façon fine à des besoins en intégration et s'appuyer pour cela sur des solutions middlewares ouvertes, légères et outillées[dRAE+07, SZ00, Kaj04].

Ces nouvelles exigences fortes sont mal traitées par les EAI traditionnels et appellent des solutions d'intégration nouvelles.

2.3 Enterprise Service Bus « ESB »

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.

Un service logiciel est vu comme une entité logicielle qui peut être utilisée grâce à sa description. Le consommateur du service l'utilise sans avoir connaissance de la technologie sous-jacente pour son implantation ainsi que de sa plate-forme d'exécution. De plus, le service ne connaît pas le contexte dans lequel il va être utilisé par le client. Cette indépendance à double sens est une propriété forte des services qui facilite le faible couplage.

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. D'une part, il existe de nombreuses technologies à service 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.

Les ESB se positionnent comme les EAI 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 fonctionnelle 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, etc.). Une partie importante du code d'intégration est ainsi dédiée à la gestion des erreurs.

Les ESB sont présentées comme des solutions d'intégration légères, plus souples, plus ouvertes. De nombreuses études ont cependant révélé la vraie nature de nombreux ESB actuels qui apparaissent très divers et finalement peu innovants. Nous distinguons en fait deux approches architecturales. La première se définit comme une extension d'un serveur Java EE. Plus précisément, cette solution consiste à développer un ESB au-dessus d'un serveur d'applications Java EE. C'est par exemple le cas de WebSphere ESB d'IBM ou d'Aqualogic ESB de BEA-Oracle. Le premier intérêt de cette approche est bien sûr de réutiliser le modèle Java EE et le serveur existant. Le résultat est cependant conséquent en taille. De plus, le modèle Java EE n'est pas parfaitement adapté à la médiation. Une seconde approche est de développer des middlewares dédiés à la médiation. De nombreux ESB relèvent de ce courant, parmi lesquels Mule (Codehaus), Camel (Apache), ServiceMix (Apache) ou encore PetalsLink (ObjectWeb). Ces deux derniers ESB reposent sur JBI (Java Business Integration, JSR-208) qui standardise l'utilisation d'un orchestrateur pour gérer

les échanges de messages entre des fonctions de médiation ainsi que les interactions avec un client ou un fournisseur de service externes à la chaîne de médiation.

Quelle que soit l'approche, les ESB sont très orientés par la technologie. Les chaînes de médiation résultant de l'assemblage des fonctions de médiation restent difficiles à concevoir, déployer et maintenir. Elles sont également difficiles à faire évoluer et à réutiliser. Il apparaît que les ESB cités précédemment n'ont pas apporté satisfaction vis-à-vis des exigences exprimées. Nous pensons qu'il y a clairement un besoin de séparation des préoccupations : les opérations de médiation plus complexes et très techniques (collecte, synchronisation, routage, envoi, etc.) ne devraient pas être au même niveau que les opérations associées aux aspects métier de l'intégration (traduction, transformation, enrichissement, etc.). Plus précisément, les détails plus techniques devraient être « masqués » par le middleware afin que les concepteurs puissent se focaliser sur les aspects métier.

3 Objectifs de cette thèse

Cette thèse se situe dans le cadre de l'informatique orientée service et se concentre sur la notion de médiation telle que mise en place dans les « Enterprise Service Bus ».

Son objectif est de fournir un cadre conceptuel et programmatique pour le développement d'opérations de médiation. Nous visons à définir un modèle architectural modulaire et homogène permettant le développement de solutions de médiation flexibles, évolutives, dynamiques et administrables. Nous visons également à fournir une implantation robuste et efficace de notre approche, utilisable dans des contextes industriels réels.

Un des principaux objectifs de ce travail est de définir et séparer clairement les concepts liés aux opérations de médiation. Pour cela, nous nous sommes appuyés sur les patrons d'intégration (Enterprise Integration Pattern) aujourd'hui communément utilisés dans les applications d'entreprises. Notre objectif étant de fournir un modèle de conception et d'implantation répondant facilement à ces patrons d'intégration.

Pour atteindre nos objectifs, nous proposons d'introduire un nouveau modèle à composant spécifique à la médiation. Ce modèle à composant définit formellement la façon de construire des unités de médiation appelées « médiateurs » ainsi que la façon de les développer et de les composer. Nous définissons également une plate-forme d'exécution spécifique associée à ce modèle composant.

Ce modèle à composant spécifique porte le nom de Cilia et est disponible en open source. Cilia est aujourd'hui utilisé dans plusieurs projets, notamment au sein d'Orange et de Schneider Electric.

L'approche introduite par Cilia est modulaire et homogène. Cilia donne ainsi la possibilité d'étendre et de modifier facilement des applications de médiation, apportant aux experts la maîtrise et la flexibilité nécessaires au développement et à la maintenance d'applications de médiation industrielles.

4 Structure du document

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

- le chapitre 2 présente l'informatique orientée service. Après un certain nombre de définitions, nous présentons les technologies essentielles d'aujourd'hui. Nous montrons ensuite que la mise en oeuvre de cette approche est variée en présentant différentes plateformes : Web Services, OSGi puis 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 patterns d'intégration usuellement utilisés aujourd'hui. Nous présentons également les approches et outils existant pour l'intégration de services logiciels.

La contribution se divise en quatre parties :

- le chapitre 4 expose la problématique, les objectifs et donne une vision d'ensemble de notre approche. Nous expliquons le concept de modèle et composant et introduisons notre modèle à composant spécifique à la médiation, à savoir Cilia.
- le chapitre 5 fournit une description détaillée de Cilia.
- le chapitre 6 détaille l'implémentation de Cilia.
- le chapitre 7 illustre l'utilisation de notre framework par des cas d'application.

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

Première partie

Etat de l'art

2

Approches à services

Sommaire

1 **Services Logiciel** 32
2 **Architecture** 35
3 **Composition de services** 39
4 **Technologie à services** 42
5 **Integration de services** 51
6 **Synthèse** 57

Dans le chapitre précédent, nous avons introduit la notion de services intégrés à l'usager et l'importance de la notion d'intégration pour la mise en place et la maintenance de ces services numériques. Nous avons également présenté les défis auxquels les fournisseurs de services intégrés étaient confrontés.

Ce deuxième chapitre présente les approches à services. Elles se sont développées depuis quelques années et proposent des solutions aux défis de l'informatique moderne que sont l'hétérogénéité et le dynamisme croissant des applications. Ces architectures sont particulièrement prometteuses, aussi bien dans le domaine des Système d'Information que des systèmes pervasifs. Si elles ne résolvent pas tous les problèmes, elles tendent à simplifier sensiblement les travaux d'intégration.

1 Services Logiciel

1.1 Généralités

De nouvelles approches de développement sont régulièrement proposées par le Génie Logiciel (GL) de façon à faciliter et améliorer la production et la maintenance des systèmes logiciels. En général, les nouvelles propositions ne remplacent pas complètement les approches existantes. Au contraire, elles s'appuient souvent les unes sur les autres pour fournir des paradigmes plus riches, plus adaptés aux besoins.

L'approche à services, ou « Service-Oriented Computing (SOC) », est apparue récemment. Elle définit la notion de service logiciel comme brique de base fournissant des fonctionnalités et pouvant être utilisée pour construire des applications. Un service peut être implanté suivant diverses technologies, comme des objets ou des composants. Ceci est illustré par la figure 2.1 issue de travaux d'IBM. IBM[Kee04] propose une architecture en trois couches - la couche la plus basse est celle des objets, suivie par la couche des composants qui s'appuient sur une solution objet pour réaliser leurs fonctionnalités et enfin la troisième couche qui permet d'exposer les fonctionnalités fournies par des composants sous la forme de services.

On voit sur la Figure 2.1 qu'un service fournit une fonctionnalité à travers une interface qui regroupe un ensemble d'opérations et qui réalise en même temps la séparation entre le service et son implémentation. L'implémentation d'un service est réalisée par un ou plusieurs composants.

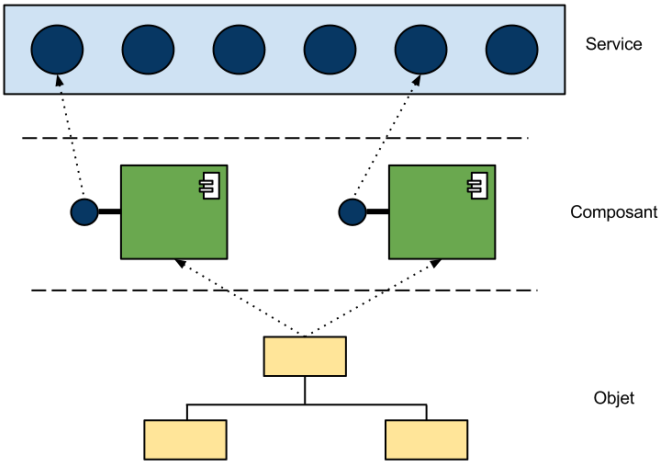

FIGURE 2.1 – Object, composants et services

Grâce à cette séparation entre la fonctionnalité fournie et son implémentation, l'approche à services permet une évolution plus rapide voir même transparente des systèmes. Tant que la fonctionnalité du service n'est pas modifiée, l'implémentation du service peut être modifiée autant de fois que nécessaire. En intégrant des services hétérogènes et la standardisation des mécanismes de collaboration des services, l'approche à services essaie de résoudre les problèmes critiques auxquels le génie logiciel doit répondre : l'intégration des systèmes hétérogènes et l'évolution continue.

1.2 Definition

Le terme service fait partie de notre vocabulaire quotidien. Le dictionnaire de la langue définit un service comme « une action réalisée par une entité pour une autre ». L'utilisation d'un service dans le monde réel donne souvent lieu à une négociation : avant de l'utiliser, il est nécessaire de connaître ses capacités exactes et ses caractéristiques d'utilisation (disponibilité, coût de son utilisation, ...) et de passer un accord avec le fournisseur du service pour définir le coût, la qualité attendue, etc.

Un service logiciel est un type de service particulier. Généralement parlant, un service logiciel est une entité logicielle qui est mise à la disposition d'autres applications. Pour être effectivement utilisé, un service est décrit et publié de façon à ce que d'éventuels utilisateurs puissent le trouver. Comme tous services de la vie courante, un service logiciel peut donner lieu à une négociation et la mise en place d'un contrat sur la qualité désirée. L'utilisateur peut ensuite exécuter le service et obtenir la fonctionnalité attendue.

De nombreuses définitions ont été proposées pour caractériser la notion de service logiciel. Voyons les plus communément citées. Le consortium OASIS propose la définition suivante :

> « A service is a mechanism to enable access to one or more capabilities, where the access is provided using a prescribed interface and is exercised consistent with constraints and policies as specified by the service description ».[BH06]

Cette définition présente le concept de service comme un mécanisme d'interaction entre les clients et les capacités proposées par le fournisseur de services. Cette interaction est réalisée en utilisant la description de service sous la forme d'interfaces d'interaction. La description du service permet aux consommateurs de connaître la politique d'accès au service et ses restrictions.

M. Papazoglou a proposé la définition suivante :

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

Un service logiciel selon Papazoglou est une entité logicielle qui peut être utilisée grâce à sa description. L'utilisation d'un tel service, fait par un consommateur de service, est faite sans avoir connaissance de la technologie utilisée pour son implantation, ainsi que de la technologie d'exécution. Cette indépendance facilite le faible couplage entre le service et le consommateur.

L'intérêt de cette approche, que le service soit local ou distant, est de permettre un faible couplage entre le fournisseur et le consommateur du service, permettant ainsi l'utilisation de services de divers fournisseurs. De plus, les détails d'implémentation des services sont transparents pour son utilisation. C'est la responsabilité de chaque fournisseur de service de résoudre ses dépendances et fournir une interface bien précise et autonome qui sera le seul point d'accès pour son utilisation. Les services sont par nature des entités sans état. En d'autres termes, l'interaction

entre le client et le fournisseur doit être individuelle, et les paramètres requis pour le service doivent être suffisants pour accomplir son exécution.

Arsanjani[AA06] définit les services logiciels de la façon suivante :

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

Cette définition identifie les principales interactions qui permettent l'utilisation des fonctionnalités offertes par un service. Une étape de recherche permet à un client de découvrir la description du service correspondant à ses critères. Cette description lui permet de savoir comment réaliser la liaison avec le service et éventuellement de réaliser l'invocation du service. L'exécution de ces trois étapes peut être différée même au moment de l'exécution de l'application cliente. Un environnement à services est un environnement dynamique par nature même car des descriptions de services peuvent être publiées ou retirées à tout moment. Cette situation est appelée disponibilité dynamique des services car, entre le moment de la spécification de l'application cliente et le moment de son exécution, plusieurs services compatibles avec les besoins de cette application peuvent apparaitre ou disparaitre. La réalisation tardive de la liaison entre l'application cliente et le service rend l'architecture de l'application cliente reconfigurable. En effet, d'une exécution de l'application cliente à une autre l'architecture de celle-ci peut être différente car les services utilisés peuvent être, à cause de la disponibilité dynamique, différents.

Il nous apparaît que les différents auteurs s'accordent sur les faits suivants :

1. - Un service est une entité logicielle fournissant des fonctionnalités spécifiées dans une description de service.
2. - La description de service décrit aussi bien les capacités fonctionnelles et non-fonctionnelles du service.
3. - En utilisant cette spécification, les clients du service peuvent rechercher, sélectionner et invoquer le service qui répond mieux à ses critères à ce moment donné.

Nous verrons par la suite que les différentes implémentations des principes de l'approche orientée services débouchent sur des points de variation, tels que l'information utilisée pour la description des services, le traitement de la disponibilité dynamique des services, le moment de réalisation des liaisons entre services, etc.

Il apparaît enfin que la notion de service n'est pas attachée à une technologie, au contraire l'indépendance des plates-formes d'implantation est mise en avant dans toutes les définitions. Pour certains, l'approche orientée service est avant tout un style d'architecture[SG96, Gar93] ou un patron d'architecture. Cela signifie que l'approche à service définit essentiellement un ensemble de principes et de mécanismes qui permettent de découpler clients et fournisseurs en facilitent les évolutions à l'exécution.

2 Architecture

2.1 Principaux acteurs

L'objectif de l'approche à services est la construction d'applications à partir d'entités logicielles indépendantes, tout en assurant un faible couplage entre ces entités. La définition, publication, utilisation des services se fait dans une infrastructure bien définie qui explicite les interactions possibles entre fournisseurs et utilisateurs. Cette infrastructure s'appelle une architecture à services (*Service Oriented Architecture*). Le group CBDI [SW04] propose la définition suivante :

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

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

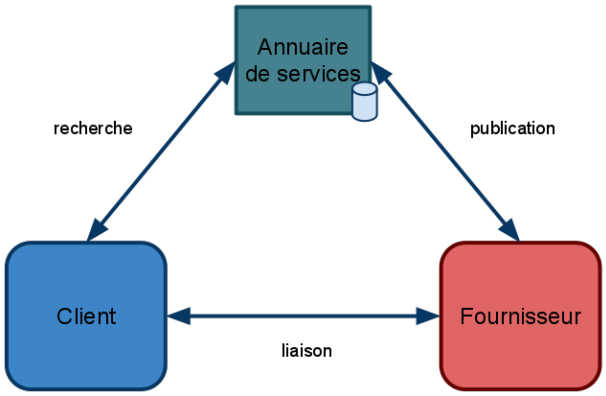

FIGURE 2.2 – Acteurs fondamentaux de l'approche à service.

Toutes les architectures à services reposent sur trois acteurs de base et doivent mettre en œuvre leurs interactions (voir figure 2.2). Ces acteurs sont :

- les fournisseurs de service qui décrivent des services selon une spécification bien établie et les communiquent à une entité centralisatrice, souvent appelée registre ou annuaire de services ;
- l'annuaire de services qui contient l'ensemble des descriptions des services proposés et valides à un instant donné ;

– les consommateurs de service, encore appelés clients, qui utilisent des services après les avoir sélectionnés au sein du registre (ou toute entité regroupant des services).

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

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

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

Un consommateur de service, interroge le registre de services pour s'enquérir des services disponibles qui correspondent à ses besoins. La découverte d'un service est réalisée grâce à la description du service disponible dans l'annuaire. Après avoir sélectionné le service qu'il veut utiliser, le consommateur du service peut, dans certains cas, négocier auprès du fournisseur les termes suivant lesquels il peut utiliser ce service. A la fin de la négociation, un accord de service[AFM05] est réalisé entre le consommateur et le fournisseur. La plupart du temps, cet accord de service contractualise les termes de l'utilisation du service par le consommateur sans garantie totale du résultat. Grâce aux informations disponibles dans la description du service, le consommateur de service peut, dès lors, réaliser la liaison et appeler les fonctionnalités du service.

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

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

2.2 Architectures à services

Ces interactions entre consommateurs et fournisseurs de services sont réalisées au sein d'une architecture à service (SOA pour « *Service Oriented Architecture* ») qui fournit un environnement d'intégration et d'exécution de services. Comme illustré par la figure ci-dessous, les différents éléments d'une telle architecture sont usuellement séparés en deux catégories[Mar08] :

- Les mécanismes de base qui permettent la publication, découverte, composition, négociation, contractualisation, invocation des différents services ;
- Les mécanismes additionnels qui assurent la prise en charge de besoins non-fonctionnels tels que la sécurité, les transactions, la qualité de service. Le nombre et la nature des aspects non fonctionnels divergent en fonction des architectures à service.

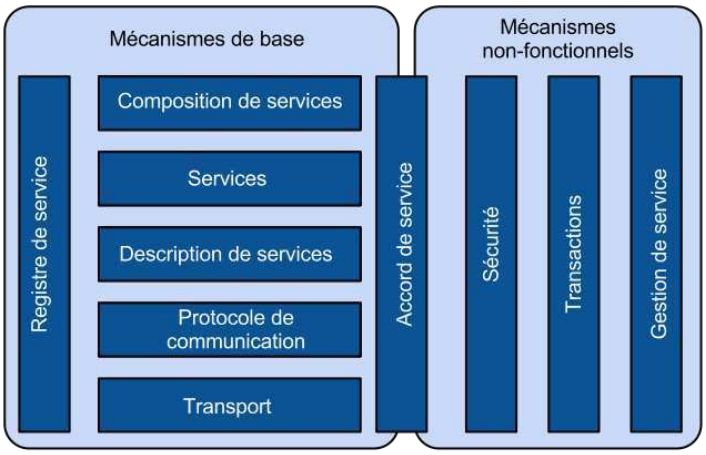

FIGURE 2.3 – Architecture à services (SOA)

La partie fonctionnelle d'un environnement d'intégration et d'exécution de services fournit les formalismes et les mécanismes nécessaires à la mise en place du patron d'interaction de base entre fournisseurs et utilisateurs :

- L'invocation d'un service suppose la mise en place d'une communication entre le client et le service invoqué. Cette communication est réalisée à l'aide d'éléments chargés d'assurer le transport des requêtes et des réponses concrétisant la collaboration entre le client et le service utilisé (Protocole de communication et Transport dans la figure mentionnée)
- La description de service est réalisée en utilisant un langage de description spécifique qui permet aux éventuels consommateurs de services de connaitre les capacités fonctionnelles et non fonctionnelles d'un service mais aussi la manière dont ils doivent l'invoquer
- Le registre de services héberge les descriptions des services mises à disposition par divers fournisseurs
- La composition de services fournit les mécanismes nécessaires pour assembler des services au sein d'une application.

Parmi les éléments non-fonctionnels supportés par un environnement d'intégration et d'exécution de services, nous pouvons en mentionner deux particulièrement important :

– L'accord de service (ou contrat de service) qui représente les termes de l'utilisation d'un service par un consommateur de services. Les termes spécifient la fonction fournie et attendue par un consommateur de services mais aussi la façon dont cette fonction est délivrée en spécifiant par exemple un niveau de disponibilité ou de performance (latence, fiabilité) ;
– Les éléments de sécurité mis en place lors d'une sélection et d'une exécution de service. L'utilisation des mécanismes de sécurité peut être demandée par les fournisseurs de services qui veulent gérer l'accès aux services qu'ils fournissent, ou par des consommateurs de services souhaitant, par exemple, garantir l'intégrité ou la confidentialité des données transmises.

2.3 Dynamisme

Nous avons présenté la vision communément admise de l'approche à services dans la section précédente en présentant les acteurs et les interactions. Cependant, certaines définitions introduisent des interactions supplémentaires pour ce que l'on appelle l'approche à services dynamique. Cette approche s'intéresse aux modifications de l'environnement d'exécution des services. Il a donc été ajouté deux primitives à l'approche à services qui sont les suivantes :

• le retrait de service qui signale qu'un fournisseur de service n'est plus en mesure de proposer son service.

• la notification qui informe les consommateurs de l'arrivée ou du départ d'un fournisseur qui propose un service répondant à leurs besoins.

Si une architecture à services prend en compte ces deux nouvelles primitives, les consommateurs de services seront capables de choisir à l'exécution leur service et surtout d'en changer si nécessaire. De nouvelles capacités dynamiques sont disponibles et nous pouvons construire des applications conscientes du contexte dynamique. Dans le cas d'applications ubiquitaires, le dynamisme et les notifications associées sont indispensables. La prise en compte de l'évolution dynamique de l'environnement est un nouvel avantage que l'on ajoute à ceux de l'approche à services.

Ensuite, les mécanismes de compositions proposés par un SOA dynamique doivent être capables de gérer l'arrivée et le départ des services utilisés, et donc assurer la conformité des services fournis en fonction de ces événements. Plus particulièrement, la composition dans un SOA dynamique doit être capable de gérer la structure de la composition afin de déterminer quels sont les services manquants ou à remplacer. Finalement, la partie gestion et supervision doit être capable de vérifier en permanence la cohérence du système sous-jacent en fonction des critères de l'application. De manière transversale, un SOA dynamique doit permettre la vérification et la gestion des propriétés non-fonctionnelles telles que la sécurité ainsi que la qualité de service. Cependant, effectuer tous ces traitements tout en gérant le dynamisme reste aujourd'hui extrêmement complexe. Peu d'intergiciels implémentent les différentes couches nécessaires à un SOA dynamique.

3 Composition de services

3.1 Problématique

La construction d'applications en utilisant une approche à services, du point de vue architectural, repose sur la composition de services. La composition de services est le mécanisme qui permet l'intégration des services pour construire applications à base de services [SD05]. Le résultat de la composition de services peut être un nouveau service, appelé service composite. Dans ce cas, la composition est dite composition récursive ou hiérarchique.

Cependant, pour passer d'un ensemble de services à une composition de services correctement structurée, il faut suivre un certain nombre d'étapes, de la spécification à la composition concrète qui peut être directement exécutée :

1. la définition de l'architecture fonctionnelle : cette phase est faite pour identifier les fonctionnalités attendues pour l'application résultant de la composition de services ;
2. l'identification des services : selon les fonctionnalités attendues, on détermine les services nécessaires à la composition ;
3. la sélection des services : à partir des services identifiés à l'étape précédente, il faut sélectionner les services qui répondent correctement aux besoins ainsi que les implantations adaptées ;
4. la médiation entre services : même si à l'étape précédente, les services les plus adaptés ont été sélectionnés, il n'est pas toujours possible de les assembler tels quels. Il faut souvent ajouter de la médiation, par exemple sémantique ou syntaxique ;
5. le déploiement et l'invocation des services : une fois la composition correctement réalisée, il faut déployer la nouvelle application. La liaison entre les consommateurs et fournisseurs de services peut être statique ou dynamique. Le consommateur sait comment accéder aux services dans la liaison statique. Une liaison dynamique permet au consommateur de découvrir la localisation du service à l'utilisation, en utilisant l'annuaire de services par exemple.

Il existe plusieurs discriminants pour caractériser des compositions de services. En particulier, deux discriminants sont communément utilisés :

- le contrôle de la composition qui définit la logique de coordination des services identifiés pour résoudre un problème global,
- le type de liaison, ou d'appel, qui définit la manière dont les services impliqués dans la composition interagissent.

Nous nous concentrons ici sur le premier aspect lié à la composition. La logique de coordination des services peut être extrinsèque ou intrinsèque aux services. Ces deux possibilités de gestion du contrôle définissent deux styles de compositions : la composition par procédés, aussi appelée composition comportementale, et la composition structurelle. Ces deux styles de composition sont présentés dans les parties suivantes.

3.2 Composition par procédé

La composition de services par procédés consiste à faire l'assemblage de services selon un ordre et un flux d'exécution[Pel03]. L'exécution d'une composition par procédés est réalisée par un coordinateur de services. L'utilisation d'un tel coordinateur implique une composition avec une logique d'exécution qui sera interprétée par ce coordinateur.

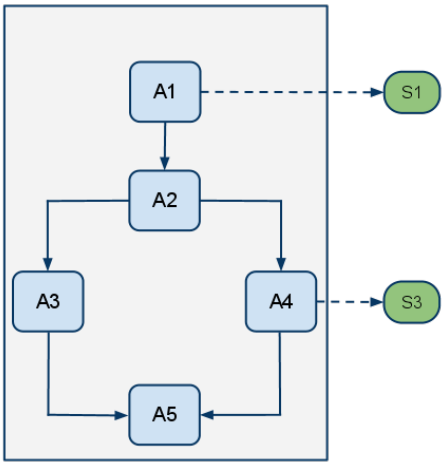

FIGURE 2.4 – Composition par procédés

La composition par procédés (voir figure 2.4) est décrite dans un langage spécifique qui est interprété par le coordinateur. Ce langage, graphique ou textuel, permet de décrire les services composés. Sa caractéristique la plus importante est qu'il permet de décrire aussi la logique d'exécution de l'application dans un langage de haut niveau.

Dans une composition par procédés, les services sont à l'extérieur de la description de la composition. La communication vers les services n'est pas déclarée dans une composition par procédés. C'est le moteur d'exécution ou coordinateur qui fait le passage de messages entre les services. De même, le moteur d'exécution fournit les mécanismes nécessaires à l'alignement syntaxique ou sémantique des messages.

WS-BPEL[Pas05] est un langage de procédés basé sur la technologie XML, tout comme les autres standards des services Web. WS-BPEL permet de construire des procédés interprétables et exécutables par un moteur d'orchestration. Un procédé est composé d'activités qui s'enchaînent grâce à des échanges de données. Les activités peuvent être de deux types : basiques ou complexes. Les activités basiques sont des types de base comme *invoke* pour appeler un service Web, *receive* pour attendre une invocation, *reply* pour une réponse... A partir de ces types de base, il est possible de créer des activités composites avec des structures de construction du contrôle du flot de données : *flow* pour une ou plusieurs activités concurrentes, *sequence* pour une séquence d'activités, *switch* pour des conditions, *while* pour une boucle... Il faut quand même noter que l'échange de données n'existe pas en tant que tel, puisqu'il faut passer par des affectations de variables entre les activités. APEL est un autre langage de composition par procédés qui s'appuie sur la notion d'activité[DEA97]. Où une activité est une tâche atomique ou composite.

3.3 Composition structurelle

Par opposition à la composition par procédés, le contrôle dans une composition structurelle est exprimé à l'intérieur des services (voir figure 2.5). Le contrôle n'est alors connu que du développeur et les seules informations qu'il possède sont celles concernant les fonctionnalités que le service fournit et celles que le service requiert. Dans le cas de la composition structurelle, les services sont donc clairement identifiés avec leurs interactions. Il faut à l'assemblage résoudre les dépendances syntaxiques et sémantiques entre les composants pour s'assurer de la validité de la composition. C'est pourquoi pour définir le fonctionnement de la composition, le développeur livre aussi la logique de coordination, qui peut être, par exemple, sous forme de classes Java.

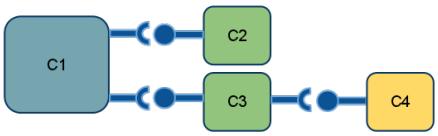

FIGURE 2.5 – Composition structurelle

La logique d'exécution, ou d'appels des services, est gérée par chaque composant qui requiert des services. Le flux d'exécution est ainsi inconnu. L'architecte de l'application peut identifier les services utilisés dans une application. Néanmoins, l'identification de la séquence d'exécution reste difficile.

Ce type de composition est effectué principalement à partir de l'usage de langages de description d'architectures formalisés (ADL pour ses sigles en anglais). L'ADL est un langage pour décrire des architectures d'applications. Dans l'approche à services, l'ADL est utilisée pour décrire les composants d'une application, les services utilisés et les liaisons ou méta-informations qui décrivent les mécanismes de résolution de dépendances de services.

SCA[Cha07] est une solution de composition structurelle de services. SCA propose un langage de description d'architectures pour faire la composition de composants. Dans cette approche, les composants sont la seule unité de composition. SCA est une approche hiérarchisée. La composition de composants, dite composite, est donc aussi une unité de composition.

iPOJO[EHL07, Hal08] (acronyme de injected Plain Old Java Objects) est un modèle à composants orientés services pour la programmation. Ce type de modèle combine les avantages de deux approches à composants et à services. iPOJO repose sur la plate-forme à services OSGi. La plate-forme OSGi permet de construire des applications dynamiques et modulaires grâce à une couche de modules et une couche de services gérés. Les composants en iPOJO sont déclarés en spécifiant les services requis et, dans certains cas, les services fournis. Le framework utilise la notion de conteneur qui gère les aspects techniques et non-fonctionnels du composant. Ce conteneur est responsable du registre de services fournis, ainsi que de la recherche et de la liaison vers les services requis.

OSGi et ipojo sont présentés plus en détail dans la suite de ce document.

4 Technologie à services

Il existe plusieurs technologies à services. L'approche à service n'est pas exclusive aux systèmes repartis, de manière qu'il est possible de construire des architectures à services en utilisant n'importe quel langage de programmation. Le projet Apache Celix [1] est un exemple d'un framework à services implémenté en langage C.

Parmi eux, nous avons choisi d'en présenter trois : les services Web[CS02], les services OSGi et les composants orientés service de iPOJO. Notre choix a été principalement motivé par l'étendue de leur utilisation dans le monde industriel et celui de la recherche. Les services Web représentent actuellement l'ensemble de standards les plus connus pour la réalisation des applications Internet. Cette technologie repose sur des standards très populaires, tels que le XML ou le protocole HTTP. Les services OSGi, la deuxième technologie que nous présentons, sont de plus en plus utilisés pour la réalisation des applications à services.

4.1 Services Web

Probablement, la technologie la plus connue est les Services Web. Cette technologie est sans doute la plus utilisée dans l'industrie et dans l'académie. Le succès des Service Web est dû au fait qu'ils s'appuient sur des standard ouverts pour le transport et pour le format de messages, ce choix permet l'interopérabilité entre diverses technologies. Un seul client peut utiliser les services de deux fournisseurs différents. Ces deux services peuvent être développés en utilisant des technologies différentes.

L'usage le plus populaire de Services Web est basé sur l'utilisation du protocole HTTP comme la couche de transport, et pour le transfert de messages le protocole SOAP[W3C]. Le protocole HTTP est le protocole de transport le plus utilisé. Ce choix est dû au fait que la majorité des dispositifs le supportent ; en plus, il y a une expertise gagnée au fil du temps par les industriels et le monde académique. Cette technologie n'est pas la seule, récemment l'usage des middlewares de messagerie, comme JMS, commence à prédominer aussi comme couche de transport.

Le succès des Services Web est aussi dû au faible couplage entre le consommateur et le fournisseur de service ; ce faible couplage permet l'interopérabilité entre diverses technologies, par exemple Java, .NET [2], etc. L'interface du service est sous la forme d'une description de service, le langage le plus populaire est WSDL[New02, W3c07a]. Ce langage, basé sur XML, permet de décrire les opérations fournies par le service, ainsi que l'information indispensable pour son utilisation.

Le registre de Service Web est fait en utilisant un annuaire UDDI[OAS]. Cet annuaire fournit un mécanisme de registre et de localisation de services. Un annuaire UDDI peut être composé de plusieurs nœuds serveurs UDDI. Les consommateurs communiquent avec l'annuaire pour localiser un service, puis l'annuaire répond avec l'information indispensable afin que le consommateur puisse interagir avec le fournisseur de service.

1. Apache Celix http ://incubator.apache.org/celix/
2. http ://www.microsoft.com/net

Un service Web reçoit une requête de la part d'un client, traite cette requête et à la fin du traitement envoie sa réponse. Néanmoins, la réalisation interne de cette fonctionnalité peut être plus complexe. Pour cela, le service peut utiliser les fonctionnalités d'autres services fait dont le client n'est pas conscient. Cela nécessite une coordination entre les différents appels des services utilisés.

Les services Web acceptent deux styles de communication - synchrone et asynchrone [PD04]. Les clients d'un service Web synchrone expriment leur requête comme un appel de méthode avec un ensemble d'arguments et attendent, avant de continuer leur exécution, la réponse à cette requête. Ce type de communication demande un couplage plus fort entre le client et le service, car le client attend toujours l'obtention de sa réponse.

Un deuxième type de communication est la communication asynchrone réalisée avec des messages. Quand un client invoque de manière asynchrone un service, il envoie un message de gros grain (appelé aussi document) au service. Celui-ci le reçoit, le traite et peut décider de répondre ou non. Le client continue son exécution sans attendre la réponse du service invoqué et, de cette manière, il est moins fortement couplé au service invoqué. La réponse peut arriver après une période de temps considérable. C'est le cas par exemple d'un client qui place une commande auprès d'un service Web d'un fournisseur et qui recevra sa réponse quelques jours après.

La description d'un service est la clé de la réalisation du fort découplage entre un consommateur de service et un fournisseur de service. En décrivant dans un format standard les propriétés fonctionnelles d'un service, la description de service permet de réaliser l'abstraction des différents langages de programmation utilisés pour réaliser les implémentations des services. De plus, le consommateur d'un service ne doit pas connaitre des détails relatifs à la plate-forme d'exécution qui héberge le service fourni par le fournisseur de service.

Pour un service Web, la fonctionnalité est décrite à travers un fichier WSDL(voir Figure2.6) (Web Services Description Language) [CCMW01]. La description WSDL d'un service Web est indépendante d'une plate-forme et d'une technologie donnée et décrit la fonctionnalité du service (ses opérations), la localisation du service (l'adresse réseau du service) et comment le service doit être appelé (le type de données et les protocoles d'accès). En utilisant la description WSDL, un client peut découvrir un service Web et peut invoquer n'importe quelle opération fournie par le service.

La description WSDL est divisée en deux parties : la définition de l'interface du service (service abstrait) et la description de l'implémentation (service concret) :

- La définition de l'interface du service décrit les opérations fournies par le service, la définition de leurs paramètres et des types de données ;
- La description de l'implémentation associe l'interface abstraite du service avec une adresse réseau et avec un protocole spécifique d'accès.

```
<wsdl:types>
    <schema targetNamespace="http://apache.org/hello_world_xml_http/bare/types"
            xmlns="http://www.w3.org/2001/XMLSchema"
            elementFormDefault="qualified">

        <element name="responseType" type="xsd:string"/>
        <element name="requestType" type="xsd:string"/>
        <element name="myComplexStruct" type="x1:myComplexStructType"/>
    </schema>
</wsdl:types>

<wsdl:message name="greetMeRequest">
    <wsdl:part element="x1:requestType" name="me"/>
</wsdl:message>

<wsdl:message name="greetMeResponse">
    <wsdl:part element="x1:responseType" name="theResponse"/>
</wsdl:message>

<wsdl:portType name="Greeter">
    <wsdl:operation name="greetMe">
        <wsdl:input message="tns:greetMeRequest" name="greetMeRequest"/>
        <wsdl:output message="tns:greetMeResponse" name="greetMeResponse"/>
    </wsdl:operation>
</wsdl:portType>
```

Définition des types de données

Définition des messages

Définition d'un port avec une opération

Définition des entrées et des sorties

FIGURE 2.6 – Exemple d'un fichier WSDL

La définition du service abstrait (l'interface d'un service) décrit d'une manière indépendante d'une plate-forme ou une technologie d'implémentation les fonctionnalités fournies par le service. Cette définition abstraite peut ultérieurement être instanciée par plusieurs implémentations concrètes. La définition du service abstrait contient la partie réutilisable de la définition d'un service, c'est-à-dire les opérations et les messages traités par le service. Les principaux éléments d'une description WSDL d'une interface de service sont `<types>`, `<message>`, `<portType>`, `<operation>`.

L'élément `<types>` d'un fichier WSDL permet de définir ou de référencer des définitions externes XML des types de données. Pour définir des types de données, les développeurs peuvent utiliser les types primitifs définis par XML (`integer`, `string`, `float`, `long`, `boolean`, `short`) ou définir leurs propres types de données complexes en utilisant ces types primitifs.

4.2 OSGi

OSGi est une spécification de plate-forme à services proposée par le consortium OSGi. Cette spécification décrit un framework s'appuyant sur Java comme plate-forme d'exécution. Elle avait pour but à l'origine de proposer une plate-forme pour les passerelles réseaux et résidentielles. Ces passerelles sont limitées en taille mémoire et en capacité d'exécution. Des enthousiastes ont proposé d'autres implémentations en différents langages, c'est le cas du projet Apache Celix.

La spécification OSGi décrit une plate-forme d'exécution Java qui supporte le déploiement d'applications extensibles et modulaires. Les applications, dans OSGi, sont représentés par un artefact appelé « bundle ». Les artefacts sont des objets JARs (de Java ARchive) qui contiennent des méta-informations qui le rend un « bundle OSGi ».

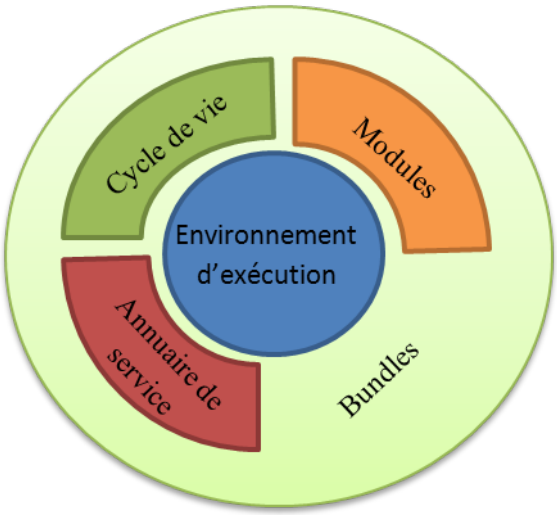

FIGURE 2.7 – Framework OSGi

La Figure 2.7 présente l'architecture d'une plateforme OSGi [All]. L'environnement d'exécution de cette architecture est la spécification de la machine virtuel Java (JVM). Comme exemple de JVM, nous pouvons citer machines basé sur les profiles Java comme J2SE (Java Standard Edition), CDC (Connected Device Configuration), CLDC (Connected Limited Device Configuration). Comme présenté dans la Figure 2.7, l'architecture d'une plateforme OSGi est composée de diverses couches, les plus importantes sont la couche de modules, la couche de cycle de vie et la couche de services.

4.2.1 La couche modules

La couche modules décrit les politiques de partage de classes entre les différentes bundles. Cette couche est celle qui permet de construire des applications modulaires et extensibles à l'exécution. Les modules, dans OSGi, sont les bundles qui contiennent une méta-information qui sera exploitée par cette couche modules. Cette méta-information contient les instructions de partage de ressources pour chaque bundle. C'est-à-dire, les classes que le bundle exporte ainsi que les

classes que le bundle importe.

4.2.2 La couche de cycle de vie

La couche de cycle de vie ajoute la possibilité d'installer, d'initialiser, d'arrêter et de mettre à jour les bundles dynamiquement. Cette couche permet de gérer des aspects dynamiques liés au déploiement.

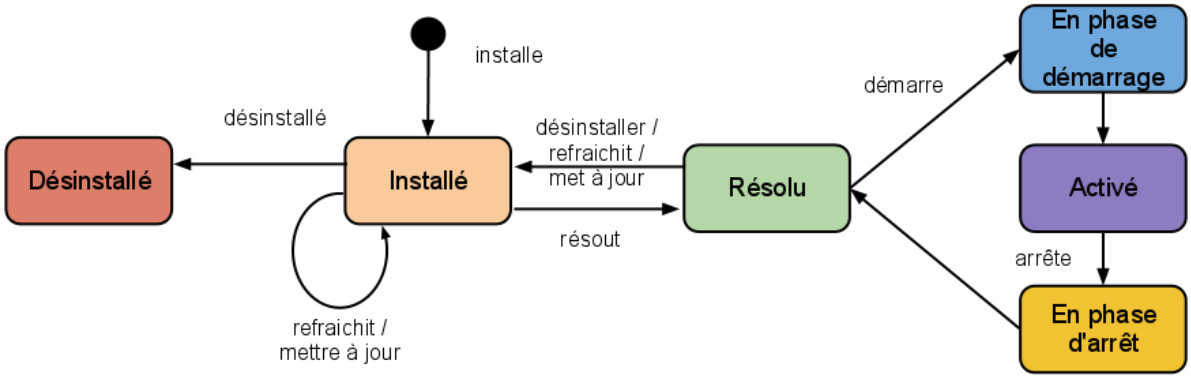

FIGURE 2.8 – Cycle de vie du bundle

La Figure 2.8 présente le diagramme d'état du cycle de vie de bundles OSGi. La couche de cycle de vie gère l'état de chaque bundle et elle permet d'effectuer les changements d'état à tout moment. C'est la couche de modules qui est importante pour gérer ces transitions. Ceci est dû au fait que la résolution d'un bundle est réalisée uniquement si toutes ses dépendances de ressources sont résolues par d'autres modules (*i.e.* bundles OSGi).

4.2.3 La couche services

La couche de services ajoute un annuaire de service. Cet annuaire permet de construire des applications ou des bundles qui prennent en considération le dynamisme. En d'autres termes, cette couche permet le partage d'objets, dit services, entre bundles. Chaque bundle peut enregistrer des objets, ou plus proprement dit des services, dans l'annuaire de services. La seule contrainte est de désenregistrer le service quand l'objet n'est plus disponible, par exemple quand le bundle qui le contient a été arrêté. La Figure 8 présente l'approche traditionnelle des services sur OSGi. Cependant, OSGi prend aussi en considération le dynamisme de services. Ceci grâce à la couche modules et à la couche de cycle de vie, qui permet l'ajout et la suppression de bundles à l'exécution. En d'autres termes, les bundles peuvent enregistrer des services et lors du départ du bundle, les services doivent être désenregistrés du registre.

Ces trois couches (modules, cycle de vie et services) permettent la construction d'applications dynamiques, modulaires et plus faciles à faire évoluer. Cependant, afin d'obtenir des applications avec ces caractéristiques, il est indispensable de considérer un bon découplage entre bundles

ainsi que de prendre en considération le dynamisme. Néanmoins, la prise en compte est à la fois complexe et répétitive. Des outils sont apparus pour répondre à cette complexité comme Service Binder [CH03, Off05] et iPOJO [EHL07, Hal08].

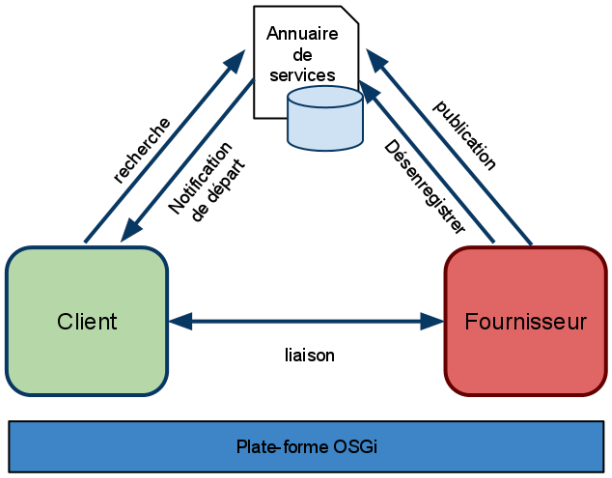

FIGURE 2.9 – SOA sur OSGi

La plate-forme OSGi est centralisée et son exécution est limitée à une machine virtuelle Java, de sorte que les fournisseurs de services, les consommateurs et l'annuaire (ou registre de services) soient dans le même espace de mémoire du JVM. De plus, les spécifications de services correspondent à des interfaces Java. Ainsi, les consommateurs ont des dépendances d'interfaces et non d'implémentations. Cependant, le mécanisme de communication entre le client et le fournisseur de service est fait sous la forme d'appels à des méthodes Java.

Le consortium OSGi a mis à disposition diverses spécifications autour d'OSGi. Notamment, il y a un ensemble de spécifications qui ciblent divers cas d'usage. Comme par exemple la spécification pour des serveurs http, d'autres pour l'interaction et la découverte de dispositifs physiques (capteurs et actuateurs), ainsi que pour l'importation et l'exportation de services distants[BLE10]. Ces services externes peuvent être localisés soit dans d'autres plates-formes OSGi, soit ils sont proxys de services d'autres technologies (UPnP [3], Jini [4], Services Web, etc).

3. Universal Plug and Play
4. Actuellement sur le nome Apache River

4.3 iPOJO

Le projet iPOJO[EHL07] est un modèle à composants orientés services. Ce modèle combine les avantages de deux approches à composants et à services. Ce projet a été développé au sein de l'équipe ADELE de l'université de Grenoble. Il est actuellement développé dans le cadre d'un sous-projet du projet Apache Felix, qui est lui-même une implémentation de la spécification OSGi R4. Cependant, le framework iPOJO n'impose pas l'utilisation de la plateforme Felix. C'est-à-dire, il est possible d'utiliser iPOJO dans d'autres implémentations OSGi comme Equinox [5] et Knopflerfish [6].

Le modèle iPOJO simplifie la complexité de développement d'applications qui s'exécuteront sur plateformes OSGi. Son approche consiste à faciliter la création d'applications en s'appuyant sur des concepts d'architectures à services dynamiques. Les contraintes d'iPOJO, principalement pour sa dépendance avec OSGi, sont : premièrement le langage de programmation imposé (Java) ; et deuxièmement, une exécution centralisée et locale.

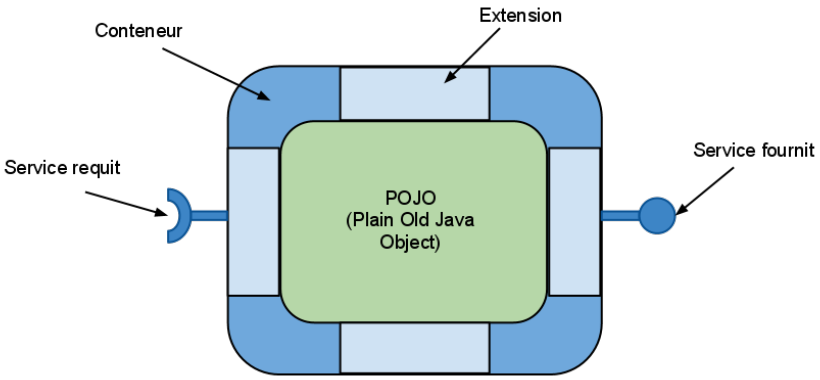

FIGURE 2.10 – Architecture d'un composant iPOJO

Le framework iPOJO repose sur la notion de conteneur (voir Figure 2.10), ce conteneur contient un objet de type POJO (Plain Old Java Object). Dans un traitement pendant la compilation, les classes Java sont « ipojonizes ». C'est-à-dire, les classes sont manipulées, à niveau du bytecode de Java, et instrumentées pour ajouter des extensions d'inspection et introspection à l'exécution. Néanmoins, cette manipulation reste transparente à la fois pour le développeur du POJO et aussi pour les possibles utilisateurs du composant.

Une des principales contributions d'iPOJO est son modèle extensible et la séparation de préoccupations ainsi obtenue. Les développeurs de composant se concentrent sur la logique fournie par le composant. En d'autres termes, le développeur du composant réalise la logique métier dans une classe Java sans prendre en considération les détails techniques comme le dynamisme et l'exportation ou la demande de services. C'est le conteneur qui est chargé de réaliser ces aspects techniques à l'exécution.

5. Eclipse Equinox : http ://www.eclipse.org/equinox/
6. Knopflerfish http ://www.knopflerfish.org/

Les caractéristiques de ce modèle à composant basé sur des services iPOJO sont les suivantes :

- un modèle et une machine d'exécution : iPOJO propose un modèle et une machine d'exécution fortement liés. Ceci a pour but de maintenir certaines informations du modèle de composants à l'exécution afin de les exploiter et de réaliser des opérations lors de l'exécution en respectant toujours le modèle ;
- une architecture à services dynamique et isolation : iPOJO permet de considérer toutes les caractéristiques d'une architecture à services dynamiques (grâce à OSGi). Néanmoins, iPOJO va plus loin et propose des mécanismes d'isolation de services dynamiques qui seront utilisés dans un contexte fermé ;
- un modèle de développement simple : iPOJO propose de simplifier autant que possible le développement de composants. En fait, tous les détails techniques de l'approche à services, ainsi que du dynamisme associé, ne font pas partie du développement du composant. En plus, grâce à une approche de tissage à la compilation et à l'exécution, ces détails techniques sont chargés par le framework ;
- un langage de composition structurelle : iPOJO propose un langage de composition structurelle basé sur XML. La différence entre ce type de composition et une composition traditionnelle est principalement que les liaisons dans iPOJO ne sont pas décrites en termes d'implémentation du composant requis, mais en termes de spécification d'interface de service requis. Ainsi, iPOJO étend les mécanismes pour réaliser des compositions et s'appuie aussi sur des aspects (de Aspect-oriented Programming) ;
- des fonctionnalités d'introspection et reconfiguration dynamique : L'objectif principal d'iPOJO est de construire des applications dynamiques, voir dans des contextes dynamiques et mobiles. Néanmoins, afin de réaliser des opérations de reconfiguration, il est important de connaitre certains détails de l'application à l'exécution. iPOJO propose des mécanismes d'introspection pour connaitre ces détails à l'exécution et si besoin, de réaliser des opérations de reconfiguration pour mieux s'adapter au contexte à l'exécution ;
- des mécanismes d'extension : une autre caractéristique est la capacité d'extension. Cette capacité est double : premièrement, iPOJO permet d'étendre les capacités non fonctionnelles qui seront traités à l'exécution. Comme exemple nous pouvons citer des capacités de logging, de persistance et de sécurité. La deuxième capacité d'extension permet de définir de nouveaux types de composants. Comme exemple nous pouvons citer les composants composites, qui sont une composition isolée, sous la forme d'un composant.

Ces caractéristiques font d'iPOJO une option pour faciliter le développement d'applications dynamiques sur OSGi. La plateforme OSGi commence à gagner du terrain dans l'industrie et dans l'académie, et dans ce cadre, plusieurs acteurs ont choisi iPOJO.

L'extensibilité et la modularité des composants iPOJO est possible grâce à un concept ajouté appelé « handler ». Ces éléments sont en charge de réaliser des opérations non-fonctionnelles des composants, comme la publication et découverte de services. En plus, ces « handlers » sont construits comme des composants iPOJO, ceci donne comme résultat une extensibilité à plusieurs niveaux et permet la réutilisation des « handlers ».

Grâce à ces extensions, iPOJO permet de définir explicitement les stratégies de sélection de services. Ces stratégies permettent d'obtenir, à la conception ou à l'exécution, le service désiré. De telles stratégies peuvent se fonder sur des aspects fonctionnels mais aussi sur des propriétés non fonctionnelles. Le choix d'un service est fait sur des propriétés comme la qualité de service que le service offre, la bande passante ou sur des informations de déploiement (même machine, réseau local, externe). Lorsque la sélection est retardée à l'exécution, il est possible de prendre en compte le contexte d'exécution.

Au cours du temps, les stratégies peuvent évoluer. En utilisant OSGi, par exemple, les services peuvent être choisis selon un classement [BH07] calculé à l'exécution. On parle de « service ranking ». iPOJO propose des mécanismes au sein du modèle à composant permettant de donner explicitement une « valeur » à chaque service utilisable lors de l'exécution. De plus, si un nouveau service avec un meilleur potentiel apparait, iPOJO permet de faire le changement de façon transparente pour le consommateur. D'autres travaux se basent sur la qualité de services [PPMM11, BdMAS09] pour faire des adaptations à l'exécution.

Dans les mécanismes de résolution de dépendances de base, iPOJO propose les modèles suivants :

- simples ou agrégées ;
- obligatoires ou optionnels ;
- filtrés ;
- triés ;
- dynamiques, statiques ou à priorité dynamique.

Il y a eu récemment d'autres travaux pour étendre ces mécanismes. Notamment, dans [TDR08], il est proposé des « handlers » qui supportent la coupure temporaire de services. Ces extensions permettent configurer un délai d'attente pendant une coupure de service. Une autre extension [Oza10], permet d'ajouter des métriques d'évaluation de services pendant son utilisation. Ceci permet à chaque consommateur de qualifier les services pour sa future utilisation.

5 Integration de services

5.1 Introduction

L'informatique orientée service est un domaine relativement jeune dans lequel il reste encore certains points durs importants. L'intégration de services, notamment, est un aspect crucial qui suscite de nombreux travaux. En effet, si l'approche à service facilite l'intégration d'applications ou d'équipements grâce à sa souplesse et à l'utilisation de protocoles standards de communication, la problématique d'intégration n'est pas entièrement résolue. Dès que l'on sort de domaines fermés bien maîtrisés, il est nécessaire d'effectuer des opérations de médiation au sens large du terme pour pouvoir effectivement appeler un service dans de bonnes conditions.

De nombreux projets proposent d'ajouter de façon plus ou moins transparente du code coté client pour utiliser au mieux un service. Ce code prend souvent la forme d'un proxy qui prend en charge un aspect non fonctionnel lié à la médiation et qui est créé statiquement ou dynamiquement. Les propositions diffèrent en fonction des aspects traités, des méthodes de création des proxies, du dynamisme de l'approche, etc.

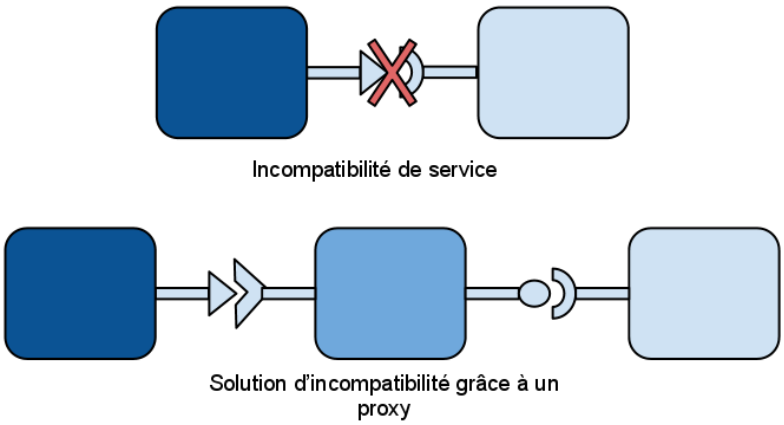

FIGURE 2.11 – Intégration à base de proxies

Le but de cette section est d'étudier les approches à base de proxies. Nous verrons de façon détaillée dans le chapitre suivant des approches à base de middleware pour l'intégration au sein d'applications à services.

Le principe de ces travaux est simple. Chaque problématique d'intégration, c'est-à-dire un aspect non-fonctionnel, est spécifiée dans un formalisme adapté (un langage, un modèle, etc.), puis il y a un processus de génération de code ou de tissage entre le code métier et du code non-fonctionnel qui résout la problématique d'intégration.

5.2 Secure FOCAS

L'incompatibilité entre un consommateur et un fournisseur de services peut être sémantique ou syntaxique. Cependant, d'autres aspects doivent être pris en compte pour un usage correct des services. Secure FOCAS[CLB08] est un projet qui propose de résoudre l'incompatibilité due à un aspect non-fonctionnel assez important : la sécurité. Ce travail est une extension d'un projet de modélisation et d'exécution d'activités composées sur un approche par procédés, appelée orchestration. L'approche de ce travail s'appuie sur un des principes du génie logiciel : la séparation des préoccupations. L'innovation réside en la séparation entre les modèles d'orchestration (métier) et, le modèle non-fonctionnel (modèle de la sécurité). Ces deux modèles sont décrits pendant l'étape de développement. Un expert du métier défini le modèle d'orchestration, pendant qu'un autre expert de sécurité est en charge de modéliser les aspects de sécurité.

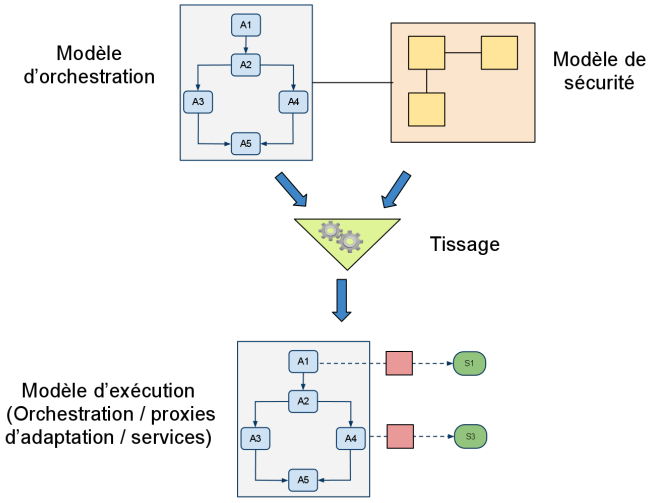

FIGURE 2.12 – Approche de Secure FOCAS

La Figure 2.12 présente la vision globale de l'approche de Secure FOCAS. Un fois que les deux que modèles sont construits par les experts (du métier et de la sécurité), un processus de tissage est réalisé dans le but de générer les proxies d'adaptation à la politique de sécurité. Ce tissage prend en compte les deux modèles pour construire ces proxies. Le résultat est un modèle exécutable de l'orchestration instrumenté pour communiquer avec les services sécurisés.

Ce travail aborde une problématique de poids dans l'industrie à savoir la sécurité. En plus, la séparation de préoccupations et les liaisons entre modèles avec des objectifs indépendants offrent une solution élégante au problème d'intégration. Les auteurs se proposent de traiter d'autres aspects récurrents lors de l'intégration en se basant sur le même approche, tel que la surveillance, le logging et la persistance. Tout en gardant un modèle métier, dit d'orchestration, et en ajoutant d'autres modèles non-fonctionnels.

Même si cette approche arrive à résoudre un problème d'intégration de façon élégante, la composition de plusieurs modèles non-fonctionnels avec un seul modèle d'orchestration reste en pratique un tâche lourde et difficile à concevoir, voire dans certains cas, impossible.

5.3 TACC Proxies

Les problématiques d'intégration dues à l'hétérogénéité des dispositifs ainsi que de services est une des principaux problèmes récurrents dans l'ubiquitaire. L'approche TACC (pour transformation, aggregation, caching et customization)[FGCB98]présente une solution pour l'interaction entre des clients hétérogènes et des services d'Internet. L'intérêt de cette approche est de fournir un proxy qui soit capable d'adapter le flot d'information du service à un client (dispositif) avec des limitations, comme exemple d'affichage.

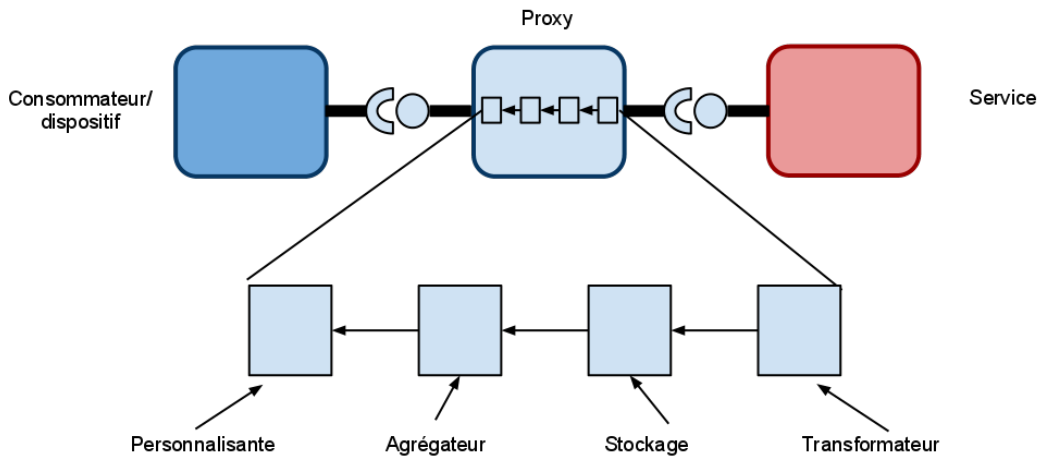

FIGURE 2.13 – TACC Proxies

La solution cible des services d'Internet avec des caractéristiques permettant de faire :

- de la transformation, comme le filtrage, la conversion de format de donnée, la compression avec perte, etc. ;
- de l'agrégation, comme la collecte d'information de diverses sources, comme les moteurs de recherche ;
- du stockage, pour le contenu original aussi comme pour le contenu transformé ;
- de la personnalisation, pour maintenir un base de donnés pour gérer les préférences des utilisateurs.

La composition d'applications (voir Figure 2.13) consiste à enchainer plusieurs travailleurs (workers dans TACC) qui réalisent des tâches spécifiques, comme la transformation d'un type de donnée vers un autre. Cet enchainement de tâches permet de réaliser l'alignement du flot d'information qui transite du service vers le client. Cet travail traite surtout de l'alignement vers dispositifs avec des contraintes de bande passante ou d'affichage[PLF+01]. Cette approche ne prend pas en considération l'intégration et la composition de divers services. Cependant, les auteurs proposent la même approche d'enchainement de tâches dans d'autres projets comme Paths[KF00], qui est la machine tournante de SWORD[PF02]. SWORD permet de réaliser une composition de services en considérant des conditions d'entrée et de sortie pour chaque service.

5.4 SCENE

Le travail sur SCENE (*Service Composition Execution Environment*)[CNM06] présente un langage de règles qui étend le langage de composition par procédés BPEL. Ce langage de règles est basé sur DROOLS[dro]. Dans ce langage, il est spécifié le déclenchement de scripts au moment où certaines activités sont exécutées. Un script peut, par exemple, reconfigurer la composition faite en BPEL.

Cette approche permet d'étendre une composition par procédés afin d'identifier les activités statiques et les activités dynamiques. Les activités statiques sont spécifiées pendant le développement ainsi que les ressources que cette activité utilise (*e.g.* un service externe). Les activités dynamiques par contre, sont instrumentées pendant le développement (avec un fichier de script) afin d'exécuter des instructions pour localiser dynamiquement les services concrets.

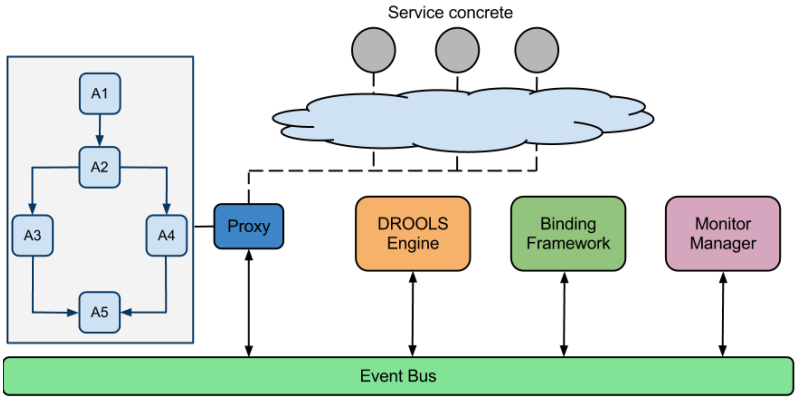

FIGURE 2.14 – 13 Architecture de la plate-forme SCENE

Les activités instrumentées avec des règles SCENE interagissent avec un proxy (voir Figure 13) instancié pour chaque activité. Si le proxy ne contient pas un service concret lors de son exécution, il envoi un évènement qui est intercepté par la machine de règles (« *monitor manager* »). S'il existe une règle valide pour cette activité, il peut déclencher l'exécution du « binding framework » pour réaliser la liaison avec un autre service concret.

Ce projet ne traite pas directement la problématique d'intégration. En revanche, il adresse un autre problème qui est celui de l'adaptation à l'exécution. Cette approche permet de réaliser la sélection de services selon certaines propriétés applicatives (comme la localisation). De plus, l'ensemble d'éléments de l'architecture SCENE permet d'intégrer un consommateur et un fournisseur de services grâce à la possible restructuration de la composition BPEL. SCENE est un composant d'un framework pour la composition de services Web adaptables[BNGG07]. En particulier, en utilisant la même approche, et certains technologies en commun, d'autres auteurs de la même institution présentent des solutions d'auto-récupération de flots d'exécution (compositions procédural)[BGG04, BGP07, BG05]. L'approche est similaire : une machine de règles est déclenchée pour certaines activités, puis les actions restructurent la composition pour s'auto-récupérer. Donc, ces nouveaux travaux adressent l'incompatibilité entre le client et le fournisseur

de services, en modifiant l'architecture de la composition.

5.5 Travaux avec ontologies

La communauté scientifique, dans l'ère de l'informatique, a récemment fait l'usage d'ontologies pour formaliser la description de l'information sur les artefacts logiciels fabriqués[ZDP09]. Cette utilisation d'ontologies permet d'enrichir les artefacts afin d'exploiter plus facilement cette connaissance par des entités logicielles ainsi que par des humains. Dans le contexte de l'intégration d'applications, cette approche a été utilisée pour traiter la problématique de l'hétérogénéité sémantique de services. La proposition consiste à enrichir avec des informations sémantiques les services et les publications, ainsi que les demandes de services. Ensuite, lors de la recherche d'un service, un processus de correspondance localise dans l'annuaire le service qui satisfait au mieux les besoins du client. Plus tard, pendant l'invocation de services, un processus de médiation transforme les messages envoyés par le client, en utilisant les ontologies du client, vers une version compréhensible par le fournisseur de service, dites ontologies du service. Finalement, la réponse suit un chemin de transformation inverse. La Figure2.15 montre une architecture générale d'une approche d'intégration à base d'ontologies. L'agent réalise les opérations de correspondance et de transformation de messages en utilisant les ontologies du client et du service.

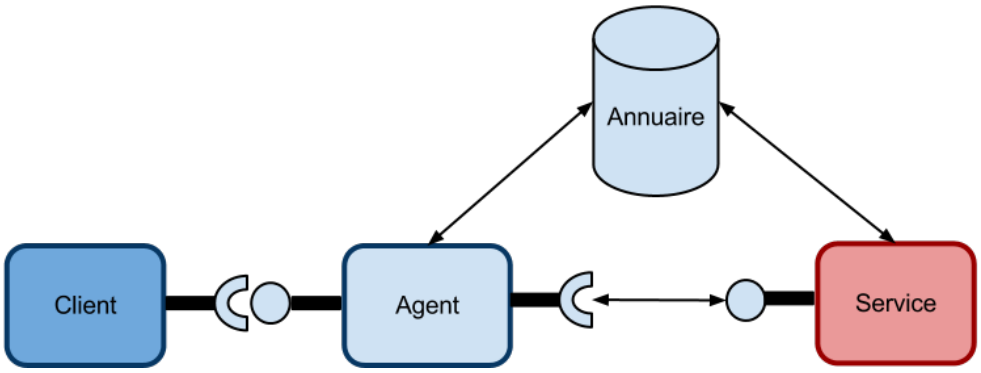

FIGURE 2.15 – Architecture d'intégration en utilisant ontologies

Il y a plusieurs travaux basés sur cette approche. Les différences résident principalement dans l'architecture de l'infrastructure ou du broker, ou dans les outils et techniques utilisés pour décrire les ontologies. Le projet WSMX[HCM+05] par exemple, fournit une architecture capable de gérer les services Web traditionnels, ainsi que les services Web sémantiques. L'architecture est divisée en deux aspects principaux : le registre de services et l'exécution de services. Dans le registre de services sont spécifiés les ontologies, les médiateurs et les objectifs du service. Lors de l'exécution de service, une découverte de service est effectuée pour répondre au mieux aux besoins du client (correspondance sémantique) ; ensuite l'invocation vers ce service est effectuée. Les médiateurs décrits précédemment sont utilisés pour réaliser correspondance ontologique du consommateur et du fournisseur de service.

Un autre travail est IRS-II[Cab05]. L'architecture de cette dernière présente un ensemble de

médiateurs et leur interaction qui sont chargés d'effectuer trois tâches spécifiques. Les médiateurs de processeur en charge de l'invocation de service ainsi que de la résolution de conflits pour faire l'invocation. Les médiateurs d'objectifs chargés de résoudre les conflits lors de la sélection de service. Finalement les médiateurs de données qui sont chargés de résoudre les conflits en relation avec les ontologies.

5.6 Synthèse

L'approche à services est clairement un candidat idéal pour aborder la problématique d'intégration de systèmes hétérogènes. Ceci grâce au faible couplage obtenu entre le consommateur et le fournisseur ce qui permet de cacher les détails d'implémentation dans les deux directions. Néanmoins, cette approche en elle-même ne peut pas résoudre cette problématique. Lors de l'usage et la conception d'architectures basées sur des services, des solutions d'intégration de services ont été soumises. L'approche la plus utilisée est l'usage de proxies d'intégration qui sont chargés de réaliser des opérations d'alignement des messages qui transitent entre les entités à intégrer. Entre ces approches, comme nous avons présenté, il existe des solutions pour traiter des problèmes spécifiques. Par exemple le projet Secure FOCAS vise l'alignement de propriétés non-fonctionnelles (la sécurité). Alors que TACC vise l'intégration d'un type spécifique d'applications : les services d'Internet. D'un autre coté, SCENE par exemple, aborde une problématique importante de nos jours : l'adaptabilité d'applications à l'exécution. Néanmoins, la restructuration de la composition pour résoudre des détails d'interopérabilité ne semble pas une bonne séparation de préoccupations. Finalement, les travaux d'ontologies résolvent l'hétérogénéité sémantique des services. Néanmoins, la complexité est déplacée lors de la définition des ontologies ainsi que des règles qui permettent de faire l'appariement et la transformation d'ontologies.

Ils existent plusieurs approches et travaux qui abordent la problématique d'intégration de services hétérogènes. Dans cette section nous montrons des travaux qui traitent des problèmes concrets à base d'intermédiaires ou proxies. Dans le chapitre suivant, nous allons montrer des architectures d'intégration qui présentent des infrastructures permettant de composer les opérations d'intégration.

6 Synthèse

L'informatique orientée service est un domaine relativement récent et de nombreux défis restent encore à relever. Cela explique certainement le nombre de conférences internationales abordant la notion de services, parfois de façon exclusive tels que SCC, ICWS ou ServiceWave, ainsi que le nombre de projets et travaux en cours [7].

L'approche à services est un style architectural pour la construction d'applications utilisant des entités logicielles faiblement couplées. Cette approche repose sur des liaisons de type client/serveur entre consommateurs et fournisseurs de service. Un principe de base de cette approche consiste à ne pas montrer les détails d'implémentation du service aux consommateurs de sorte que la description du service est la seule connaissance à partager. Cette description doit être autosuffisante : elle décrit les capacités d'un service et comment l'utiliser.

Il existe principalement deux approches de composition pour construire des applications en utilisant des services. La première approche appelée « par procédés » repose sur un langage décrivant l'ordre et le flux d'exécution de services. Cette approche s'appuie sur une machine d'exécution qui dirige l'exécution de la composition. La deuxième approche appelée « composition structurelle » consiste à assembler entre eux les services sélectionnés. Dans cette approche, la logique d'exécution de l'application réside dans la logique métier du composant consommateur.

Plusieurs technologies se basent sur l'approche à services. La technologie des Services Web est certainement la plus utilisée. Cette technologie s'appuie sur des standards Internet de communication et de format de données pour la communication avec des services distants. La technologie présentée par l'alliance OSGi propose un modèle basé sur les services dynamiques dans le contexte d'une machine virtuelle Java.

L'importance de l'approche à services est double. D'abord, elle permet de construire des applications en utilisant des briques logicielles avec un certain degré de découplage entre elles. Ensuite, avec des primitives de dynamisme, elle rend possible le développement d'applications pouvant plus facilement évoluer à l'exécution en fonction de la disponibilité des services.

Néanmoins, il y a des défis, toujours d'actualité, qui sont traités par la communauté scientifique. Ces défis sont liés aux nouveaux domaines d'application, tels que la domotique et l'informatique ubiquitaire. Ces nouveaux domaines ajoutent des exigences de dynamisme et de mobilité, qui doivent être prises en compte lors de la composition d'applications, voire lors de l'intégration de services. Lors de l'intégration de services, la couche d'intégration doit apporter la souplesse nécessaire permettant de supporter des adaptations à l'exécution.

Dans ce chapitre nous avons abordé au début les concepts de base sur l'approche à service ainsi que la contribution de cette approche dans les environnements dynamiques. Pour finir, nous avons vu les défis actuels autour de l'intégration de services et l'adaptabilité comme solution aux problèmes liés au dynamisme. Dans le chapitre suivant, nous abordons plus en détail les différentes architectures d'intégration.

7. voir `http://cordis.europa.eu/fp7/ict/ssai/home_en.html`

3

Architectures d'intégration

Sommaire

1 **Intégration** 60
2 **Architectures d'intégration** 63
3 **Patrons d'intégration** 75
4 **Solutions existantes** 82
5 **Synthèse** 92

Pour que les architectures à services se mettent en place à grande échelle au sein des entreprises, il est nécessaire de disposer des techniques et outils capables de gérer l'intégration de services hétérogènes. La couche logicielle chargée de cette l'intégration, souvent appelée couche de médiation ou d'intermédiation, accueille tous les traitements nécessaires à la communication effective entre applications non préparées à cela.

Le but de ce chapitre est de définir la notion de médiation, d'examiner les différents styles d'architectures définis jusqu'à aujourd'hui ainsi que les patrons de conception associés. Il s'agit également d'étudier les intergiciels disponibles aujourd'hui pour la mise en place de médiation dans les architectures à services.

1 Intégration

1.1 Introduction

L'intégration d'applications a toujours constitué un des domaines les plus importants en informatique. Le besoin en intégration apparaît dès lors que l'on veut construire de nouvelles applications en utilisant des applications existantes accessibles depuis un réseau. Ces applications déjà présentes sont souvent appelées applications « legacy ».

L'activité d'intégration est récurrente ; elle devient inéluctable dès lors qu'une entreprise souhaite étendre son système d'information de façon à fournir de nouveaux services, internes ou externes. Ces nouveaux services doivent en effet généralement interopérer avec les applications en place afin de s'appuyer sur les données d'entreprise et respecter les procédures et usages internes. Il est bien évidemment impensable de dupliquer, ou de redévelopper, les applications existantes à chaque création de nouvelle application.

L'activité d'intégration est devenue de plus en plus prépondérante ces dernières années. On demande en effet aux Systèmes d'Information une souplesse et une évolutivité accrue afin de permettre la création de nouveaux services de plus en plus fréquemment et de plus en plus rapidement. Cette activité est également stressée par l'explosion des applications pervasifs. Ici, une des exigences premières est d'être capable d'intégrer des capteurs, des applications, des services distants (sur le Web par exemple) pour fournir des services de plus haut niveau. Ce type d'intégration, fondamental aujourd'hui, est compliqué par le fait que les éléments à intégrer sont dynamiques et souvent très hétérogènes.

L'intégration d'applications est une activité complexe pour une raison simple : les applications ou les équipements devant être utilisées par une nouvelle application n'ont pas été conçus pour cela, ni même, en général, pour simplement communiquer entre eux. Il en résulte de fortes différences dans la manière de représenter les données, l'absence d'interfaces appropriées, l'utilisation de protocoles de communication différents, la manipulation de données communes avec absence de moyens de synchronisation, des évolutions à des rythmes différents...

L'approche orientée service représente désormais un support privilégié pour l'intégration d'applications. Les services Web en particulier permettent en effet de résoudre en partie les problèmes liés aux protocoles de communication en fournissant un protocole unificateur basé sur les technologies Internet. Ceci représente certes une partie limitée des problèmes liés à l'intégration d'applications mais permet néanmoins d'aller vers plus d'uniformité dans la description et la communication des applications d'entreprise.

Ceci dit, pour que les architectures à services se mettent en place à grande échelle au sein des entreprises, il sera nécessaire de disposer des outils adaptés à la gestion de l'intermédiation. Cette couche logicielle, qui accueille tous les traitements nécessaires à la communication effective entre applications non préparées à cela, est depuis toujours le nerf de la guerre.

1.2 Opérations d'intégration

Le rôle de la couche d'intégration entre applications est de mettre en place des opérations permettant la communication, la compatibilité, la synchronisation entre les différentes applications devant être mises en relation.

Pratiquement, la médiation met en œuvre des traitements tels que :

- la communication. Le but premier de cette fonction est de permettre à des applications utilisant des protocoles de communication différents d'interopérer. Il s'agit ainsi d'assurer des transformations de protocoles au sens d'un pont réseau. Il s'agit également d'améliorer le découplage entre applications en masquant, par exemple, les adresses réseaux des applications qui sont alors identifiées par un nom symbolique. L'approche à services sous-tend ce genre de politique, où les liaisons aux services se font au dernier moment.
- l'alignement syntaxique. Il s'agit ici d'assurer l'utilisation d'un même format de données par les applications à intégrer. Cela peut se faire à la volée lors de chaque communication d'application à application ou en passant par un format pivot. Selon cette dernière approche, toutes les données échangées sont transformées en un format unique ; cette approche réduit le nombre de transformations à effectuer puisque le nombre de transformateurs est égal au nombre d'applications à intégrer.
- l'alignement sémantique. Il s'agit ici de rapprocher des applications qui ont des divergences plus profondes liées à des définitions différentes de la sémantique de certaines fonctions et comportements. Cet alignement est beaucoup plus difficile à réaliser et repose souvent sur des notions émergentes de liens entre ontologies.
- l'ajout de propriétés non fonctionnelles. Il est souvent important pour répondre aux besoins d'intégration initiaux d'assurer certaines qualités telles que la sécurité ou la disponibilité. Ces propriétés ne peuvent pas être entièrement assurées par les applications qui restent souvent fermées. Il est dès lors nécessaire de les traiter en partie au niveau de la couche d'intermédiation.
- la persistance de données. Il est important de garder trace de tous les échanges réalisés entre les applications. La couche de médiation peut prendre en charge le logging de toutes les requêtes, données, réponses qui transitent. Encore une fois, pour des raisons de séparation claire des aspects, il est préférable d'assigner cette tâche à la couche de médiation et non aux applications elles-mêmes.
- le suivi des échanges à des fins de supervision. Il s'agit ici de récolter des données au niveau de la couche de médiation afin d'alimenter des systèmes de supervision qui sont utilisés pour vérifier que la qualité de service (latence, débit...) attendue est bien atteinte.
- l'introduction de code métier. La couche d'intermédiation peut servir à ajouter du code métier comme, par exemple, accéder à une base de données pour extraire les données nécessaires à l'invocation d'un service. Cette approche, qui peut s'avérer très pratique, est toutefois à éviter ; elle introduit une grande confusion dans l'architecture où les notions de code métier et de code technique se mélangent.

1.3 Besoin en adaptation

La couche d'intermédiation nécessaire à l'intégration d'applications doit répondre aujourd'hui à des besoins toujours plus complexes. Ceci est dû à l'émergence de nouveaux domaines et de nouveaux contextes d'intégration demandant notamment des qualités importantes de dynamisme.

En particulier, il est aujourd'hui de plus en plus demandé de gérer deux formes d'évolution très fréquentes présentées ci-après :

- Evolution des interfaces des applications à intégrer. Ces évolutions traduisent généralement des modifications fonctionnelles ou non fonctionnelles des applications. Elles peuvent aussi correspondre à de simples évolutions techniques (passage de RMI aux services Web par exemple).
- Evolution des besoins non fonctionnels. Ces évolutions traduisent souvent de nouvelles exigences concernant les applications intégratrices. Par exemple, on peut demander la mise en place de nouvelles propriétés de sécurité.

Pour expliciter ces deux formes d'évolution, mettons-nous dans un contexte d'architectures à services. Le premier cas, celui lié à l'évolution des interfaces, correspond à une modification du contrat de service. En effet, l'évolution porte sur la façon d'appeler le service ou sur certaines des propriétés afférentes. Cette évolution demande au consommateur de s'adapter au nouveau service, ou plus précisément à sa nouvelle interface. Répondre à ce problème demande de modifier la couche d'intermédiation de façon à prendre en compte les nouvelles interfaces. L'approche la plus communément utilisée est de faire un adaptateur qui résolve les changements d'interface lors de l'exécution. Un problème lié à ce type d'adaptation est de prendre connaissance des évolutions d'interface. La notification de ce type d'évolution n'est pas toujours faisable facilement et peut induire sans un coût supplémentaire important pour certaines technologies. Comme exemple, le travail [FAB06]utilise des médiateurs pour faire l'alignement d'interfaces à l'exécution. Ainsi, les ontologies [DL08, JWY+09, JWY+10] ont été utilisés pour attaquer ce problème d'incompatibilité d'interfaces.

La seconde forme d'évolution concerne la modification de certaines propriétés d'intégration. La qualité de l'intégration elle-même peut effectivement évoluer. Par exemple, il peut être nécessaire de sécuriser différemment une composition de services, de mettre en place des traces (des « logs »), d'ajouter des mécanismes améliorant la disponibilité, etc. Il faut comprendre que les besoins en qualité logicielle sont parfois très contextuels et donc soumis à une forte dynamique. Cela demande de mettre en place de telles évolutions de façon très réactive et parfois de manière fréquente.

Les deux types d'évolution présentés ici demandent de modifier la couche d'intermédiation, c'est-à-dire les opérations de médiation présentées précédemment. De plus, il est de plus en plus nécessaire d'effectuer ces modifications de façon dynamique, c'est-à-dire sans interrompre les traitements d'intégration en cours.

2 Architectures d'intégration

2.1 Principes

Comme nous l'avons indiqué précédemment, le problème de l'intégration d'applications (ou d'équipements) est déjà ancien. Les applications patrimoniales (« legacy applications ») sont au cœur de la problématique d'intégration. Les entreprises préfèrent bien évidemment utiliser ces applications existantes plutôt que d'en construire de nouvelles. Ce choix entre réutiliser ou ré-implémenter une application se fonde principalement sur deux facteurs :

- le coût de développement ainsi que les risques liés au remplacement d'un système qui fonctionne ;
- le coût de migration de l'information d'une application ancienne vers une nouvelle.

Historiquement, la plupart des travaux sur l'intégration ont cherché à créer une couche explicite permettant de séparer les applications (équipements) à intégrer. En effet, laisser les opérations d'intégration au sein des applications devant interopérer ne passe pas à l'échelle et est difficile, voire impossible, à maintenir.

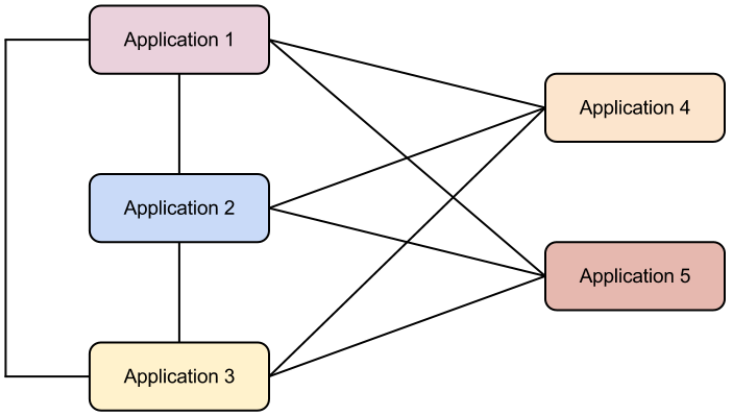

FIGURE 3.1 – Intégration point à point

Comme illustré par la Figure 3.1 ci-dessus, le couplage entre applications peut devenir très important. Le nombre de lien nécessaire pour avoir une connectivité complète dans un environnement d'intégration comprenant n applications est donnée par la formule suivante :

$$\frac{n(n-1)}{2}$$

Dans des environnements réels, notamment pervasifs, où les applications et équipements à intégrer se comptent par dizaine, le nombre de connexions devient prohibitif.

La solution très rapidement proposée par les acteurs de l'intégration d'applications (IBM par exemple) a été de placer un élément informatique entre les différentes applications. Ceci est illustré par la Figure 3.2. On voit tout de suite que le nombre de liens nécessaires tombe à n pour un environnement d'intégration constitué de n applications. Surtout, les évolutions ne concernent

que les liens entre applications mises à jour (interface, propriétés non fonctionnelles) et l'élément de centralisation.

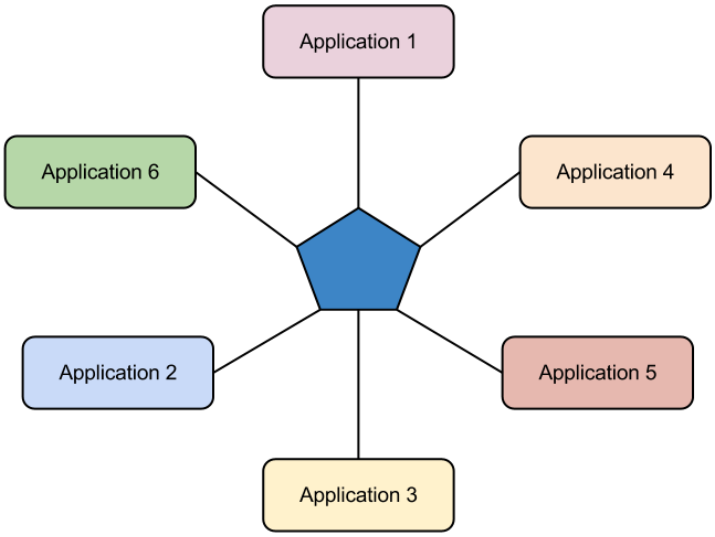

FIGURE 3.2 – Intégration centralisé

Le rôle de l'élément centralisateur est de mettre en place les différentes opérations de médiations nécessaires à l'interopérabilité entre les applications. Dans le pire des cas, on peut avoir besoin de développer $\frac{n(n-1)}{2}$ connecteurs comme dans la solution d'intégration point à point. Cependant, l'approche centralisée offre les avantages suivants :

- un faible couplage entre applications ;
- un seul point de configuration et de maintenance ;
- l'usage de services techniques au sein de l'infrastructure d'intégration ;
- le non duplication de services techniques ;
- l'ajoute d'opérations non-fonctionnelles dans la couche d'intégration.

Cette approche est désormais une solution architecturale pour la construction d'infrastructures d'exécution, qui sont dédiées à l'intégration d'applications. En effet, la plupart des architectures d'intégration reposent sur cette approche. Ainsi, diverses solutions *middlewares*, comme infrastructure d'intégration, ont été proposées et utilisées comme la couche répondant aux problématiques d'intégration.

Par la suite, nous montrons les diverses approches à base de *middlewares* qui sont utilisées comme infrastructures d'intégration. Premièrement, nous montrons les *middlewares* de communication, notamment les *middlewares* de messagerie et leur usage comme infrastructure d'intégration. Deuxièmement, les solutions EAI (*Enterprise Application Integration*). Ensuite, les *middlewares* d'intégration à services, notamment les ESB (*Enterprise Service Bus*). Pour finir, nous montrons des travaux qui ont introduit les architectures à base de médiateurs lors de l'intégration, principalement pour l'intégration de données. Nous montrons cette dernière approche, car il sert comme base lors de l'assemblage d'une solution d'intégration.

2.2 *Middlewares* de communication

La première approche proposée pour mettre en place cette architecture reposait sur la notion de *middleware* de communication. Le mot *middleware* est un terme très générique[Nau68] désignant toute couche informatique située entre des applications et des ressources (OS, réseau) et offrant des services techniques facilitant le développement, déploiement ou exécution des applications.

Les *middlewares* ont été très largement utilisés en support des systèmes distribuées et possiblement hétérogènes[Ber96, Ber93]. Selon [Gho06], un « système distribué » est une application avec les caractéristiques suivantes :

- premièrement, le système est composé de plus d'un processus. Chaque processus gère son propre fil d'exécution ;
- deuxièmement, il existe un mécanisme de communication entre ces processus ;
- puis, chaque processus doit avoir son propre espace de mémoire ;
- et finalement, il existe une interaction entre les processus afin de réaliser un but en commun.

Ces caractéristiques ne limitent pas un « système distribué » comme un système reparti dans un réseau. Ainsi, un « système distribué » peut être exécuté dans une seule machine et la communication peut être sous n'importe quel mécanisme de « communication inter-processus ». (IPC en anglais pour *Inter-process communication*) Néanmoins, l'usage plus commun est de situer les « systèmes distribués » dans un réseau où les processus s'exécutent sur plusieurs machines.

L'intérêt des solutions *middleware* est donc de faciliter le développement d'applications, en fournissant des abstractions, qui masquent l'hétérogénéité et la distribution, voire la répartition des processus, ainsi que la complexité programmatique de bas niveau.

Les MOM (acronyme de *Message Oriented middleware* en anglais) sont des *middlewares* qui permettent l'échange de messages dans les systèmes repartis. Les fonctionnalités principales d'un MOM sont les suivantes[EFGK03] :

- Transport de messages entre processus : les messages se caractérisent pa un en-tête technique propre à la technologie employée et des données transportées par le *middleware* et qui peuvent être formatée de diverses façons ;
- l'application émettrice d'un message et l'application réceptrice du message n'ont pas besoin d'être actives en même temps. La file d'attente reçoit le message de l'application émettrice et le stocke jusqu'à ce que l'application réceptrice vienne lire le message ;
- routage : certains MOM permettent de faire de routage pour livrer les messages. Le routage peut être de deux types, routage de niveau transport : pour transmettre les messages vers le réseau et communiquer avec autres MOMs, ou routage applicatif, pour identifier le composant qui va recevoir le message ;
- transformation : certains MOM permettent de faire des opérations de transformation du message qu'ils transmettent, en d'autres termes, le consommateur et le producteur de messages peuvent être conçus pour produire et traiter messages dans différents formats ;

– persistance : certains MOM font la persistance des messages qui sont en transit, quand les consommateurs ne sont pas exécutés en même temps que le producteur, le MOM peut stocker les messages pour sa future livraison.

Les MOM ont deux modes de fonctionnement principaux :

– Point à point : une application produit des messages et une application les consomme. Les messages ne sont lus que par un seul consommateur. Une fois qu'un message est lu, il est retiré de la file d'attente.
– Publication/souscription (par abonnement) : les applications consommatrices des messages s'abonnent à un *topic* (sujet, catégorie de messages). Les messages envoyés à ce *topic* restent dans la file d'attente jusqu'à ce que toutes les applications abonnées aient lu le message.

Les architectures de type publication/souscription (voir Figure 3.3) sont populaires pour construire des grands systèmes repartis qui prendraient en compte les futures évolutions pour passer plus facilement à l'échelle. Le principe est simple : un client, dit consommateur, souscrit pour recevoir les événements que l'intéressent, et le fournisseur, dit producteur, publie les événements qui seront reçus par les composants intéressés.

La gestion de messages et de souscripteurs est réalisée par un composant dédié à la gestion d'événements. L'utilisation d'un tel gestionnaire d'événements permet, entre autres [EFGK03] : un découplage spatial, les producteurs et les souscripteurs n'ont pas nécessairement une référence directe entre eux, en plus ils peuvent être exécutés sur des machines reparties ; il permet aussi un découplage temporel, l'exécution de consommateurs et de producteurs n'est pas nécessairement en même temps ; et finalement il permet un découplage de synchronisation, la transmission d'événements ne bloque pas l'exécution de producteurs.

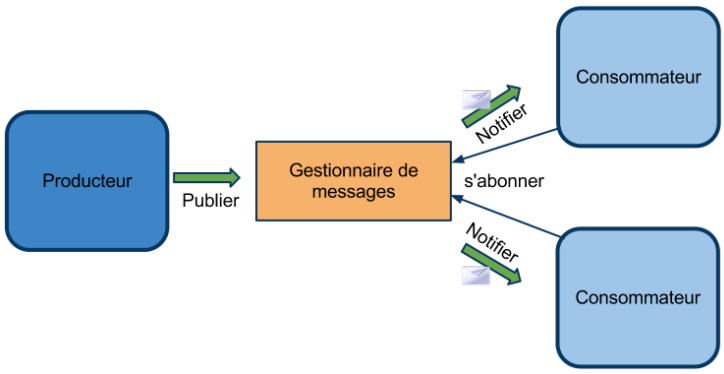

FIGURE 3.3 – Architecture publication/souscription

Le mécanisme de base est l'utilisation de « *topics* » ou « sujets » comme identification des messages. Dans cette approche basée sur les *topics*, les consommateurs réalisent un abonnement auprès du gestionnaire d'événements pour recevoir les messages des *topics* que les intéressent. Les producteurs publient les messages dans certains *topics*. Dans cette approche, c'est le gestionnaire qui fait la corrélation entre les messages publiés sur les *topics* et les consommateurs qui sont intéressés.

Les files d'attente de messages, ou en anglais Message Queuing (MQ), sont des solutions pour transmettre des messages entre le producteur et les consommateurs de même que les systèmes de type producteur/souscripteur. La différence réside dans la façon de livrer les messages. Dans les MQ, le producteur laisse un message dans une queue donnée, le consommateur, qui ne s'exécute pas nécessairement en même temps, demande les messages à la queue, et ils sont livrés de façon ordonnés, c'est-à-dire, il suive un comportement de : premier arrivé, premier servi.

Les MOMs sont fortement utilisés par les industriels. Il existe plusieurs implémentations, avec certaines caractéristiques en commun. Par exemple JMS, est une API standard pour Java. Cet API spécifie comment les composants repartis peuvent communiquer entre eux en utilisant les messages. Il y a plusieurs implémentations de JMS comme : Apache ActiveMQ [1], Joram [2], HornetQ [3], WebSphere MQ [4], parmi d'autres.

JMS est critiqué à cause de son incapacité à faire interagir les différentes implémentations entre elles. C'est ainsi que sont apparu des spécifications de *middlewares* à messages comme AMQP[Vin06]. Cette spécification décrit à la fois le comportement du fournisseur de messages comme celui du consommateur, ce qui permet aux diverses implémentations d'interagir entre elles.

Dans le monde de l'intégration d'applications, les *middlewares* basés sur des messages sont rapidement devenus populaires en raison du découplage fourni entre composants. En fait, le MOM est devenu l'ossature dans certaines solutions d'intégration actuelles. En effet, ce type de *middleware* est un choix naturel pour certaines problématiques d'intégration.

Les MOMs peuvent être utilisés comme l'infrastructure qui permet d'intégrer des systèmes à base d'une approche « pipes-and-filter », où chaque filtre réalise des opérations d'intégration. Dans ce cas, c'est le MOM qui facilite la communication entre les filtres, c'est-à-dire, le MOM réalise la fonction des « pipes ». En plus, grâce à ces caractéristiques, certains systèmes peuvent interagir, même si ils ne sont pas exécutés au même temps.

Néanmoins, utiliser seulement les *middlewares* à messages n'est pas suffisent pour faire l'intégration de systèmes. Au moins, il est nécessaire d'avoir des adaptateurs pour les systèmes à intégrer. En effet, la gestion de tels systèmes intégrés se limite à l'administration du *middleware* de messagerie.

1. Apache ActiveMQ http ://activemq.apache.org/
2. OW2 Joram http ://joram.ow2.org
3. JBoss HornetQ http ://hornetq.org
4. IBM WebSphere MQ http ://www-306.ibm.com/software/integration/wmq/

2.3 Enterprise Application Integration

Les solutions EAI (*Enterprise Application Integration*) proposent l'intégration d'applications via une architecture centralisée caractérisée par un seul point d'interaction pour les applications (Figure 3.4). Toute la complexité de mise en œuvre de telles solutions réside dans la maintenance de ce seul point d'interaction qui caractérise une architecture de type hub-and-spoke. Les solutions de type EAI classique fournissent souvent trois mécanismes qui permettent la construction d'un système intégré. Ces mécanismes sont la traduction, le routage, et souvent l'interprétation de règles de haut niveau.

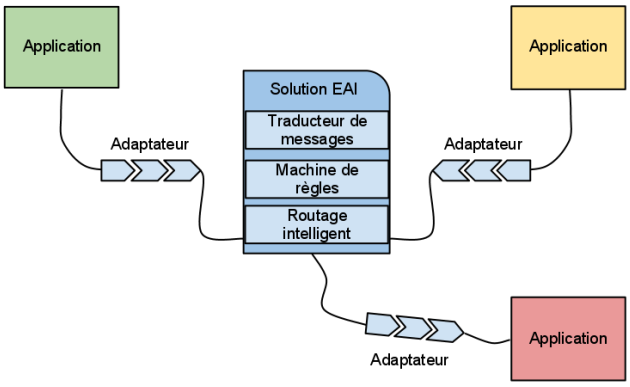

FIGURE 3.4 – EAI pour faire interagir applications

2.3.1 Traducteur de messages

La traduction de messages est une de principales caractéristiques qu'une solution d'intégration doit offrir. Dans une architecture EAI, le traducteur de messages est une couche qui comprend le format de toutes les applications. Cette couche permet faire donc des traductions « à la volée » lors qu'un message venant d'un système quelconque est destiné à un autre système connecté au EAI. Cette traduction est indispensable lorsque les systèmes à intégrer « parlent un langage différent de données ». Selon David Linthicum[Lin00], la couche de traduction est la « Pierre de Rosette » du système intégré.

Les EAI fournissent principalement deux mécanismes de traduction. La première est une traduction de représentation. Cette conversion permet de traduire la structure ou format de l'information sans changer la signification. Le deuxième type de traduction permet de faire la conversion de données, les modifier et aussi la création de nouvelles données.

2.3.2 Routage Intelligent

Le routage intelligent, aussi appelé routage dynamique ou routage basé sur le contenu est aussi important lorsqu'une intégration est composée de plus de deux systèmes. Plus précisément, plus de deux processus. Au moment où un message arrive à la solution EAI, l'entité chargée de

faire le routage analyse le message et choisi le destinataire du message. Par exemple, il l'envoie à l'entité de transformation ou à un autre système.

2.3.3 Machine de règles

Certaines solutions EAI fournissent des machines de traitement de règles. Ce type de machines, en combinaison avec le routage, facilite les possibilités d'intégration et ajoute une flexibilité facile à concevoir. La machine de règles permet principalement de décrire la logique métier d'intégration. Elle permet principalement de définir des instructions basées sur des prédicats logiques, notamment les IF-THEN-ELSE. La plupart de machines de règles permettent d'interpréter des scripts au lieu d'exécuter programmes d'instructions précompilées. Néanmoins, certains EAIs compilent les scripts à la volée pour garantir un meilleur temps d'exécution. Comme exemple de règle, une règle peut spécifier que certaines données soient routées à une application selon une information contextuelle (l'heure où le message est envoyé), et dans une autre situation le même type de donnée sera envoyé à deux autres systèmes. Ce type de logique d'intégration peut être réalisé avec une machine de règles.

Ces trois mécanismes de base fournis par les solutions EAI permettent de spécifier la logique d'intégration. Mais, afin de réaliser la connectivité entre l'EAI et les systèmes, est nécessaire d'un autre mécanisme. Ce mécanisme est à base de connecteurs ou d'adaptateurs. Ce mécanisme est composé d'entités logicielles qui connaissent la façon d'interagir avec les systèmes à intégrer, et ils sont les points d'union vers la solution EAI. Souvent, les EAI fournissent un format de donnée commun. Ainsi, chaque adaptateur transforme chaque message reçu dans ce format commun. La majorité des fournisseurs de solutions d'intégration livrent un ensemble d'adaptateurs vers les applications les plus connues. En plus, ils livrent des APIs (*Application Programming Interface*) qui permettent de construire des adaptateurs pour réaliser la connectivité vers d'autres applications. Il est bien connu qu'un des facteurs de succès d'une solution d'intégration est la liste d'adaptateurs livrés, ainsi que la facilité d'utilisation de l'API pour construire des nouveaux.

Un défi des EAI est la connectivité vers des systèmes qui ne sont pas conçus pour communiquer avec d'autres entités logicielles. Pour résoudre ce défi, les architectes doivent appliquer des patrons de connectivité, par exemple la construction des wrappers. Le problème s'amplifie quand il n'existe pas un accès au code source de ces systèmes, dits « legacy ». Dans ce dernier cas, une approche de rétro-ingénierie doit être appliquée.

L'adoption de solutions EAI se rencontre avec les problèmes suivants :

- la rapide évolution d'applications requiert des changements constants des applications, en conséquence des systèmes intégrés et de la solution d'intégration ;
- la nécessité des experts dans le domaine de l'intégration d'applications, aussi comme des expertes dans les systèmes intégrés ;
- pas de consensus de standards, chaque solution EAI fournit ses propres protocoles, outillages et mécanismes, souvent fermés.

2.4 Enterprise Service Bus

L'apparition des technologies à services, plus précisément les Services Web et ses standards associés, a permis l'émergence des ESB (*Enterprise Service Bus*) qui traitent de l'intégration d'applications patrimoniales et prennent la forme de *middlewares* d'intégration de services. Les applications patrimoniales, de même que les nouvelles applications, sont ainsi exposées sous forme de services connectés à l'ESB. L'ESB fournit un annuaire pour enregistrer ces services ainsi qu'un ensemble de services techniques pour faciliter le développement du code d'intégration (voir Figure 3.5).

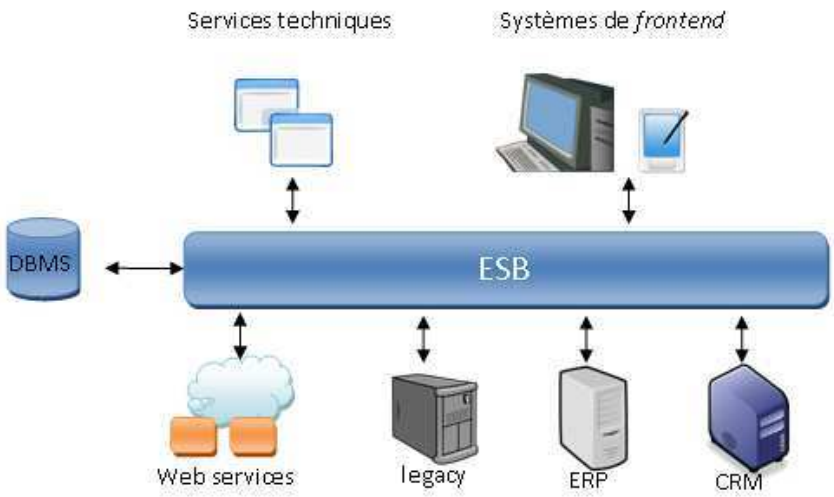

FIGURE 3.5 – Architecture ESB

Nous pouvons voir les ESB comme des solutions EAI plus évoluées, ceci grâce au fait que les ESB s'appuient sur des protocoles de communication standards, ainsi comme des formats standards de données. Ces dernières caractéristiques permettent d'affronter une problématique des EAI traditionnelles, où chaque fournisseur propose ses formats de données et de communication, souvent privés.

Les ESB de nos jours s'appuient fortement sur des standards Web. Les Web Services, par exemple, sont souvent les entités de composition, ainsi comme les entités techniques qui réalisent des opérations non fonctionnelles. De même, l'usage de XML comme format structurant de l'information permet d'homogénéiser le passage de messages, ainsi comme les opérations réalisées aux données. Par exemple, l'usage du langage XSLT[W3c07b] permet de réaliser des opérations de transformation de messages XML. Du même, le langage XPATH[CS] permet de réaliser des recherches dans les documents XML.

L'usage des standards est un facteur clé pour l'acceptation des solutions ESB. L'utilisation du format de messages XML, par exemple, découple même l'usage de langages de programmation, grâce à qu'il y a des parseurs dans presque tous les langages existants. En plus, l'utilisation de protocoles d'Internet comme couche de transport facilite l'interaction entre entités hétérogènes. Ces

protocoles d'Internet sont devenus ubiquitaires, et presque tous les dispositifs, avec des capacités de communication, le supportent.

Les services techniques ou les outils fournis par les *middlewares* d'intégration, que cela soit des EAI ou des ESB, visent à simplifier l'intégration des applications patrimoniales. En particulier, ces *middlewares* fournissent bien souvent des adaptateurs vers plusieurs systèmes existants, par exemple :

- adaptateurs vers les ERPs les plus populaires (SAP, Oracle e-Buisiness Suite) ;
- adaptateurs vers des services d'Internet (FTP, SMTP, XMPP, ...) ;
- adaptateurs de systèmes de gestion de base de données (MySQL, Oracle DB,...).

Certains ESB fournissent un *middleware* de messageries pour lier de façon lâche les applications et autorisent, de plus, des ponts de communications entre ces *middlewares* via des adaptateurs particuliers. Ces facilités conduisent à la définition d'architectures d'intégration dites snowflake[Bal] dans lesquelles coexistent plusieurs points d'interaction distribués qui sont bien souvent gérés de façon centralisée.

Les solutions de type ESB commencent à devenir populaires, et non seulement pour les grandes entreprises, comme étaient les solutions de type EAI traditionnelles, mais aussi pour les petites et moyennes entreprises (PME). En fait, cette popularité est due au fait que certains ESB actuelles (notamment des ESB open source) sont composées de divers « frameworks », qui répondent aux besoins bien définis. Cette modularité permet d'utiliser seulement certains éléments nécessaires pour une solution d'intégration concrète. Par exemple ServiceMix, qui est un ensemble de frameworks qui facilitent l'intégration. Il est composé d'un *middleware* de messagerie (Apache ActiveMQ), un framework d'intégration (Apache Camel), et un framework de connectivité (Apache CXF). Tous ces frameworks peuvent être utilisés séparément pour attaquer des problèmes d'intégration où il n'est pas indispensable avoir une infrastructure complète et lourde pour réaliser une solution d'intégration. En plus, un bon découplage de préoccupations, une correcte utilisation des frameworks, et des bonnes pratiques architecturales, permet de réintégrer d'autres frameworks lors d'évolutions d'applications.

Globalement, les ESB ne se différencient pas fondamentalement des EAI si ce n'est qu'ils s'appuient sur des standards de communication (pile WS-*, en particulier) et qu'ils facilitent la mise en place d'architectures de type snowflake.

2.5 Intégration de données

Le domaine de l'intégration de base de données a été marqué par de nombreux travaux de recherche. Les travaux de Gio Wiederhold [Wie92, WG97, Wie07]menés dans les années 80 et 90 font référence dans ce domaine. Ceux-ci ont posé pour la première fois les concepts d'une architecture d'intégration appelée « architecture de médiation » (Voir Figure 3.6). L'architecture de médiation, selon G. Wiederhold, se présente comme un assemblage de modules appelés « médiateurs » permettant de passer progressivement de données persistantes dans des sources de données hétérogènes à des données de plus haut niveau d'abstraction dites « métier ». Ces données apparaissent typiquement comme les objets « métier » d'une application.

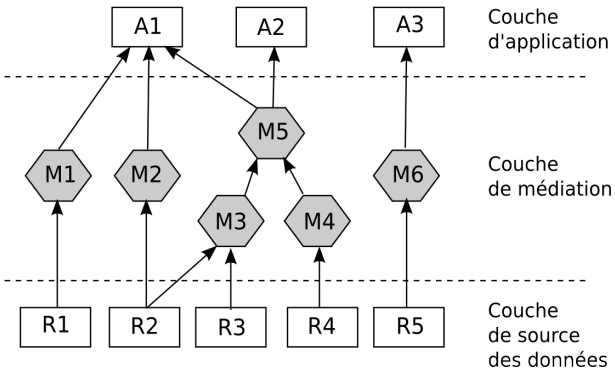

FIGURE 3.6 – Architecture de médiation

Les travaux de G. Wiederhold ont profondément influencé les modèles et les produits dédiés à l'intégration d'application. En particulier, ils ont été précurseurs des nouvelles architectures « 3-tiers » dans les Systèmes d'Information, découplant les applications « métier » des bases de données.

Peu de résultats directement issus des travaux de G. Wiederhold se sont matérialisés dans l'industrie de *miiddleware*. Nous pouvons néanmoins citer le *middleware* objet-relationnel PowerTiers de Persistence Software au début des années 90, qui associait un objet à une vue définie sous la forme d'une requête en algèbre relationnelle. En fait, le problème semble venir du fait que les bases de données relationnelles ne sont pas un très bon vecteur pour la problématique d'intégration qui se base essentiellement sur les notions de liaison et de désignation. L'issue de ce côté est venue d'une couche « *middleware* » réalisant une abstraction « objet » de ces bases de données, telle que Java EJB (*Enterprise Java Beans* - objet métier de l'entreprise) ou JDO (*Java Data Object*). L'avènement de cette couche *middleware* standardisant l'accès aux bases de données marqua d'ailleurs la fin d'une vision d'un Système d'Information centré sur ses bases de données.

Les bases de données fédérées (FDBMS pour l'acronyme en anglais)[SL90] traitent de la problématique de l'hétérogénéité dans les bases de données. Les applications voulant accéder aux bases de données communiquent directement avec le FDBMS (Figure 3.7) qui propose un modèle de données et un langage de requêtes homogènes. La couche d'abstraction apportée par le FDBS homogénéise donc l'interface de communication entre des applications clientes et des

bases de données hétérogènes.

Plus précisément, les requêtes des applications sont décomposées en sous-requêtes qui sont adressées dans le langage approprié aux bases de données visées. Le système fédéré doit agréger et transformer les réponses aux différentes sous-requêtes précédemment envoyées de façon à fournir à son tour une réponse unique.

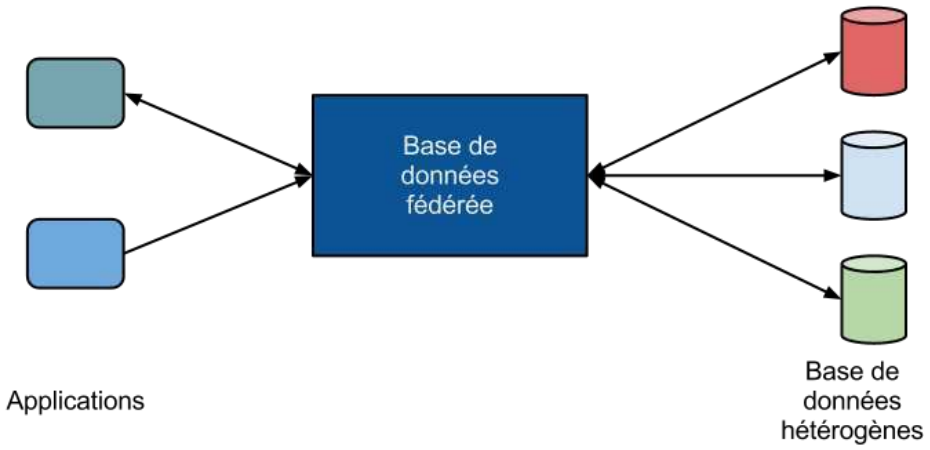

FIGURE 3.7 – Base de données fédérées

Une autre approche pour l'intégration de données est la mise en place d'une entité de stockage commune, basée sur un format compréhensible par tous. Des données de différentes sources sont ainsi répliquées dans un stockage commun ; ce dernier devient le point d'accès unique pour les applications.

L'entrepôt de données (*data warehouse*) est la solution la plus connue de cette approche. L'entrepôt est en charge de chercher les données directement dans les différentes sources de données et de faire les opérations d'intégration nécessaires.

Les différentes applications clientes n'interagissent qu'avec l'entrepôt de données qui leur fournit une vision cohérente des données. L'interaction se fait normalement dans un seul sens. Les applications ne font que des requêtes de demande de données et ne modifient pas ces mêmes données. Les modifications se font directement sur les sources de données. Les entrepôts de données sont ainsi utilisés pour faire des analyses, générer des rapports, explorer des données, etc.

Les travaux de la médiation ont beaucoup influencé le domaine de l'intégration. Même si cette approche s'adresse originalement à l'intégration de données, le mettre en œuvre pour résoudre des problèmes d'intégration de système ajoute certains avantages. Notamment la séparation des préoccupations, ainsi que le découplage. En plus, la construction d'une couche de médiation en faisant une composition de tâches de médiation permet la construction de systèmes modulaires et plus faciles à faire évoluer.

2.6 Synthèse

Les architectures d'intégration ont évolué dans le cours du temps, de même que les systèmes et les besoins des entreprises. Lorsque les systèmes logiciels évoluent, les architectes changent en suivant de paradigmes pour répondre aux nouveaux besoins. Souvent, une mauvaise planification du système intégré complexifie les évolutions futures des applications. C'est le cas de solutions d'intégration ad-hoc. Pour éviter cette difficulté, les architectes ont choisi d'utiliser des couches type *middleware* qui permettent, à certains niveaux, d'anticiper les évolutions.

L'usage des *middlewares* de messagerie (MOM), permet d'ajouter un découplage entre les applications à intégrer. Ce découplage peut être spatial, temporel, ainsi que logique. Le découplage est grâce à des opérations d'intégration ou d'adaptation lors de l'enchainement de messages d'un système vers une autre. Néanmoins, l'usage d'un tel *middleware* n'assure pas une solution d'intégration de qualité pour diverses raisons : premièrement, il y a un manque de modèles de développement et de gestion de cycle de vie. Ce qui fait que chaque architecte ou développeur spécifie son propre processus. Ceci complexifie les évolutions futures ainsi que la maintenance.

Les approches de type EAI, par rapport aux approches basées seulement sur des *middlewares* de messagerie, sont spécialisées pour attaquer les problèmes d'intégration. Ces solutions spécialisées permettent de définir des architectures, en même temps que de modèles de conception et d'outillages spécialisés à l'intégration. Néanmoins, les solutions de type EAI restent lourdes et complexes à mettre en œuvre.

Les solutions de type ESB apparaissent avec l'avènement d'approches à services et des standards associés (WS-* et XML), ainsi comme l'usage des protocoles de communication d'Internet. Néanmoins, la plupart de solutions de type ESB clochent des certains défauts comme les solutions EAI classiques. La complexité d'usage, ainsi que la difficulté de mettre en œuvre pour problèmes d'intégration dehors l'usage classique dans l'industrie (ubiquitaire, santé. . .).

Les nouveaux domaines d'application, comme l'ubiquitaire et la santé, nécessitent aussi des solutions d'intégration. Néanmoins, l'utilisation des solutions d'intégration actuelles ne répond pas aux nouveaux besoins de ces domaines. Il est nécessaire des solutions d'intégration qui prend en compte le dynamisme des environnements, ainsi comme une capacité d'adaptation pour répondre à une rapide adoption d'entités matérielles ainsi que des logiciels.

3 Patrons d'intégration

Tous les efforts et les connaissances acquises lors de la construction de solutions d'intégration depuis des années ont enrichi une large bibliothèque de patrons d'intégration. Un patron est la solution à un problème récurrent de conception. G. Hohpe et B. Woolf [HW08] ont fait un grand travail d'organisation de certains patrons de conception les plus utilisés pour résoudre les problèmes d'intégration. L'organisation et la formalisation de tels patrons ont été bien acceptées par les industriels et les fournisseurs de *middlewares* d'intégration. Pour la majorité des fournisseurs, ils vantent la possibilité d'implémenter tel ou tel patron en utilisant facilement les outillages du *middleware*. De ce fait, le succès d'un *middleware* d'intégration ainsi que la bibliothèque d'adaptateurs de systèmes courants et l'outillage disponible sont influencés par la mise en œuvre de tels patrons.

Les patrons d'intégration présentés dans le livre de G. Hohpe se basent principalement sur les systèmes de messagerie (généralement appelés *Message Oriented Middleware*). G. Hohpe justifie l'utilisation des systèmes de messagerie car ils offrent des avantages indispensables pour faire interagir les applications patrimoniales des entreprises. Parmi ces avantages, nous trouvons : la communication répartie, la communication asynchrone, le découplage entre les systèmes, la gestion de files d'exécution et la fiabilité de la communication entre autres. Tous ces avantages dépendent du fournisseur du système de messagerie ; néanmoins, il y a un consensus non formalisé de certaines de ces caractéristiques communes.

Certains patrons sont intrinsèques au *middleware* d'intégration utilisé, tels que les patrons d'interaction (*event-based*, *topic-based*, *request-reply*,...). Cependant, d'autres patrons sont appliqués selon la nature des applications patrimoniales à intégrer. Par la suite, nous faisons une brève classification de certains patrons que nous considérons comme importants. Plutôt que de suivre la catégorisation faite par Hohpe, nous plaçons les patrons d'intégration dans les catégories suivantes :

- la réception de messages,
- la transformation de messages,
- le routage de messages,
- la gestion de systèmes.

Des informations supplémentaires concernant les patrons d'intégration et leurs implantations possibles sont disponibles sur le site de G. Hohpe.

3.1 Réception de messages

Les patrons de réception de messages sont divisés en deux sous-catégories : patrons « d'acquisition de messages » et patrons « d'ordonnancement de messages ». Nous distinguons ces deux sous-catégories puisque le traitement de messages est fortement relié à la façon dont ils sont obtenus (acquisition de messages) et à leur identité précise (identification de messages).

La première sous-catégorie appelée « patron d'acquisition de messages » vise à exprimer la façon dont les messages sont obtenus. En résumé, ces patrons expriment l'interaction entre le producteur ou la source de messages et le consommateur de messages. Nous pouvons trouver certains patrons d'interaction implémentés par les *middlewares* d'intégration (comme le patron de publication/souscription et de requête/réponse). Néanmoins, certains autres patrons doivent être implémentés lors de la mise en œuvre de l'intégration de systèmes.

Le polling consumer, par exemple, est un patron qui répond au besoin d'avoir un contrôle total sur l'acquisition des messages. L'implantation de ce patron permet aux applications de requérir les messages au moment où ils en ont besoin. Ce polling de messages peut être déclenché par diverses techniques : une contrainte temporelle, une détection de changement d'état, ou la réception d'un nouveau message.

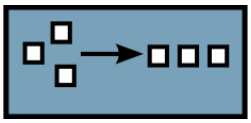

FIGURE 3.8 – Patron ordonnanceur (du livre d'EIP)

La deuxième sous-catégorie appelée « patrons d'ordonnancement de messages » vise l'analyse et l'identification des messages à l'arrivée. Cette analyse permet l'identification des messages dans le but de réaliser un traitement correct au bon moment et dans l'ordre correct.

Le patron appelé correlation identifier est un patron de référence de cette sous-catégorie. Ce patron définit une solution à l'identification des messages qui sont corrélés. Par exemple, pendant une interaction requête/réponse, l'application doit identifier à quelle requête correspond chaque réponse reçue. Cette identification est faite par un étiquetage unique d'identification de corrélation que chaque message doit contenir.

Nous classifions aussi le « patron resequencer » dans la sous-catégorie « patrons d'ordonnancement ». Ce patron aborde le problème de mettre en ordre les messages reçus hors séquence par l'analyse des messages. Il y existe certaines variantes de ce patron. Ces variantes se différencient par leur gestion de l'identification des messages pour leur ordonnancement.

3.2 Transformation de messages

Les patrons de « transformation de messages » visent à résoudre les problèmes d'alignement (syntaxique et sémantique) entre les diverses applications à intégrer. Les solutions de ces patrons consistent à appliquer des modifications aux messages afin qu'ils soient reçus dans le format (et la sémantique) attendu par les consommateurs.

Un patron de référence est l'aggregate pattern. Ce patron vise la construction d'un message par la composition de divers messages. Ce patron est couramment utilisé avec un patron « d'identification de messages ». Par exemple, pour identifier quels sont les messages corrélés, il est possible d'utiliser le patron de correlation identifier. Un autre patron couramment utilisé est le splitter pattern. A la différence de l'aggregate, ce patron vise à découpler un message composé de plusieurs sous-messages. Nous pouvons trouver dans cette catégorie les patrons :

- le content enricher, pour ajouter des méta-informations à chaque message ;
- l'agrégateur, pour construire un nouveau message composé à partir de divers messages reçues ;
- le translator, pour traduire la signification de messages.

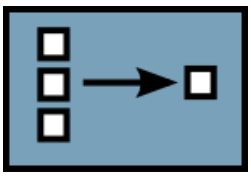

FIGURE 3.9 – Aggregateur (du livre d'EIP)

En résumé, les patrons de transformation abordent les problèmes liés à l'application d'une fonction aux messages pour réaliser une modification de forme, de contenu ou de sens afin qu'ils soient consommés par d'autres composants. Si l'on suit rigoureusement cette définition, le patron de filtrage (*filter pattern*) peut être placé dans cette catégorie et non pas dans la catégorie de routage de messages présentée par G. Hohpe. Le filtrage correspond à l'exécution d'une fonction d'élimination des messages qui n'intéressent pas l'application consommatrice.

3.3 Routage de messages

Ces patrons ont pour but de résoudre les problèmes récurrents à la livraison de messages. De la même façon que la catégorie de réception de messages, certains problèmes de routage sont abordés par les caractéristiques fournies par les *middlewares* d'intégration. Le load balancer pattern se place dans cette catégorie.

Le patron de référence de cette catégorie est le content-based routing. Ce patron consiste à analyser le contenu de chaque message ainsi que le résultat de cette analyse qui permet d'identifier la destination correcte du message. Une variante de ce patron est le type-based routing, qui consiste à identifier la destination correcte selon le type de message qui transite.

FIGURE 3.10 – Content-Based Router (du livre de EIP)

Le patron appelé recipient list est un patron de routage de messages qui consiste à gérer une liste de destinations. Ce comportement ressemble à un multicast qui consiste à envoyer le même message à plusieurs destinations. Une variante de ce patron est le dynamic router. Ce patron répond au problématique de gérer la liste de récepteurs qui change au cours de l'exécution selon la disponibilité des récepteurs de messages. Une variante d'un routage dynamique appelé routing slip consiste à analyser chaque message. Ce patron exprime que chaque message doit contenir des méta-informations qui décrivent qui doit être le récepteur de ce message. En résumé, les patrons de routage abordent la problématique de la sélection du consommateur.

3.4 Gestion de systèmes

Certains patrons sont utilisés pour aborder la problématique concernant la surveillance du bon fonctionnement de la solution d'intégration d'applications. Parmi ces patrons, nous trouvons les patrons de débogage, de contrôle, de surveillance et de stockage de l'historique.

Le patron de référence est le Wire Tap. Ce patron spécifie comment faire l'inspection des messages qui transitent entre les composants. La solution proposée consiste à placer un composant d'interception entre le producteur et le consommateur de messages. Ce problème peut être aussi résolu par le patron recipient list en ajoutant le composant d'interception dans la liste des récepteurs.

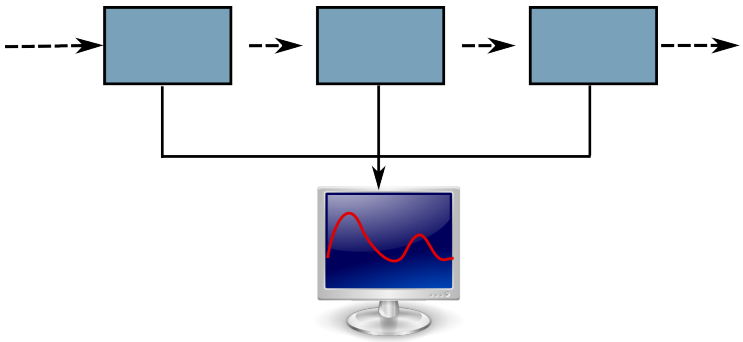

FIGURE 3.11 – Wire Tap (du livre EIP)

Un autre patron, le patron Control bus, aborde la problématique de la gestion des systèmes d'intégration. Ce patron spécifie l'utilisation du système de messagerie utilisé par les applications interconnectées en utilisant divers canaux exclusifs à la gestion.

3.5 Patrons composés

On peut considérer qu'une intégration est simple lorsqu'un seul patron d'intégration suffit pour faire interagir deux systèmes.

L'intégration de systèmes patrimoniaux n'est en général pas aussi facile et doit généralement passer par une analyse approfondie de ces systèmes. Cette analyse implique la compréhension du fonctionnement des systèmes, l'analyse du type de données manipulées, les patrons d'interaction possibles ainsi que plusieurs autres aspects à identifier. Le résultat de cette analyse permet de construire une solution d'intégration pour ces systèmes. La solution d'intégration peut être interprétée comme une composition de divers patrons d'intégration. Un architecte expérimenté peut ainsi identifier les patrons nécessaires et guider la mise en œuvre de cette composition de patrons d'intégration.

Un exemple d'une composition de patrons d'intégration est le patron split/aggregate. Ce patron est une combinaison de deux patrons importants : le split et l'aggregate. De plus, il est couramment combiné à d'autres patrons. Dans la Figure3.12, la composition de patrons d'intégration combine deux autres patrons : le translator et le content-based routing.

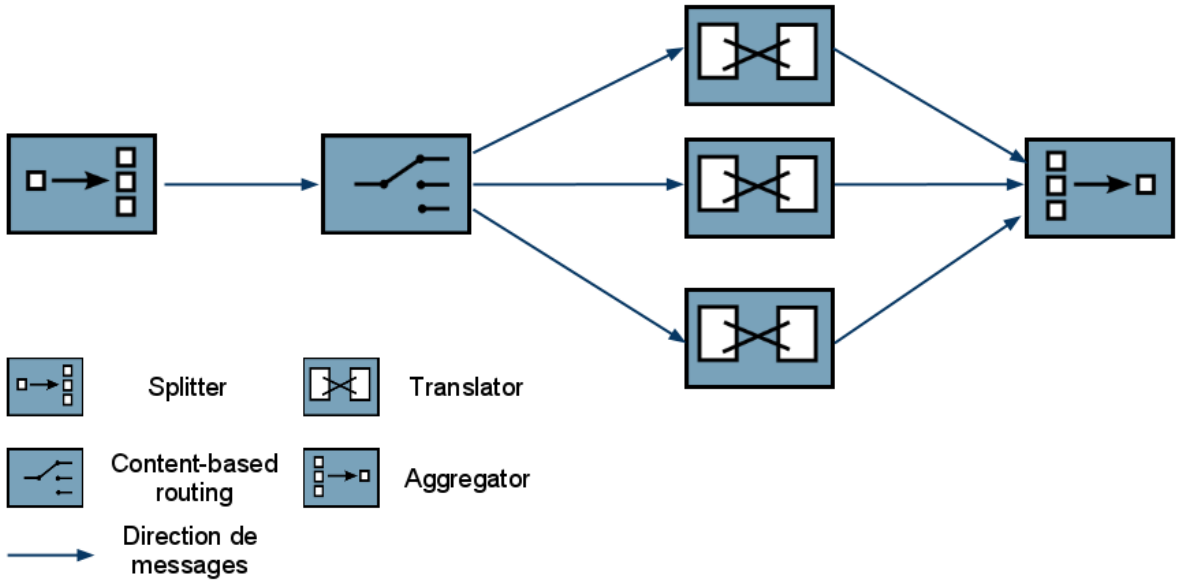

FIGURE 3.12 – Split/Aggregate - patron composé

3.6 Synthèse

Le Table 3.1 montre un récapitulatif des divers patrons d'intégration et leurs catégories. La liste des patrons présentée dans ce tableau n'est pas exhaustive et ne comprend pas tous les patrons présentés dans le livre de G. Hohpe. De plus, la catégorisation que nous présentons diverge de celle présentée dans le livre puisque nous avons choisi de la centrer sur les messages en montrant en particulier l'impact que l'implantation de ces patrons a sur les messages.

Les patrons de conception représentent une connaissance de grande valeur pour les concepteurs d'applications. L'expertise acquise depuis des années a permis de constituer une large bibliothèque de patrons qui répondent aux problématiques les plus récurrentes dans divers domaines d'application. De ce fait, le domaine d'intégration n'est pas une exception. G. Hohpe fait une classification des divers patrons d'intégration en utilisant les systèmes de messagerie. Nous définissons une catégorisation par rapport à trois grandes problématiques de la conception : la réception, le traitement et la livraison. En plus, nous pouvons ajouter une quatrième catégorie qui est liée à la gestion de systèmes d'intégration.

Patron de Réception		**Patron de transformation**	**Patron de routage**	**Patron de gestion**
Patron d'acquisition	**Patron d'ordonnancement**			
Polling consumer	Correlation identifier			
Observer pattern	Resequencer	Mapping	Message-type based	Control bus
Topic based	Message expiration	Filter	Recipient list	Channel purger
Message bridge		Content enricher	Dynamic routing	
Durable subscriber		Aggregator	Multicast	
		Splitter	Load balancer	
		Normalizer	Routing slip	

TABLE 3.1 – Liste de patrons d'intégration

Les patrons d'intégration, comme leur nom l'indique, sont les patrons qui abordent la problématique d'intégration de systèmes qui sont faiblement couplés. Une solution mise en œuvre pour résoudre cette problématique d'intégration est de composer un ensemble de ces patrons d'intégration. Cela explique l'importance de l'opération de composition des patrons.

Les patrons d'intégration ont influencé le développement des middlewares d'intégration. Les premiers outils d'intégration, tels que les EAI ou les *Message Brokers*, fournissaient des services techniques afin de faciliter le développement. Parmi ces services, nous pouvons citer les services de transformation, de routage et de messagerie. Cependant, la conceptualisation des solutions d'intégration était de la responsabilité du concepteur et le succès de la solution reposait clairement sur son expérience. Ce besoin d'expertise est une des principales raisons pour laquelle les fournisseurs de middlewares d'intégration proposent des méthodes de composition basées sur les patrons d'intégration. Cette nouvelle proposition consiste à fournir des composants prédéfinis. Ces composants permettent de mettre en œuvre certains patrons d'intégration, de minimiser la complexité de développement et d'augmenter la réutilisation.

Dans ce qui suit, nous décrivons les principaux *frameworks* d'intégration qui s'appuient sur une composition basée sur les patrons d'intégration.

4 Solutions existantes

4.1 Introduction

Dans les sections précédentes, nous avons vu des solutions architecturales d'intégration, ainsi comme les patrons d'intégration qui répondent aux problématiques récurrentes d'intégration. Ces patrons d'intégration peuvent être appliqués dans ces solutions architecturales, EAI ou ESB. Même si ces solutions sont souvent composées de plusieurs éléments et technologies, nous faisons emphase sur les EIP et leur composition. C'est la composition de ces patrons qui réponde vraiment à l'intégration. D'autres éléments des ESB, ainsi comme des EAI, comme les MOM, les services d'administration et les adaptateurs, facilitent la mise en œuvre, comme la gestion et le passage à l'échelle.

Dans la suite, nous présentons des solutions d'intégration qui proposent des mécanismes de composition de tâches de médiation. Les critères d'analyse des solutions existantes sont les suivants :

- la composition : pour décrire comment ces solutions existantes permettent de lier les tâches de médiation, dites tâches d'intégration (liaison interne). Ainsi, les styles d'interaction entre composant et les méthodes d'interaction avec des entités externes (liaison externes). De plus, la façon de décrire la composition. (e.g. Langages de description d'architecture) ;
- le cycle de vie : pour décrire les options disponibles pour chaque étape de cycle de vie, depuis la conception jusqu'à la maintenance de systèmes intégrés ;
- L'adaptabilité : pour décrire quelle sont les options que chaque framework propose pour faire évoluer les applications, soit au runtime (à la volée), soit statique. Ainsi qu'identifier ses caractéristiques pour adresser le dynamisme et la mobilité des nouveaux environnements.

4.2 Spring Integration

Principes

Spring Integration est un middleware d'intégration proposé par SpringSource. Aujourd'hui il est détenu par VMware. Cette solution d'intégration est un nouveau projet du portfolio de solutions de Spring. Le modèle de programmation, qui est basé sur la technologie de Spring, est destiné à l'intégration de systèmes basé sur l'échange de messages et sur les patrons EIP.

Les principaux éléments de Spring Integration dans une application (voir Figure3.13) sont :

- Le *Message*. Cet élément correspond à une enveloppe contenant un objet Java (*payload*) et des métadonnées (*header*). Le payload peut être de n'importe quel type. Le header est composé d'informations comme l'identificateur du message, la date d'expiration, etc.
- Le *Channel*. Cet élément représente le pipe dans une architecture « *pipes-and-filters* ». Il définit la façon dont les messages transitent entre les endpoints (voir ci-dessous).
- L'*Endpoint*. Ce composant réalise les opérations de médiation pour les messages qui transitent.. Ces éléments réalisent des tâches de manipulation de données, de transformation de données, de filtrage de données, de l'identification des canaux de sortie (routage) et de la connectivité vers des systèmes externes (JMS, HTTP, mail, etc.).

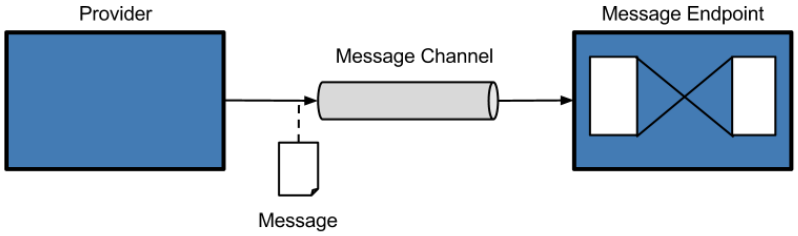

FIGURE 3.13 – Architecture d'un application Spring Integration

Composition

Spring Integration présente un modèle d'intégration simple avec une architecture « *pipes-and-filters* ». Les pipes (*channels*) permettent la liaison entre deux composants, et les filters (*endpoints*) réalisent diverses tâches, par exemple de routage, d'adaptation vers des systèmes externes, et de traitement des messages. Ceci aboutit à un certain mélange des préoccupations dans les endpoints. Cependant,avoir seulement deux concepts simplifie le développement théoriquement.

Spring Integration s'appuie fortement sur le modèle de programmation de Spring. En conséquence, les endpoints, ainsi que les channels sont des beans (Spring beans) spécialisés. Spring Integration fournit un espace de nommage (*namespace*) XML comme une extension au langage de base de Spring. Cette extension ajoute un nouveau langage pour décrire les composants et décrire les liaisons entre ces composants.

L'équipe de *Spring Integration* a fait un grand effort pour fournir une bibliothèque de patrons

déjà implémentés et prêts à l'utilisation. En plus, elle fournit un ensemble d'adaptateurs vers des systèmes divers, allant des protocoles de communication, des gestionnaires de bases de données et, plus récemment, des réseaux sociaux. Un autre grand avantage est le grand nombre d'utilisateurs de la technologie de Spring ainsi que l'interopérabilité existante entre toutes les solutions fournies.

Cycle de vie

Une solution d'intégration, en utilisant Spring Integration, peut être déployée sur différents environnements d'exécution. Elle peut être déployé par exemple dans un serveur d'applications, mais aussi dans une passerelle OSGi, ou encore comme une application de type standalone. La seule restriction existante est que les capacités de calcul et de ressources du dispositif qui le contient soient suffisantes.

Les frameworks fournis par Spring sont pensés pour faciliter le développement d'applications. Néanmoins les caractéristiques fournies pour reconfigurer les applications lors de l'exécution sont minimes. La version 2.5 de Spring Integration fournit un mécanisme de contrôle d'applications (appelé control bus). Ce mécanisme permet aux administrateurs de faire des appels de méthodes de certains composants, de la même façon que JMX, mais en utilisant un canal de messagerie.

Adaptabilité

Afin de faire des adaptations au runtime Spring Integration s'appuie d'un mécanisme appelé « de control ». Cependant, ce mécanisme permet seulement de modifier certains paramètres de l'application. Cet mécanisme est le seul disponible pour faire des adaptations au runtime. Sinon, pour une reconfiguration plus complexe, comme la restructuration de l'architecture, la seule option disponible est d'arrêter et de redémarrer l'application pour effectuer des mises à jour.

4.3 Camel

Principes

Camel est un projet open source de la Fondation Apache. C'est un moteur de routage et de médiation fortement influencé par les patrons d'intégration EIP. Il fournit à la fois, un modèle de développement et un grand nombre d'adaptateurs et de patrons d'intégration prêts à l'usage. Ce projet est fortement soutenu par les industriels. C'est le moteur « d'orchestration » fourni par le serveur d'applications JOnAS ainsi que les ESB open source ServiceMix et FuseESB.

Apache Camel repose sur les trois éléments suivants (voir Figure3.14) :

- Un « *Endpoint* » peut être, soit une source de données, soit un consommateur de données. Un endpoint est identifié par un URI (*Uniform Resource Identifier*). L'*endpoint* source est identifié dans la description de routage grâce au mot-clé « from (URI) ». Le endpoint de destination est identifié dans la description de routage grâce au mot-clé « to (URI) ». En général, un endpoint est chargé de l'adaptation vers les différents protocoles de communication.
- Un « *Processor* » prend en charge de nombreuses fonctions incluant le traitement des messages, le routage, le filtrage et la transformation de données. En pratique, les *processors* permettent mise en application des patrons d'intégration.
- Un « *Route* » est une composition de « *processors* » et d'« *endpoints* ».

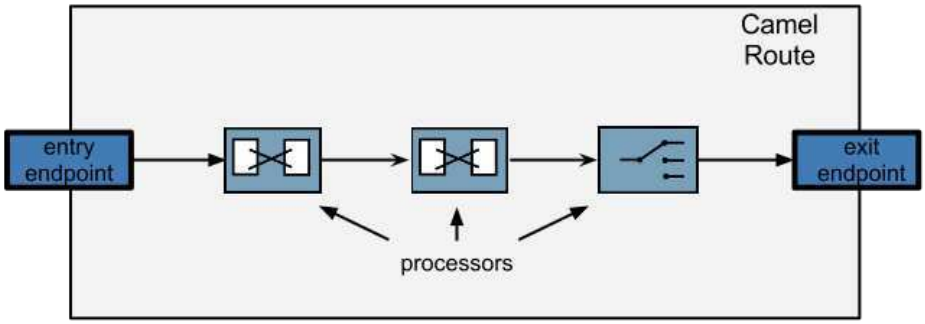

FIGURE 3.14 – Architecture d'un application Camel

Composition

Camel fournit plusieurs options de composition. La plus recommandée est appelée « Camel Java DSL ». Ce langage est sur la forme d'une API Java « fluide » (*fluent API* en anglais). Cette API permet une programmation guidée par l'enchaînement de méthodes. Le seul but de ce type de programmation est une lecture du code plus compréhensible. De plus, la composition devient facile dans les environnements de développement grâce aux options de complétion automatique. Une autre option de composition est l'utilisation d'extensions du langage XML de la technologie fournie par Spring. L'utilisation de Camel combinée avec la technologie de Spring permet l'usage

de l'inversion de contrôle ainsi que l'interopérabilité avec les services fournis par le portfolio de Spring.

Pendant la composition, c'est-à-dire la définition des routes, les liaisons entre les composants sont faites par l'usage des URI (*Uniform Resource Identifier*). Cette composition de routage est faite par l'enchaînement des URI qui identifient chaque composant sous la forme d'un endpoint. En fait, une URI peut identifier un « bean » de Spring, un objet Java, un Service Web, un protocole de transport, etc.

Cycle de vie

Une solution d'intégration, en utilisant Camel, peut être déployée sur différents environnements d'exécution, tels que les conteneurs de servlets comme Tomcat, les passerelles OSGi comme Apache Felix ou Equinox, les serveurs d'applications J2EE comme JOnAS. Il a également été intégré avec l'ESB ServiceMix, le broker de messagerie Apache ActiveMQ et aussi avec le framework de communication et de création de services Apache CXF.

L'administration de routes sur Camel est faite par l'utilisation d'une console d'administration de routes. Cette console permet aussi la surveillance d'événements lors de l'exécution. Une autre fonctionnalité est la possibilité d'arrêter et de redémarrer les routes ainsi que d'en créer des nouvelles ou encore de modifier des routes existantes. Néanmoins, la modification de routes existantes est possible seulement si nous déchargeons et chargeons de nouveau les routes.

Adaptabilité

L'enchainement, c'est-à-dire, les routes construites peuvent être connues lors de l'exécution. Ainsi, ces routes peuvent être modifiées. Néanmoins, les modifications sont semi-dynamiques : la chaîne de médiation doit être complètement arrêtée, ensuite réaliser les modification et finalement redémarrer. En un mot, Camel permet un mécanisme de basia pour la construction d'applications adaptables. La seule restriction est que ces modifications doivent être faites pendant que l'application est arrêté.

4.4 Mule ESB

Principes

Mule ESB est un framework d'intégration Open Source proposé par Mulesoft. Ce framework propose une architecture d'intégration en trois niveaux (cf. Figure 3.15). Le premier niveau, dit applicatif, contient les services avec lesquelles on veut interagir (la logique métier). Le second niveau, dit d'intégration, permet de définir des opérations de médiation comme le routage, l'agrégation, la transformation. Le dernier niveau gère les aspects liés à la communication.

- Les principaux éléments d'un « service » Mule ESB sont les suivants :
- Un *Endpoint* est chargé de la communication entre les différents composants de la chaîne de médiation. L'*endpoint* est situé dans le niveau de communication.
- Un *Service Component* spécifie la logique métier.
- Un *Inbound* permet de manipuler le message reçu avant qu'il ne soit passé au service avec le quel on veut communiquer. L'*inbound* est situé dans le niveau d'intégration.
- L'*Outbound* permet de manipuler le message après son traitement effectué par le service. L'*outbound* est situé dans le niveau d'intégration.

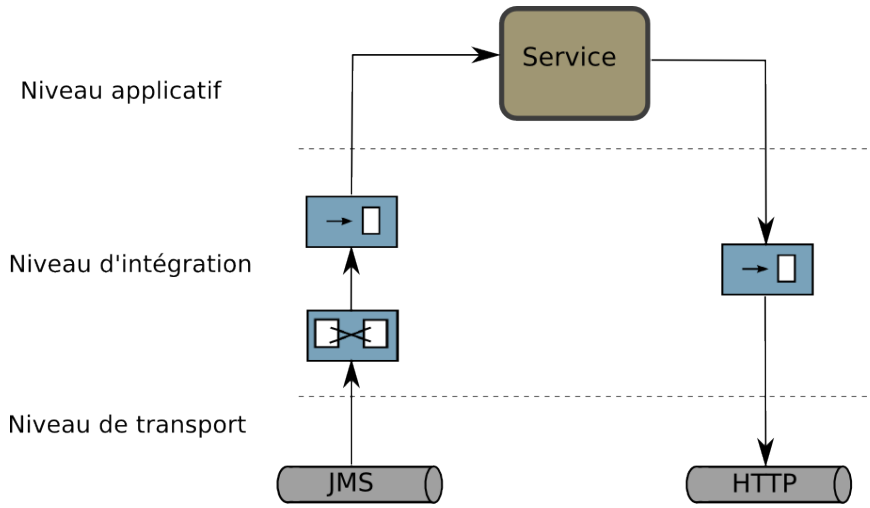

FIGURE 3.15 – Architecture de Mule 2

Composition

Le niveau d'intégration, composé par l'*inbound* et l'*outbound*, est une composition d'opérations en chaîne séparées en deux sections. Premièrement, les opérations, qui sont enchaînées dans la section d'*inbound*, réalisent les transformations nécessaires aux données reçues par l'*endpoint* d'entrée : ceci afin que ces données soient bien consommées par le *Service Component*. Ensuite, les opérations enchaînées dans la section d'*outbound* font une transformation afin d'envoyer les données dans le bon format vers l'*endpoint* de sortie.

La version 2 de Mule fournit une architecture bien définie et flexible pour aborder le problème

d'intégration d'applications. Cette flexibilité permet au développeur de construire des modules de médiation avec plusieurs éléments dans le niveau d'intégration. La richesse de Mule est principalement l'ensemble des composants prêts à l'usage dans le niveau d'intégration, ainsi que la liste d'endpoints fournis pour interconnecter des systèmes externes.

La version 3 de Mule présente un nouveau style architectural basé sur le flux de messages, appelé « *flow* ». Le flux dans Mule est une composition de tâches enchaînées depuis une source, comme un endpoint, vers une destination, comme un service ou un autre endpoint. Ce flux est beaucoup plus flexible que la composition d'un service présenté dans la version précédente. Cette flexibilité est due au fait que la composition n'impose pas un ordre pour les opérations, ni l'utilisation d'un *Service Component*.

Mule s'appuie sur le framework Spring et ses capacités d'extension pour fournir un langage de composition en XML. D'ailleurs, la version 3 de Mule permet de faire des compositions de services, ainsi que des compositions basées sur le style flux. La liaison entre chaque composant à l'intérieur d'un flux ou d'un service est faite directement et elle est gérée par le runtime de Mule et l'inversion de contrôle fournie par Spring. Cependant, la liaison entre services ou entre les systèmes externes est faite par les endpoints du niveau transport.

Cycle de vie

Certains outils sont fournis par MuleSoft pour gérer le cycle de vie des applications basées sur Mule : un IDE (*Integrated Development Environment*) pour faciliter le développement, un outil pour le déploiement dans plusieurs serveurs en même temps. Bien qu'une application Mule puisse être déployée dans un conteneur d'applications, elle peut aussi être de type standalone. De plus, une console d'administration personnalisable est disponible pour l'analyse de messages ainsi que l'analyse de la performance et le réglage de paramètres du système.

Adaptabilité

Mule ne présente pas de possibilités avancées pour la construction d'applications adaptables. La seule option disponible est la modification des paramétrages du système lors de l'exécution. Sinon, la seule possibilité est l'arrêt et le redémarrage de l'application pour faire des mises à jour.

4.5 Comparaisons

Les patrons d'intégration ont fortement influencé les middlewares d'intégration. En conséquence, certains middlewares d'intégration fournissent des langages de description d'architectures les intégrant explicitement. Le projet Apache Camel est un exemple où l'on perçoit bien l'influence des patrons d'intégration sur le langage de description de routes. Les routes dans Camel représentent l'architecture des solutions de médiation qui répondent à la problématique d'interaction des applications.

La Table 3.2présente les middlewares d'intégration d'applications présentés dans cette section et certaines caractéristiques. Nous nous intéressons principalement à deux aspects importants : la composition et l'adaptabilité. La composition comprend les aspects liés à l'assemblage de composants. L'adaptabilité comprend, quant à elle, les possibilités de modification lors de l'exécution. Les caractéristiques du cycle de vie sont aussi présentées puisque la composition et l'adaptabilité font partie du processus de développement et de reconfiguration qui font eux-mêmes partie du cycle de vie.

La séparation des préoccupations, la modularité et l'anticipation des évolutions sont les principes du génie logiciel qui permettent de minimiser la complexité et de mettre en œuvre les systèmes ubiquitaires. D'une part, la séparation des préoccupations et la modularité guident le développement pour une meilleure compréhension et une meilleure répartition du code métier. D'autre part, l'anticipation aux évolutions lors de l'exécution permet une flexibilité pour réagir selon le dynamisme et la mobilité présente dans les systèmes ubiquitaires.

La séparation des préoccupations doit permettre d'identifier et de classer séparément les aspects fonctionnels et non fonctionnels. Par exemple, l'aspect de routage, ne doit pas être dans la même classification que l'aspect d'adaptation. Dans les cas de l'intégration, cela revient à catégoriser les diverses tâches récurrentes telles que la communication, le routage, etc. Les aspects non-fonctionnels, comme la surveillance, doivent être toujours bien séparés des aspects fonctionnels de l'intégration, comme la transformation. Dans les frameworks présentés, il y a certaines tâches que ne sont pas bien séparées, ce qui donne l'impression d'un mélange de préoccupations. Spring Integration fait bien la distinction entre liaison et composant métier. Néanmoins, le composant métier peut être à la fois : un transformateur de données ainsi qu'un descripteur de routage. Le runtime de Camel prendre en charge les aspects de routage dans le runtime, les liaisons entre composants internes sont implicites et les liaisons vers des systèmes externes sont faites par les URI. Néanmoins, il n'y a pas un vrai modèle conceptuel de référence à part celui des endpoints pour les liaisons externes et celui de processor pour tout le reste des opérations. La version 2 de Mule, qui présente le Service Component comme entité d'intégration, présente un modèle intéressant qui fait une séparation claire des préoccupations avec une architecture à trois niveaux. Néanmoins, la version 3 de Mule présente un nouveau modèle de composition basée sur le « flux ». Nous trouvons cette nouvelle approche plus flexible mais elle détruit le modèle bien structuré du Service Component.

La modularité est un aspect assez récurrent et abordé dans tous les middlewares d'intégration.

L'utilisation du framework Spring permet d'importer divers fichiers de configuration. Néanmoins cet import est explicite. Camel, Spring Integration et Mule utilisent Spring Integration pour permettre la construction d'applications modulaires.

Nous pouvons classer une anticipation aux évolutions en trois niveaux : le premier où les changements sont faits en modifiant le code de l'application ; le deuxième qui est une anticipation minimale ou hors ligne, où l'architecture peut être modifiée sans modifier le code existant ; le troisième où les applications peuvent s'adapter lors de l'exécution, sans arrêter l'application. Une adaptation idéale devrait être faite dans le troisième niveau ; c'est-à-dire que les évolutions sont faites au fur et à mesure lors de l'exécution. Cependant, l'anticipation aux évolutions des middlewares d'intégration étudiés sont de niveau deux ; c'est-à-dire que les évolutions peuvent être effectuées hors ligne sans modifier le code des composants existants tant qu'il existe une bonne modularité et une bonne séparation des préoccupations dans le code existant.

Catégorie	Sous-Categorie	Spring Integration	Camel	Mule ESB
Composition	Liaison interne	Utilisation de canaux de communication	Basé sur URIs, chaque composant a un URI	directe, description implicite
	Liaison externe	Utilisation d'adaptateurs de systèmes externes	Basé sur URIs, chaque composant a un URI	Composants spéciaux (Endpoint), liaison par URI ou par configuration en XML
	Style d'interaction	point-à-point, publication/souscription	Requête/réponse, Event Message	requête/réponse, point-à-point
	Description d'architecture	Langage de description de composants basé sur l'EIP, connections sur le nommage de canaux. Langage basé sur XML (extension de Spring)	Basé sur le routage. DSL basé sur les EIP. Différents options de DSL (Spring XML, Java , Scala).	basé sur les flux, connexion implicite basé sur l'ordre de définition de composants.
Cycle de vie	Développement	Modèle basé sur Spring. Code + fichiers XML. Maven et Ant pour gérer le développement de projets	Editeur spécialisé, Maven et Ant pour gérer le développement de projets. Option d'utiliser le modèle de programmation Spring.	IDE spécialisé, Maven et Ant pour gérer le développement de projets.
	Déploiement	Sur un conteneur d'applications, une plateforme OSGi, ou tout seul	Sur un conteneur d'applications, une plateforme OSGi, ou tout seul	Sur un conteneur d'applications, tout seul. Gestion de déploiement multiple
	Surveillance (Administration)	Utilisation d'un bus de control. Canaux spécifiques à la gestion et surveillance	Fournit d'intercepteurs, monitor d'activités et un web console monitor	Console d'administration : surveillance personnalisé, analyses de messages, analyses de performance et réglage, control de services (stop/start)
	Reconfiguration	Statique (hors l'exécution). Approche : décharge, modifie, chargé de nouveau	Possible lors de l'exécution. Approche : arrêter/modifier/démarrer.	Statique (hors l'exécution). Approche : décharge, modifie, chargé de nouveau
Adaptabilité	Dynamisme	Difficilement due à la technique de reconfiguration	Semi dynamique. Il faut arrêter la route complet pour modifier.	Difficilement due à la technique de reconfiguration
	Mobilité	Non définie	Non définie.	Non définie
	Passage à l'échelle	Oui, statique.	Oui.	Oui, statique.
	Modèle de l'exécution	Non	navigation de routes.	Non

TABLE 3.2 – Résumé de middlewares d'intégration basées sur des EIP

5 Synthèse

Le besoin d'intégration n'est pas un problème récent. L'évolution des entreprises et de la technologie amène la nécessité de construire de nouveaux systèmes. Néanmoins, pour minimiser les coûts, les industriels préfèrent réaliser l'adaptation des systèmes existants avec les nouveaux systèmes.

Les architectures d'intégration ont évolué au cours de temps. Les premières approches abordées étaient centrées sur des données. Ces approches se focalisent sur l'accès pour qu'ils soient consommés par des applications tierces. Une fois abordés ces problèmes d'accès, la communauté scientifique concentre ses efforts sur la représentation et la transformation des données. Cela pour faire des alignements requis par des applications. Des architectures élégantes proposent de découpler l'accès aux données de son usage, grâce à l'utilisation des couches intermédiaires, dites de médiation, qui se concentrent à réaliser des opérations propres aux données pour sa correcte utilisation par les applications.

Plus récemment, en raison de la complexité des systèmes et les besoins et de l'importance des applications existantes, les approches d'intégration se centrent sur l'intégration de systèmes et leur logique métier. Les solutions d'intégration commencent à se concentrer sur l'interaction entre systèmes. Architecturalement, ces solutions sont vues comme des solutions EAI, et plus récemment, avec l'avènement des approches à services, comme des solutions ESB.

Dans ce chapitre, nous avons vu de différentes architectures d'intégration. Nous nous sommes intéressés aux architectures d'intégration entre systèmes. Qui nous amène à faire une emphase sur des patrons d'intégration. Les patrons, dans certains degrés, influencent les architectures des solutions d'intégration. Nous avons terminé ce chapitre par une comparaison des solutions qui prenait en compte les patrons d'intégration comme solution a cette problématique. Comme résultat, nous avons constaté que ces solutions présentent une bonne approche de point de vue du génie logiciel lors de la conception. Néanmoins, les défis présentés précédemment, sur l'adaptabilité et dynamisme, sont peu abordés.

Dans le chapitre suivant, nous présentons notre proposition. Nous considérons la composition de patrons d'intégration comme un élément clé lors de la conception d'une solution d'intégration. Cela nous à amené à la définition d'un modèle à composant qui prend en compte la composition de tels patrons, ainsi qu'une machine d'exécution que nous permet d' attaquer les problématiques actuelles : le dynamisme et la considération des évolutions futures.

Deuxième partie

Proposition

4

PROPOSITION

Sommaire

1 **Problématique** 96
2 **Objectifs** 98
3 **Introduction sur les composants logiciels** 99
4 **Notre approche : CILIA** 102
5 **Synthèse** 109

Dans la première partie de cette thèse, nous avons présenté un état de l'art sur la problématique d'intégration, et plus particulièrement sur l'intégration de services. Dans un premier temps, nous avons examiné l'approche à services et les défis associés à l'intégration et à la composition de services. Nous avons constaté qu'un des enjeux majeurs de nos jours est celui de l'intégration dans les environnements ubiquitaires. Ceci est dû à la mobilité et au dynamisme qui sont intrinsèques à ce domaine. Ensuite, nous avons examiné des solutions architecturales qui répondent aux besoins d'intégration. Plus précisément, nos avons étudié les patrons d'intégration les plus utilisés aujourd'hui. Nous avons constaté un manque au niveau de la gestion du dynamisme et de l'adaptabilité des solutions actuelles.

La seconde partie de cette thèse présente notre proposition qui introduit un modèle à composant spécifique à la médiation qui répond à la problématique d'intégration entre services hétérogènes. De plus, cette approche aborde les défis actuels de dynamisme en fournissant une souplesse de reconfiguration à l'exécution.

Dans le premier chapitre de cette seconde partie, nous rappelons le contexte de notre travail et présentons une vision globale de notre approche. D'abord, nous discutons la problématique et les besoins dans les nouveaux domaines d'application. Ensuite, nous présentons des constats qui guident notre approche ainsi que les objectifs à aborder.

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 au sein des applications logicielles. L'informatique orientée service permet ainsi d'aborder de nouveaux domaines d'application, caractérisés notamment par de fortes contraintes de dynamisme.

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). Aujourd'hui de plus en plus d'applications, d'équipements, de ressources de toutes sortes sont exposées sous forme de services. On doit malheureusement noter que les technologies à services se sont également multipliées et que les environnements à services se caractérisent de plus en plus par leur grande hétérogénéité.

Nous avons alors abordé un problème majeur soulevé par le succès des services logiciels : l'intégration. 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 sémantique et les problèmes liés aux aspects non fonctionnels (sécurité, *logging*, performance, etc.).

Naturellement, nous nous sommes ensuite intéressés à la problématique d'intégration dans son ensemble. Il est apparu que l'intégration est depuis bien longtemps une préoccupation majeure des entreprises. Depuis l'apparition de l'informatique distribuée avec RPC (« Remote Procedure Call »)[BN84], il est devenu tout à fait classique de faire communiquer des applications au sein des Systèmes d'Information pour la mise en place d'applications nouvelles à des coûts raisonnables, c'est-à-dire sans tout redévelopper. Différents environnements d'intégration ont été proposés avec le temps. Les EAI (« *Enterprise Application Integration* ») reposent sur une architecture logique centralisée où toutes les interactions entre applications se font via un point de concentration unique. Les ESB (« *Enterprise Service Bus* ») développent une approche très similaire mais en l'appliquant exclusivement aux services.

Comprendre la notion de patron d'intégration est essentiel pour aborder la problématique d'intégration aujourd'hui. Les patrons d'intégration constituent un recueil de bonnes pratiques permettant de résoudre des problèmes récurrents en intégration. De nombreux projets d'intégration peuvent ainsi se réduire en un assemblage intelligent de patrons d'intégration bien connus et dont les propriétés sont clairement documentées dans différents ouvrages ([HT04, HW08] par exemple).

Il est d'ailleurs intéressant de noter que tous les outils d'intégration actuels, qu'il s'agisse d'EAI ou bien d'ESB, implantent les patrons d'intégration essentiels. Ils fournissent même, en général, des solutions plus ou moins génératives pour faciliter la mise en place de ces patrons. Aujourd'hui, la facilité de mise en œuvre des patrons d'intégration est clairement un critère d'évaluation majeur lors du choix d'un outil d'intégration.

Pour autant, il apparaît que les outils et approches actuels ne permettent pas de traiter les nouveaux problèmes d'intégration de façon satisfaisante, notamment ceux liés au cas épineux de l'informatique ambiante, des réseaux sociaux ou encore de l'informatique dans les nuages. En effet, ces nouveaux domaines ont fait évoluer la problématique d'intégration de façon radicale et de nouveaux défis très structurants sont apparus.

Plus précisément, nous avons fait les constatations suivantes :

- **Les besoins en intégration sont plus forts que jamais mais différents**. Les services numériques envahissent notre quotidien et sont de plus en demandés, que cela soit par les usagers, le « marketing », les décideurs, etc. Le développement de ces services comprend une forte composante directement liée à la problématique d'intégration logicielle. Cependant, les besoins précis (dynamisme, légèreté) ne sont plus ceux des systèmes d'information.
- **L'intégration est une tâche trop lourde**. Intégrer des services demande un niveau de compétence rarement disponible. Dans la pratique, les couches d'intégration sont réalisées aujourd'hui par des équipes spécialisées, souvent surchargées. Le modèle économique associé aux services numériques exige rapidité et flexibilité et ne peut pas être mis en œuvre par ce type d'organisation lourde et centralisée.
- **L'intégration est une tâche de plus en plus complexe**. Historiquement, l'intégration se concentrait sur des applications au sein d'un système d'information. Comme nous l'avons indiqué, la situation a évolué de façon radicale et il s'agit dorénavant d'intégrer des ressources présentes dans l'ensemble de notre environnement : maisons, bâtiments tertiaires, usines, voitures, trains, etc.
- **L'hétérogénéité des ressources est de plus en plus grande**. L'arrivée des services n'a rien changé à l'affaire : les environnements informatiques sont caractérisés par une hétérogénéité sans cesse croissante. C'est à la couche d'intégration qu'il revient de gérer la grande diversité des standards, des protocoles de communication, des supports réseaux et bien sûr des sémantiques.
- **La dynamicité des ressources est de plus en plus grande**. Les nouveaux domaines tels que l'informatique ambiante, les réseaux sociaux ou l'informatique dans les nuages se caractérisent par une grande dynamicité. Les ressources à intégrer apparaissent et disparaissent, les interfaces et les propriétés de ces ressources évoluent également de façon fréquente et de manière autonome.
- **Les besoins liés à l'intégration évoluent également**. Pour une même application, les besoins en intégration évoluent. Cela signifie par exemple que les propriétés non fonctionnelles associées à une intégration ne sont pas figées et demandent des adaptations au cours du temps.
- **L'arrêt de fonctions d'intégration est de moins en moins accepté**. De nombreux services demandent une disponibilité continue. Les interruptions de service sont en effet couteuses dans de nombreux domaines. Ceci complexifie grandement les opérations de maintenance, qu'elles soient de nature corrective ou évolutive.

2 Objectifs

L'objectif de notre travail est de fournir des environnements logiciels de conception et d'exécution de solutions d'intégration permettant d'aborder les domaines dynamiques, hétérogènes et contraints. Il s'agit également de faire en sorte que la création et la maintenance des solutions d'intégration soient aisées et naturelles et ainsi accessibles à des non spécialistes de l'intégration.

De façon plus détaillée, l'objectif de notre travail est de définir et d'implanter des environnements de conception et d'exécution répondant aux exigences suivantes :

- **Faciliter la conception des solutions d'intégration**. Comme nous l'avons constaté, la conception de solutions d'intégration est une tâche trop lourde et trop complexe aujourd'hui. Nous souhaitons donc définir des méthodes et des outils facilitant autant que possible la conception de ces solutions d'intégration. En particulier il est important de s'adresser aux spécialistes des domaines à intégrer et non pas seulement aux spécialistes des technologies d'intégration.
- **Fournir un modèle de développement « 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.
- **Faciliter l'implantation des patrons d'intégration**. Comme nous l'avons vu, les patrons EIP (Enterprise Integration Patterns) jouent un rôle majeur aujourd'hui. Notre proposition doit donc faciliter leur mise en œuvre et leur association. L'assemblage et la manipulation de patrons doit donc pouvoir se faire de façon naturelle.
- **Permettre les évolutions dynamiques**. La capacité de faire évoluer une solution d'intégration sans l'interrompre est une exigence majeure pour aborder les domaines que nous visons. Cette exigence est bien sûr très structurante et difficile à réaliser. Elle demande de connaître l'état interne d'une solution d'intégration et la capacité de l'adapter tout en conservant données et flots de contrôle.
- **Faciliter la gestion des erreurs**. Il s'agit d'une exigence de première importance, d'autant plus que l'intégration renvoie à des enchaînements d'opérations impliquant des services distants. De nombreuses erreurs peuvent ainsi se produire au niveau des opérations d'intégration elles-mêmes ou bien lors de la communication avec les services distants (absence de réponse, réponses incorrectes ou partielles, etc.). En pratique, une partie importante du code d'intégration est ainsi dédiée à la gestion des erreurs.
- **Limiter la taille des environnements d'exécution**. Dès lors que l'on quitte le monde des systèmes d'information, les ressources pour l'exécution d'applications sont plus faibles. Il nous faut alors être capable d'exécuter des solutions d'intégration sur des cibles relativement contraintes telles que des passerelles ou des mini-PC.
- **Faciliter l'installation et la gestion**. Comme une solution d'intégration doit être réalisable « à la demande » lors de la création d'un nouveau service intégré, l'installation, la configuration et l'administration doivent être simples et supportées par des outils appropriés.

3 Introduction sur les composants logiciels

Avant de présenter l'approche que nous avons mise en place pour répondre aux objectifs fixés, nous allons faire une brève introduction sur la notion de composant logiciel puisque notre proposition s'appuie en grande partie sur ce paradigme.

La programmation orientée composant (Component-Based Software Engineering en anglais) est une discipline du génie logiciel qui est destinée à la construction d'applications par la composition de briques logicielles prédéfinies. Cette approche s'inspire d'autres disciplines d'ingénierie, comme par exemple le génie électronique, où la construction de nouveaux dispositifs se fait en grande partie par assemblage de composants électroniques préexistants.

L'approche à composants est définie ainsi par Clemens Szyperski :

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

Cette définition décrit les composants comme des unités de composition caractérisées par leurs interfaces et leurs dépendances. Szyperski présente les composants comme des boîtes noires, mais il ne décrit pas le rôle du composant dans l'architecture. Une autre définition, qui nous semble plus complète, est proposée par Heineman et Councill [HC01] :

> « A software component is a software element that conforms to a component model and can be independently deployed and composed without modification according to a composition standard »

Cette seconde définition introduit la notion de modèle à composant qui revêt une importance majeure dans le développement effectif de toutes applications à composants.

3.1 Modèle à composant

Un modèle à composant est la spécification des standards et les conventions imposées aux développeurs de composants [BBB+00] pour la définition, l'implantation et l'assemblage des composants.

Dans un modèle à composant, les composants sont les entités logicielles que l'on peut déployer et exécuter. Le modèle définit un ensemble de règles à respecter pour décrire et implanter les composants. Les composants peuvent être décrits dans un langage de description de composants mais aussi, directement dans un langage de programmation. Les EJB, par exemple, sont souvent développés en suivant une convention d'usage d'annotations dans le code.

De nombreux modèles à composants permettent de spécifier certains aspects non fonctionnels et de générer le code correspondant. Dans les EJB, par exemple, les transactions ou la sécurité sont ainsi simplement spécifiés par les développeurs. Ainsi, à la compilation, le code nécessaire à leur prise en compte (des indirections par exemple) est ajouté.

Les descriptions de composants doivent contenir toute l'information nécessaire à leur exécution et à leur assemblage. Il est nécessaire, en particulier, de définir formellement les interfaces d'interaction entre composants. Une interface peut être basée sur des opérations ou sur des ports. Dans le premier cas, l'interface fournit une description détaillée des opérations qui peuvent être utilisées par d'autres composants pour interagir avec eux. Cette description précise notamment les paramètres et leur type. Le second cas est communément utilisé pour mettre en place des approches événementielles où le niveau de compatibilité est plutôt sémantique. Cela signifie que les composants émetteur et récepteur doivent s'accorder sur les valeurs possibles des événements.

Une notation permettant de décrire des assemblages de composants et leurs configurations est appelé un ADL (pour Architecture Description Language en anglais) [Cle96]. Un ADL peut être formel ou semi-formel (comme UML par exemple) et impose un ensemble de règles pour les compositions. On trouve deux grandes familles d'ADL : les approches génératives qui sont utilisées pour produire du code dans une plate-forme cible [GLV+03, RABA, MKB+04, RCBO06] et les approches interprétées où les ADL sont interprétés par un framework d'exécution[LV95, EHL07].

Les patrons d'interaction expriment le rôle de chaque composant pendant l'échange de messages d'une composition. Trois patrons sont majoritairement employés :

- Requête/Réponse : cette interaction est de type client/serveur où le client envoie une requête (i.e. un appel de méthode) et le serveur retourne une réponse. Ce patron peut être appliqué en utilisant des interfaces basées sur des opérations ou sur des ports. L'utilisation de ports implique que le message passé dans la requête contient l'information du port pour recevoir la réponse.
- Push : cette interaction est déclenchée par un composant émetteur. Ce composant déclenche l'exécution du composant passif en lui envoyant des données. Le type d'interface dans une interaction de type push est surtout basé sur des ports.
- Pull : cette interaction est déclenchée par le composant récepteur. C'est ce composant récepteur qui fait une requête au composant émetteur qui est passif. Le type d'interface dans une interaction de type pull est surtout basé sur des opérations.

Les patrons d'interactions peuvent être bloquants ou non bloquants (synchrone ou asynchrone). Le patron requête/réponse est couramment bloquant quand les interfaces sont basées sur des opérations. Par contre, il est non bloquant quand les interfaces sont des ports. Le patron push peut être bloquant ou non bloquant, alors que le patron pull est surtout bloquant.

Les liaisons entre composants ont ainsi toujours une direction : un composant actif prend le contrôle et le composant passif suit l'interaction. L'opération de lier deux composants peut être automatique ou descriptive. Une liaison descriptive est décrite à l'aide d'un ADL qui spécifie le composant source (actif) et son interface, ainsi que le composant cible (passif) et son interface. Une liaison automatique est faite grâce à une déduction de composant cible et de son interface. Cette déduction est faite afin de satisfaire les besoins du composant source. Comme exemple d'une liaison automatique, on peut citer la liaison dans une approche à service.

3.2 Framework d'exécution

Le développement d'applications en suivant un modèle à composant est fortement influencé par le framework d'exécution. Le framework d'exécution est l'ensemble des techniques et technologies utilisées pour exécuter les applications fondées sur le modèle à composant. Ce framework peut utiliser des techniques d'interprétation ou de compilation. Il apporte souvent des restrictions au développement de composants. Par exemple, le langage de programmation utilisé ou la plate-forme d'exécution.

Le framework d'exécution participe activement à l'exécution d'applications basées sur le modèle à composant. A partir du déploiement de composants, le framework prend en charge les mécanismes de gestion de cycle de vie des composants et des ressources. Par exemple l'initialisation, l'arrêt et parfois la résolution de dépendances.

Les modèles à composant spécifient le cycle de vie d'applications, c'est-à-dire, comment concevoir une application en suivant le modèle à composant, comment déployer l'application, ainsi que les possibilités de maintenance lors de son exécution.

- La conception. Pendant la conception d'applications, certains modèles à composant fournissent des outils pour le développement. Ces outils peuvent être des IDE (Integrated Développement Environnement en anglais), des outils de génération de code, des compilateurs ou des outils d'emballage. L'utilisation de tels outils est importante pour un usage correct des modèles à composant. Pour certains modèles, les outillages sont indispensables pour maîtriser effectivement le modèle, alors que pour d'autres, l'outillage est une option qui facilite le développement d'applications. En l'absence d'outillage, les développements de composants et d'applications conformes à un modèle à composant donné sont réalisés par des experts de ce modèle.
- Le déploiement. Une approche à composant permet de construire des applications en utilisant des composants indépendants et parfois développés par des tiers. En conséquence, l'utilisation d'un composant peut provoquer des dépendances vers d'autres composants. Cette dépendance entre composants est abordée par des outils de déploiement intelligent. Ces outils font la résolution de dépendances automatiquement. Un autre exemple d'outillage de déploiement est les outils d'approvisionnement. Ces outils font le déploiement automatique de nouveaux composants. Néanmoins, cette approche ne gère pas la résolution des dépendances, mais elle fournit un nouveau service qui peut être composé par un ensemble de composants. Un dernier exemple est l'approche de déploiement distribué [HM06, Fou] où l'on peut spécifier le nœud à déployer pour chacun des composants.
- L'administration. Les facilités de gestion du cycle de vie d'une application à composants peut comprendre la phase de maintenance. En d'autres termes, certains modèles à composant fournissent des outillages d'administration et de paramétrage. Citons comme exemples d'outils de gestion les Consoles Web, les Shells d'administration[SMF+09], ou encore des services exposés qui peuvent être exploités par systèmes tiers [JN08, LLB+09].

4 Notre approche : CILIA

4.1 Vision globale

Nous avons développé un modèle à composant spécifique à l'intégration logicielle, nommé CILIA.

Ce modèle repose sur des composants, appelés médiateurs, et sur un langage d'assemblage de ces médiateurs. Il comprend également un framework dynamique. La conception de CILIA a été influencée par :

- Les bonnes pratiques de Génie Logiciel : CILIA a pour objectif d'être modulaire, simple et vise à mettre en place une bonne séparation des préoccupations,
- Les middlewares adaptatifs : CILIA a pour objectif d'utiliser au mieux les techniques d'introspection et d'intercession de façon à mettre en place la dynamicité,
- Les patrons d'intégration (EIP) : CILIA a pour objectif de rendre l'implantation des patrons aussi naturelle et directe que possible.

Il en résulte un modèle à composant homogène, adapté au métier de l'intégration et un framework d'exécution modulaire et flexible qui permet la modification, l'ajout, le retrait ou le remplacement à chaud de tous médiateurs.

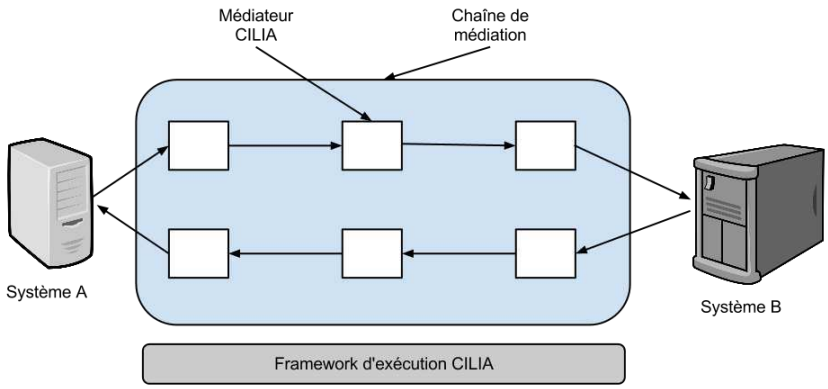

FIGURE 4.1 – Vision globale de CILIA.

Comme illustré par la Figure 4.1 ci-dessus, CILIA repose sur les principes suivants :

- Une solution d'intégration prend la forme d'un assemblage de médiateurs qui communiquent par flots de données. Cet assemblage forme une chaîne reliant les applications à intégrer,
- Un médiateur réalise une unique opération dite de médiation à partir de données reçues et transmet le résultat au(x) médiateur(s) suivant(s). Il peut s'agir d'une opération de filtrage, d'alignement sémantique, d'enrichissement de données, de sécurisation, etc.
- L'architecture logique d'intégration est centralisée puisque toute médiation passe par le framework CILIA. Cela dit, le framework CILIA peut être distribué. Ceci permet un bon découplage entre applications à intégrer.

4.2 Composants CILIA

Les composants CILIA sont des entités exécutables qui réalisent des tâches de médiation, d'où leur nom. L'objectif de ces tâches est de permettre l'interaction effective entre deux entités logicielles développées indépendamment et qui, a priori, n'ont pas de connaissance l'une de l'autre.

Comme indiqué précédemment, le rôle d'un modèle à composant est de définir un formalisme et des règles pour guider et contraindre le développement de composants. Dans le cas de CILIA, un composant est défini comme suit :

> « 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. »

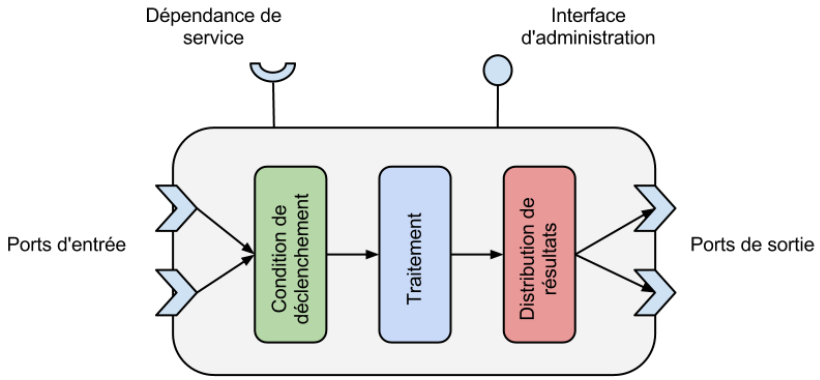

FIGURE 4.2 – Structure des composants « médiateur ».

De façon plus précise, les ports permettent de recevoir les données à traiter et de stocker les résultats. Les ports d'entrée et de sortie sont typés. Cela signifie qu'un port ne peut recevoir que des données d'un même type. Les ports de sortie permettent la connexion vers les autres médiateurs (via leurs ports d'entrée) ou vers les applications à intégrer.

Les dépendances de code sont exprimés sous forme de dépendances de services, ce qui permet une résolution à l'exécution. Les dépendances de services expriment ainsi des besoins fonctionnels qui doivent être complètement résolus pour pouvoir effectuer les traitements. Ceci est directement inspiré des approches à services précédemment présentées. Une dépendance permet d'interagir avec un service externe pour affiner une méthode de calcul ou stocker une donnée à distance par exemple.

Enfin, les interfaces d'administration permettent de gérer le cycle de vie des médiateurs. Nous verrons par la suite que le framework d'exécution associé à ces composants permet de modifier ou de remplacer à chaud les médiateurs CILIA.

La structure interne d'un médiateur a été conçue de façon à faciliter le travail des concepteurs et des développeurs de solutions d'intégration. En effet, un médiateur permet d'exprimer trois aspects essentiels à toutes opérations d'intégration :

- **Quand traiter les données ?** La condition de déclenchement est l'algorithme qui prend la décision de lancer le traitement des données, c'est-à-dire qu'il déclenche l'exécution du principal algorithme du médiateur en fonction, par exemple, des données reçues ou du temps.
- **Comment traiter les données ?** Le traitement des données est la fonction cœur du médiateur qui met en œuvre un algorithme de médiation nécessaire à l'intégration. Cela peut être une transformation, un filtrage, un enrichissement, une traduction, etc.
- **A qui transmettre les données ?** La fonction de distribution des résultats dépose les données traitées vers les ports de sortie. Nous verrons que ceci permet de mettre en œuvre différents types de routage au sein des chaînes de médiation tels que le routage basé sur le contenu ou le multicast.

Nous avons vu précédemment (chapitre 3) que les patrons d'intégration peuvent se répartir en trois catégories : les patrons de réception, les patrons de transformation et les patrons de routage. Le modèle des composants CILIA permet d'implanter ces différents patrons en rassemblant les préoccupations liées d'un point de vue « métier » au sein de modules homogènes, les médiateurs. Cela permet d'éviter une dispersion des traitements et l'apparition de concepts hétérogènes comme dans les frameworks présentés au long du chapitre 3.

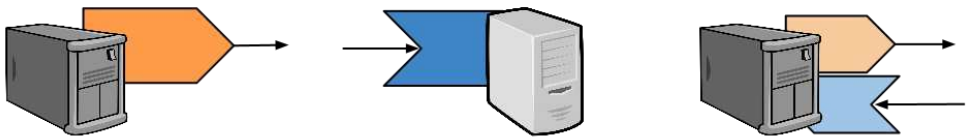

FIGURE 4.3 – Adaptateur d'entrée, de sortie et d'entrée/sortie (de gauche à droite)

CILIA définit cependant des connecteurs de façon à interagir avec les applications à intégrer : cet type de connecteurs sont sous la forme d'adaptateurs. Les adaptateurs sont chargés de la communication entrante ou sortante d'une chaîne de médiation. Ils définissent une primitive de communication vers l'extérieur de la chaîne de médiation.

CILIA définit trois adaptateurs de base qui se caractérisent par leur la primitive de communication :

- Les adaptateurs d'entrée traitent uniquement de messages entrant dans la chaîne (in-only) ;
- Les adaptateurs de sortie traitent uniquement de messages sortant dans la chaîne (out-only) ;
- Les adaptateurs d'entrée/sortie gèrent des interactions de type requête/réponse en sortie de chaîne (request/reply).

Les adaptateurs reprennent une partie seulement de la structure interne d'un médiateur. Par exemple, la condition de déclenchement n'existe pas sur les adaptateurs d'entrée.

4.3 Configuration d'architectures en CILIA

La configuration d'une application de médiation en CILIA est appelé chaîne de médiation. En d'autres termes, une chaîne de médiation est l'ensemble de composants de médiation, médiateurs et adaptateurs, et les interconnexions entre eux. Cette composition est chargée de réaliser toutes les opérations nécessaires pour faire inter-opérer un ou plusieurs éléments logiciels.

Une chaîne de médiation est dirigée par les données (*data-flow-driven* en anglais). Plus précisément le modèle de calcul est le suivant :

> Le calcul est initié par un médiateur particulier, appelée adapteur, qui fait le lien avec les applications clientes et transmet des données à un premier médiateur. Ensuite, chaque médiateur applique un traitement sur les données reçues dès que les conditions de déclenchement sont satisfaites et propage les résultats vers les médiateurs qui lui sont connectés. Le calcul se termine sur un adapteur de sortie.

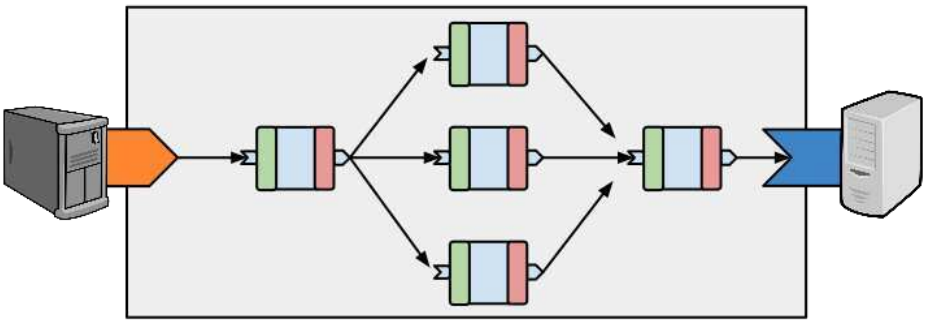

FIGURE 4.4 – Exemple de chaîne de médiation.

Ce style d'architecture permet de construire des applications dirigées par les données où les composants sont inactifs jusqu'au moment de la réception de données. Ce style présente un découplage important entre les systèmes à intégrer mais aussi au sein même de la solution d'intégration.

La séparation des préoccupations est d'ailleurs l'un des aspects clé dans CILIA. Les médiateurs sont en effet bien séparés les uns des autres et réalisent des opérations métier bien distinctes. Egalement, il y a séparation entre les opérations de traitement des données et les opérations techniques de synchronisation, de transport des données, etc.

Enfin, il convient de noter que les flots de données ont un sens bien défini. Chaque composant dans une chaîne de médiation a une direction de communication. Les composants ne peuvent pas gérer des communications dans deux sens différents : par exemple, deux médiateurs ne peuvent échanger suivant un mode « *peer-to-peer* ».

Afin de réaliser des chaînes qui ont du sens et qui sont familières aux experts de l'intégration, l'assemblage de composants doit suivre un patron d'intégration dès que cela est possible. Nous montrerons dans la suite de ce document comment implanter les patrons d'intégration majeurs en utilisant simplement CILIA.

4.4 Framework d'exécution

Le framework d'exécution du modèle à composant CILIA permet d'exécuter les médiateurs et les chaînes de médiation. Comme indiqué précédemment, sa construction s'est inspirée des middlewares adaptatifs et de la technologie des composants orientés service.

Le framework CILIA maintient un modèle structurel à l'exécution des applications de médiation et peut introspecter les applications à l'exécution. Cette connaissance permet de faire des reconfigurations des chaînes de médiation à chaud, c'est-à-dire lors de son exécution et sans interrompre l'exécution.

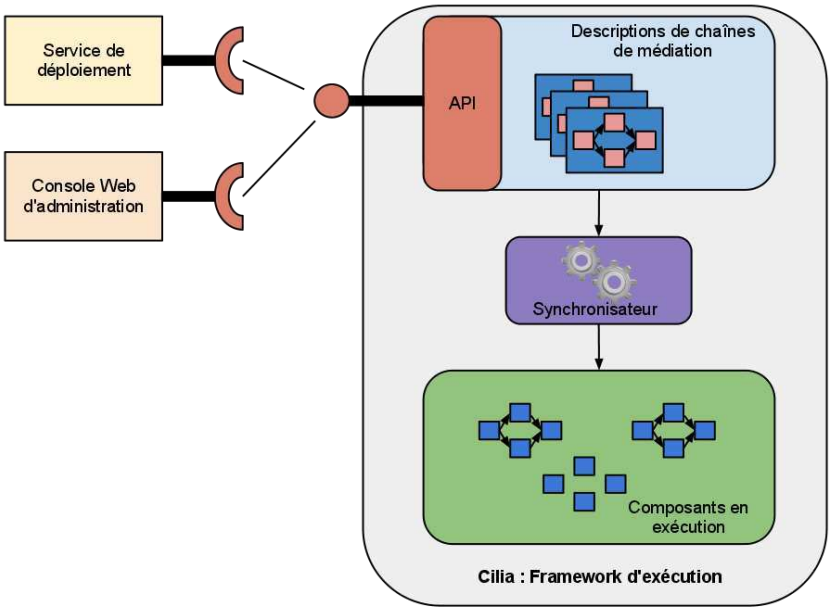

FIGURE 4.5 – Framework d'exécution

Une brève représentation du framework d'exécution est fournie par la figure 4.5 ci-dessus. Il est composé principalement de trois éléments :

- le premier, un conteneur de description de chaînes de médiation. Cet élément contient les descriptions de chaînes de médiations. Ces descriptions sont faites en utilisant, par exemple, un langage de description ;
- ensuite, une API qui permet l'interaction des outillages avec les descriptions d'applications de médiation ;
- après, l'ensemble d'instances de composants de médiation. Ces objets sont la réification des descriptions de chaînes contenues dans le conteneur de descriptions. Ces objets sont chargés de réaliser les opérations de médiation pour intégrer des applications hétérogènes ;
- et finalement, un synchronisateur. Cet élément est chargé de réifier les descriptions d'architectures de chaînes de médiation comme instances de composants.

La figure montre aussi deux éléments comme exemple. Un service de déploiement et une console Web d'administration. Ces deux éléments utilisent l'API du framework d'exécution pour créer de chaînes de médiation ou pour les administrer : modifier des paramétrages ou restructurer l'architecture.

Le framework d'exécution permet d'avoir les structures des chaînes de médiation principalement pour deux raisons : 1) pour monter en abstraction de l'information des applications et 2) pour manipuler des détails des architectures à l'exécution sans interagir directement avec les composants en exécution. Cette fonctionnalité est inspirée sur des travaux de *middleware* reflexives[KCBC02, BCG04].

De plus, la manipulation obtenue grâce à une API d'administration permet la construction de logiciels plus complexes, par exemple, les gestionnaires autonomiques. Un gestionnaire autonomique peut être chargé d'introspecter le comportement de l'application de médiation et réagir selon une analyse à tel comportement. Par exemple, modifier la structure d'une chaîne de médiation à l'exécution.

Cette faisabilité ajoutée par l'API nous permet d'aborder des défis actuels dans des environnements ubiquitaires et les nouveaux domaines d'application. Dans ces environnements il y a un fort dynamisme et mobilité qui peut influencer le comportement d'applications et pousser des évolutions continues.

4.5 Cycle de vie

Le cycle de vie associé au modèle à composant (voir 4.6) CILIA comprend la conception de composants de médiation, l'assemblage de chaînes de médiation, le déploiement dans une plateforme d'exécution et la gestion des chaînes de médiation à l'exécution.

La conception d'applications est réalisée en deux étapes : 1) le développement de composants de médiation, c'est-à-dire la définition des algorithmes qui effectuent les opérations de médiation et 2) la composition des chaînes de médiation. Cette conception est bien sûr conforme au modèle à composant CILIA à tous les niveaux de développement.

Le résultat d'assemblage est un artefact prêt au déploiement. En plus, une configuration du déploiement est fréquemment nécessaire pour spécifier des détails plus concrets, comme par exemple la localisation d'une base de données. Le framework d'exécution, présenté dans la section suivante, prend le contrôle de l'application lors de son déploiement.

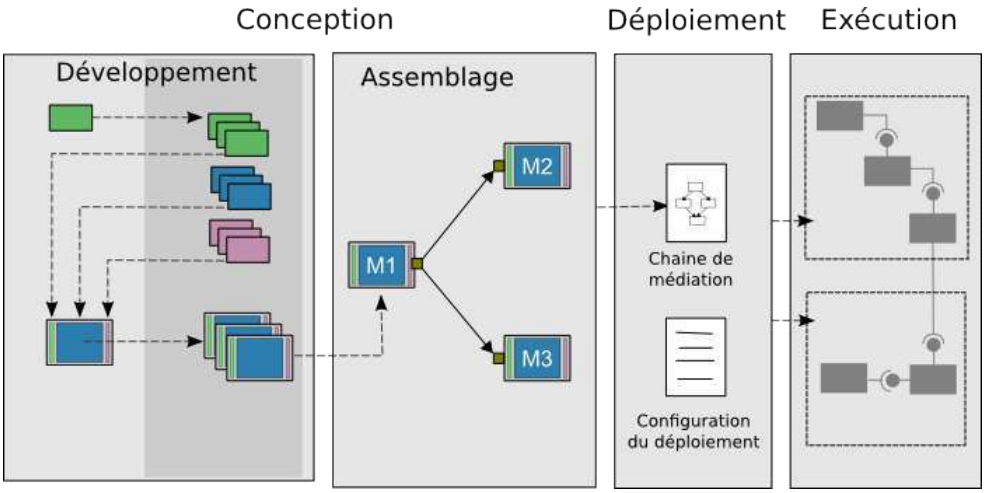

FIGURE 4.6 – Cycle de vie

Comme présenté dans cette section, les médiateurs sont des entités qui répondent à trois questions d'intégration. Cette séparation nous permet de construire des algorithmes indépendants, où selon les généralités de conception, peuvent être utilisé dans divers contextes, c'est-à-dire dans divers chaînes de médiation. Ainsi, cette modularité encourage le développement d'entités généralistes grâce à la construction indépendante de chaque entité. En plus, la compatibilité entre ces entités au niveau métier est seulement sémantique.

La modularité et la séparation des préoccupations sont des principes fondamentaux du génie logiciel que permet facilement d'anticiper les évolutions futures, ainsi que réaliser des adaptations au moment de l'exécution. Ceci est réalisé grâce au fait que les architectures conçues permettent d'échanger des composants et d'ajouter de nouveaux composants sans impacter les composants existants.

5 Synthèse

Dans ce chapitre, nous avons présenté la vision générale de notre approche : un modèle à composant spécifique à la médiation nommé CILIA. Ce modèle vise la conception de couches de médiation dynamiques. L'un des objectifs du modèle à composant CILIA est d'exploiter les connaissances en intégration, notamment formalisées par les patrons d'intégration EIP, pour fournir un modèle répondant aux principes de génie logiciel (abstraction, modularité, séparation de préoccupations, simplicité notamment). Ceci afin de produire du code de médiation de qualité qui soit facile à maintenir et à faire évoluer à l'exécution.

La prise en considération des principes de génie logiciel permet de construire des solutions d'intégration modulaires et flexibles. Cette modularité est effective aussi bien à la conception qu'à l'exécution. Les composants, de même que certains éléments internes, peuvent être des modules indépendants. La séparation des préoccupations est un des aspects le plus importants de CILIA. Le modèle sépare en effet les opérations métier (comme la transformation, l'alignement sémantique, etc.) des opérations purement techniques (comme la communication, l'ordonnancement, etc.).

Le modèle CILIA permet de définir des applications de médiation, c'est-à-dire les architectures et les médiateurs à un haut niveau d'abstraction, grâce à la manipulation des concepts spécifiques au domaine de la médiation pendant la spécification ainsi que pendant le développement.

En conclusion, CILIA répond aux défis exprimés dans la section 3 dans ce chapitre :

- il est centré sur un domaine : il facilite donc la conception des solutions d'intégration ;
- il prend en compte l'expertise du domaine : il permet ainsi l'implantation naturelle des EIP ;
- il prend en compte des défis actuels : il autorise donc des évolutions dynamiques ;
- il est simple et basé sur un nombre limité de concepts : il est ainsi ouvert et utilisable ;
- il est embarquable : il peut donc traiter le domaine de l'informatique pervasive ;
- il est introspectable : il rend ainsi la gestion des erreurs plus aisée ;

Le chapitre suivant présente les détails de notre proposition.

5

Le modèle à composant CILIA

Sommaire

1 **Vers des composants d'intégration** 112
2 **Composants CILIA** 113
3 **Assemblage CILIA** 118
4 **Configuration CILIA** 121
5 **Framework d'exécution CILIA** 123
6 **Cycle de vie CILIA** 125
7 **Synthèse** 129

Dans le chapitre précédent, nous avons présenté une vision globale pour le développement de solutions d'intégration. Nous avons introduit un modèle à composant spécifique à la médiation, nommé CILIA, ainsi que sur le framework d'exécution associé.

Nous débutons ce chapitre en mettant l'accent sur la classification de patrons d'intégration qui a fortement influencé ce modèle à composant spécifique appelé CILIA. Ensuite, nous décrivons le modèle à composant en deux parties. La première décrit les concepts du modèle à composant, ainsi que la composition, à l'aide de méta-modèles (section 2, section 3 et section 4). La deuxième partie présente le cycle de vie du modèle à composant ainsi que le framework d'exécution (section 5 et 6).

1 Vers des composants d'intégration

Le métier de l'intégration aujourd'hui est très fortement influencé par les patrons d'intégration définis pour la plupart dans [HW08]. Ces patrons synthétisent une expertise précieuse acquise et validée par une multitude de projets en intégration logicielle. Ces patrons offrent ainsi des solutions pérennes à des problèmes d'intégration récurrents que nous nous devons d'utiliser.

Nous avons catégorisé les patrons d'intégration de la manière suivante :

- patrons de réception de données : ces patrons traitent de l'acquisition des données à intégrer et de leur préparation pour un futur traitement. Comme exemples de patrons, nous pouvons citer le « *correlation identifier* » ou encore le « *resequencer* » ;
- patrons de traitement de données : ces patrons permettent la réalisation d'opérations de médiation sur les données. Comme exemples de patrons, nous pouvons citer le « *translator* », le « *content enricher* », le « *aggregator* » ou encore le « *splitter* » ;
- patrons de routage de données : ces patrons traitent du routage de données une fois traitées. Citons par exemple les patrons « *content-based dispatcher* », « *dynamic routing* » et « *routing slip* » ;
- patrons d'administration : ces patrons répondent à la problématique de la gestion ou de la surveillance du bon fonctionnement de l'application. Comme exemples de ces patrons, nous trouvons le « wire-tap » et le « control bus ».

Nous avons également vu qu'une solution d'intégration peut être vue comme une composition de ces patrons. Ainsi :

> *Toute solution d'intégration doit mettre en œuvre les trois types de patrons d'intégration : réception de données, traitement de données, et routage de données.*

Il nous semble dès lors évident que tout outil d'intégration se doit de suivre ces patrons ou, tout du moins, permettre leur mise en place de façon aisée et naturelle. Dans cette thèse, nous proposons un modèle à composant spécifique au domaine de l'intégration visant à remédier aux limites des approches actuelles. Plus précisément, notre modèle vise à répondre aux objectifs de simplicité, d'ouverture, de dynamicité, d'embarquabilité, et d'adéquation naturelle au métier décrit dans le chapitre précédent.

Ce modèle porte le nom de CILIA. Comme tout modèle à composant, CILIA se caractérise par un langage de définition des composants, par un langage d'assemblage des composants, par un langage de configuration et par un langage de gestion du cycle de vie des composants.

CILIA définit les notions de « médiateur », de « connecteur » et de « chaîne de médiation » pour référer aux composants, aux assemblages et aux configurations. Ces concepts sont présentés en détail dans ce chapitre ainsi que le cycle de vie d'une application d'intégration développée en CILIA et le framework d'exécution de support.

2 Composants CILIA

2.1 Notion de médiateur

Un médiateur est un composant, c'est-à-dire une entité exécutable, qui réalise une « tâche de médiation ». Une telle tâche comprend la réception et la mise en forme de données, l'application d'une opération de traitement des données et le routage du résultat vers d'autres médiateurs ou vers les applications à intégrer.

Le but de notre approche est ainsi de regrouper un ensemble de traitements spécifiques à la médiation dans une seule structure homogène. Cette structure, le médiateur, a du sens pour les experts du métier et permet des manipulations plus modulaires lors de la conception et la maintenance des solutions d'intégration.

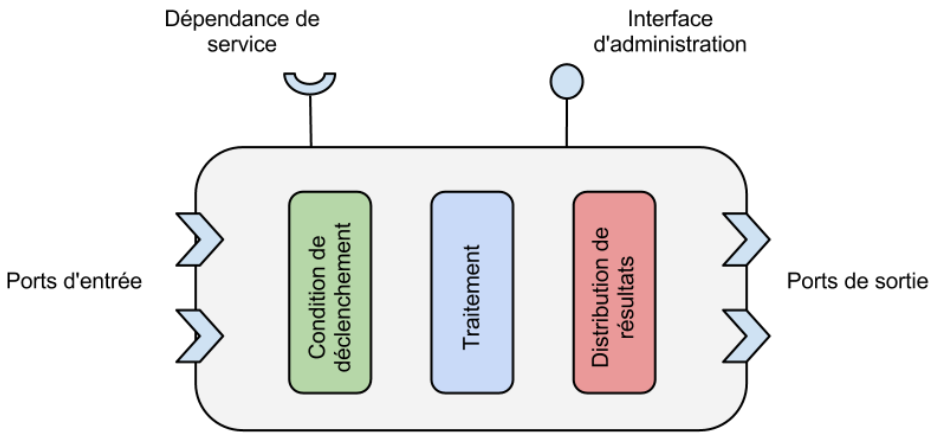

FIGURE 5.1 – Médiateur CILIA

Nous définissons un médiateur comme suit :

> *« Un médiateur CILIA est un composant constitué de ports d'entrée et de ports de sortie typés et d'une opération de médiation. L'opération de médiation est définie par une fonction de déclenchement, une fonction de traitement de données et par une fonction de routage des résultats. 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. »*

Comme illustré par la figure ci-dessus, la fonction de déclenchement est appelée « *scheduler* ». Son rôle est de recueillir les données au niveau des ports d'entrée et de les transmettre au bon moment et sous une forme appropriée à la fonction de traitement.

La fonction de traitement est appelée « *processor* ». Son rôle est de réaliser les opérations d'intégration proprement dite.

La fonction de routage des résultats, enfin, est appelée « dispatcher ». Son rôle est de déposer les résultats dans les ports de sortie adéquats et de déclencher les envois.

Les ports permettent de faire le lien avec des applications à intégrer ou avec d'autres médiateurs. Ils sont typés : Seuls des ports de même type peuvent être connectés. Les types sont spécifiques à une application d'intégration et véhiculent une sémantique précise. Ils sont définis à l'aide d'un symbole unique au sein d'une application. Pour une solution d'intégration donnée, tous les types doivent être définis de façon non ambiguë.

Les dépendances de code sont exprimées sous forme de dépendances de services, ce qui permet une résolution à l'exécution. Les dépendances de services expriment ainsi des besoins fonctionnels qui doivent être complètement résolus pour pouvoir effectuer les traitements. Ceci est directement inspiré des approches à services précédemment présentés. Une dépendance permet d'interagir avec un service externe pour affiner une méthode de calcul ou stocker une donnée à distance par exemple.

Les interfaces d'administration permettent de gérer dynamiquement le cycle de vie des médiateurs.

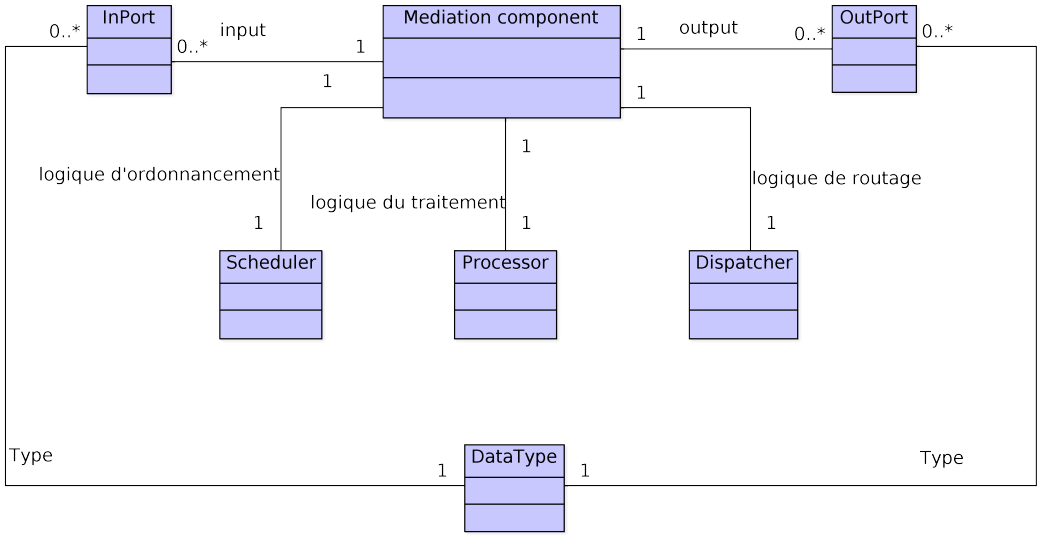

FIGURE 5.2 – Modèle d'un médiateur CILIA

Cette définition des composants de médiation CILIA, synthétisée par la figure ci-dessus, permet de regrouper les trois aspects d'une opération d'intégration au sein d'une même structure. En effet :

1. un médiateur est capable de gérer l'arrivée de données hétérogènes provenant de plusieurs sources de façon asynchrone,
2. un médiateur est capable d'appliquer un traitement d'intégration aux données reçues,
3. un médiateur est capable de gérer le cheminement des résultats pour de futurs traitements.

La logique de fonctionnement est simple : les données à intégrer sont déposées sur les ports d'entrée en respectant les contraintes de type. Lorsque la condition de déclenchement est satisfaite, le traitement est lancé avec les données nécessaires. A l'issue du traitement, les résultats sont routés vers les ports de sortie appropriés.

2.2 Rôle du *scheduler*

Une problématique majeure dans toute intégration est l'acquisition des données à traiter, leur regroupement suivant des critères temporels et/ou logiques et leur mise en forme. Cela soulève en effet des défis aussi bien techniques que métier. En particulier, il est nécessaire de traiter des problèmes complexes liés aux protocoles et patrons de communication, à l'association de données cohérentes, à la gestion des synchronisations, etc.

Le rôle du *scheduler* au sein d'un médiateur CILIA est de prendre totalement en charge cette problématique. Pour cela, un scheduler doit avoir les capacités suivantes :

- accéder aux données présentes sur ports d'entrée ;
- stocker les données reçues qui ne nécessitent pas un traitement immédiat ;
- analyser de façon périodique ou événementielle les données reçues et évaluer la possibilité de lancer la fonction d'intégration ;
- déclencher le traitement des données en transmettant les données pertinentes au moment opportun.

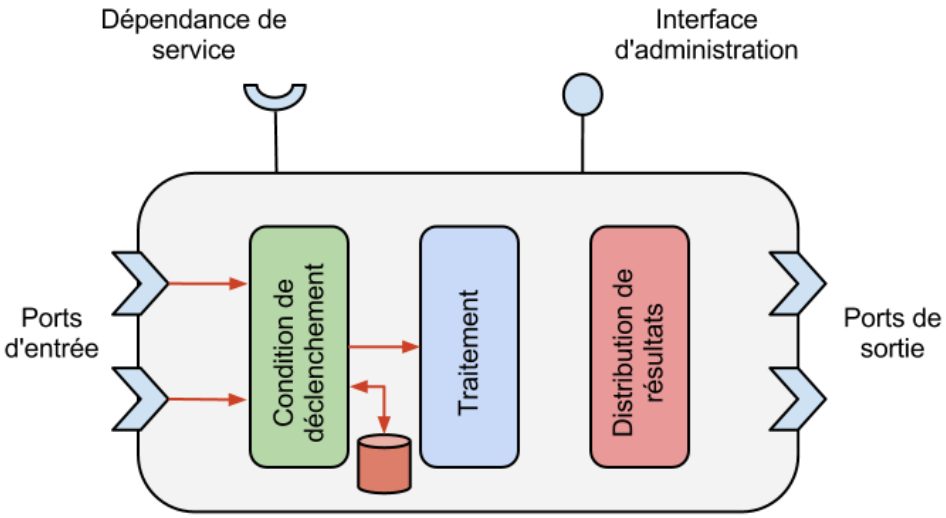

FIGURE 5.3 – Scheduler CILIA

La fonction de stockage des données est particulièrement importante puisque les données reçues ne sont pas toutes traitées immédiatement. Ainsi, le scheduler doit conserver les données reçues tant qu'elles ne sont pas traitées ou qu'une limite de taille ou de temps n'est pas atteinte. Le *scheduler* doit donc pouvoir écrire dans un support de stockage et accéder à ce support. Pour des raisons de performance, il est préférable que le support de persistance se trouve sur la même machine que le médiateur.

Nous verrons par la suite que CILIA fournit un framework de développement permettant de guider et de contraindre la définition des schedulers. Ceci est important à la fois pour simplifier le travail des développeurs des solutions d'intégration mais également pour garantir un fonctionnement optimal des médiateurs lors de l'exécution.

2.3 Rôle du *processor*

Le *processor* implante la fonction cœur du médiateur. C'est la fonction qui applique l'opération de médiation proprement dite sur les données. Typiquement, un *processor* peut réaliser les types d'opération suivants :

- transformation : cette opération consiste à modifier les données de telle sorte que la sémantique des données résultantes peut être complètement différente de celle des données d'entrée ;
- filtrage : cette opération permet de réduire la quantité ou la taille des données transférées. Le filtrage s'applique sur les données entrantes et baisse leur nombre ou leur complexité ;
- enrichissement : cette opération est utilisée pour ajouter de l'information aux données sans changer leur signification ;
- traduction : cette opération modifie la forme des données mais en gardant leur signification.

Un *processor* correspond typiquement à un algorithme implanté par une fonction ou une méthode. Cette fonction/méthode doit ainsi être appelée avec les bons paramètres. En fonction du niveau de généricité désirée, la variété des paramètres possible est plus ou moins grande.

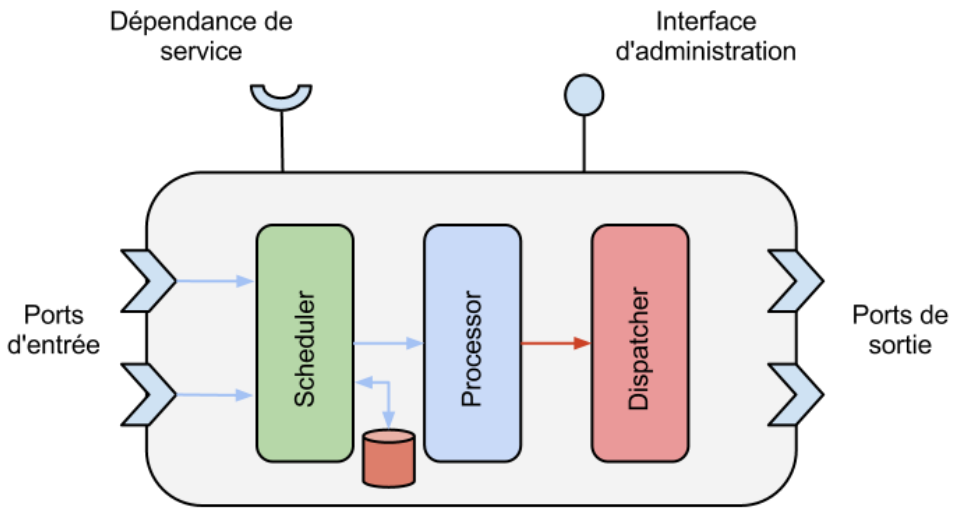

FIGURE 5.4 – Processor CILIA

Comme indiqué précédemment, c'est au *scheduler* de déterminer les paramètres du *processor* et le moment de l'appel. On comprend que scheduler et processor sont intimement liés : le *scheduler* connaît les attentes du *processor* pour lui fournir les paramètres attendus. Par contre, il ne connaît pas le traitement appliqué. De façon similaire, le *processor* ne veut pas connaître les problèmes de synchronisation et de mise en forme auquel le *scheduler* est confronté. Il veut simplement être appelé de façon appropriée.

Comme illustré par la figure ci-dessus, le *processor* transmet ses résultats au *dispatcher*. Ainsi, il ne soucie pas de l'utilisation futur des ses résultats. Il « se contente » de les assembler de manière cohérente et de les passer au *dispatcher*.

2.4 Rôle du *dispatcher*

La seconde problématique majeure dans les solutions d'intégration modulaires est le routage, c'est-à-dire l'orchestration des différentes opérations de médiation. Lorsqu'une opération est terminée, les résultats doivent être envoyés au(x) prochain(s) médiateur(s) ou aux applications à intégrer. Comme pour l'acquisition des données, le routage des données soulève des défis techniques et métier complexes. En particulier, il peut être nécessaire de dupliquer des données, d'analyser le contenu des résultats pour décider des routes, etc.

Le rôle du *dispatcher* au sein d'un médiateur Cilia est de traiter entièrement cet aspect. Le *dispatcher* implémente ainsi la logique de sortie des résultats du médiateur. Comme nous l'avons vu précédemment, la spécification d'un composant de médiation inclut la description d'un certain nombre de ports de sortie. La fonction du *dispatcher* consiste donc à choisir dans quel port de sortie chaque message doit être livré.

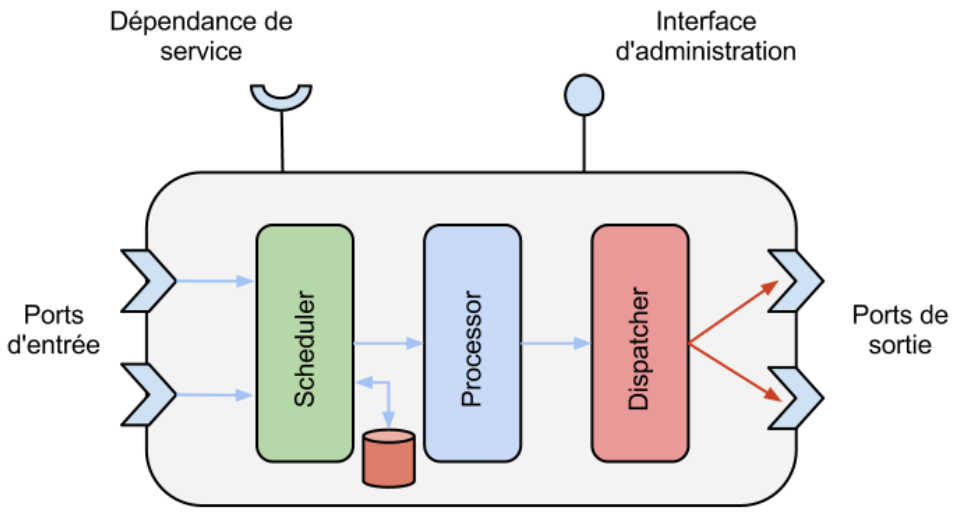

FIGURE 5.5 – dispatcher CILIA

Séparer le dispatcher du processor est une décision de conception très structurante. Elle permet de clairement séparer les préoccupations du processor et les préoccupations du dispatcher. Le processor se focalise sur des fonctions de médiation, c'est-à-dire sur du code purement métier. Il ne se préoccupe pas de la destination des ses résultats : c'est la raison d'être du dispatcher.

Cette approche permet de plus facilement faire évoluer un médiateur et favoriser la maintenance en général. Cela permet aussi, le cas échéant, de réutiliser les algorithmes du processor pour constituer de nouveaux médiateurs.

On voit ici que l'approche Cilia permet une bonne séparation des préoccupations au niveau logiciel mais aussi au niveau humain. En effet, on comprend bien que les schedulers (et les dispatchers) seront généralement implantés par un spécialiste technique de l'intégration alors que les processors seront implantés par des spécialistes du métier des applications à intégrer.

3 Assemblage CILIA

3.1 Notion de connecteur

Cilia définit un langage permettant d'assembler des médiateurs. Ce langage repose sur la notion de connecteur. Un connecteur permet de lier deux médiateurs ou un médiateur et une ressource à integrer. Il se caractérise par le protocole de communication utilisé et par un ensemble de propriétés permettant de définir l'interaction mise en place entre les médiateurs. Ces propriétés permettent de préciser, par exemple, si une interaction est locale ou distribuée ou le niveau de sécurisation souhaité pour les échanges.

Comme nous l'avons indiqué précédemment, les médiateurs se caractérisent par des interfaces basées sur la notion de port typé. Toutes les interactions entre les médiateurs se font via les ports. Les connecteurs sont donc liés aux médiateurs par leurs ports d'entrée et de sortie.

Les types associés aux ports permettent une vérification minimale de la compatibilité de deux médiateurs. Précisons néanmoins ici que l'idée de CILIA n'est pas d'associer deux médiateurs sur cette seule base : les médiateurs doivent être conçus pour fonctionner ensemble et partager la sémantique des données échangées.

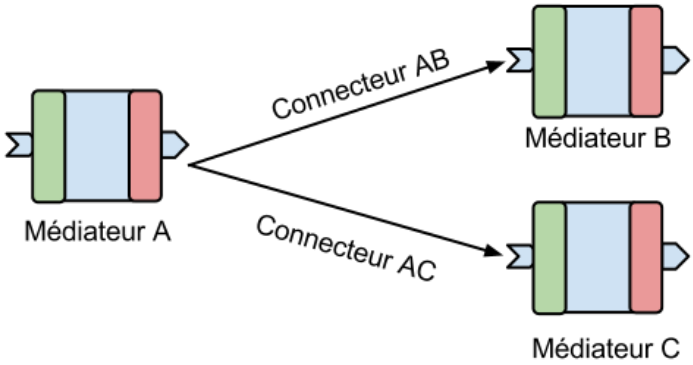

FIGURE 5.6 – Exemple de connecteur entre deux médiateurs.

CILIA fait la distinction entre deux types de connecteurs :

- Les connecteurs internes qui permettent de relier deux médiateurs via leur ports d'entrée et de sortie possédant les mêmes types,
- Les connecteurs externes qui permettent de relier des médiateurs et les ressources à intégrer telles que des applications, des équipements, etc.

Ces deux types de connecteurs sont nécessaires pour former des solutions d'intégration. Les connecteurs externes font le lien avec l'extérieur. Ils sont complexes mais souvent réutilisables en partie. Les connecteurs internes permettent de construire les enchaînements de transformations qui caractérisent l'intégration. Ils sont souvent spécifiques à l'intégration traitée et sont moins facilement réutilisables.

3.2 Connecteurs internes

Les connecteurs internes permettent de faire communiquer deux médiateurs et, ainsi, d'enchainer des opérations de médiation.

La Figure 5.7 présente des exemples de connections internes, c'est-à-dire liaisons entre médiateurs. Ces liaisons suivent divers paradigmes de communication, à savoir :

- une liaison directe (a) met en place une interaction synchrone de type client/serveur entre les deux médiateurs à connecter,
- une liaison à base de message (b) met en place une interaction asynchrone. Cette liaison est généralement réalisée par l'intermédiaire d'une facilité de gestion de messages (« *Message Oriented Middleware* » en anglais).
- une liaison par mémoire partagée (c) met également en place une interaction asynchrone mais en utilisant un service de persistance. Un tel service peut correspondre à une base de données ou un buffer en mémoire par exemple.

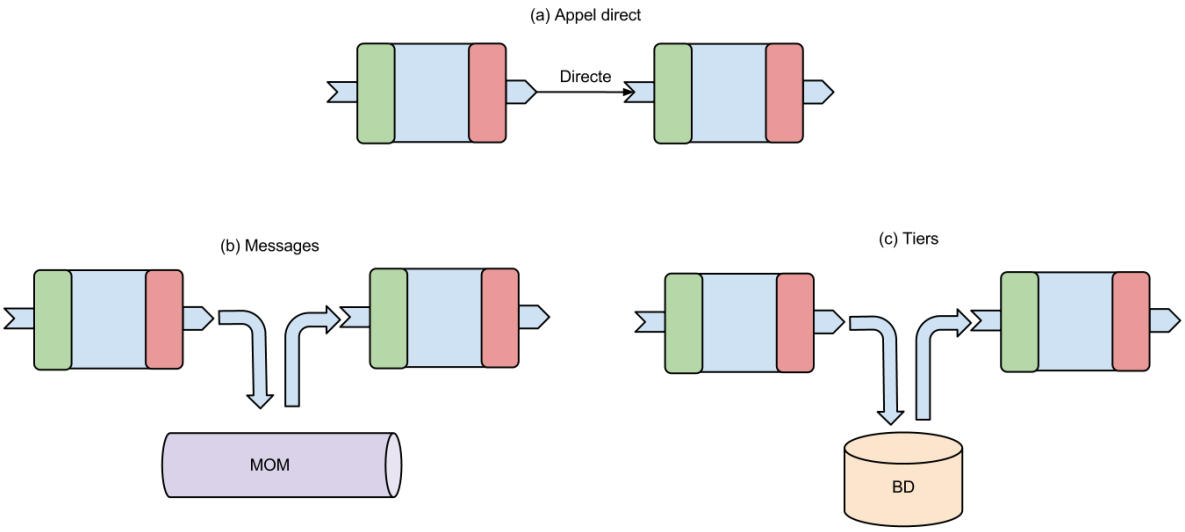

FIGURE 5.7 – Liaisons possibles entre médiateurs.

CILIA repose sur un patron de linkage pour réaliser des connections internes. Par exemple, un « *linker* » de *middleware* à messages basé sur des *topics*. Ce « *linker* » doit être capable de configurer le médiateur émetteur et le médiateur récepteur avec le *topic* correspondant. Ainsi, il doit paramétrer par exemple la direction IP du serveur MOM et le port IP.

La liaison n'est pas décrite dans le composant de médiation, ce qui permet de maintenir un faible couplage entre les composants. De plus, il permet une généralisation de connecteurs. C'est-à-dire qui soient utilisées en plusieurs chaînes de médiation.

3.3 Connecteurs externes

Un connecteur externe permet de faire communiquer un médiateur et une ressource à intégrer. Un point fondamental est que les ressources possèdent le contrôle sur les données à intégrer : les ressources émettent les données et consomment les données à leur rythme. L'application d'intégration ne contrôle que le transfert de ces données, dans les deux sens.

Les ressources externes peuvent correspondre à des applications spécifiques mais aussi à des systèmes plus génériques. Il peut s'agir par exemple de systèmes de fichiers, de systèmes de gestion de base de données, d'intergiciels ou encore de progiciels comme des ERP ou des CRM. Dans ces derniers cas, il est intéressant de développer des connecteurs externes réutilisables (par configuration).

Dans tous les cas, les connecteurs externes doivent communiquer avec des systèmes possédant leur propre format de données, leur propre protocole de communication et leur propres APIs. Développer des connecteurs externes peut être lourd et, en ce sens, leur réutilisation devient très pertinente.

Dans CILIA, ils existent trois types d'adaptateurs, à savoir :

- les connecteurs d'entrée,
- les connecteurs de sortie,
- les connecteurs d'entrée/sortie.

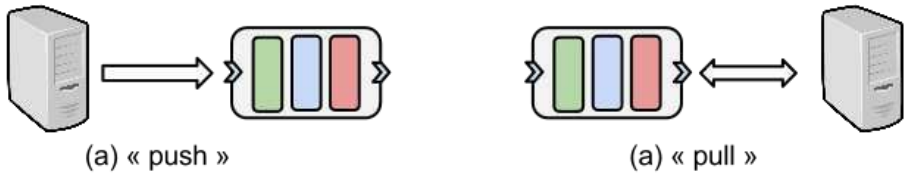

FIGURE 5.8 – Sens de la connexion pour les connecteurs d'entrée.

Les connecteurs d'entrée sont en charge de la communication entrante vers les applications d'intégration. Comme nous l'avons décrit précédemment, des messages produits ou contrôlés par des systèmes externes sont réceptionnés par l'application d'intégration qui prend le contrôle de ces messages.

Comme illustré par la figure ci-dessus, il existe deux types de connecteurs d'entrée qui correspondent aux deux styles d'interaction classiques « *push* » et « *pull* ». Dans le style « *push* », l'application externe initialise l'échange d'information. En revanche, dans le style « *pull* », c'est le connecteur qui effectue une demande d'information au système externe. Par exemple, cette initialisation peut être déclenchée de façon périodique.

Les connecteurs de sortie, comme leur nom l'indique, sont en charge des données sortant des applications d'intégration. Ils doivent livrer ces données aux ressources à intégrer. Une fois livrées, les données sont sous le contrôle du système cible. Tout comme les connecteurs d'entrée, les connecteurs de sortie peuvent utiliser le mode « *push* » et « *pull* ».

4 Configuration CILIA

Dans un modèle à composant, une configuration est un assemblage de composants configurés formant une application. En CILIA, une configuration s'appelle une chaîne de médiation. Une chaîne de médiation est un assemblage de médiateurs, de connecteurs et de ressources.

Une chaîne de médiation est dirigée par les données (*data-flow-driven* en anglais). Le modèle de calcul est le suivant :

> *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.*

La topologie d'une chaîne de médiation prend la forme d'un graphe avec deux contraintes fortes. Tout d'abord, on ne peut lier que des ports de sortie et des ports d'entrée pour la connexion de médiateurs. Par ailleurs, les cycles ne sont pas autorisés. Une chaîne de médiation peut contenir un ou plusieurs points d'entrée ainsi qu'un ou plusieurs points de sortie. Ces points sont les extrémités de la chaine de médiation.

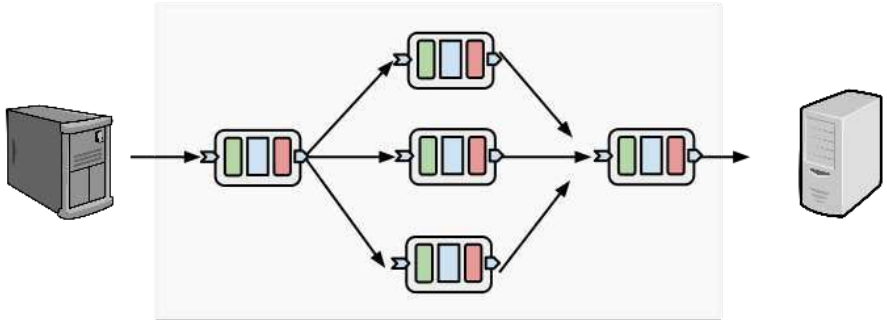

FIGURE 5.9 – Exemple de chaîne de médiation.

Le langage de configuration de CILIA prend la forme d'un ADL (« *Architecture Description Language* » en anglais). De façon générale, un ADL permet de définir une architecture à base de composants et de connecteurs et de spécifier un ensemble de propriétés spécifiques aux composants et aux connecteurs. Un langage de description d'architecture fournit les éléments pour représenter une ou plusieurs vues architecturale se focalisant sur une préoccupation particulière. La description d'une architecture à l'aide d'un ADL peut être compréhensible par une machine ou transformable en un format compréhensible. Ce format peut être analysé et utilisé afin d'automatiser l'implémentation de certains aspects des systèmes logiciels. C'est justement ce qui nous permet de générer des exécutables à partir d'une configuration Cilia.

Nous avons ainsi défini un méta-modèle ainsi qu'un langage concret pour spécifier les architectures des solutions d'intégration. Plus précisément, une chaîne est composée d'un ensemble de médiateurs M et d'un ensemble de connecteurs internes B et de connecteurs externes A :

$$\begin{aligned} chain &= < M, A, B > \\ M &= \{m_1, ..., m_i, ..., m_{in}\} \\ A &= \{a_1, ..., a_j, ..., a_{jn}\} \\ B &= \{b_1, ..., b_k, ..., b_{kn}\} \end{aligned}$$

Une liaison b_k est une relation entre un port de sortie d'un composant de médiation $mc1$, et un port d'entrée d'un composant de médiation $mc2$. Une restriction dit que le composant de médiation $mc1$ doit être différent au composant de médiation $mc2$.

$$\begin{aligned} b_k &= mc1 \rightarrow mc2, \forall mc1 \neq mc2 \; . \; Est\ une\ liaison\ de\ mc1\ vers\ mc2 \\ mc1 &= m1 \vee a1, \forall m1 \in M, \forall a1 \in A \\ mc2 &= m2 \vee a2, \forall m2 \in M - \{mc1 \ni m1\}, \forall a2 \in A - \{mc1 \ni a1\} \end{aligned}$$

Tous les éléments d'une chaîne de médiation peuvent contenir des propriétés paramétrables. Ainsi, un *processeur* par exemple, peut contenir une propriété spécifique à une chaîne de médiation concrète. Cependant, les propriétés dépendent fortement de l'implémentation.

Un médiateur m_i peut être constitué des propriétés de chacun de ses constituants, c'est-à-dire, des propriétés de son scheduler, de son processor, ainsi que de son dispatcher.

$$m_i =< prop(s_i), prop(p_i), prop(d_i) >$$

Où, $prop(x)$ est une fonction qui retourne l'ensemble de propriétés de x. Par exemple, un *scheduler* peut contenir une propriété « *période en millisecondes* » pour déclencher le traitement périodiquement, ainsi qu'une autre pour le délai d'initialisation après la création du composant. Pour le *processor*, par exemple, il peut contenir une propriété pour définir l'emplacement d'un fichier « XSLT » afin de réaliser une transformation d'un message XML vers un autre. Et finalement pour le *dispatcher*, il peut contenir des propriétés pour paramétrer le routage.

5 Framework d'exécution CILIA

Le framework d'exécution d'un modèle à composants est chargé de fournir les fonctions suivantes aux composants qu'il gère :

- Support pour la gestion du cycle de vie aussi bien au niveau des composants que des configurations. Ceci peut inclure, par exemple, des fonctions de création ou d'élimination au niveau des implantations ou au niveau des instances.
- Support pour certaines propriétés non fonctionnelles telles que la persistance par exemple.
- Support pour l'administration et l'autogestion. Ceci peut inclure des facilités d'introspection, d'intercession et d'adaptation.

Le framework d'exécution CILIA est composé d'un API réflexif, d'un « méta-level », d'un « base-level », d'un gestionnaire de synchronisation et d'un gestionnaire de données. Cette architecture permet la modification à chaud des médiateurs et des chaînes de médiation.

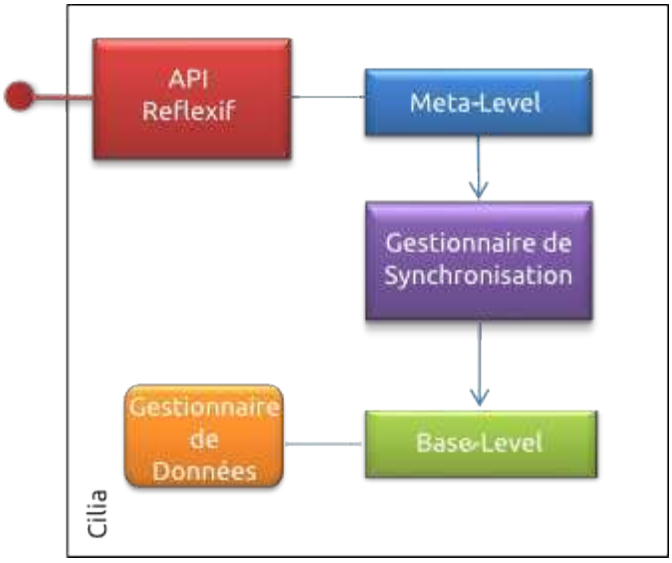

FIGURE 5.10 – Framework d'exécution CILIA

L'« **API réflexive** » permet de créer des chaînes de médiation en utilisant un langage de programmation. Cet élément de l'architecture du framework d'exécution de CILIA est un pilier important. Cet API permet de réaliser l'assemblage des chaînes de médiation en respectant le modèle présenté dans les sections précédentes. Ainsi, nous pouvons considérer cette API comme un DSL interne [Fow06]. Les opérations fournies par cet API sont principalement :

- créer une chaîne de médiation ;
- ajouter un composant de médiation dans une chaîne de médiation ;
- éliminer un composant de médiation dans une chaîne de médiation ;
- ajouter une liaison dans une chaîne de médiation ;
- éliminer une liaison dans une chaîne de médiation ;

Le « **méta-level** » est un conteneur de descriptions d'applications de médiation. Chaque application en exécution est représentée par une description abstraite de l'architecture. Cette description abstraite contient la liste des composants de médiation, ses configurations de paramétrage, ainsi que les configurations de connecteurs entre composants. Les objets qui ont cette information sont construits grâce à l'« API reflexive ». En bref, cette couche contient toute la connaissance de la structure des applications de médiation. Cette information de modèles d'application nous permet d'interagir avec les chaînes de médiation sans entrer dans les détails techniques de l'implémentation de chacun de ses composants et ses constituants. Grâce à ce haut niveau d'abstraction offert par les modèles des applications à l'exécution, la manipulation de concepts propres à la médiation devient possible et facile à l'exécution. Cette manipulation est indispensable lors de l'adaptation d'applications à l'exécution. Ainsi, elle est indispensable pour réaliser des évolutions sans arrêter l'application. Cette dernière est utile pour des applications où il est indispensable d'offrir un service et évoluer sans arrêter l'exécution. De même, on peut réaliser des évolutions quand il y a une carence d'administrateurs sur site pour les réaliser, et c'est donc l'utilisateur final qui va paramétrer les changements dans un langage d'haut niveau d'abstraction, par exemple une interface d'utilisateur Web.

Le « **base-level** » est le framework d'exécution proprement dit, contenant les instances des composants. Les instances, à ce niveau, sont le résultat de la réification de l'information contenue dans le modèle. Ces instances sont des objets qui interagissent afin de réaliser les opérations de médiation. Pour chaque médiateur il y a un objet scheduler, un objet processor et un objet dispatcher. De plus, pour chaque liaison fait dans le niveau modèle, il y a un ensemble d'objets techniques qui sont chargés d'assurer les échanges de données entre les objets du médiateur.

Le **gestionnaire de synchronisation** est un élément majeur chargé de réifier la description de chaque chaîne de médiation contenue dans la couche du modèle (« méta-level »). Aussi, ce gestionnaire maintient à jour les objets de la couche d'objets (« base-level ») en surveillant les modifications de chaque modèle de chaîne. Il est chargé aussi de faire l'instanciation de composants de médiation et aussi d'effectuer les opérations du cycle de vie des composants telles que l'initialisation, la destruction, mais aussi le paramétrage. Ce gestionnaire permet de découpler le « base-level » du « meta-level » et son comportement leur est caché.

Chaque chaîne de médiation contient un **gestionnaire de données**. Les objets dans le « base-level » interagissent avec cet gestionnaire de données. Plus précisément sont les objets de schedulers qui l'utilisent. L'importance de ce gestionnaire est qu'il permet d'échanger un médiateur complet par un autre sans perdre de messages. Une seule contrainte, assez importante, est que nous ne pouvons pas changer le gestionnaire de données à l'exécution. Les seules entités qui nous pouvons échanger sont les médiateurs ainsi que les liaisons.

6 Cycle de vie CILIA

Les modèles à composants ont toujours un cycle de vie associé. Le cycle de vie représente les différentes étapes et les règles à suivre pour une conception correcte d'une application à composants. Pour certains modèles, le cycle de vie est implicitement lié à la technologie utilisée. Dans d'autres cas, le cycle de vie peut s'abstraire de la technologie cible. C'est le cas de CILIA où nous spécifions un cycle de vie abstrait qui peut être appliqué à diverses technologies d'exécution. Avec l'hypothèse, bien sûr, que le framework d'exécution puisse être implanté pour les technologies en question.

Nous séparons le cycle de vie en trois grandes phases : la conception, le déploiement et, enfin, la gestion ou administration. Nous allons détailler par la suite chacune de ces phases.

6.1 Conception

La phase de conception d'applications de médiation se réalise en plusieurs étapes (Figure 5.11). Chacune des ces étapes permet, de façon incrémentale, de produire des artefacts qui seront déployés afin d'exécuter une chaîne de médiation.

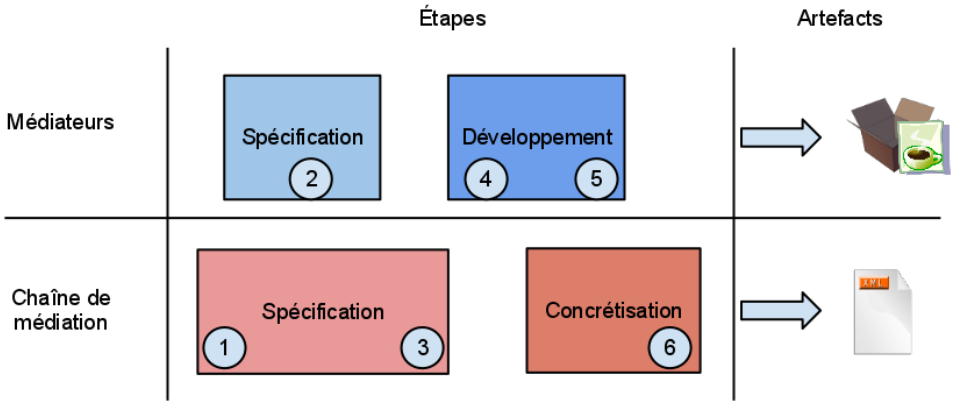

FIGURE 5.11 – Etapes pour la production d'une application CILIA.

Étapes de spécification.

1. Description de la chaîne de médiation, éventuellement avec des médiateurs existants
2. Spécifications complètes des nouveaux composants de médiation

Étape de développement

3. Implémentation des médiateurs concrets en définissant ses constituants.
4. Implémentations de chacun de ses constituants spécifiés lors de l'étape de spécification.

Étape de concrétisation

5. Configuration de la chaîne de médiation,en définissant les paramètres nécessaires pour tous les médiateurs et connecteurs.

6.1.1 Étape de spécification

L'objectif de cette étape est d'obtenir la spécification d'une chaîne de médiation pour répondre à une certaine problématique d'intégration. Pendant cette étape, les architectes identifient chacun des composants de médiation qui sont nécessaires pour définir la chaîne de médiation. Cette identification peut aboutir à la réutilisation de composants de médiation existants, mais elle peut aussi avoir comme résultat des spécifications de nouveaux composants de médiation à développer.

Ce sont ces spécifications de nouveaux composants de médiation qui attirent notre attention. Cette étape permettra de produire des composants éventuellement réutilisables dans le futur. C'est ce qu'on appelle la généralisation des composants et c'est la responsabilité des concepteurs de l'assurer. Néanmoins, cette généralisation n'est pas toujours possible. Dans la majorité des cas les composants de médiation sont spécifiques à une seule problématique d'intégration.

Une spécification de médiateur doit contenir au moins les éléments suivants :

- le nom du composant de médiation,
- la catégorie du composant de médiation,
- une référence vers une spécification du *processor*,
- une référence vers une spécification du *scheduler*,
- une référence vers une spécification du *dispatcher*,
- l'ensemble des ports d'entrée et de sortie avec les types associés.

Comme nous pouvons noter, les spécifications de médiateurs font référence à d'autres spécifications (de *processor*, de *scheduler* et de *dispatcher*). Chaque spécification de ces constituants doit contenir l'information suivante :

- le type de constituant (soit *scheduler*, *processor*, ou *dispatcher*),
- le nom du constituant,
- son ensemble de propriétés fonctionnelles,

Les spécifications des adaptateurs sont plus simples que les spécifications de médiateurs. Les adaptateurs ont des constituants d'ordonnancement (*scheduler*) et de routage (*dispatcher*) implicites. Ainsi, les éléments à décrire pour une spécification d'adaptateurs sont :

- le type d'adaptateur (d'entré, de sortie, ou d'entrée/sortie) ;
- le nom d'adaptateur ;
- l'ensemble de propriétés fonctionnelles de l'adaptateur.

Une fois les spécifications des composants de médiation complètes, la spécification de la chaîne de médiation peut être enrichie. Néanmoins, la complétude de la chaîne de médiation n'est pas garantie, jusqu'à ce que tous les composants de médiation, ainsi que ses constituants, soient développés. Ceci est dû à certaines propriétés, principalement non fonctionnelles, qui sont spécifiées lors du développement. Ces propriétés peuvent être paramétrables dans la description de la chaîne de médiation.

6.1.2 Étape de développement

Pendant l'étape de développement, chacune des spécifications de composants de médiation est enrichie avec sa logique d'exécution. En d'autres termes, les algorithmes des composants sont décrits dans un langage de programmation. De plus, l'empaquetage du résultat développé est réalisé pendant cette étape, et cela selon la technologie utilisée.

Ces algorithmes correspondent à chacun des constituants du composant de médiation. Ainsi, chaque médiateur doit contenir :

- le nom du composant de médiation (il est obligatoire, et il est le même que celui décrit dans la spécification),
- un namespace du composant,
- la catégorie du composant de médiation (le même que dans la spécification),
- une référence vers un *processor* (obligatoire),
- une référence vers un *scheduler* (obligatoire),
- une référence vers un *dispatcher* (obligatoire),
- les ports d'entrée, ainsi que les ports de sortie avec les types de données acceptés.

Aussi, chacune des implémentations des constituants de composants de médiation doit contenir :

- Le type de constituant,
- le nom de constituant,
- un namespace du constituant,
- la référence vers la logique du constituant (dans un langage de programmation)
- l'ensemble de propriétés fonctionnelles, ainsi que non fonctionnelles.

6.1.3 Étape de concrétisation

La dernière étape consiste à finaliser les détails de la chaîne de médiation. Ces détails sont principalement liés à des aspects non fonctionnels qui sont identifiés lors de l'étape d'implantation. Comme exemple de ces aspects non fonctionnels, nous pouvons citer des paramètres spécifiques à une bibliothèque JMS utilisée lors de l'implémentation, paramètres de *logging*, etc.

Nous faisons le constat suivant : la conception de chaînes de médiation suit une approche de développement incrémentale. La définition d'une chaîne de médiation correspond aux premières étapes, cependant, elle n'est finalisée qu'à la dernière étape, là où nous connaissons tous les détails d'implémentation de composants nécessaires pour son exécution. De même, chaque composant de médiation, comme ses constituants, est défini en deux étapes. La première est dédiée à l'identification des principales propriétés fonctionnelles spécifiques à la problématique d'intégration. La deuxième, lors de l'implémentation, permet d'identifier des détails techniques liés principalement à la technologie utilisée. C'est après cette dernière étape que la chaîne de médiation peut être concrétisée avec tous les paramètres nécessaires, et qu'elle devient alors prête pour l'exécution. Cependant, il est toujours possible d'identifier tous les paramètres nécessaires lors de la spécification, sans que l'étape de l'implémentation impacte les décisions prises pendant la spécification.

6.2 Déploiement

Le résultat obtenu lors de la conception est constitué principalement de deux types d'artefacts. Le premier est l'ensemble des composants de médiations. Ceci correspond à l'implémentation des algorithmes (le code source) en plus de la description des composants et ses constituants (méta-information). Le deuxième type d'artefact correspond au résultat de la spécification de la chaîne de médiation. Cet artefact prend la forme d'un seul fichier qui contient l'architecture de la chaîne de médiation décrite dans le langage de description d'architectures d'applications Cilia, nommé DSCilia.

Dans notre proposition, les premiers types d'artefacts correspondent à un ou plusieurs fichiers JAR prêts à être déployés dans une passerelle OSGi. Le deuxième type d'artefact correspond à un fichier basé sur XML, qui est conforme au langage DSCilia.

Le déploiement des artefacts d'implémentations de composants de médiation ainsi que de la chaîne de médiation consiste à le rendre visible pour le framework d'exécution. Notre proposition fournit un mécanisme de déploiement à chaud, où nous pouvons mettre divers artefacts à l'exécution. Par exemple, nous pouvons déployer la chaîne de médiation même s'il manque des artefacts d'implémentation à déployer. Et c'est le framework qui gère le cycle de vie de l'application. Puis, une fois que toutes les dépendances sont résolues, la chaîne de médiation pourra être utilisée en intégralité. De plus, grâce aux capacités d'OSGi, il est possible de faire replacement des artefacts à l'exécution. Cela permet, par exemple de déployer une nouvelle version d'un composant de médiation qui corrige certains bugs.

6.3 Administration

Le modèle d'exécution fournit des mécanismes qui permettent la gestion de chaînes de médiation à l'exécution. Nous proposons une API d'administration qui peut être utilisée pour divers systèmes de gestion d'applications de médiation. Exemples de gestionnaires utilisant cet API sont :

- des instructions dans un *shell* ;
- un portail web de gestion ;
- un gestionnaire autonomique ;
- un service OSGi qui peut être exploité par d'autres outillages.

En général, toutes ces capacités permettent principalement deux tâches importantes dans des environnements dynamiques et adaptables ;

- le paramétrage d'applications de médiation
- la réalisation d'une restructuration de l'architecture de chaînes de médiation.

Les opérations qui permettent d'effectuer ces capacités sont les suivantes :

- ajouter une chaîne de médiation à chaud ;
- arrêter et éliminer complètement une chaîne de médiation à chaud ;
- reconfigurer des paramètres de composants de médiation dans une chaîne de médiation ;
- éliminer et ajouter des composants de médiation ;
- éliminer et créer de liaisons entre composants de médiation.

7 Synthèse

Dans ce chapitre nous avons décrit les détails sur le modèle à composant CILIA, ainsi que sur le modèle d'exécution. Nous avons montré les concepts qui ont introduit la définition de composants de médiation. Ces concepts sont surtout une classification de patrons d'intégration qui ont été appliqués depuis des années pour répondre aux problématiques récurrentes d'intégration. Ainsi, un *scheduler*, un *processor* et un *dispatcher* sont définis pour construire des composants de médiation. Nous avons montré aussi les capacités de composition de CILIA et ses capacités d'extension basée sur le modèle.

Nous avons expliqué aussi le cycle de vie associé à la conception d'applications de médiation. Nous avons fait l'emphase que ce cycle de vie, de même que le modèle à composant sont abstraits et ils peuvent être appliqués à diverses implémentations de CILIA en utilisant différentes technologies. Ce cycle de vie aussi met en évidence que la conception de solutions d'intégration exige une expertise. Cette expertise des concepteurs qui permettront de construire des composants généralistes.

Finalement, nous avons expliqué la structure du modèle d'exécution qui permet la manipulation de concepts de haut niveau d'abstraction à l'exécution. Cette manipulation nous permet de construire des systèmes auto-adaptables ainsi qu'évolutifs à l'exécution.

Dans le chapitre suivant, nous présentons les détails d'implémentation d'un prototype développé afin de valider le modèle à composant ainsi que le framework d'exécution. De plus, nous présentons des exemples de composants de médiation et les outillages élaborés.

6

RÉALISATION

Sommaire

1 **Technologies de base** 132
2 **Exemple de support** 134
3 **Langage de spécification** 136
4 **Framework de développement** 147
5 **Framework d'exécution** 157
6 **Cycle de vie** 161
7 **Synthèse** 165

Dans le chapitre précédent, nous avons développé les détails de notre proposition : le modèle à composant CILIA ainsi que le framework d'exécution associé.

Nous avons aussi décrit, dans le chapitre précédent, la proposition comme un modèle à composants indépendant d'une plate-forme cible. Dans ce chapitre nous présentons les détails de l'implémentation de référence de CILIA.

Nous débutons ce chapitre en mettant l'accent sur les technologies de base utilisées, leurs caractéristiques et leurs avantages pour développer des applications dynamiques. Dans la suite, nous montrons les détails d'implémentation des composants de médiation ainsi que leurs constituants. Nous détaillons l'assemblage de chaînes de médiation et le langage de description d'architectures de médiation appelé DSCilia. Ensuite, nous montrons les détails d'implémentation du framework d'exécution et finalement, les outils disponibles pour le cycle de vie associé à CILIA.

1 Technologies de base

Nous avons entièrement développé un framework d'exécution supportant le modèle à composant CILIA. Ce framework repose sur les technologies à service OSGi et iPOJO. Nous verrons plus avant que nous avons profité des capacités d'extension d'iPOJO pour traiter au mieux nos besoins. Plusieurs raisons nous ont amenés à faire ce choix. Par la suite, nous détaillons les caractéristiques de chacune de ces technologies et les raisons qui nous ont motivé pour leur utilisation.

Il est important de préciser ici que le modèle à composant CILIA peut être implanté en utilisant d'autres technologies. Nous verrons en particulier que nous avons également implanté CILIA sur des Java SunSpot caractérisés par de faibles empreintes mémoire.

1.1 OSGi

La plate-forme d'exécution de l'implémentation de référence de CILIA est basée sur OSGi. Comme nous avons vu dans le chapitre 2, le framework OSGi permet l'exécution d'applications à services dynamiques. Il repose sur la notion de « *bundle* » qui apporte la modularité nécessaire à l'ajout et au retrait de code à la volée. Pour rappel, le framework OSGi est constitué principalement de trois couches :

- la couche module, concrétisée par les *bundles*, qui nous permet de définir l'importation et exportation de code dans chaque bundle ;
- la couche de service, qui permet de connecter des objets Java en utilisant les mécanismes de base d'une approche à service (publication, recherche, liaison) ;
- la couche de cycle de vie, qui permet l'installation, le démarrage et d'autres opérations aux bundles.

Ces trois couches du framework OSGi nous permettent de construire des applications dynamiques et modulaires. En fait, nous pouvons structurer des applications en plusieurs bundles « plus au moins » indépendants, et seulement déployer les bundles nécessaires pour une application concrète. En plus, il est possible de faire des actualisations de code à la volée, comme d'ajouter de nouveaux bundles au fur et à mesure. Nous nous appuyons donc sur ces avantages pour l'implantation de CILIA.

1.2 iPOJO

Comme nous l'avons indiqué, l'implémentation de référence de CILIA est basée sur iPOJO. Le modèle à composants iPOJO permet la construction d'applications à services au dessus d'OSGi. Il vise essentiellement à masquer la complexité de développement d'applications dynamiques et à faciliter la gestion du cycle de vie des composants, voire des services.

Les composants CILIA, ainsi que les constituants sont une spécialisation de composants iPOJO. Pour mieux comprendre la structure interne des composants CILIA, nous détaillons les particularités d'iPOJO ainsi que ses points d'extension.

Un composant iPOJO est composé d'un code, ou logique fonctionnelle, ainsi que d'un conteneur qui gère des aspects non-fonctionnels. iPOJO gère ces aspects non-fonctionnels à l'aide d'un constituant spécial appelé « *handler* ». Un composant peut contenir autant de handlers que d'aspects non-fonctionnels à gérer. Le framework iPOJO fournit un ensemble de handlers de base pour :

- gérer l'injection de propriétés,
- gérer la publication de services,
- gérer les dépendances de services,
- gérer le cycle de vie de composants,
- fournir des détails techniques de l'architecture des composants et de leurs connexions (pour l'administration et le débogage).

Il est intéressant de noter que les *handlers* sont eux-mêmes implantés sous la forme de composants iPOJO. En conséquence, un handler est capable d'utiliser d'autres *handlers*.

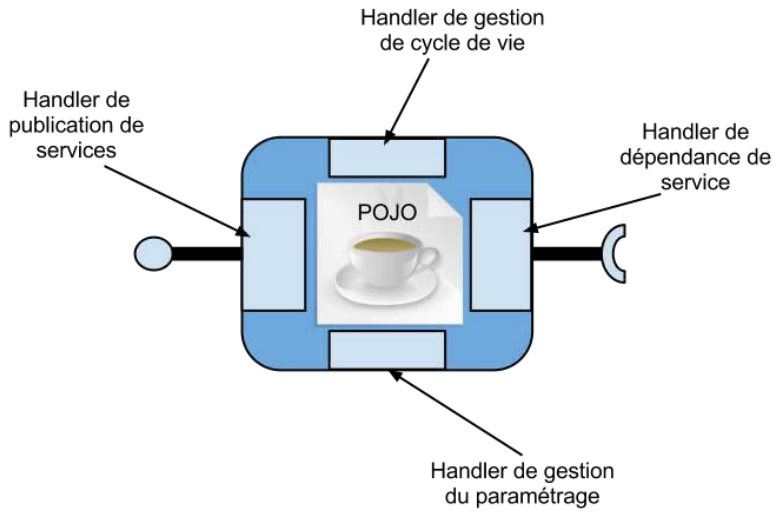

FIGURE 6.1 – Architecture d'un composant iPOJO

Les *handlers* représentent ainsi un mécanisme d'extension fourni par iPOJO pour gérer des aspects non-fonctionnels. Cela permet la construction de POJOs plus propres avec un code qui gère seulement les aspects métiers. Dans notre implantation, nous allons nous appuyer sur ces capacités d'extensibilité d'iPOJO. Chaque constituant d'un composant de médiation sera implanté sous la forme d'un composant iPOJO et pourra profiter des *handlers* iPOJO existants.

Dans iPOJO, chaque définition de composant est assurée par un service de fabrique (*factory* en anglais) enregistré dans OSGi. Ce service de fabrique prend en charge l'instanciation du conteneur et de l'ensemble des *handlers* du composant. Nous nous appuyons sur ce mécanisme et nous avons défini de nouveaux types de services de fabrique. Plus précisément, nous avons défini une fabrique pour les composants de médiation, une fabrique pour chaque constituant d'un médiateur (*scheduler*, *processor*, *dispatcher*), et une fabrique pour les connecteurs entre composants de médiation.

2 Exemple de support

Tout au long de ce chapitre, nous nous appuierons sur un exemple unique pour illustrer les différents points de notre proposition.

Cet exemple traite de l'intégration de trois applications bien connus de nos jours, à savoir Jabber, Twitter et Facebook :

- **Jabber** [1] est un protocole ouvert de messagerie instantanée,
- **Twitter** [2] est un service de « *microblogging* » qui permet de poster des messages (des « *post* ») contenant un maximum de 140 caractères,
- **Facebook** [3] est un service de réseau social qui permet de partager des informations de types variés (texte, image, video) avec d'autres personnes appelés « contacts ».

Le problème d'intégration que l'on souhaite résoudre avec cet exemple est le suivant :

> *Un utilisateur utilise régulièrement Jabber pour communiquer avec ses amis. Il dispose d'un compte Facebook et d'un compte Twitter. Souvent, il souhaite partager les répliques drôles ou intéressantes via Facebook et Twitter. Pour éviter d'utiliser un navigateur Web pour publier les messages qui sont échangés via Jabber, il est intéressant de lui proposer une interface directement via Jabber pour qu'il publie ses extraits de conversations.*

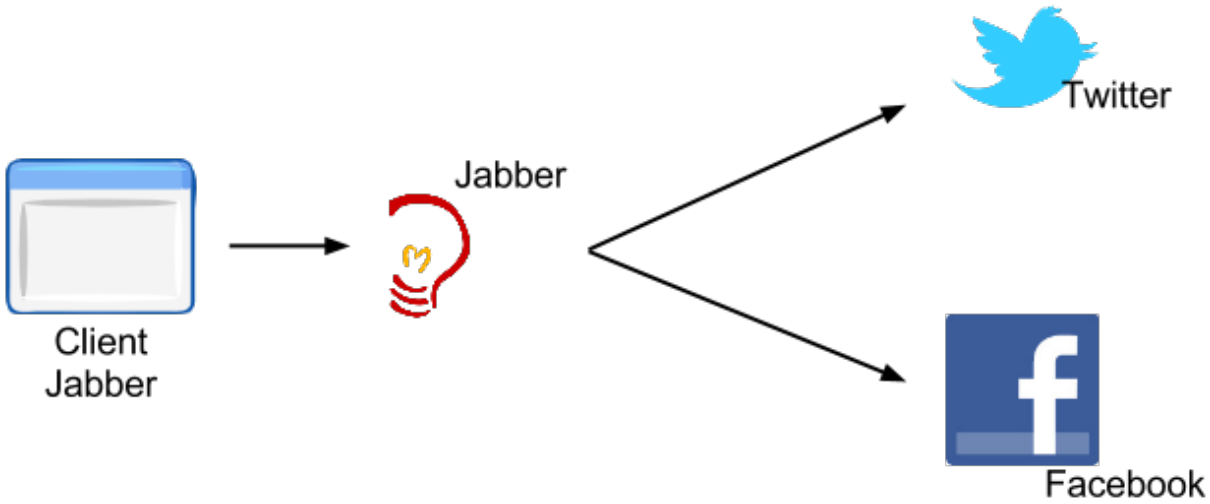

FIGURE 6.2 – Exemple d'intégration utilisé dans ce chapitre.

Cet exemple est illustré par la figure ci-dessous. Il est en fait relativement pertinent de nos jours puisque les internautes ont de plus en plus de mal à maintenir une certaines cohérence entre leurs différents blogs, réseaux sociaux, pages internet, ... On peut penser qu'il sera bientôt nécessaire d'automatiser le maintien de cohérence entre ces différents applicatifs. Cela repose bien sûr sur des solution d'intégration dynamiques.

Notre intégration doit prendre en compte trois exigences majeures :

1. **http ://www.jabber.org/about/**
2. **https ://twitter.com/about**
3. **http ://www.facebook.com/**

- l'utilisateur interagit avec une application client Jabber pour envoyer des messages ;
- l'utilisateur peut envoyer plusieurs messages, par exemple de dialogues, et il veut qu'ils soient agrégés dans un seul *post* ;
- néanmoins, le service de Twitter n'accepte pas des messages de plus de 140 caractères.

Comme illustré par la figure 6.3 ci-après, ce problème est classiquement résolu de la façon suivante :

- un premier adaptateur Jabber est défini pour récupérer les données issues du protocole Jabber,
- un médiateur est utilisé pour transformer les données récupérées de la façon requise pour Twitter ainsi que pour Facebook ,
- un adaptateur Twitter est chargé de communiquer vers le compte Twitter, mais il envoie des messages que si leur longueur est inférieure à 140 caractères,
- et, finalement, un autre adaptateur Facebook est chargé d'envoyer tous les messages vers le service Facebook.

Nous avons simplifié l'exemple afin de se concentrer sur les détails apportés par Cilia. Ainsi, un seul médiateur est suffisant. Une solution plus complète contient, par exemple, un autre médiateur chargé d'extraire l'information requise dans un message XML issu du protocole Jabber. On pourrait aussi distinguer deux médiateurs de transformation : un pour s'adapter au format Twitter et un autre pour s'adapter au format Facebook.

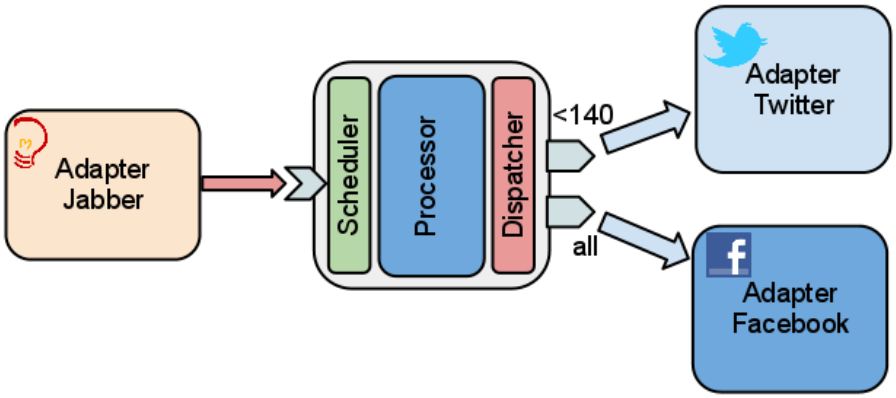

FIGURE 6.3 – Solution Cilia pour notre exemple d'intégration.

Cet exemple nous permet de définir un médiateur spécialisé qui contient :

- un scheduler de minuterie qui déclenche le traitement de tous les messages séparés par un délai minimal (par exemple 30*s*),
- un *processor* qui fait la concaténation de tous les messages corrélées par le *scheduler*,
- et un *dispatcher* qui envoie tous les messages vers un port connecté vers l'adaptateur Facebook et aussi qui envoie les messages de moins de 140 caractères vers l'adaptateur Twitter.

3 Langage de spécification

Nous avons défini un langage permettant de définir les différents éléments de CILIA : les médiateurs, les constituants des médiateurs (*scheduler*, *processor*, *dispatcher*), les connecteurs et les chaînes de médiation. Ce langage repose sur des grammaires XML que nous présentons ici.

3.1 Définition d'un « médiateur »

Nous avons défini un langage qui permet la définition des composants de médiation. Ce langage est basé sur le méta-modèle présenté par la Figure 6.4. Une spécification de médiateur est interprétée à l'exécution et utilisée pour la spécification des chaînes de médiation. Une caractéristique des composants de médiation est qu'ils n'ont pas de logique ou de code associé. Ceci est dû au fait que chacun des constituants est indépendant. La logique est définie dans des fichiers différents. Cependant, la spécification d'un médiateur doit contenir une référence vers chacun des trois constituants.

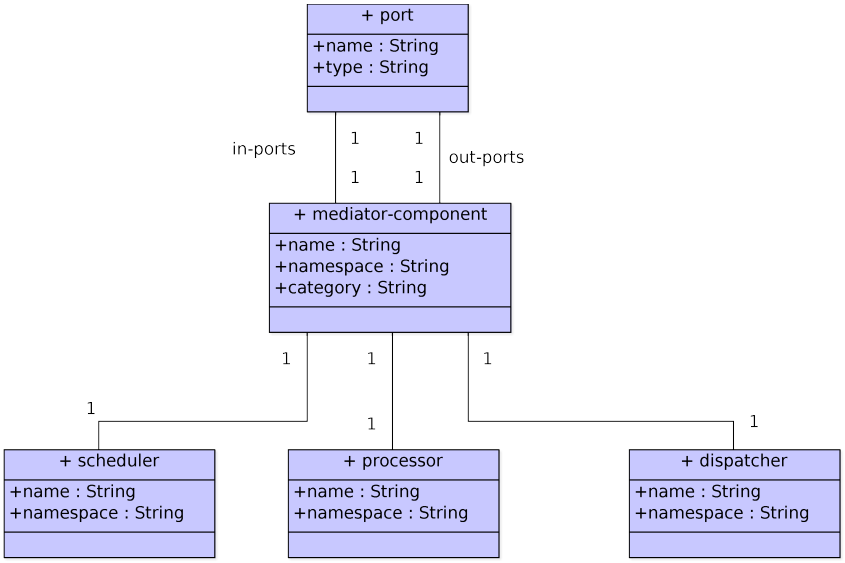

FIGURE 6.4 – Méta-modèle de spécifications de médiateurs

Les éléments du langage de spécification de médiateurs en XML sont définis comme suit :

<mediator-component> := décrit un médiateur. Il contient cinq sous-nœuds : *out-ports*, *in-ports*, *scheduler*, *processor* et *dispatcher* et trois attributs : *name*, *namespace* et *category*.

<in-ports> := décrit l'ensemble de ports d'entrée. Il contient des éléments de type *port*.

<out-ports> := décrit l'ensemble de ports de sortie. Il contient des éléments de type *port*.

<port> := décrit un port de médiateur. Il contient deux attributs : *name* et *type*.

<scheduler> := décrit la référence vers un constituant *scheduler*. Il contient deux attributs : *name* et *namespace*.

<processor> := décrit la référence vers un constituant *processor*. Il contient deux attributs : *name* et *namespace*.

<dispatcher> := décrit la référence vers un constituant *dispatcher*. Il contient deux attributs : *name* et *namespace*.

Dans l'exemple que nous avons mentionné précédemment, nous devons construire une médiateur constitué d'un scheduler effectuant une corrélation temporelle, d'un processor réalisant l'agrégation de messages corrélées et d'un dispatcher envoyant les messages vers un port pour les messages de moins de 140 caractères et à un autre port pour tous les messages. Voici l'exemple de la description du composant de médiation et de ses caractéristiques :

```
<mediator-component name="AggregatorExample"
    namespace="tesis.example" category="example">
    <in-ports>
        <port name="jabber.port" type="Text" />
    </in-ports>
    <scheduler name="timer-scheduler" namespace="tesis.example" />
    <processor name="aggregator-processor" namespace="tesis.example" />
    <dispatcher name="example-dispatcher" namespace="tesis.example" />
    <out-ports>
        <port name="fb.port" type="Text" />
        <port name="tw.port" type="Text" />
    </out-ports>
</mediator-component>
```

Ce médiateur est bien sûr caractérisé par des ports d'entrée et de sortie. Il possède un port d'entrée appelé « jabber.port » qui sera connecté à l'adaptateur Jabber et deux ports de sortie : un qui sera connecté à l'adaptateur Facebook et un second connecté à l'adaptateur Twitter.

3.2 Définition d'un « *scheduler* »

Le *scheduler* est un des constituants où la logique est un peu plus complexe puisqu'il doit gérer la façon de stocker les données reçues et définir à quel moment déclencher le traitement de ces données stockées. Nous avons défini un langage basé en XML qui permet de spécifier des schedulers. Ce langage est basé sur le méta-modèle de la Figure 6.5.

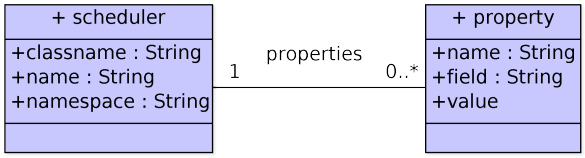

FIGURE 6.5 – Méta-modèle de spécification de scheduler

Les éléments du langage de spécification de *schedulers* en XML sont définis comme suit :

<scheduler> := décrit un *scheduler*. Il contient principalement un sous nœud appelé properties. Il doit également contenir trois attributs : name, namespace et classname.

<properties> := décrit l'ensemble de propriétés du *scheduler*.

<property> := décrit une propriété. Il contient principalement trois attributs : *name* pour le nom de la propriété, *field* pour le champ affecté par la propriété et *value* pour une valeur par défaut.

Pour continuer avec l'exemple présenté dans la section précédente, nous nous sommes intéressés à un scheduler capable d'identifier des messages provenant de Jabber et de détecter les messages devant être corrélés. L'agrégation est ensuite réalisée par le processor. Une technique de corrélation simple consiste à corréler des messages qui ont un faible écart au niveau du temps de réception ; c'est-à-dire les messages qui ont, par exemple, un écart de moins de 30s.

Ainsi, nous avons défini une scheduler nommé « timer-scheduler ». Sa spécification est la suivante :

```
<scheduler classname="test.example.TimerScheduler" name="timer-scheduler"
    namespace="tesis.example">
    <properties>
        <property name="period" field="period" />
    </properties>
</scheduler>
```

Ce scheduler a une propriété appelée « period » qui sert à configurer la minuterie.

La spécification présentée fait explicitement référence à une classe Java appelée « TimerScheduler ». Cette classe implémente la logique d'ordonnancement. Nous présenterons par la suite le framework que nous avons défini pour faciliter l'implantation de cette classe.

3.3 Définition d'un « *processor* »

La logique du processeur est le cœur du médiateur. Le plus souvent, cette logique est spécifique à une problématique donnée. Néanmoins, il est possible d'identifier des processors généralistes avec certaines propriétés réutilisables. Nous montrons ici le langage basé sur XML qui permet de spécifier les processors. Ce langage est basé sur le méta-modèle de la Figure 6.6.

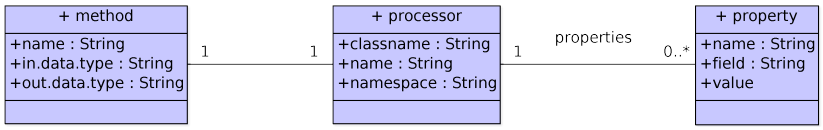

FIGURE 6.6 – Méta-modèle de spécification de *processor*

Les éléments du langage de spécification de *processor* en XML sont définis comme suit :

<processor> := décrit un *processor*. Il contient principalement un sous nœud appelé *properties*. Il doit également contenir trois attributs : *name*, *namespace* et *classname*.

<properties> := décrit l'ensemble de propriétés du *processor*.

<property> := décrit une propriété. Il contient principalement trois attributs : *name* pour le nom de la propriété, *field* pour le champ affecté par la propriété et *value* pour une valeur par défaut.

<method> := décrit le méthode qui fait le traitement. Il contient trois attributs : *name*, pour le nom de la méthode ; in.data.typ, pour le type de donnée de paramètre et out.data.type, pour le type de donnée retourné par la méthode.

Poursuivons notre exemple d'intégration. Nous avons ainsi créé un processor réalisant une concaténation de toutes les données corrélées sur une période donnée (cette corrélation est définie par le scheduler). Le processor reçoit comme paramètre une liste contenant tous les messages corrélés et il retourne comme résultat un objet unique contenant la concaténation.

La description d'un tel processor est la suivante :

```
<processor classname="test.example.AggregatorProcessor" name="aggregator-processor"
    namespace="tesis.example">
    <method name="aggregate" in.data.type="java.util.List"
        out.data.type="fr.liglab.adele.cilia.Data" />
</processor>
```

Cette spécification fait référence à une classe Java appelée « AggregatorProcessor ». Nous en détaillerons l'implantation dans une prochaine section.

3.4 Définition d'un « *dispatcher* »

Le langage de spécification des *dispatchers* ressemble au langage de spécification des *schedulers*. Ce langage est basé sur le méta-modèle présente par la Figure 6.7.

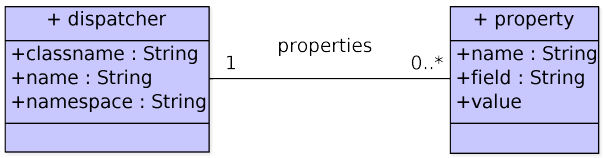

FIGURE 6.7 – Méta-modèle de spécification de dispatcher

Les éléments du langage de spécification de *dispatcher* en XML sont définis comme suit :

<dispatcher> := décrit un *dispatcher*. Il contient principalement un sous nœud appelé properties. De plus, il doit contenir trois attributs : *name*, *namespace* et *classname*.

<properties> := décrit l'ensemble de propriétés du *dispatcher*.

<property> := décrit un propriété. Il contient principalement trois attributs : *name* pour le nom de la propriété, *field* pour le champ affecté par la propriété et *value* pour une valeur par défaut.

Pour l'intégration de Jabber, Twitter et Facebook, nous avons décidé d'implémenter un *dispatcher* effectuant le routage selon le nombre de caractères du message à livrer. La spécification de ce dispatcher est la suivante :

```
<dispatcher classname="test.example.ExampleDispatcher"
    name="example-dispatcher" namespace="tesis.example">
</dispatcher>
```

3.5 Définition d'un connecteur externe

Les connecteurs externes nous permettent de communiquer vers des ressources externes comme, par exemple, des systèmes de fichiers, des progiciels, des bases de données, ou encore des services Internet tels que Facebook et Twitter. La connectivité vers l'extérieur d'une chaîne de médiation est réalisée par une entité appelée « *adapter* ». Un adapter est la composition d'un médiateur minimal et d'un code de connexion vers l'extérieur. Un médiateur minimal est constitué d'un scheduler « immédiate », d'un processeur « vide », et d'un dispatcher « *multicast* ».

3.5.1 Adapteurs d'entrée

L'entrée dans une chaîne est prise en charge par une entité appelée « collector » qui fait ainsi partie intégrante des adapteurs d'entrée. Le langage de spécification des adaptateurs doit donc permettre de définir cette nouvelle entité. Le langage est basé sur le méta-modèle présenté par la Figure 6.8.

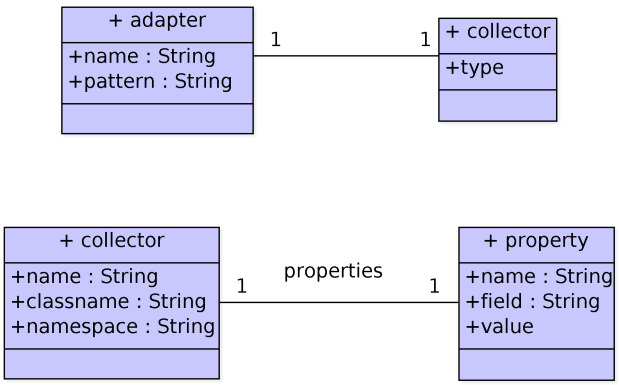

FIGURE 6.8 – Méta-modèle d'adaptateur d'entrée et de collector.

Les éléments du langage de spécification des adaptateurs en XML sont définis comme suit :

<adapter> := décrit un *adapter*. Il contient principalement un sous nœud appelé *collector*. Il doit également contenir deux attributs : *name* et *pattern*.

<collector> := décrit le la référence vers l'entité qui réalise la connexion. Il y a un seul attribut nommé *type*. Cet attribut doit correspondre au nom du *collector*.

Comme nous pouvons le constater, les adaptateurs contiennent une référence vers un « *collector* ». Nous avons également décrit un langage permettant de spécifier des collectors en XML :

<collector> := décrit un *collector*. Il contient principalement un sous nœud appelé properties. De plus, il doit contenir trois attributs : *name*, *namespace* et *classname*.

<properties> := décrit l'ensemble de propriétés.

<property> := décrit une propriété. Il contient principalement trois attributs : *name* pour le nom de la propriété, *field* pour le champ affecté par la propriété et *value* pour une valeur par défaut.

Pour poursuivre l'exemple d'intégration Jabber, Twitter et Facebook, nous allons détailler la spécification d'un adaptateur d'entrée. Le seul adaptateur d'entrée de cet exemple est celui lié à Jabber. Le style d'interaction est de type push. L'adaptateur Jabber est notifié lorsqu'il reçoit un nouveau message. Cet adaptateur contient une logique de connexion faite par un élément appelé « jabber-collector ».

```
<collector name="jabber-collector" classname="test.example.JabberCollector"
    namespace="tesis.example">
    <properties>
        <property name="user.name" field="login"/>
        <property name="user.password" field="pwd"/>
        <property name="service.host" field="host"/>
        <property name="service.port" field="port"/>
    </properties>
</collector>

<adapter name="jabber-adapter" pattern="in-only">
    <collector type="jabber-collector" />
</adapter>
```

Dans cet exemple, le *collector* a quatre propriétés de paramétrage :

- une propriété « user.name » pour réaliser une connexion au service de messagerie instantané,
- une propriété « user.password » pour le mot-clé de l'utilisateur,
- une propriété « service.host » pour le hostname du serveur de messagerie instantanée,
- une propriété « service.port » pour le port Internet pour communiquer avec le serveur.

3.5.2 Adapteurs de sortie

La sortie d'une chaîne est prise en charge par une entité appelé « *sender* ». Les adaptateurs de sortie sont ainsi constitués d'une entité de ce type. Le langage de spécification d'adaptateurs doit donc permettre de définir cette nouvelle entité. Le langage qui permet de spécifier ces connecteurs est basé sur le méta-modèle présenté dans la Figure 6.9

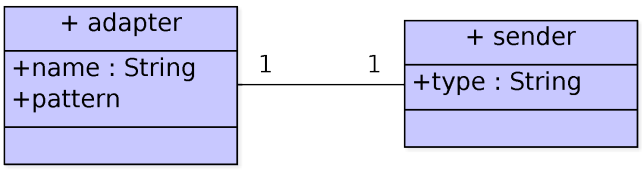

FIGURE 6.9 – Méta-modèle d'adaptateur de sortie et de sender

Les éléments du langage de spécification des adaptateurs de sortie en XML sont définis comme suit :

<adapter> := décrit un *adapter*. Il contient principalement un sous nœud appelé *sender*. De plus, il doit contenir deux attributs : *name* et *pattern*.

<sender> := décrit la référence vers l'entité qui réalise la connexion. Il y a un unique attribut nommé *type*. Cet attribut doit correspondre au nom du *sender*.

Comme nous pouvons le constater, les adaptateurs contiennent une référence vers un « *sender* ». Nous avons défini un langage permettant de décrire des senders en XML :

<sender> := décrit un nouveau *sender*. Il contient principalement un sous nœud appelé properties. De plus, il doit contenir trois attributs : *name*, *namespace* et *classname*.

<properties> := décrit l'ensemble de propriétés.

<property> := décrit une propriété. Il contient principalement trois attributs : *name* pour le nom de la propriété, *field* pour le champ affecté par la propriété et *value* pour une valeur par défaut.

Reprenons notre l'exemple. Nous nous appuyons sur l'adaptateur Twitter. Cet exemple montre une caractéristique importante héritée d'iPOJO, qui est l'utilisation de *handlers* pour gérer des aspects non-fonctionnels. Il s'agit dans ce cas de la dépendance vers un service OSGi. Ce handler est configuré avec la balise « *requires* » d'iPOJO. Pour l'implémentation, nous nous appuyons aussi sur un service OSGi Twitter existant développé par le projet OW2 Chameleon [4].

```
<sender classname="fr.liglab.adele.cilia.components.senders.ConsoleSender"
    name="console-sender" architecture="false">
</sender>

<adapter name="console-adapter" pattern="out-only">
    <sender type="console-sender" />
</adapter>
```

3.5.3 Adaptateurs d'entrée/sortie

Les adaptateurs d'entré/sortie, à la différence des deux types d'adaptateurs précédents, permettent de définir un style d'interaction de type requête/réponse.

Cependant, cet adaptateur demande que cela soit l'application externe qui émette la requête et, donc, que la chaîne de médiation fournisse la réponse.

Le langage qui permet de spécifier ces connecteurs est basé sur le méta-modèle présenté par la Figure 6.10 ci-après.

Les éléments du langage de spécification d'adaptateurs d'entré/sortie en XML sont définis comme suit :

<io-adapter> := décrit un adaptateur d'entré/sortie. Il contient principalement un sous nœud appelé *properties*. Il doit également contenir deux attributs : *name* et *classname*.

4. http ://wiki.chameleon.ow2.org/xwiki/bin/view/Main/WebHome

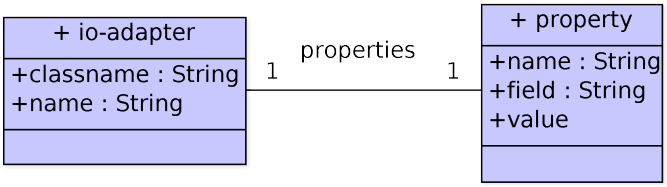

FIGURE 6.10 – Méta-modèle d'adaptateur d'entre/sortie

<properties> := décrit l'ensemble de propriétés du constituant.

<property> := décrit une propriété. Il contient principalement trois attributs : *name* pour le nom de la propriété, *field* pour le champ affecté par la propriété et *value* pour une valeur par défaut.

Dans notre exemple d'intégration de Jabber, Twitter et Facebook, il n'y a aucun adaptateur de ce type. Pour néanmoins illustrer cet aspect, nous présentons l'exemple d'un adaptateur fournissant un service OSGi. Pour cela, nous utilisons les capacités d'iPOJO pour publier l'adaptateur comme un service. Ainsi, dans la description de l'adaptateur, une balise appelée « *provides* » est utilisée. Cette balise correspond à un *handler* iPOJO chargé de la publication d'un service OSGi.

```
<io-adapter classname="test.exemple.DemoAdapterImpl"
    name="PriceAdapter" >
    <ipojo:provides />
</io-adapter>
```

3.6 Définition d'un connecteur interne

Dans la section précédente, nous avons montré que pour construire des connecteurs d'entrée et des connecteurs de sortie, nous avons introduit de nouvelles entités appelées « *collector* » et « *sender* ».

Le choix de réaliser ces entités dans le framework CILIA nous a permis d'obtenir des éléments réutilisables grâce à la généralisation de caractéristiques communes. Ainsi, pour réaliser les liaisons internes entre composants de médiation, le framework Cilia s'appuie aussi sur ces entités collector et sender.

Nous réalisons la connexion de deux composants grâce à un « *linker* ». Le langage qui permet de spécifier ces linkers est basé sur le méta-modèle présenté par la Figure 6.11.

Les éléments du langage de spécification d'adaptateurs d'entré/sortie en XML sont définis comme suit :

<linker> := décrit un *linker*. Il contient principalement trois sous nœuds : *properties*, *collector* et *sender*. Il doit également contenir deux attributs : *name* et *classname*.

<properties> := décrit l'ensemble de propriétés.

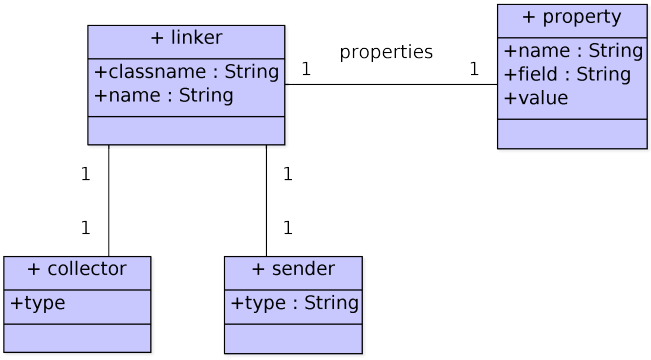

FIGURE 6.11 – Méta-modèle d'un linker

<property> := décrit une propriété. Il contient principalement trois attributs : *name* pour le nom de la propriété, field pour le champ affecté par la propriété et value pour une valeur par défaut.

<sender> := décrit la référence vers l'entité qui réalise la connexion. Il y a un seul attribut nommé *type*. Cet attribut doit correspondre au nom du *sender*.

<collector> := décrit la référence vers l'entité qui réalise la connexion. Il y a un seul attribut nommé *type*. Cet attribut doit correspondre au nom du *collector*.

Ci-après, un exemple d'un linker avec les *collector* et *sender* associés :

```
<sender classname="components.sender.ea.EventAdminSender" name="ea-sender"
    architecture="true">
    <properties>
        <property name="topic" field="m_topic" />
    </properties>
    <requires field="m_eventAdmin" />
    <callback transition="validate" method="started" />
    <callback transition="invalidate" method="stopped" />
</sender>

<collector classname="components.collector.ea.EventAdminCollector"
    name="ea-collector" architecture="true">
    <properties>
        <property name="topic" field="m_topics" />
        <property name="filter" field="ldapfilter" />
    </properties>
    <callback transition="validate" method="started" />
    <callback transition="invalidate" method="stopped" />
</collector>

<linker classname="components.binding.ea.EABindingService" name="event-admin">
    <collector type="ea-collector" />
    <sender type="ea-sender" />
</linker>
```

3.7 Définition d'une chaîne de médiation

La configuration d'une chaîne de médiation est réalisée en suivant un langage de description d'architectures appelé DSCilia. Ce langage permet de décrire l'ensemble de médiateurs d'une chaîne ainsi que leurs connexions et les adapteurs. Il permet également de configurer ces différents paramètres.

Un fichier DSCilia est basé sur une grammaire XML et il peut contenir autant de chaînes que nécessaire. Chaque chaîne est définie par un identificateur unique (usuellement un nombre) et par ses constituants. Il faut noter qu'il n'y a pas de partage de composants entre divers chaînes CILIA : des médiateurs configurés n'appartiennent qu'à une seule chaîne.

Les éléments du langage de spécification des chaînes de médiation CIlia en XML sont définis comme suit :

<cilia> := représente le nœud racine d'un fichier DSCilia.

<chain> := représente le nœud qui délimite la description d'une chaîne de médiation CILIA. Un seul fichier DSCilia peut contenir autant de chaînes que souhaité. Néanmoins, chaque chaîne doit être identifiée par un nom unique.

<mediators> := représente le nœud qui inclut toutes les instances de médiateurs qui appartiennent à une chaîne.

<adapters> := représente le nœud qui inclut toutes les instances d'adaptateurs qui appartiennent à une chaîne.

<bindings> := représente le nœud qui inclut toutes les liaisons entre tous les composants de médiation.

Les balises présentées ci-avant permettent d'organiser l'ensemble des éléments d'une chaîne de médiation CILIA. Nous verrons que ces différents éléments sont ensuite interprétés par le framework d'exécution pour la création des structures à l'exécution.

Les instances de médiateurs doivent contenir :
- un nom, un type et, optionnellement, le namespace du type de médiateur,
- les propriétés de paramétrage du *scheduler*,
- les propriétés de paramétrage du *processor*,
- les propriétés de paramétrage du *dispatcher*.

Le paramétrage des adaptateurs est plus simple. Ceci est simplement dû au fait que ses constituants (*scheduler*, *processor* et *dispatcher*) ne possèdent pas de paramètres spécifiques. Cependant, le langage permet d'ajouter des paramètres pour configurer les éléments de connectivité (*collector*, *sender*). Cela est nécessaire dès que l'on souhaite utiliser des protocoles de communication particuliers par exemple.

La chaîne de médiation qui répond au problème d'intégration suivi tout au long de ce chapitre est définie comme suit :

```
<cilia>
        <chain id="myChain-1">
                <mediators>
                        <mediator-instance type="AggregatorExample" id="aggregator">
                                <scheduler>
                                        <property name="period" value="30" />
                                </scheduler>
                        </mediator-instance>
                </mediators>
                <adapters>

                        <adapter-instance type="jabber-adapter" id="jabber">
                                <property name="user.name" value="issac" />
                                <property name="user.password" value="123qwerty" />
                                <property name="service.host" value="talk.google.com" />
                                <property name="service.port" value="5222" />
                        </adapter-instance>

                        <adapter-instance type="facebook-adapter" id="facebook" />

                        <adapter-instance type="twitter-adapter" id="twitter" />


                </adapters>
                <bindings>
                        <binding from="jabber:exit" to="aggregator:jabber.port" />
                        <binding from="aggregator:fb.port" to="facebook:std" />
                        <binding from="aggregator:tw.port" to="twitter:std" />
                </bindings>
        </chain>
</cilia>
```

Cet exemple montre une chaîne de médiation appelée « myChain-1 ».

Cette chaîne de médiation est constituée d'un médiateur avec une identification unique nommé aggregator et trois adaptateurs : un adaptateur d'entrée déterminé par l'identificateur unique « jabber », un adaptateur de sortie déterminé par l'identificateur unique « facebook » et, finalement, un autre adaptateur de sortie déterminé par l'identificateur unique « twitter ».

Nous pouvons voir sur cet exemple que, lorsque l'on configure une chaîne de médiation, il est possible de paramétrer les instances des différents composants de la chaîne. C'est le cas du médiateur aggregator et de l'adaptateur jabber.

Cet extrait de code met également en évidence le langage utilisé pour spécifier les liaisons entre médiateurs, d'un port de sortie vers un port d'entrée.

4 Framework de développement

4.1 Implantation des types de données

Nous avons défini un conteneur de données. Ce conteneur prend la forme d'une classe Java qui renferme un objet représentant le message à transmettre d'un constituant à un autre, ou d'un médiateur à un autre dans une chaîne de médiation. Le conteneur permet aussi de stocker des méta-informations utiles lors des différents traitements. La classe implantant cette notion de conteneur est appelée « Data » (fr.liglab.adele.cilia.Data), voir Figure 6.12. Un objet de type Data peut contenir les méta-informations suivantes :

- le contenu de la donnée (un objet Java) ;
- la date et l'heure de création ;
- le nom sémantique ;
- le type de donnée sémantique ;
- le nom du port par lequel il est arrivé ;
- et toute autre information qui, pour une application précise, soit indispensable.

Les objets Data peuvent être enrichis lors de leur traitement. Par exemple, lors de son activation au sein d'un médiateur, le processor est capable d'ajouter des méta-informations qui seront pertinentes pour une autre médiateur dans la chaîne.

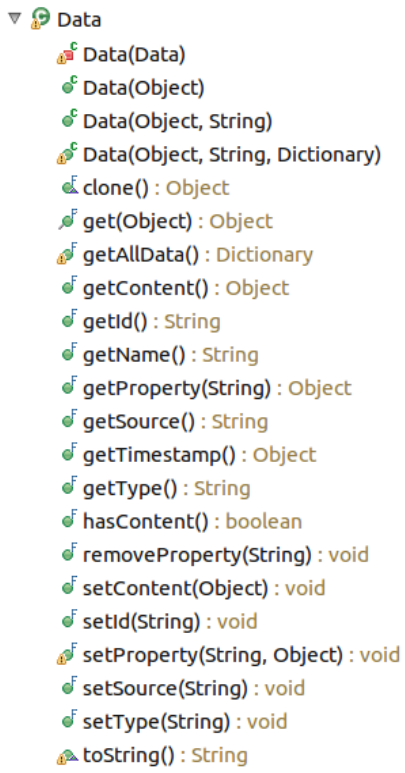

FIGURE 6.12 – API de la classe fr.liglab.adele.cilia.Data

4.2 Implantation des « *schedulers* »

Nous pouvons identifier trois types de *schedulers* :
- les schedulers basés sur le temps,
- les schedulers basés sur une analyse des données reçues,
- et les scheduler basés sur un événement externe.

Le framework CILIA propose des classes abstraites pour faciliter l'implémentation de *schedulers* des deux premiers types.

Pour le premier type de scheduler, une classe abstraite appelée « AbstractPeriodicScheduler » est chargée de gérer les aspects temporels. Pour que cette classe devienne concrète, deux méthodes doivent être implantées : « startTimer() » et « timeOut(..) ». C'est le framework d'exécution qui, en fonction de ces deux méthodes, se chargera d'appeler le processor au bon moment.

La deuxième classe abstraite est appelée « AbstractCorrelationScheduler ». Cette classe permet de construire des schedulers fondés sur une analyse des données reçues. Plus précisément, à chaque fois qu'une donnée est reçue sur un port d'entrée, un traitement est déclenché pour décider de l'appel du processor. Pour ce faire, il faut implanter la méthode « checkCompletness(..) » qui doit contenir l'algorithme d'analyse de données.

Dans la Figure 6.13, nous présentons les APIs de deux classes abstraites. Les méthodes qui restent à développer sont mises en évidence.

FIGURE 6.13 – Classes abstraites de schedulers

Dans notre exemple d'intégration Jabber, Twitter et Facebook, nous devrons implanter un *scheduler* qui réalise une corrélation de messages séparés par un écart temporel inferieur à 30 secondes.

Ainsi, il convient d'implanter la logique suivante :

1. pour chaque message reçu, le *scheduler* le stocke et active une minuterie (30 secondes dans notre cas),
2. si un autre message arrive entre temps, le scheduler stocke aussi ce message et il réactive la minuterie (une nouvelle fois 30 secondes),
3. une fois délai de 30 secondes dépassé, le scheduler déclenche le traitement de tous les messages qui sont stockés en les transmettant au processor. Il se remet alors en attente de nouveaux messages (retour au point 1).

Puisque ce *scheduler* est basé sur des aspects temporels, nous avons implanté une classe Java qui hérite d'une classe abstraite « AbstractPeriodicScheduler » (et la rend donc exécutable, c'est-à-dire instanciable, pour le framework d'exécution).

Le code de la méthode est le suivant :

```
public class TimerScheduler extends AbstractPeriodicScheduler {

    public  TimerScheduler() {
        schedulerPoolExecutor = new ScheduledThreadPoolExecutor(1);
    }
    /**
     * Injected field by the framework.
     * It specifies the time between correlated messages.
     */
    private long period;
    /**
     * Method called when time is running out.
     * It process all received data.
     */
    public void timeOut(List stockedData) {
        process(stockedData);
    }
    /**
     * Method called when new data arrives.
     * It call the parent method to stock the incoming data
     * and initializes the timer.
     */
    public void notifyData(Data data) {
        super.notifyData(data);
        //Each time a data arrives, it initialize the timer.
        startTimer();
    }
    /**
     * Starts the timer.
     */
    public void startTimer() {
    /**
     * Stops the timer.
     */
    private void stopTimer() {
}
```

4.3 Implantation des « *processors* »

La classe qui implante un *processor*, à la différence d'un *scheduler*, n'est pas obligée d'étendre une classe abstraite ni d'implanter une interface particulière. Néanmoins, il y a certaines conventions à respecter pour une utilisation correcte par le framework d'exécution.

Ces conventions sont les suivantes :

- une méthode principale de traitement doit être définie,
- cette méthode reçoit un seul paramètre et elle doit retourner comme résultat un unique objet.

Dans notre exemple d'intégration de Jabber, Facebook et Twitter, nous avons décidé de développer un processor chargé de réaliser une agrégation. Ainsi, il reçoit une liste d'objets Data, et il retourne comme résultat un objet Data.

Voici la classe processor correspondant à notre exemple :

```
public class AggregatorProcessor {

    public Data aggregate(List<Data> listOfData) {
        StringBuffer sb = new StringBuffer();
        for (int i = 0; i < listOfData.size(); i++) {
            Data current = listOfData.get(i);
            sb.append(current.getContent());
            sb.append("\n");
        }
        Data ndata = new Data(sb.toString());
        return ndata;
    }
}
```

4.4 Implantation des « *dispatchers* »

Le *dispatcher* est le seul constituant de médiation qui est obligé de connaître les ports de sortie du médiateur qui le contient. Cette connaissance peut être établie grâce à des propriétés ou à partir de la logique de dispatching. Par exemple, un *dispatcher* basé sur le contenu peut être configuré par un ensemble de tuples de la forme <condition, port>, où la condition est une opération booléenne sur le contenu et le port, le port de sortie où envoyer la donnée si la condition est vérifiée.

Nous pouvons identifier trois types de dispatcher :

- le *dispatcher* qui envoie le résultat à tous les ports connectés,
- le *dispatcher* qui analyse le contenu de l'information de façon générique, c'est-à-dire que les données sont compréhensibles par un analyseur générique,
- le *dispatcher* métier plus spécifique qui ne peut pas être implanté de façon générique. Le dispatcher renferme une connaissance directement liée à l'application traitée. Par exemple, la décision de routage peut être déterminée par une information externe comme une base de données, un service externe à la chaîne ou une information spécifique aux systèmes à

intégrer.

Le framework de développement CILIA propose une unique classe abstraite pour les *dispatchers*. En fait, pour les trois types de dispatchers identifiés, il ne reste à implémenter qu'une seule méthode.

Ainsi, le développement d'un *dispatcher* suit les conventions suivantes :

- il doit étendre une classe abstraite,
- il doit faire des appels à des méthodes de cette classe abstraite.

Le *dispatcher* doit étendre une classe abstraite nommée « CiliaDispatcher ». Précisément, il doit implémenter la méthode « *dispatch* » qui reçoit un paramètre de type Data (fr.liglab.adele.cilia.Data). Mais, le plus important est qu'il doit appeler une méthode « *send* » qui reçoit deux paramètres : le premièr est le nom du port de sortie auquel s'adresser et le second est la donnée à livrer.

Dans notre exemple, le *dispatcher* doit connaître les ports de sortie. Nous avons déclaré que le médiateur a deux ports de sortie. Un port appelé « tw.port » lié à l'adaptateur Twitter et un port « fb.port » lié à l'adaptateur Facebook. La logique de routage est la suivante :

- Si (text.lenght < 140) alors envoi vers « tw.port »,
- Si (text.lenght > 0) alors envoi vers fb.port.

Le code de ce *dispatcher* est le suivant :

```
public class StringLenghtDispatcher extends CiliaDispatcher {

    private final int TWITTER_MAX = 140 ;

    /**
     * @param context
     */
    public StringLenghtDispatcher(BundleContext context) {
        super(context);
    }
    /**
     * Method called when ready to route.
     * @param data data to route.
     * @throws CiliaException
     */
    public void dispatch(Data data) throws CiliaException {
        int count = count(data);
        if (count < TWITTER_MAX) {
            send("tw.port", data);
        }
        send ("fb.port", data);
    }
    /**
     * Count the characters in data content.
     * @param data
     * @return
     */
    public int count(Data data) {
        int count = String.valueOf(data.getContent()).length();
        return count;
    }

}
```

4.5 Implantation des connecteurs externes

4.5.1 Connecteurs d'entrée

Le code des connecteurs externes d'entrée prend la forme de collectors. Dans les sections précédentes, nous avons en effet précisé que les adaptateurs d'entrée sont implantés en partie grâce à un collecteur qui réalise la logique d'adaptation.

CILIA permet de construire des adaptateurs d'entrée conformes aux deux styles d'interaction les plus communs :

- le style d'interaction nommé « *pull* » qui récupère des données par appel à une application externe,
- le style « *push* » où c'est l'application externe qui envoie des messages à l'adaptateur.

FIGURE 6.14 – Classes abstraites de collectors.

Afin de simplifier le développement de ces entités entièrement techniques (et souvent très complexes), le framework de développement CILIA fournit deux classes abstraites (voir Figure 6.14) qu'il convient de concrétiser.

Il s'agit des classes suivantes :

- un « AbstractCollector » qui est la classe abstraite principale qui permet d'implémenter un collector. En bref, cette classe contient le code technique qui permet l'interaction avec le framework CILIA ;
- un « AbstractPullCollector » qui contient du code technique pour simplifier une interaction *pull* de type périodique où pour chaque période de temps, le collecteur réalise une collecte de données sur le système externe.

Dans notre exemple d'intégration de Jabber, Twitter et Facebook, nous avons développé un collector appelé « jabber-collector ». La classe du *collector*, contenant le code de connexion, étend la classe abstraite de CILIA appelée « AbstractCollector ».

Afin d'utiliser l'API de communication vers Jabber, cette classe implante une interface appelée « MessageListener ».

Le code de cette méthode est le suivant :

```
public class JabberCollector extends AbstractCollector implements MessageListener{
    //Fields injected by the framework.
    private String user;//
    private String login;
    private String pwd;
    private String host;//"talk.google.com"
    private int port;//5222
    private String service; //gmail.com
    /**
     * To connect to the Jabber servcer.
     */
    XMPPConnection connection;
    /**
     * Method called when a new message is received.
     */
    public void processMessage(Chat chat, Message message) {
        Data receivedData = null;
        if (message.getType() == Message.Type.chat) {
            receivedData = new Data("message", message.getBody());
            super.notifyDataArrival(receivedData);
        }
    }
    /**
     * Connect to the Jabber Server.
     * @throws XMPPException
     */
    public void connect() throws XMPPException {

}
```

Dans notre code d'exemple, une méthode nommée « processMessage (. . .) » est appelée par la bibliothèque Jabber. Une fois cette méthode appelée, le *collector* notifie au *scheduler* la réception d'un nouveau message en appelant la méthode « notifyDataArrival (. . .) ».

4.5.2 Connecteurs de sortie

Le code des connecteurs externes de sortie prend la forme de *senders*. Dans les sections précédentes nous avons en effet indiqué que les adaptateurs de sortie sont implantés en partie grâce à un sender qui réalise la logique d'adaptation.

Les adaptateurs de sortie permettent de construire des entités conformes à un seul style d'interaction, le push, où l'adaptateur d'une chaîne de médiation envoie des messages à l'application externe. Etant donné que ce style ne requiert pas de complexité technique, ni d'interaction avec le conteneur du composant, il suffit d'implémenter une interface pour que le framework appelle correctement la méthode fournie par l'application externe.

Dans notre exemple, nous avons introduit un adaptateur de sortie qui utilise la logique d'adaptation d'un sender vers Twitter.

La classe *sender* (code de connexion) implante une interface de CILIA appelée « ISender ». Cette interface impose l'implantation d'une méthode appelée « send(..) » qui prend comme paramètre l'objet Data contenant le message à envoyer au service Twitter.

Le code du *sender* Twitter est le suivant :

```
public class TwitterSender implements ISender {

    /**
     * Service OSGi injected par iPOJO
     */
    TwitterService service;

    /**
     * Method called when sending a tweet.
     */
    public boolean send(Data arg0) {

        try {
            service.publishTweet(getTweet(arg0));
        } catch (Exception e) {
            e.printStackTrace();
        }

        return true;
    }

    private String getTweet(Data data) {
        String ntweet = String.valueOf(data.getContent());
        return ntweet;
    }

}
```

4.5.3 Les connecteurs d'entre/sortie

Le code de connexion des connecteurs externes de type entrés/sortie est réalisé dans un adaptateur spécial. Cet adaptateur est chargé de transformer une interaction requête/réponse en une interaction push. Ceci est fait de façon à ce que l'adaptateur permette à une application externe à la chaîne de médiation d'interagir avec la chaîne. Plus précisément, il permet d'exposer la fonctionnalité d'une chaîne de médiation comme un service.

Cet adaptateur est très différent des deux précédents. Néanmoins, nous avons cherché simplifier son développement autant que possible. Ainsi, pour construire un adaptateur d'entrée/sortie, il existe une classe abstraite proposant un cadre structurant pour le code. Cette classe abstraite propose des méthodes qui permettent de transformer une interaction du style push (pour l'interaction entre médiateur) en un style d'interaction de type requête/réponse.

Dans notre exemple d'intégration de Jabber, Twitter et Facebook, il n'y a aucun adaptateur de ce type. Pour l'illustration, nous présentons un exemple simple de service OSGi qui fournit une interface appelée « DemoAdapterI ».

Le code de notre adaptateur est le suivant :

```
public class DemoAdapterImpl extends AbstractIOAdapter implements DemoAdapterI {


    public DemoAdapterImpl(BundleContext context) {
        super(context);
    }

    /**
     * Method of the DemoAdapterI Interface.
     */
    public String getPrice(String params) {
        //it sends the incoming data to first mediator on the chain and wait for the result.
        //This result comes from  another mediator.
        Data ndata = new Data(params);
        Data resultdata = invokeChain(ndata);
        String result = String.valueOf(resultdata.getContent());
        return result;
    }

    public void receiveData(Data data){
        super.receiveData(data);
    }

    public Data dispatchData(Data data) {
        return super.dispatchData(data);
    }

}
```

L'exemple présenté ci-dessus montre un adaptateur d'entrée/sortie. Il implémente une interface métier (DemoAdapterI) et une méthode de cette interface appelée « getPrice(...) ». Dans cette méthode, une méthode de la classe abstraite « invokeChain(...) » est appelée. Cette méthode envoie une donnée de type Data à un premier médiateur connecté à l'adaptateur. Puis, l'adaptateur reste bloqué un certain temps pour attendre le résultat provenant d'un dernier médiateur dans la chaîne.

4.6 Implantation des connecteurs internes

Les connecteurs internes permettent de lier deux médiateurs au sein d'une chaîne de médiation. La connexion dans CILIA est réalisé par des entités *collector* et *sender*.

Le *sender* et le *collector* sont implantés de la même façon que les connecteurs des adaptateurs. Ceci permet, par exemple, de créer un *linker* d'événements avec le *sender* et *collector* et, en plus, de créer un adaptateur de sortie d'événements et un adaptateur d'entrée d'événements. L'élément *linker*, par contre, doit être un service OSGi qui implante une interface appelée « CiliaBindingService ».

L'approche utilisée pour la liaison entre médiateurs est ainsi basée sur les éléments suivants :

- le composant émetteur doit contenir un constituant de type *sender*,
- le composant récepteur doit contenir un constituant de type *collector*.
- une entité appelée *linker* est chargée d'ajouter chaque constituant (*sender* et *collector*) au bon médiateur,
- les constituants sont configurés par le *linker* à l'exécution afin d'assurer une communication correcte.

Par exemple, un *linker* d'un service d'événements basé sur des « *topics* » est chargé d'ajouter un *sender* d'événements et un *collector* d'événements aux composants. De plus, il les configure afin que le *sender* écrive dans un *topic* et que le *collector* lise ou s'abonne à ce même topic. Néanmoins, les deux constituants doivent contenir une logique correcte qui assure le bon fonctionnement. Ceci est dû au fait que le framework et le *linker* ne sont chargés que des configurations, et non de la cohérence de la sémantique d'ensemble.

Dans la section précédente, nous avons présenté la spécification d'un linker basé sur le service « Event Admin [5] » d'OSGi.

Voici un extrait de code de ce *linker* :

```
public class EABindingService extends CiliaBindingServiceImpl implements
CiliaBindingService {
    protected static final String collectorProperty = "collector.topic";
    protected static final String senderProperty = "topic";

    public Dictionary getProperties(Dictionary collectorProperties,
            Dictionary senderProperties, Binding b) {

        Dictionary properties = new Properties();

        String topic = getTopic(b);
        topic = topic.replace(" ", "_");

        return properties;
    }

    private String getTopic(Binding b) {
}
```

En bref, lors qu'on étend la classe « CiliaBindingServiceImpl », il suffit d'implanter une méthode appelée « getProperties() ». Ce code sert à configurer le sender et le collector du médiateur émetteur et du médiateur récepteur respectivement.

5. http ://felix.apache.org/site/apache-felix-event-admin.html

5 Framework d'exécution

Comme nous l'avons expliqué, le framework d'exécution est composé d'un « API réflexif », d'un « méta-level », d'un « gestionnaire de synchronisation », d'un « base-level » et d'un « gestionnaire de données ».

L'« **API réflexif** » est une entité très importante. En première, il permet l'assemblage de chaînes de médiation, et deuxièmement, il permet de réaliser la reconfiguration aux chaînes de médiation existantes.

Une caractéristique à faire noter est que cet API suit les règles du modèle d'assemblage et donc il peut être considéré comme un Java DSL (ou *internal DSL* [Fow06]). Cela signifie que l'API permet de décrire l'assemblage ou configuration d'architecture des applications de médiation.

Dans l'implémentation présentée dans cette section, nous avons développé diverses classes qui font partie de l'API réflexif. Notamment, nous avons construit une classe pour chaque entité du modèle. En particulier, il existe une classe appelée « *Chain* », qui contient toutes les méthodes nécessaires pour ajouter des éléments, ainsi que pour réaliser des liaisons. Il existe aussi une classe appelée « *Mediator* » et une autre classe appelée « *Adapter* », qui contiennent des méthodes pour configurer les composants, et il existe aussi une classe appelé « *Binding* ».

Dans une section précédente, nous avons montré un langage de description de chaînes de médiation en XML (le fichier DSCilia). Le parseur de ces fichiers utilise cet API pour construire les chaînes de médiation à l'exécution. Une fois qu'une chaîne de médiation est instanciée, il est toujours possible d'utiliser ce même API pour inspecter l'architecture des applications et, si nécessaire, les modifier.

La Figure 6.15 montre une exemple typique de l'API réflexif. Une analyseur de fichiers DSCilia s'appuye sur cet API pour produire des objets qui correspondent à la chaîne de médiation décrit en DSCilia.

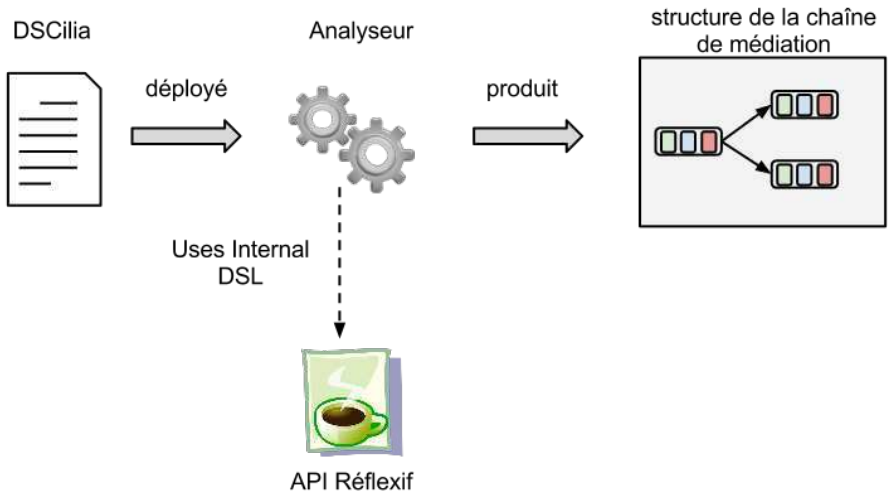

FIGURE 6.15 – Exemple de l'utilisation de l'API réflexif

Le **méta-level** est un ensemble d'objets descriptifs des chaînes de médiation. Ainsi, une description d'architecture exprimée avec DSCilia est transformée à l'exécution comme un ensemble d'objets dans le « méta-level » qui représentent cette même architecture. Les objets dans le méta-level, sont des objets construits en utilisant l'API réflexif.

L'objet le plus important dans le méta-level est un celui qui implémente une interface appelée « CiliaContext ». CILIA fournit un objet de ce type sous la forme d'un service OSGi qui, de fait, est un conteneur de chaînes de médiation. Les méthodes de ce conteneur permettent d'ajouter une chaîne de médiation, de l'initialiser et de l'éliminer le cas échéant. En bref, ce conteneur est un gestionnaire de chaînes de médiation.

La figure 6.16 montre, comme exemple, l'API de la classe « Chain » ainsi que l'API de l'interface « CiliaContext ».

```
Chain
  adapters
  bindings
  lockObject
  mediators
  Chain(String, String, String, Dictionary)
  add(Adapter) : boolean
  add(Mediator) : boolean
  bind(Port, Binding) : Binding
  bind(Port, Port) : Binding
  bind(Port, Port, Binding) : Binding
  dispose() : void
  getAdapter(String) : Adapter
  getAdapters() : Set
  getBindings() : Set
  getBindings(MediatorComponent, MediatorComponent) : Binding[]
  getMediator(String) : Mediator
  getMediators() : Set
  inputBind(Port, Binding) : Binding
  outputBind(Port, Binding) : Binding
  removeAdapter(String) : boolean
  removeMediator(Mediator) : boolean
  removeMediator(String) : boolean
  unbind(Binding) : boolean
  unbind(Binding[]) : void
  unbind(MediatorComponent, MediatorComponent) : boolean
  unbind(String, String) : boolean
  unlinkBindingsFromChain(Binding[]) : void
```

```
CiliaContext
  CILIA_VERSION
  addChain(Chain) : Chain
  addChainListener(String, ChainListener) : void
  createMediatorSpecification(String, String, String) : MediatorSpecification
  getAllChains() : Set
  getChain(String) : Chain
  getCiliaVersion() : String
  getMutex() : ReadWriteLock
  removeChain(Chain) : void
  removeChain(String) : void
  removeChainListener(String, ChainListener) : void
  start() : void
  startChain(Chain) : void
  startChain(String) : void
  stop() : void
  stopChain(Chain) : void
  stopChain(String) : void
```

FIGURE 6.16 – API de CiliaContext et Chain

Le **gestionnaire de synchronisation** est une entité interne au framework CILIA. En fait, chaque classe du méta-level a une classe gestionnaire. Ainsi, au moment d'initialiser une chaîne de médiation, le framework crée une instance d'un gestionnaire pour cette chaîne ; laquelle fait une instance d'un gestionnaire pour chacun des éléments de la chaîne. Ainsi, il y a un gestionnaire pour chaque médiateur, un gestionnaire pour chaque adaptateur et un gestionnaire pour chaque liaison.

Les gestionnaires sont chargés de maintenir la cohérence entre le méta-level et le base-level. Ils utilisent le patron « observer » pour être notifiés des modifications au niveau méta-level. Par exemple, le gestionnaire de chaînes est notifié lorsque l'on ajoute une connexion à l'exécution. Le

gestionnaire de chaînes ajoute un nouveau gestionnaire de liaison qui sera chargé de réaliser les ajustements dans le base-level (Figure 6.17).

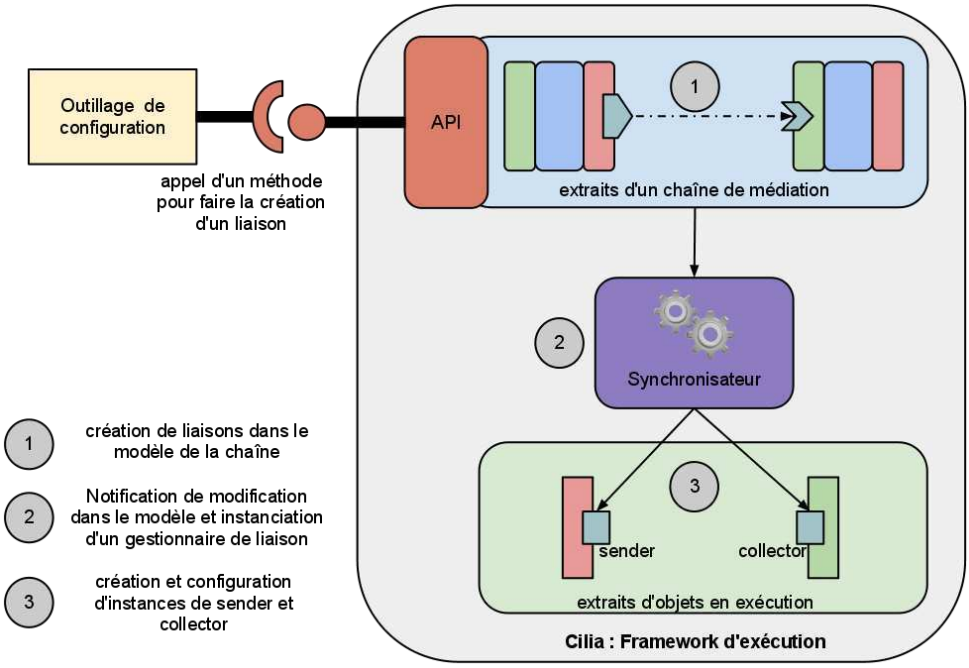

FIGURE 6.17 – Exemple de la création d'un liaison entre médiateurs

Le **base-level** est constitué des instances des composants. Ces instances n'ont pas de relations directes vers le méta-level puisque c'est le gestionnaire de synchronisation qui est chargé de maintenir la cohérence à l'exécution. Les instances du base-level comprennent les instances des constituants (implémentant des interfaces CILIA ou étendant des classes abstraites fournies) ainsi que les instances des conteneurs de composants. Chaque conteneur de médiateurs, par exemple, maintient l'interaction entre chaque constituant. Ainsi, le conteneur est notifié lorsque le *scheduler* déclenche le traitement et c'est le conteneur qui envoie les paramètres au *processor*. Ensuite, le conteneur reçoit le résultat du traitement et il le transmet au *dispatcher*.

Le **gestionnaire de données** est un autre élément important du framework d'exécution. Dans la version actuelle de CILIA, il y a un gestionnaire de données pour chaque chaîne de médiation. Ce gestionnaire fait l'usage de la mémoire volatile. Cependant, nous avons assuré un faible couplage entre gestionnaire et support de stockage grâce à l'approche à services (service local fondé sur OSGi). Ainsi, nous envisageons d'implémenter divers gestionnaires pour des besoins précis, par exemple, un gestionnaire basé sur des bases de données ou un autre basé sur un système de fichiers répartis (par exemple Hadoop [6]).

Chaque médiateur d'une chaîne de médiation a droit à un espace de mémoire sur ce gestionnaire. Ainsi, au moment où un *scheduler* demande des données, le gestionnaire livre seulement les données qui se trouvent dans son espace de mémoire. Nous avons découpé ce gestionnaire du

6. http ://hadoop.apache.org/

composant de médiation, plus précisément des *schedulers*, de façon à permettre plus de souplesse au moment de réaliser des reconfigurations à l'exécution. Par exemple, lorsqu'un médiateur est échangé par un autre, le framework bloque l'accès aux données pendant la migration. Puis, une fois la migration réalisée, l'espace de mémoire du nouveau médiateur permet de récupérer les données de l'ancien médiateur. Ceci fait la supposition que les deux médiateurs (l'ancien et le nouveau) acceptent les mêmes types de données. Si ce n'est pas le cas, la migration doit être accompagnée d'une autre reconfiguration plus complexe.

Nous avons réparti toutes les capacités de CILIA sur plusieurs bundles OSGi. Ainsi, le framework d'exécution est principalement divisé en trois grands modules :

- le module core comprend l'ensemble des classes Java de base. Ces classes permettent la construction de chaînes de médiation. De plus, ce module contient les interfaces Java des services de base comme, par exemple le service de l'API réflexive,
- le module du runtime comprend l'ensemble des classes et composants iPOJO qui permettent de transformer une spécification de chaîne de médiation en instances concrètes de composants iPOJO. Ce module contient le synchronisateur de la couche de modèles vers la couche d'objets,
- le module de compendium comprend des constituants de base comme certains services qui permettent de simplifier le développement d'autres constituants. Il s'agit, par exemple, des services de gestion de documents XML, ainsi que l'exécution d'instructions XPATH et XSLT.

Enfin, il existe d'autres modules qui nous permettent la gestion à l'exécution :

- module d'analyse de fichiers DSCilia,
- module de déploiement de fichiers DSCilia,
- modules de gestion de chaînes de médiation par commandes dans un Shell OSGi (Felix Shell et GOGO Shell).

6 Cycle de vie

6.1 Outils à la conception

Les artefacts à la conception sont de deux natures différentes : il peut s'agir de spécification ou de développement. Les spécifications correspondent à des descriptions d'assemblage (ou niveau des médiateurs ou des chaînes de médiation) alors que le code correspond à du code Java conforme à des interfaces définies par le framework CILIA. La plupart des artefacts contiennent des méta-informations.

Le premier type d'artefact correspond aux descriptions de médiateurs et de leurs constituants. Il contient le code Java et de la méta-information sur chaque entité, que cela soit un médiateur ou un constituant de médiateur. Il y a un couplage fort entre la méta-information et les classes Java produites. Par conséquent, la méta-information et code sont toujours dans le même projet de développement.

Pour la construction de ces artefacts, nous nous appuyons sur une technologie Java appelée Maven [7]. Cette technologie permet la génération d'artefacts et l'utilisation de « plug-ins » capables d'effectuer des traitements automatiques sur des projets de développement. Un plug-in peut effectuer des opérations simples, comme ajouter un fichier à un projet, ainsi que des des opérations complexes, comme modifier des fichiers existants. CILIA utilise deux plug-ins : le « Maven-Bundle-Plugin », pour la génération de bundle OSGi ainsi que le « maven-ipojo-plugin » pour la manipulation de « bytecode » d'iPOJO. La configuration de ces deux plug-in est effectuée dans un fichier de configuration de projet maven appelé POM. La configuration est la suivante :

```
<plugin>
    <groupId>org.apache.felix</groupId>
    <artifactId>maven-bundle-plugin</artifactId>
    <extensions>true</extensions>
    <configuration>
        <instructions>
            <Bundle-SymbolicName>${pom.artifactId}</Bundle-SymbolicName>
            <Export-Package />
            <Private-Package>
                test.constituant.dispatcher,
                test.constituant.processor, test.constituant.scheduler
            </Private-Package>
        </instructions>
    </configuration>
</plugin>
<plugin>
    <groupId>org.apache.felix</groupId>
    <artifactId>maven-ipojo-plugin</artifactId>
    <executions>
        <execution>
            <goals>
                <goal>ipojo-bundle</goal>
            </goals>
            <configuration>
                <metadata>metadata.xml</metadata>
            </configuration>
        </execution>
    </executions>
</plugin>
```

7. http ://maven.apache.org

Le deuxième type d'artefact est le fichier de description d'architecture appelé DSCilia. Ce fichier est écrit en XML et il permet le déploiement de descriptions de chaînes de médiation à chaud. Ce fichier est obtenu à partir d'un éditeur XML. Cependant, nous avons développé un prototype d'éditeur graphique pour la génération automatique d'un fichier DSCilia. Cet éditeur permet une composition graphique de chaînes de médiation (figure 6.18).

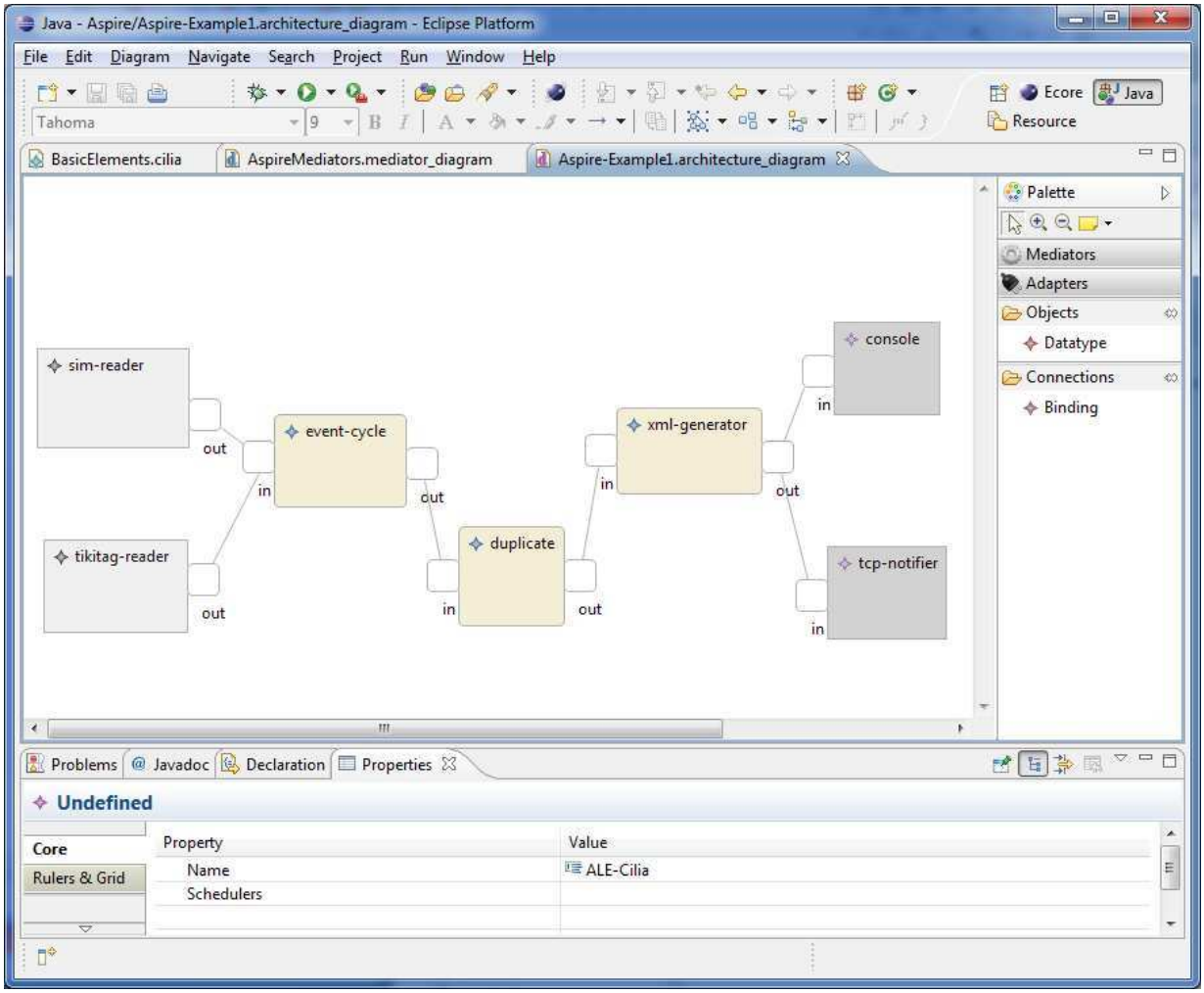

FIGURE 6.18 – Editeur de chaîne de médiation

L'usage d'un éditeur graphique permet d'éviter des erreurs communes de typage et permet d'assurer les bonnes liaisons entre composants de médiation. L'aide visuelle est importante et simplifie la compréhension, ainsi que le paramétrage.

En utilisant un éditeur de fichiers XML généraliste, il est possible de garantir une cohérence au niveau syntaxique ; ceci grâce à la définition de schéma XML qui permet la construction d'un fichier DSCilia correct.

```
torito@charron: ~/Documents/workspaces/demos/tp5-121/felix3.2.2-cilia1.2.1
  29|Active     |    1|OW2 Chameleon - DOJO Toolkit for OSGi (1.3.2.SNAPSHOT)
  30|Active     |    1|OW2 Chameleon - Apache Commons Logging Bundle (1.1.1.0003-SNAPSHOT)
  31|Active     |    1|Split/Aggregate Demo - chain services (1.0.0.SNAPSHOT)
g! cilia:help
cilia
                create
                        chain id=chainid
                        mediator chain=chainId type=namespace:mediatorType id=mediatorId
                        adapter  chain=chainId type=namespace:adapterType id=adapterId
                        binding  chain=chainId from=mediator:port to=mediator:port
                modify
                        chain id=chainid property=propertyName value=propertyvalue type=[primitive|Array|Map]
                        mediator chain=chainId id=mediatorId property=propertyName value=propertyvalue
                        adapter  chain=chainId id=adapterId property=propertyName value=propertyvalue
                        binding  chain=chainId from=mediator:port to=mediator:port property=propertyName value=propertyvalue
                remove
                        chain id=chainid
                        mediator chain=chainId  id=mediatorId
                        adapter  chain=chainId  id=adapterId
                        binding  chain=chainId from=mediator:port to=mediator:port
                start
                        chain id=chain_id
                stop
                        chain id=chain_id
                show
                        chain id=chain_id
                        mediator id==mediator_id chain=chainId
                        adapter id=adapter_id chain=chainId
                load
                        file=url


g! help
cilia:create
cilia:help
cilia:load
cilia:modify
cilia:remove
cilia:show
cilia:start
```

FIGURE 6.19 – Shell d'administration de chaînes de médiation

6.2 Outils de déploiement

A propos du déploiement, comme nous l'avons précisé précédemment, il y a deux types d'artefacts Cilia : les bundles OSGi qui contiennent le code et méta-information, et les fichiers DSCilia qui contiennent la description de chaînes de médiation. Les artefacts du premier type sont déployés grâce à des outils disponibles par la plate-forme d'exécution OSGi. Par exemple, il y a des instructions niveau *Shell* pour déployer des bundles, ainsi que des fichiers de configuration de bundles à déployer.

Nous avons développé un gestionnaire de déploiement de fichiers CILIA. Ce gestionnaire permet, par exemple, de configurer un répertoire (dans le système de fichiers) comme conteneur d'applications de médiation. Ainsi, lorsqu'on met à disposition un fichier DSCilia dans ce répertoire, le gestionnaire de déploiement fait l'ajout des chaînes automatiquement.

6.3 Outils à l'exécution

Le framework CILIA est accompagné de divers outils de gestion de chaînes de médiation à l'exécution. Ainsi, nous proposons de base les services suivants qui nous permettent de modifier à chaud les chaînes de médiation.

Un service d'instructions dans le Shell : ce service nous permet de réaliser toutes les reconfigurations disponibles à l'exécution (Figure 6.19). Ainsi, par exemple, il est possible de créer une chaîne de médiation en donnant des instructions, ajouter des médiateurs et liaisons, ainsi que mettre à jour des paramètres.

Nous avons également développé un « manager autonomique » très simple pour tester les capacités de reconfiguration à chaud de CILIA. Plus précisément, nous avons fait un manager

```
CiliaAdminService
    adapterProperty(String, String, String, String, String) : void
    bindingProperty(String, String, String, String, String, String) : void
    chainProperty(String, String, String, String) : void
    copyAdapter(String, String, String) : void
    copyMediator(String, String, String) : void
    createAdapter(String, String, String) : void
    createBinding(String, String, String) : void
    createEmptyChain(String) : void
    createMediator(String, String, String) : void
    execute(String) : void
    getAdapter(String, String) : Adapter
    getChain(String) : Chain
    getMediator(String, String) : Mediator
    mediatorProperty(String, String, String, String, String) : void
    removeAdapter(String, String) : void
    removeBinding(String, String, String) : void
    removeChain(String) : void
    removeMediator(String, String) : void
    replaceAdapter(String, String, String) : void
    replaceMediator(String, String, String) : void
    startChain(String) : void
    stopChain(String) : void
```

FIGURE 6.20 – API du gestionnaire

autonomique qui permet de modifier des chaînes de médiation lorsqu'un service apparaît ou disparaît dans le registre OSGi. Ce manager contient l'information suivante :

- la chaîne à modifier,
- un filtre LDAP pour suivre la trace d'un service OSGi,
- les actions à réaliser lorsque le service apparaît,
- les actions à réaliser lorsque le service disparaît.

Toutes ces capacités d'administration et de gestion à l'exécution sont réalisées grâce à une API proposée par le framework CILIA. Ainsi, comme nous l'avons mentionné, nous mettons à disposition un service OSGi qui permet de construire d'autres entités de gestion à l'exécution plus complexes.

La figure 6.20 présente les options disponibles dans le service réflexif d'administration de chaînes de médiation à l'exécution. Il permet, par exemple, d'éliminer des composants de médiation tout comme d'en ajouter des nouveaux, d'éliminer des liaisons et d'en créer de nouvelles. Il permet aussi de créer de nouvelles chaînes de médiation et de les configurer.

7 Synthèse

Dans ce chapitre, nous avons examiné les détails techniques de l'implémentation CILIA. Nous pouvons constater que nous proposons une version complète et prête à l'emploi, c'est-à-dire constituée d'un environnement de développement et d'un environnement d'exécution.

Cette version est basée sur OSGi et iPOJO. Ces deux frameworks permettent une grande souplesse pour le développement d'applications dynamiques. Néanmoins, les capacités d'iPOJO sont génériques et nous avons développé un conteneur d'applications qui permet de réaliser des opérations de reconfiguration à l'exécution. Ces capacités nous permettent de répondre aux problématiques liées aux nouveaux domaines d'applications, comme l'ubiquitaire, où les systèmes évoluent très rapidement.

Le tableau ci-dessous synthétise les éléments majeurs de notre implantation :

Artefact	**Nombre de lignes de code**
Cilia-core	2822
Cilia-runtime	7252
Cilia-compendium	4453
Cilia-outils	10242
Composants de médiation et constituants	2472
Editeur de composition de chaîne	38997 (la plupart générées)
Shell de commande	2632

Même si l'implémentation proposée repose sur OSGi et iPOJO, nous pouvons également implémenter un framework d'exécution en utilisant une autre technologie. Nous avons fait un prototype minimal basé sur JME (Java Micro Edition) pour construire des chaînes de médiation dans des dispositifs Java, plus concrètement, nous avons utilisé des dispositifs SunSPOT [8].

Dans le chapitre suivant, nous allons montrer des exemples réels de projets qui ont utilisé CILIA ainsi qu'une évaluation quantitative du code CILIA et des détails de performance.

8. http ://www.sunspotworld.com/

7

VALIDATION

Sommaire

1 **Introduction** 168
2 **Projet ASPIRE** 169
3 **Projet MEDICAL** 174
4 **Intégration de systèmes d'information patrimoniaux (Orange)** 179
5 **Evaluation de l'environnement d'exécution** 185
6 **Synthèse** 190

Dans le chapitre précédent, nous avons décrit les détails d'implémentation de CILIA. Nous avons présenté un exemple minimaliste, mais permettant de mettre en évidence les caractéristiques de CILIA. Pour montrer les détails d'implémentation de CILIA, nous avons au début présenté le langage qui nous permet de réaliser les spécifications de chaque entité dans CILIA, allant des composants de médiation, en passant par des entités plus techniques pour réaliser la connectivité par exemple. De plus, nous avons présenté des extraits de code produit pour construire ces entités. Ensuite, nous avons montré les détails du framework d'exécution et finalement les capacités qui nous permettent d'interagir avec CILIA lors du cycle de vie des applications.

Dans ce chapitre nous présentons la validation de notre approche. En premier, nous nous concentrons sur l'usage effectif de CILIA dans des projets collaboratifs nationaux et européens. L'utilisation de CILIA par nos partenaires nous a permis de faire progresser et de valider notre proposition. Dans cette section nous discutions les détails de l'utilisation de CILIA dans ces projets et les avantages et les apports.

Pour conclure ce chapitre, nous nous intéressons à des aspects plus quantitatifs liés au code de CILIA. Nous avons ainsi mesuré l'impact de l'usage de CILIA dans plusieurs cas de figure.

1 Introduction

Le framework CILIA présenté dans cette thèse est d'ores et déjà utilisé dans plusieurs projets collaboratifs. Cela nous a permis de régulièrement confronter CILIA aux besoins de différents domaines et de l'adapter au mieux. Cela a notamment amené la production de plusieurs versions de notre proposition au cours de ces trois dernières années.

Particulièrement, CILIA a été utilisé par les projets suivants :

- Le projet Européen ASPIRE. Le rôle de CILIA était dans ce cas de permettre le traitement et la remontée de données issues de capteurs RFID.
- Le projet Minalogic MEDICAL. Le rôle de CILIA était cette fois de traiter et de remonter sur Internet des données issues de capteurs installés dans une maison équipée pour la mise en place de services numériques.
- Un cas d'usage d'intégration de systèmes d'information patrimoniaux. Dans ce dernier cas, CILIA était utilisé comme un « Enterprise Service Bus » pour intégrer de nouvelles applications avec des applications patrimoniales de gestion des abonnés.

CILIA est également en cours d'évaluation chez Schneider Electric pour l'interopérabilité d'équipements exposés sous forme de services et par Bull SA pour la remontée et le traitement de données de supervision. Néanmoins, nous ne possédons pas encore assez d'informations sur ces deux évaluations pour présenter en détail ces expérimentations dans ce manuscrit.

CILIA est également en cours d'utilisation pour la mise en place d'interfaces multi-modales dynamiques[ANL11, ALN11]. Ici encore, nous manquons de recul et de résultats tangibles pour les détailler ici. Notons néanmoins que ces travaux suscitent un intérêt certain de la communauté IHM qui manque de supports d'exécution dynamiques et simples à programmer.

Le but de ce chapitre d'évaluation est de présenter les trois cas d'utilisation précédemment mentionnés et d'examiner en quoi l'utilisation de CILIA a été bénéfique. En particulier, nous examinons les aspects liés à la modularité, l'extensibilité, le dynamisme, l'embarquabilité et la simplicité de développement. Ces aspects sont de fait les objectifs que nous nous étions fixés lors de la conception et de l'implantation du modèle et du framework CILIA.

Pour chacune des trois expérimentations, nous présentons le problème abordé ainsi que la solution CILIA proposée. Nous discutons ensuite les aspects mentionnés ci-dessus pour évaluer l'intérêt réel de CILIA.

2 Projet ASPIRE

La technologie RFID (*Radio Frequency Identification*) permet d'identifier et de tracer des objets physiques dotés d'une étiquette appropriée, également appelée « *tag* ». Cette technologie, très populaire aujourd'hui, est utilisée au sein de nombreux processus industriels et connaît un engouement croissant. La technologie RFID permet par exemple de mettre en place le suivi de produits dans une chaîne de production ou dans une chaîne d'approvisionnement dans de nombreux domaines.

Les infrastructures RFID comprennent essentiellement des lecteurs d'étiquettes et des *middlewares* de communication pour la remontée de données. Les étiquettes contiennent un identifiant et éventuellement des données complémentaires. Les lecteurs RFID obtiennent des informations sous une forme brute. Ces données brutes doivent donc être traitées et interprétées au sein du middlewares avant d'être livrées à un système d'information.

Le projet européen ASPIRE [1] (*Advanced Sensors and lightweight Programmable middleware for Innovative Rfid Enterprise applications*) avait pour objectif de créer un middleware RFID « *open source* », ouvert et extensible permettant la construction d'applications RFID contextuellement plus riches. Le *middleware* ASPIRE est basé sur la spécification EPCGlobal [2] et propose des ajouts qui permettent la construction d'applications de traçabilité où les lectures RFID sont enrichies avec de l'information contextuelle, comme la géolocalisation ou la température.

Dans le cadre du projet ASPIRE nous avons proposé une implémentation embarqué du serveur ALE (*Application Level Event*) fondée sur CILIA. La spécification de ce module, qui fait également partie d'EPCGlobal, décrit un système chargé de réaliser l'agrégation et de monter en abstraction les données brutes obtenues par les lecteurs RFID.

2.1 Besoins du cas d'usage

Le serveur ALE tel que spécifié par EPCGlobal est composé des éléments logiques suivants :

- Des lecteurs logiques (*Logical Readers*) : ils permettent de découpler les éléments physiques (i.e. les lecteurs) du service ALE.
- Le cycle de lecture (*Read Cycle*) : Le cycle de lecture est l'unité d'interaction la plus petite dans le lecteur qui obtient des lectures de tags.
- Cycle d'événements (*Event Cycle* ou EC) : il définit des frontières temporelles qui expriment le moment pour commencer à collecter des données et le moment pour arrêter de collecter ces données qui constitueront un rapport.
- Rapport (*Reports*) : quand le cycle d'événements est arrêté, les données brutes passent par un processus de filtrage et d'agrégation. Puis un ensemble de rapports est généré.

Pour résumer, l'objectif d'un serveur ALE est d'obtenir des données brutes depuis des tags RFIDs et de produire des rapports de haut niveau d'abstraction qui sont ensuite envoyés aux clients

1. http ://www.fp7-aspire.eu/
2. http ://www.gs1.org/epcglobal

intéressés. La forme de ces rapports est également spécifiée par EPCGlobal. Ils expriment les opérations suivantes à appliquer aux données :

- *Report Set* : pour obtenir l'ensemble des lectures (tag RFID) nouvelles, c'est-à-dire qui n'appartiennent pas au cycle d'événement précédent, ou pour obtenir l'ensemble des lectures appartenant au cycle d'événements précédent et qui ne se trouve pas dans au sein du cycle courant.
- *Include and Exclude* : pour obtenir des lectures filtrées par un ensemble de patrons paramétrés. Ces filtres spécifient quelles lectures doivent être insérées dans les rapports et quelles lectures ne le sont pas.
- *Groupping* : pour obtenir des rapports où les lectures sont groupées selon un patron de corrélation. Par exemple, les lectures qui provenant d'une même entreprise ou les lectures correspondant à des objets d'un même type.
- *Data Format* : pour obtenir un rapport avec les lectures dans un format donné. Cette option permet de définir la façon de réaliser le rapport. Il peut s'agir, par exemple, de montrer seulement la quantité d'éléments lus ou d'afficher des lectures avec la valeur de l'identifiant de l'étiquette.

Dans le cadre du projet ASPIRE, il était demandé de suivre ces spécifications bien définies tout en étant capable de supporter d'autres technologies. Le middleware devait donc être extensible. De plus, il devait supporter le dynamisme et la mobilité dus à des lecteurs RFID mobiles. Ainsi, il était nécessaire de traiter l'ajout d'un nouveau lecteur sans pour autant affecter l'exécution des lecteurs déjà présents.

Ainsi donc, au sein d'ASPIRE, un serveur ALE est vu comme un middleware RFID basé sur la spécification d'EPCGlobal. En outre, il doit être :

- extensible et il pouvoir gérer d'autres sources de données comme des capteurs,
- capable de supporter la génération de rapports dans divers formats,
- capable de supporter la distribution,
- facilement embarquable.

2.2 Solution

Nous modélisons le traitement des lectures d'étiquettes de la manière suivante :

1. Agrégation des étiquettes lues par les différents lecteurs.
2. Élimination des doublons (par exemple, la même étiquette a été lue par deux lecteurs différents pendant le même cycle).
3. Calcul de l'ensemble à inclure dans le rapport. Il existe trois types de calcul, à savoir le CURRENT_SET qui ajoute les étiquettes lues au cours d'un cycle, l'ADDITION_SET qui ajoute au rapport les nouvelles étiquettes par rapport à la lecture du cycle précédent, et le DELETION_SET qui correspond aux étiquettes qui ont « disparu » par rapport à la lecture du cycle précédent.

4. Filtrage des étiquettes. On peut spécifier des filtres par rapport aux étiquettes à ajouter dans le rapport.
5. Génération du document qui normalement se fait dans un format XML. Cependant, des applications clientes peuvent choisir d'autres formats de codage pour le rapport.
6. Envoi des documents aux clients intéressés.

A partir de l'étude de la problématique d'intégration et de remontée de données, ainsi que de la spécification ALE fourni par EPCGlobal, nous avons identifié les composants de médiation CILIA suivants :

Composant	**Fonctionne Principale**	**Constituants**
Adaptateurs de lecteurs RFID	Ils servent à l'obtention de données brutes. Chaque adaptateur est chargé de la gestion du cycle de lecture (Read Cycle).	*Collector* technique de connectivité
Adaptateurs de capteurs	Ils servent à la connectivité vers les capteurs qui ne sont pas lecteurs RFID, comme un capteur de température par exemple.	*Collector* technique de connectivité
EventCycle	Il gère les cycles d'événements.	*Scheduler* périodique. *Processor* d'agrégation de lectures. *Dispatcher* multicast
Détection de doublons	Il est chargé de l'élimination de doublons	*Scheduler* immédiate. *Processor* d'élimination de doublons. *Dispatcher* multicast
Addition_Set	Il permet la génération de rapports contenant les nouvelles étiquettes par rapport à la lecture du cycle précédent.	*Scheduler* immédiate. *Processor* de génération de l'ensemble. *Dispatcher* multicast
Deletion_set	Il permet la génération de rapports qui contiennent les étiquettes qui ont « disparu » par rapport à la lecture du cycle précédent.	*Scheduler* immédiate. *Processor* de génération de l'ensemble. *Dispatcher* multicast
Filtrer	Il est chargé de filtrer les étiquettes selon un patron donné.	Scheduler immédiate. *Processor* de filtrage. *Dispatcher* multicast
Report_XML	Il est chargé de la transformation du rapport en XML.	*Scheduler* immédiate. *Processor* de transformation *Dispatcher* multicast
Adaptateur HTTP d'envoi	Il livre les rapports générés aux applications clients.	Sender http

TABLE 7.1 – Composants identifiés pour le middleware RFID

La Table 7.1 détaille les médiateurs et les adaptateurs développés pour construire des chaînes de médiation qui permettent de produire des rapports basés sur la spécification ALE. Nous pouvons constater que les médiateurs réutilisent des **schedulers** et **dispatchers** fournis par le framework. Ainsi, le concepteur se concentre sur des aspects métiers des **processors**.

Cette solution nous permet de répondre au traitement identifié précédemment et de fournir les rapports spécifiés par le serveur ALE. Néanmoins, il y a deux différences notables. Premièrement, un serveur ALE est configuré par un langage de spécification de rapport, avec l'implémentation présenté, nous sommes capables d'offrir la même fonctionnalité mais en décrivant une chaîne de médiation. Deuxièmement, la spécification ALE présente une API pour gérer l'abonnement de clients. Nous n'avons pas cependant implanté cette API car elle ne fait pas partie de nos objectifs.

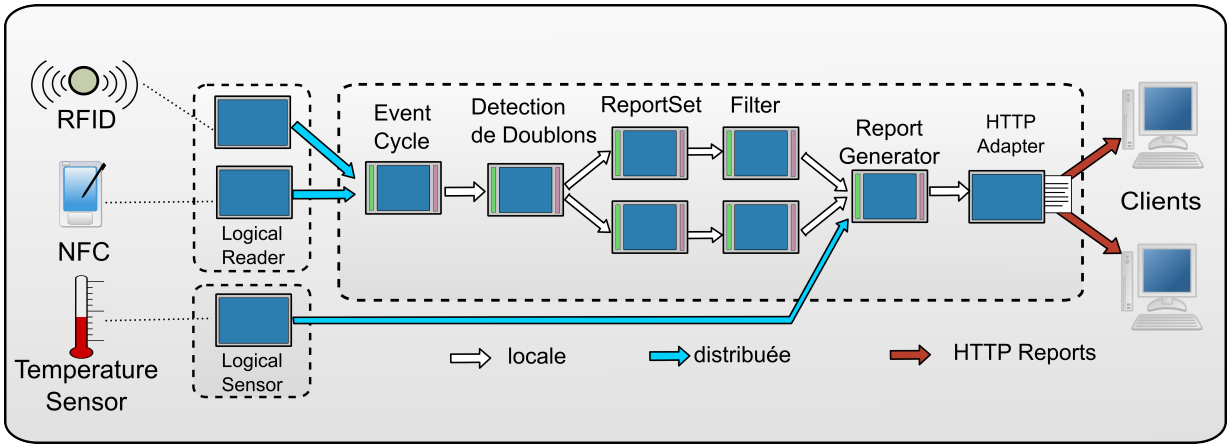

FIGURE 7.1 – Chaîne de médiation de Middleware RFID

La Figure 7.1 présente une chaîne de médiation qui répond à la génération de rapports basés sur ALE. Cette chaîne de médiation présente aussi une entité capteur qui permet d'enrichir les rapports générés. La description de cette chaîne de médiation est faite de la manière suivante :

```
<cilia>
    <chain id="AspireDemoChain">
        <mediators>
            <mediator-instance type="aspire-event-cycle" id="event-cycle">
                <scheduler>
                    <property name="delay" value="500" />
                    <property name="period" value="5000" />
                </scheduler>
            </mediator-instance>
            <mediator-instance type="aspire-duplicate" id="duplicate" />
            <mediator-instance type="aspire-addition-set" id="addition-set-1">
                <scheduler>
                    <property name="timeout" value="5000" />
                    <property name="initseq" value="1" />
                    <property name="seqfield" value="round" />
                </scheduler>
            </mediator-instance>
            <mediator-instance type="aspire-filter" id="filter" />
            <mediator-instance type="aspire-xml-generator" id="xml-generator">
                <scheduler>
                    <property name="correlation" value="($round)" />
                </scheduler>
            </mediator-instance>
        </mediators>
        <adapters>
            <adapter-instance type="aspire-logical-reader" id="logical-reader-2">
                <property name="delay" value="1000" />
                <property name="period" value="1000" />
                <property name="requires.from">
                    <property name="reader-id" value="tikitag-reader-1" />
                </property>
            </adapter-instance>
            <adapter-instance type="http-adapter" id="notifier"/>
        </adapters>
        <bindings>
            <bind from="logical-reader-2:end" to="event-cycle:begin" />
            <bind from="event-cycle:end" to="duplicate:begin" />
            <bind from="duplicate:end" to="addition-set-1:begin" />
            <bind from="addition-set-1:end" to="filter:begin" />
            <bind from="filter:end" to="xml-generator:begin" />
            <bind from="xml-generator:end" to="notifier:begin" />
        </bindings>
    </chain>
</cilia>
```

2.3 Evaluation

L'utilisation de CILIA pour cette application apporte plusieurs avantages :

- L'extensibilité permet d'étendre les fonctions disponibles. Ainsi, il est possible d'ajouter de nouveaux lecteurs physiques et logiques ou d'ajouter des entités de génération de rapport. Par exemple, le même rapport peut être livré dans un format différent comme HTML, PDF ou texte non structuré.
- L'évolution permet d'ajouter de nouvelles fonctions. Ainsi, il est possible d'ajouter de nouveaux algorithmes de génération de rapports plus complexes et plus riches, tout en garantissant que les composants actuels ne sont pas modifiés. De plus, il est possible d'ajouter d'autres entités de collecte d'information qui ne sont pas nécessairement RFID. Par exemple, des capteurs de température, d'humidité, ou de géolocalisation...

– La distribution transparente. Les composants produits sont indépendants du protocole de communication. Il est donc possible de construire une application de médiation distribuée en utilisant des connecteurs externes. Par exemple, nous pouvons déployer des lecteurs logiques de façon distribuée et c'est une infrastructure centralisée qui assure les opérations de filtrage.
– L'embarquabilité. Le framework CILIA peut être exécuté sur des dispositifs embarqués. Ainsi, le service ALE peut être un système embarqué. Nous avons de fait déployé la chaîne de médiation dans un **téléphone mobile Android** [3]. Ainsi, il est possible d'embarquer la chaîne de médiation dans un autre dispositif embarqué comme par exemple dans un moyen de transport (un camion, un bateau, ...).

3 Projet MEDICAL

Le maintien à domicile est un service souhaité dans la plupart des pays développés. En effet, la population de ces pays vieillit : en France, 16 % de la population a plus de 65 ans et 8 % a plus de 65 ans, ce qui représente aujourd'hui 4,9 millions de personnes. Les estimations prévoient que ce chiffre sera porté à 10,9 millions de personnes en 2050. De façon parallèle, le nombre de personnes en situation d'isolement et/ou en situation de dépendance augmente. Pour des raisons économiques et humaines, les collectivités souhaitent encourager ce maintien à domicile et maintenir l'autonomie des personnes. Pour cela, il est intéressant d'aménager les lieux de vie et de proposer des services innovants liés à la santé en tirant profit des nouvelles technologies de l'information.

Néanmoins, l'introduction de tels services soulève de nombreux défis liés à la technologie d'automatisation mais aussi aux coûts de la mise en œuvre et de la maintenance, voir des évolutions futures. Cependant les avancements technologiques et la maison numérique permettent de plus en plus d'envisager des environnements qui permettent de fournir un support pour ces services.

Les principaux défis techniques sont dus à l'hétérogénéité et à la volatilité de dispositifs, au dynamisme des usagers ainsi qu'à la gestion autonomique des d'applications. Celles-ci ne peuvent pas en effet être gérées par les habitants qui ne sont pas des experts techniques.

Le projet MEDICAL [4] (Middleware Embarqué D'Intégration de Capteurs et d'Applications pour les services à L'habitat) de Minalogic [5] vise la conception et le développement d'un middleware d'intégration pour les services numériques à domicile. MEDICAL se concentre notamment sur le domaine du maintien à domicile des personnes âgées ou souffrantes. Les principaux défis de ce middleware sont la capacité d'intégrer un environnement hétérogène et changeant ainsi que permettre des adaptations futures tant en assurant ses fonctions de façon continue.

CILIA est le middleware de base utilisé dans ce projet. Dans la suite, nous détaillons un cas

3. Testé sur un HTC Hero : processor 528Mhz, 288Mo RAM
4. http ://medical.imag.fr/
5. http ://www.minalogic.org/

d'usage pour le projet MEDICAL où l'utilisation de CILIA a permis de construire une application de médiation pour le suivi des habitudes des habitants d'une maison équipée. Cette application est également appelée actimétrie.

3.1 Besoins du cas d'usage

Dans le cadre du projet MEDICAL, une chaîne de médiation a été développée pour fournir un module d'actimétrie chez l'habitant. L'application d'actimétrie vise l'obtention de mesures chez l'habitant de façon non intrusive. Ensuite, l'information obtenue est analysée afin de détecter des comportements inhabituels qui peuvent correspondre à une perte d'autonomie.

L'analyse des mesures, ainsi que la consultation de l'analyse sont faites depuis un module dans une infrastructure IT. Néanmoins, les mesures sont obtenues et agrégées chez l'habitant. Dans ce contexte, l'application développée vise l'obtention de mesures et les envoie à l'infrastructure IT. En bref, l'application est distribuée comme présenté dans la Figure 7.2.

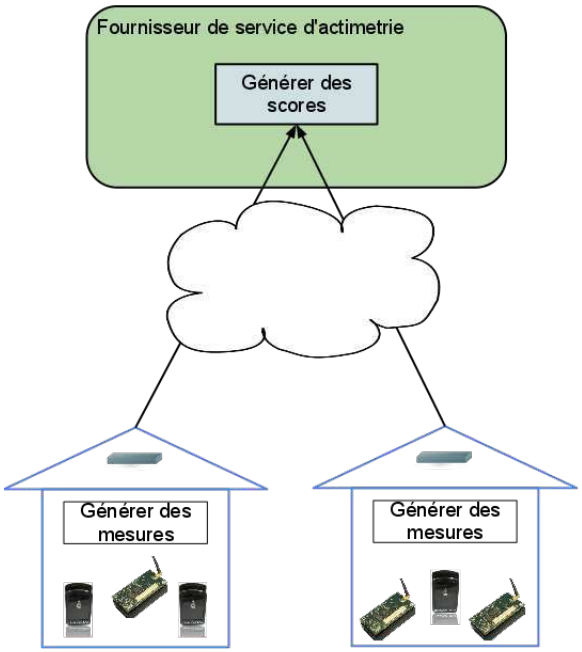

FIGURE 7.2 – Suivi des habitudes

Dans le cadre du projet MEDICAL, il est intéressant d'offrir une infrastructure applicative chez le patient. Cette infrastructure est composée de divers capteurs hétérogènes chargés de produire des mesures qui permettent de déduire le taux d'occupation des pièces. Les informations obtenues sont agrégées pour être envoyées à l'infrastructure IT chez le fournisseur du service d'actimétrie. Cependant, un traitement important doit s'opérer chez les résidents. Voici la liste d'opérations à appliquer aux données avant qu'elles soient envoyées au service d'actimétrie :

– filtrage : les mesures sont filtrées et les mesures inférieures à un certain seuil sont éliminées, ainsi que les lectures « fausses positives ».

– enrichissement : les mesures sont enrichies avec l'identifiant du résident.
– transformation : les données sont transformées pour qu'elles soient conformes au module de génération de scores d'actimétrie.

Une fois les données prêtes à l'envoi, elles sont envoyées au service distant qui est localisé dans une infrastructure IT chez le fournisseur de services.

Cette problématique d'intégration (des capteurs vers les services applicatifs) ne représente pas un défi algorithmique. Néanmoins il y a des aspects à considérer pour offrir un service intégré compétitif et profitable :

– il doit être conçu pour gérer l'hétérogénéité de dispositifs capteurs,
– il doit s'adapter aux besoins de distribution spécifiques par chaque résident,
– il doit être capable d'interagir avec des « boxes » existantes,
– les capteurs peuvent être partagés entre applications fournies par des tiers,
– il doit être embarqué dans dispositifs de type « box ».

L'application de médiation est chargée d'aborder l'hétérogénéité de capteurs et d'assurer la remontée des donnée de mesures depuis le résident jusqu'à l'infrastructure IT. Elle est aussi chargée d'augmenter le niveau d'abstraction des données brutes obtenues. Mais le plus important est qu'elle doit prendre en considération les besoins présentés ci-dessus

3.2 Solution

Nous modélisons le traitement des lectures de la façon suivante :

– obtention des données à partir des capteurs (dynamiques et hétérogènes),
– élimination des lectures indésirables, comme les « faux positifs » celles qui sont inferieures à un certain seuil,
– ajout de méta-information, principalement des informations métier, comme l'identifiant du résident,
– transformation vers un format de donnée spécifié par le module d'actimétrie,
– et finalement envoi vers l'application d'actimétrie.

Ce traitement nous amène à la spécification des composants suivants :

Composant	Fonctionne Principale	Constituants
Adaptateurs de capteurs	Ils servent à la détection de taux d'occupation des pièces	Collector technique de connectivité
Médiateur de filtrage	Il sert à détecter des lectures fausses positives	*Scheduler* immédiate. *Processor* de filtrage. *Dispatcher* multicast.
Médiateur d'enrichissement	Il ajoute des méta-informations aux données, notamment l'information du patient.	*Scheduler* périodique. *Processor* d'enrichissement. *Dispatcher* multicast.
Médiateur de transformation	Il est chargé de transformer les lectures au format désiré par le service d'actimétrie	Scheduler immédiate. *Processor* de transformation. *Dispatcher* multicast.
Adaptateur de service d'actimétrie	Il communique avec le service d'actimétrie pour lui envoyer des lectures.	Collector technique de connectivité

TABLE 7.2 – Composants identifiés pour suivre des habitudes

La Table 7.2 présente la liste des composants de médiation identifiés qui constituent une chaîne de médiation qui répond aux besoins spécifiés précédemment (voir Figure 7.3). Nous pouvons constater qu'à part le code de connectivité des connecteurs, les seules entités à implanter sont les « *processors* », ce qui est dû au fait qu'il n'y a pas besoin d'une grande complexité algorithmique ni pour le « *dispatcher* », ni pour le « *scheduler* ».

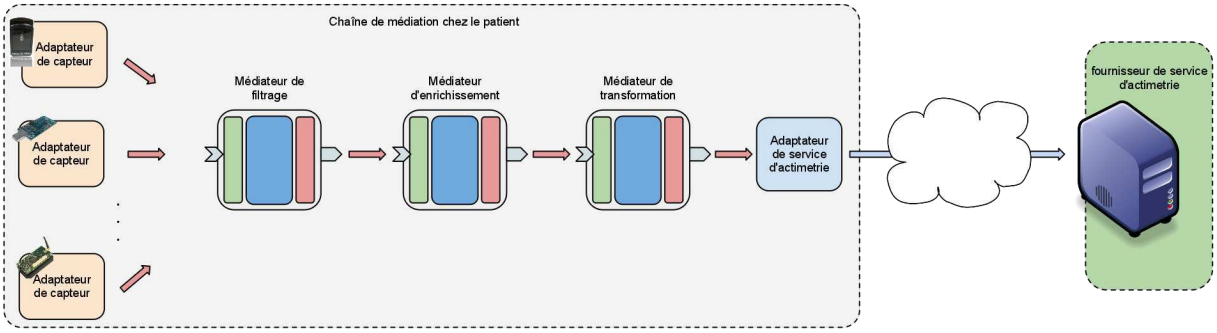

FIGURE 7.3 – Chaîne de médiation de suivi des habitudes

Cette chaîne de médiation répond partiellement les besoins mentionnés. En fait, l'application de suivi des habitudes est composée d'autres éléments combinés avec CILIA pour aborder des détails comme la découverte automatique de dispositifs, ainsi que la réification de dispositifs comme des services OSGi. Ces capacités sont obtenues grâce à une brique logicielle appelée RoSe [6] chargée de la gestion dynamique de services logiciels. La description de cette chaîne de médiation est faite de la manière suivante :

6. http ://wiki.chameleon.ow2.org/xwiki/bin/view/Main/Rose

```xml
<?xml version="1.0" encoding="UTF-8"?>
<cilia>
        <chain id="generator-mesures" type="generator-mesures">
                <!-- Liste d'adaptateurs-->
                <adapters>
                        <adapter-instance id="presence-collector" type="PresenceDetectorAdapter" />

                        <adapter-instance id="event-webservice" type="EventServiceAdapter" />
                </adapters>
                <!-- Liste de médiateurs-->
                <mediators>

                        <mediator-instance id="filter" type="MeasureFilterMediator" />

                        <mediator-instance id="enricher" type="MeasureEnricherMediator" />

                        <mediator-instance id="transformer" type="MeasureTransformerMediator" >
                                <processor>
                                        <property name="user" value="Aurelie"/>
                                </processor>
                        </mediator-instance>

                </mediators>
                <!--Liaisons-->
                <bindings>
                        <binding from="presence-collector" to="filter:in" />
                        <binding from="filter:out" to="enricher:in" />
                        <binding from="enricher:out" to="transformer:in" />
                        <binding from="transformer:out" to="event-webservice" />
                </bindings>
        </chain>
</cilia>
```

3.3 Evaluation

L'utilisation de CILIA a permis de construire une solution de suivi des habitudes qui prend en considération le dynamisme et l'hétérogénéité de l'environnement, tout en facilitant les évolutions futures (grâce notamment à la modularité).

Ainsi, nous pouvons synthétiser les avantages apportés par CILIA de la manière suivante :

- Modularité. CILIA encourage le découplage de l'application en diverses sous-tâches bien définies. Cela permet de construire des solutions de remontée de données modulaires avec une bonne séparation de préoccupations.
- Extensibilité. CILIA permet l'usage relativement aisé de capteurs hétérogènes sans affecter la logique de l'application. Il est possible d'ajouter d'autres adaptateurs de capteurs à la chaîne de médiation et, si nécessaire, il est possible d'ajouter des médiateurs afin par exemple de normaliser les messages transmis.
- Distribution. La fonctionnalité de la chaîne de médiation peut être distribuée. En fait, la connectivité vers des capteurs (les adaptateurs) est localisée dans une « box » spécial qui contient des capacités de connexion suivant plusieurs protocoles, comme X10, ZigBee, Bluetooth, etc.
- Evolutif. Un aspect souhaité était la diminution des coûts d'évolution. Une architecture modulaire avec une bonne séparation de préoccupations minimise l'impact en cas des modifications futures. Ainsi, l'approche obtenue nous permet de construire et d'ajouter des nouvelles entités de connectivité, mais aussi de médiation pour répondre aux nouveaux be-

soins. Comme par exemple l'ajout de nouveaux médiateurs avec des algorithmes différents qui calculent le taux d'occupation accompagnés des dispositifs capteurs spéciaux, tout en laissant les capteurs et la fonctionnalité existante.

– Embarquabilité. Le framework CILIA peut être exécuté sur des dispositifs embarqués. Ainsi, le module de génération de mesures chez le patient peut être un système embarqué. Nous avons déployé la chaîne de médiation dans un dispositif de type « plug computer »[7].

4 Intégration de systèmes d'information patrimoniaux (Orange)

Les solutions d'intégration ont au départ été conçues pour résoudre des problèmes d'entreprise, notamment pour les systèmes d'information. Comme indiqué dans les premiers chapitres de ce manuscrit, les systèmes étaient plutôt statiques avec une fréquence d'évolution relativement faible. Depuis quelques années, néanmoins, le rythme des avancées technologiques est tel qu'il est préférable que les systèmes d'information ainsi que les systèmes intégrées soient conçus en considérant des évolutions futures et ceci en s'appuyant sur des principes de génie logiciel. Rien n'assure évidemment une évolution forcément facile, mais cela permet de minimiser les couts de maintenance et d'évolution dans de nombreux cas.

Dans le cadre d'un workshop élaboré par l'équipe ADELE dans les installations d'Orange Labs à Meylan, nous avons fait développer à des ingénieurs d'Orange un cas d'usage portant sur l'intégration d'applications patrimoniales d'applications d'Orange[8]. L'intérêt du workshop été double. Premièrement pour montrer les besoins actuels lors de l'intégration d'applications dans des nouveaux domaines d'application. Et deuxièmement pour montrer l'utilisation de CILIA dans un cas d'utilisation d'un système d'information. Les résultats du workshop ont été très satisfaisants, principalement pour réaffirmer des collaborations de recherche autour de CILIA. Dans la suite nous détaillons le cas d'usage d'un système d'information intégré.

4.1 Besoins du cas d'usage

Le cas d'usage proposé par Orange est le suivant. Un opérateur Telecom souhaite développé et mettre à disposition de ses abonnés un nouveau service applicatif. Ce service permet aux clients de visualiser la consommation des forfaits auxquels il est abonné. Les services intégrés que l'entreprise fournit actuellement sont au nombre de trois : le service de téléphonie fixe, le service de téléphonie mobile, et le service d'internet (voir Figure 7.4). Cependant, l'application doit être facilement extensible afin d'agréger d'autres services que l'entreprise mettra à disposition dans l'avenir.

Cette application met en évidence un besoin important lorsque l'on souhaite intégrer des systèmes hétérogènes : la capacité d'extension. Cela s'obtient principalement grâce à une modularité bien identifiée, à une bonne séparation de préoccupation, et en garantissant le plus possible un

7. http ://www.ionics-ems.com/plugcomputer.html
8. http ://wikiadele.imag.fr/index.php/Cilia/Workshop

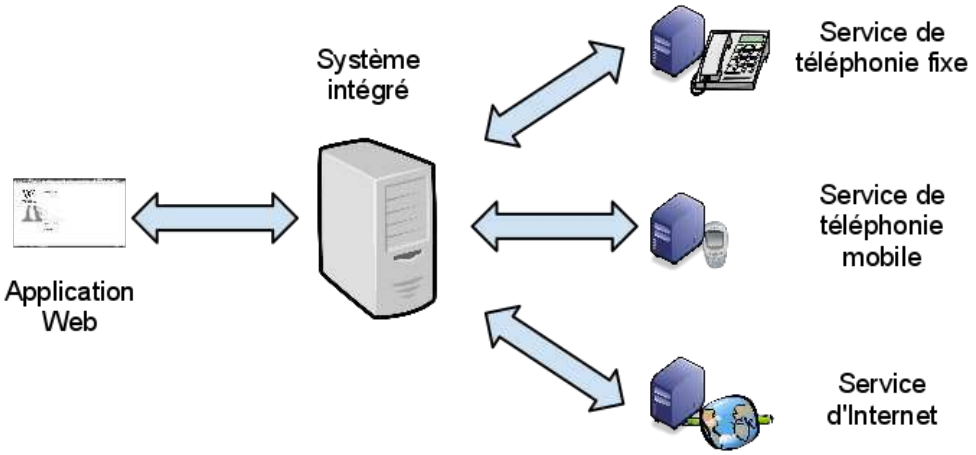

FIGURE 7.4 – Architecture d'interaction de systèmes intégrés

faible couplage entre les applications à intégrer et les opérations d'intégration.

Les besoins précis de cette application sont :

- une application « front-end » qui permet l'interaction du client en utilisant une interface Web,
- le client s'identifie et indique la liste de services qu'il veut visualiser,
- l'information est transmise aux systèmes (back-ends) choisis pour calculer les consommations actuelles,
- le résultat intégré est visualisé dans le front-end.

Les contraintes techniques suivantes doivent également être prises en compte :

- l'application front-end (Application Web) manipule un seul format de donnée (basée en XML),
- chaque application Back-end (BE) du service de mobile, de fixe et d'Internet manipule un format de données différent.

4.2 Solution

Nous modélisons le traitement des lectures de la façon suivante :

1. Le front-end envoie le message au système d'intégration.
2. Le message est transformé dans le but de préparer le message avant de réaliser le découpage.
3. Le message est découpé en sous-messages à destination des back-end
4. Les sous messages sont routés en fonction du contenu.
5. Le contenu de chaque sous-message est transformé (pour chaque BE)
6. Communication avec le service BE (pour chaque BE)
7. Regroupement des réponses associées à la requête

8. Création d'une réponse unique qui contient la consommation pour chaque forfait et le cumul des consommations.

Ainsi donc, nous avons identifié les composants de médiation suivants :

Composant	Fonctionne Principale	Constituants
Adaptateurs d'aller/retour	Il sert à interagir avec le Front-End, il reçoit la requête du front-end et il envoie le message au premier médiateur, puis il reste en attente du résultat.	
Médiateur de transformation	Il transforme le message en un message normalisé, qui sera découplé dans un traitement futur.	*Scheduler* immédiate. *Processor* de transformation. *Dispatcher* multicast.
Médiateur XML Splitter	il découple le message en trois, et chaque partie est livrée selon le contenu	*Scheduler* immédiate. *Processor* XPATH Splitter. *Dispatcher* content-based dispatcher
Médiateurs de transformation (un pour chaque back-end)	Il est chargé de transformer le message pour qu'il soit conforme au format utilisé par le back-end	*Scheduler* immédiate. *Processor* de transformation. *Dispatcher* multicast.
Médiateur de back-end	C'est un médiateur qui envoie la requête au back-end et attend la réponse.	*Scheduler* immédiate. *Processor* d'interaction avec le back-end. *Dispatcher* multicast.
Médiateur d'agrégation	Il reste en attente des messages obtenus par les back-end et il fait une agrégation.	*Scheduler* de corrélation. *Processor* d'agrégation. *Dispatcher* multicast

TABLE 7.3 – composants de médiation de système intégré

La Table 7.3 composants de médiation de système intégréeprésente la liste des composants de médiation développés pour construire une chaîne de médiation répondant aux besoins d'intégration précédemment spécifiés. Cette solution d'intégration représente bien un patron de type « split/aggregate ».

La plupart des constituants des médiateurs sont génériques. Grâce à l'usage de messages en XML ainsi qu'à l'utilisation de technologies de manipulation de documents XML, les « *processors* » de transformation sont génériques. Ils utilisent comme paramétrage un fichier XSLT qui spécifie les règles de transformation. Ainsi, le « *content-based dispatcher* » est basé en XPATH pour identifier des nœuds XML lors de l'analyse du contenu. De même le « *spliter* », reçoit comme paramètre un patron XPATH pour découpler le message XML. Ainsi, les éléments qui restent applicatifs et qui ne sont pas génériques sont l'adaptateur d'entrée et les médiateurs qui interagissent avec les Back-ends.

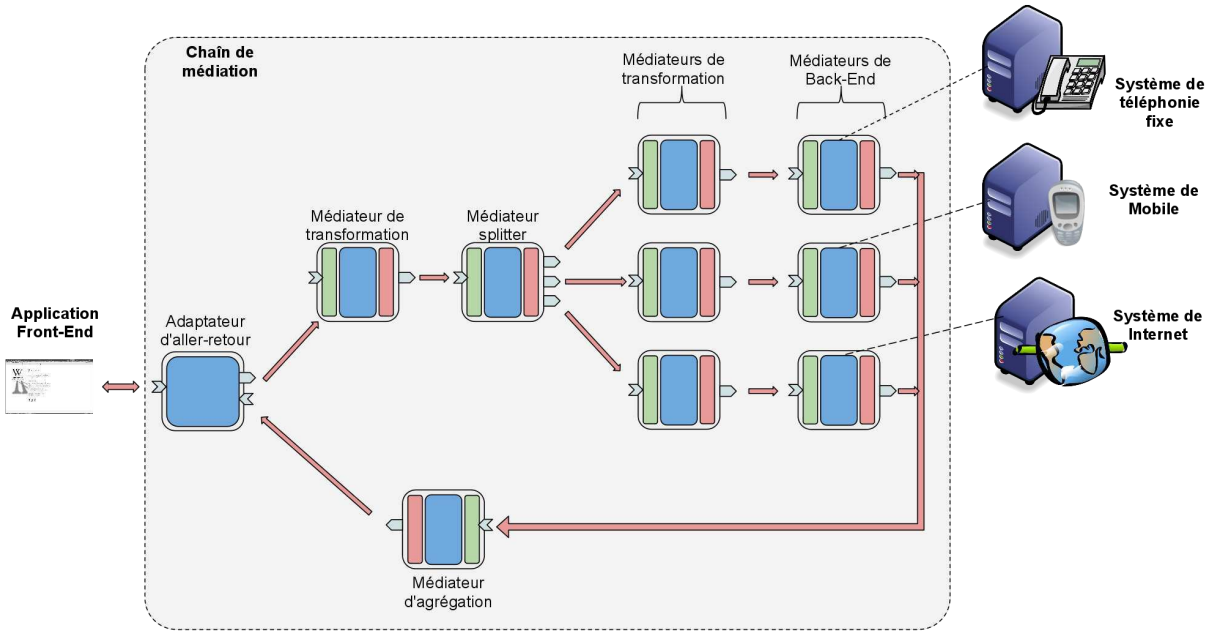

FIGURE 7.5 – Chaîne de médiation d'intégration de service de téléphonie fixe, de téléphonie mobile et d'internet

La Figure 7.5 décrit l'architecture de la chaîne de médiation qui répond à l'application d'intégration. La solution que nous avons présentée réifie les systèmes back-ends comme des services OSGi (en utilisant le patron *proxy*). Cette réification est faite en utilisant le framework RoSe. Ainsi, les processors de back-ends ont une dépendance vers un service OSGi pour interagir vers ces services distants.

La description de cette chaîne de médiation est faite de la manière suivante :

```
<cilia>
    <chain type="split-aggregate" id="DemoSplitAggregate">
        <mediators>
            <mediator-instance type="XsltTransformer" id="xslt-transformer-1" >
                <processor>
                    <property name="xslt-file" value="trans.xslt" />
                </processor>
            </mediator-instance>
            <mediator-instance type="XmlSplitterWithCorrelation" id="xml-splitter-1">
                <processor>
                    <property name="separator" value="//suiviconso-requete" />
                </processor>
                <dispatcher>
                <property name="language" value="xpath" />
                <property name="conditions">
                    <item key="//suiviconso-requete[@produit='fixe']"
                        value="fixe" />
                    <item key="//suiviconso-requete[@produit='mobile']"
                        value="mobile" />
                    <item key="//suiviconso-requete[@produit='internet']"
                        value="internet" />
                </property>
                </dispatcher>
            </mediator-instance>
            <mediator-instance type="Translator" id="translator-1" >
                <processor>
                    <property name="dictionary">
                        <item key="CLIENT-ID" value="CLIENT-IDENT"/>
                        <item key="SuiviConso" value="SuiviConsoInternet"/>
                    </property>
                </processor>
            </mediator-instance>
            <mediator-instance type="Translator" id="translator-2" >
                <processor>
                    <property name="dictionary">
                        <item key="SuiviConso" value="SuiviConsoFixe"/>
                    </property>
                </processor>
            </mediator-instance>
            <mediator-instance type="Translator" id="translator-3" >
                <processor>
                    <property name="dictionary">
                        <item key="SuiviConso" value="SuiviConsoMobile"/>
                    </property>
                </processor>
            </mediator-instance>
            <mediator-instance type="XsltTransformer" id="xslt-transformer-2" >
                <processor>
                    <property name="xslt-file" value="delproduitinternet.xslt" />
                </processor>
            </mediator-instance>
            <mediator-instance type="XsltTransformer" id="xslt-transformer-3" >
                <processor>
                    <property name="xslt-file" value="delproduitfixe.xslt" />
                </processor>
            </mediator-instance>
            <mediator-instance type="XsltTransformer" id="xslt-transformer-4" >
                <processor>
                    <property name="xslt-file" value="delproduitmobile.xslt" />
                </processor>
            </mediator-instance>
```

```
            <mediator-instance type="FacturationMobile" id="facturation-mobile-1" />
            <mediator-instance type="FacturationInternet" id="facturation-internet-1" />
            <mediator-instance type="FacturationFixe" id="facturation-fixe-1" />
            <mediator-instance type="UC1Aggregator" id="uc1-aggregator-1" />
        </mediators>
        <adapters>
            <adapter-instance type="SplitAggregateEndpoint" id="adapter"/>
        </adapters>
        <bindings>
            <binding from="adapter" to="xslt-transformer-1" />
            <binding from="xslt-transformer-1:to-splitter" to="xml-splitter-1:from-trans" />
            <binding from="xml-splitter-1:mobile" to="translator-3" />
            <binding from="xml-splitter-1:internet" to="translator-1" />
            <binding from="xml-splitter-1:fixe" to="translator-2" />
            <binding from="translator-1" to="xslt-transformer-2" />
            <binding from="translator-2" to="xslt-transformer-3" />
            <binding from="translator-3" to="xslt-transformer-4" />
            <binding from="xslt-transformer-2" to="facturation-internet-1" />
            <binding from="xslt-transformer-3" to="facturation-fixe-1" />
            <binding from="xslt-transformer-4" to="facturation-mobile-1" />
            <binding from="facturation-mobile-1" to="uc1-aggregator-1" />
            <binding from="facturation-internet-1" to="uc1-aggregator-1" />
            <binding from="facturation-fixe-1" to="uc1-aggregator-1" />
            <binding from="uc1-aggregator-1" to="adapter" />
        </bindings>
    </chain>
</cilia>
```

4.3 Evaluation

Le dynamisme est un aspect peu probable dans les systèmes d'information. Néanmoins, la préparation des évolutions est désormais un aspect important mais toujours difficile à réaliser. Une bonne approche est de s'appuyer sur les principes de génie logiciel qui encouragent la construction de systèmes modulaires, avec un faible couplage et une bonne séparation de préoccupations. Ces principes sont les piliers de CILIA.

Les avantages principaux, en plus de ceux présentés dans les projets précédents, sont :

- Généralité. La généralisation de composants et de constituants de médiation permet la réutilisation. La plupart de constituants dans cette application sont entièrement paramétrables et utilisables dans différents contextes.
- Anticipation aux évolutions. La généralité, associée à la séparation de préoccupations et la modularité, minimise les coûts des évolutions futures.

5 Evaluation de l'environnement d'exécution

Dans la section précédente, nous avons montré l'utilisation réussie du framework CILIA dans des projets nationaux et Européens. Dans cette section, nous fournissons certaines évaluations quantitatives du code produit ainsi que du framework en exécution.

5.1 Qualité de code produit

Le projet CILIA est divisé en divers artefacts appelées *bundles*. Comme nous l'avons mentionné, un bundle est un JAR (Java ARchive) spécialisé avec des méta-informations spécifiques à l'exécution dans une passerelle OSGi. Chaque bundle contient ainsi du code Java ainsi que des descriptions XML.

Le framework CILIA est constitué principalement de trois artefacts, c'est-à-dire trois bundles. Ainsi, ces bundles peuvent évoluer indépendamment sans impacter le reste. Ces artefacts sont les suivants :

- Cilia-core fournit les classes Java qui permettent la définition d'applications de médiation CILIA, dites chaînes de médiation,
- Cilia-ipojo-runtime contient le runtime du framework pour des applications de médiation CILIA basée sur iPOJO,
- Cilia-ipojo-compendium contient la définition de constituants de base fournis par CILIA, par exemple des schedulers périodiques, des dispatchers basés sur le contenu, etc.

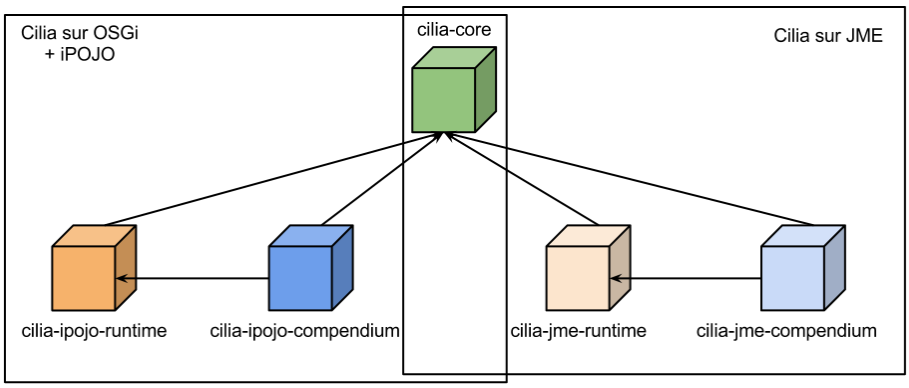

FIGURE 7.6 – Relation de bundles de CILIA

Ces trois artefacts représentent le minimum indispensable pour construire des applications basées sur CILIA. Il y a cependant d'autres artefacts importants, liés notamment à l'outillage, qui permettent, par exemple, la gestion et la reconfiguration de chaînes de médiation. Néanmoins, ces artefacts ne sont pas analysés en détail car ils ne caractérisent pas l'approche proposée, mais plutôt l'implémentation de référence de CILIA.

Dans la Figure 7.6, nous montrons la relation entre ces trois artefacts. Ainsi, nous pouvons voir l'intérêt de séparer l'artefact cilia-core. Cela nous permettra, en particulier, de développer

Bundle\Aspect	Lignes de code Java	Lignes de code XML	Nombre de classes	Nombre de packages	Ratio code/ commentaires	Code dupliqué	Pourcentage de conformité de Sonar
CILIA-core	2583	76	65	9	30.5	0.9	80.8
CILIA-ipojo-runtime	7042	259	92	12	30.7	0.9	83.6
CILIA-ipojo-compendium	4070	269	49	10	28.4	1.3	81.2

TABLE 7.4 – Résultats de l'analyse de code en utilisant SONAR

aisément d'autres implantations de CILIA. C'est du reste ce que nous avons fait en portant CILIA sur des SunSPOT [9].

Nous avons analysé le code du framework CILIA avec la plateforme d'analyse de qualité de code Sonar [10]. Cette plateforme permet de réaliser une analyse statique du code, afin d'identifier des erreurs possibles ou de signaler des améliorations possibles du code. Parmi les aspects inspectés par Sonar, nous pouvons citer :

- L'analyse du style utilisé, par exemple au niveau du nommage de classes, des méthodes, des champs, etc.
- la détection des erreurs, comme des blocs vides liés à des conditions jamais réalisées, des boucles etc.
- la détection de code mort qui n'est pas utilisé,
- la détection de code dupliqué (notamment à l'aide de « copié/collé »),
- la détection de code non optimal, par exemple par l'usage de classes obsolètes ou par un mauvais usage des objets,
- la vérification du respect de règles basées sur PMD (*Programming Mistake Detector*).

La Table 7.4 montre des résultats de l'analyse statique du code fait par le framework SONAR. Les premiers aspects analysés correspondent à l'analyse statique du code (les lignes de code, le nombre de classes par bundle et le nombre de packages). Les trois dernières correspondent à une analyse qualitative du code. Ainsi, nous pouvons voir le ratio de code documenté, le pourcentage de code dupliqué, et finalement, le pourcentage de conformité selon la configuration de SONAR.

Un aspect important de ce travail, c'est que la proposition présentée est entièrement implémentée. De plus, les artefacts ont une bonne qualité. Ce qui a permis l'utilisation de CILIA dans des différents projets comme nous l'avons présenté précédemment.

Dans la suite, nous montrons des détails quantitatifs du framework d'exécution, tel que les temps pour réaliser des reconfigurations, ainsi que le temps ajouté par le framework (*overhead*) lors de la réalisation de tâches de médiation enchainées.

9. http ://wikiadele.imag.fr/index.php/Cilia/Cilia-SunSPOT_Developping_a_new_application
10. www.sonarsource.org/

5.2 Mesures de l'impact de reconfiguration

Les défis des nouveaux domaines d'application sont associés au dynamisme de l'environnement, qui impacte les applications, et dans certains cas, l'architecture des systèmes. Les systèmes doivent donc être souvent reconfigurés. Cette reconfiguration doit être faite à chaud, c'est-à-dire sans arrêter l'application.

Cette reconfiguration à chaud est une des caractéristiques qui démarquent CILIA par rapport aux solutions d'intégration/médiation existantes. La Table 7.5montre le temps minimal nécessaire lors de la réalisation d'une reconfiguration à chaud. Ces temps ont été calculés pour une chaîne de médiation où il y a une seule liaison d'entrée et une seule liaison de sortie par composants. Ce temps de réalisation de reconfiguration s'accroit naturellement lorsqu'on réalise des opérations de reconfiguration sur des composants ayant un grand nombre de liaisons.

Opération de reconfiguration	Temps (ms)
Création de médiateur	130
Création de liaisons	60
Destruction de liaison	50
Destruction de médiateur	115
Remplacer un médiateur	200
Reconfiguration de paramétrage	7
Cloner un médiateur	140

TABLE 7.5 – Temps par reconfiguration

5.3 Mesures de l'impact de l'exécution

Nous avons ensuite cherché à mesurer l'impact de CILIA sur l'exécution d'applications de médiation, c'est-à-dire l'« overhead » généré par le framework. L'impact que nous mesurons est lié au temps d'exécution des applications.

Le calcul effectué correspond à la proposition suivante. Une application d'intégration réalise n tâches P_n pour chaque message qui transite dans une chaîne de médiation et chaque tâche prend un temps d'exécution de $t(P_n)$. Ainsi :

$$Temps\,total = overhead(fwk) + \sum_{i=1}^{n} t(P_i)$$

Le temps total d'une exécution est égal au temps de réalisation de chaque tâche auquel s'ajoute l'« *overhead* » du framework. Néanmoins, cet « *overhead* » est fortement lié au nombre de tâches de médiation. Ceci est dû au fait que l'« *overhead* » correspond principalement au temps passé entre la finalisation d'une tâche P_i et l'initialisation de la tâche suivante P_{i+1}.

La Figure 7.7 présente de manière graphique l'architecture d'une chaîne de médiation et le temps d'exécution correspondant.

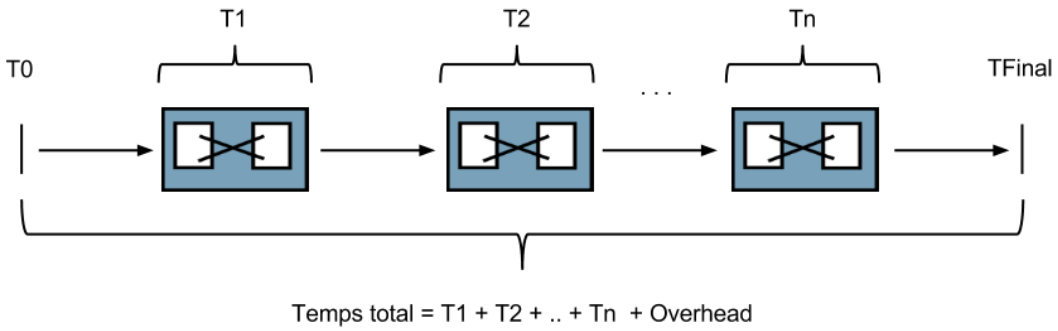

FIGURE 7.7 – Temps d'exécution d'une application de médiation

Nous fournissons, dans la suite, les détails des applications de médiation développées pour faire des mesures et des comparatifs de l'« *overhead* » généré par le framework de médiation.

5.3.1 Description des applications de test

Nous avons construit un banc d'essai qui nous permet de mesurer l'« *overhead* » de CILIA (Figure 7.8). Cette application est constituée :

- d'un composant injecteur de messages, appelé INJ
- d'un composant configurable, appelé PRC, qui prend comme paramètre une valeur X correspondant au temps d'exécution du composant.

Le composant d'injection (INJ) réalise l'algorithme suivant :

- Il génère un ensemble de messages à envoyer,
- ensuite, il envoie les messages. Ce composant peut injecter, selon sa configuration, des messages de façon asynchrones ou de façon synchrone,
- pour chaque message envoyé, il déclenche une minuterie,
- et finalement, il reste en attente des messages pour calculer le temps passé pour chaque message lors de son cheminement au travers de la chaîne.

Le composant PRC applique le processus suivant :

- il est notifié pour chaque message reçu,
- ensuite, selon sa configuration, il prend un temps d'attente afin de simuler le calcul en un temps défini,
- ensuite, la méthode appelée lors de la réception retourne le même message reçu.

Pour finaliser, l'architecture de l'application de médiation est constituée de composants PRC enchaînés.

Afin d'évaluer le framework CILIA, nous avons développé ce banc d'essai en utilisant en plus les deux frameworks suivants :

- Camel, le framework d'intégration de la forge Apache basé principalement sur des patrons d'intégration (voir chapitre 2),
- Spring Integration, le framework d'intégration basé sur le modèle de programmation Spring.

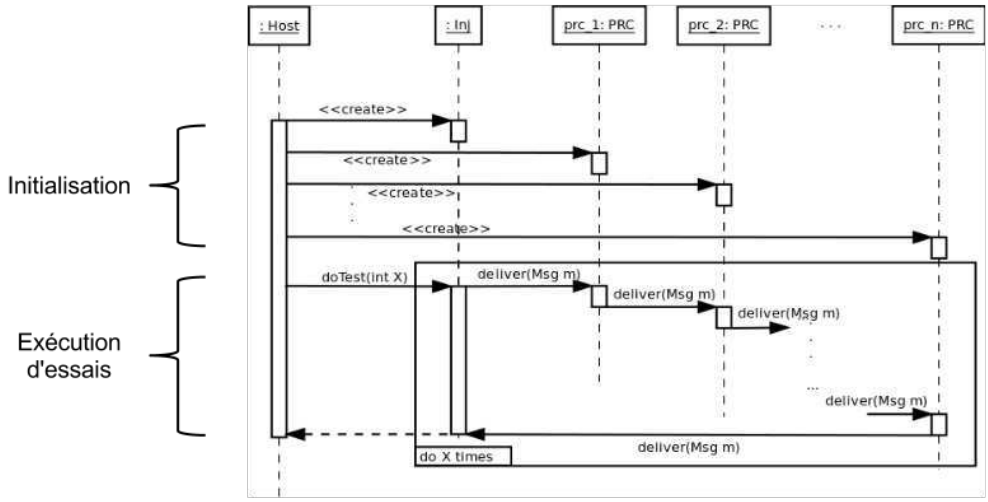

FIGURE 7.8 – Fonctionnement de banc d'essai

Nous avons choisi ces deux framework principalement car ils sont conçus pour être déployés dans des environnements dynamiques, comme c'est le cas d'une passerelle OSGi. Cependant, comme nous avons vu dans un chapitre précèdent de cette thèse, il y a une réelle faiblesse au niveau de la gestion du dynamisme dans ces frameworks. Cependant, ce manque de dynamisme n'est pas bloquant pour les tests présentés dans cette section.

Le banc d'essai est configuré avec 10 composants PRC et chacun prend un temps d'exécution de 10ms. Le nombre de composants, ainsi que le temps d'exécution choisi sont arbitraires, mais ils doivent être suffisamment grands pour identifier des différences.

5.4 Résultats

Le test a été effectué à dix reprises et chaque exécution envoie un total de 100 messages. Chaque message est étendu avec la date de son injection et, à sa sortie de la chaîne de médiation, avec le temps total de traversée.

La Figure 7.9 montre les résultats obtenus pour ces dix exécutions pour chaque framework de médiation (Camel, Spring Integration et CILIA). Chaque mesure présentée sur le graphique est la moyenne de ces dix exécutions.

Selon la configuration, le temps total d'exécution des dix composants PRC est de $100ms$. Néanmoins, chaque framework ajoute un overhead à l'exécution. En total l'overhead pour dix composants est inferieur à $2ms$ pour tous les frameworks. Cet overhead est négligeable pour ce type d'applications. En effet, le principal coût d'une application d'intégration est dû principalement aux systèmes tiers ou aux protocoles de communication, où il y a un accès intensif aux ressources.

Les résultats obtenus nous permettent de conclure que l'environnement d'exécution de CILIA, conçu depuis le début pour adresser des problèmes d'intégration dans des environnements dynamiques, se comporte de manière similaire aux frameworks d'intégration open source utilisés

dans l'industrie. Ceci réaffirme que les propriétés de modularité, ainsi que de reconfiguration au runtime, n'impactent pas les temps d'exécution des opérations de médiation.

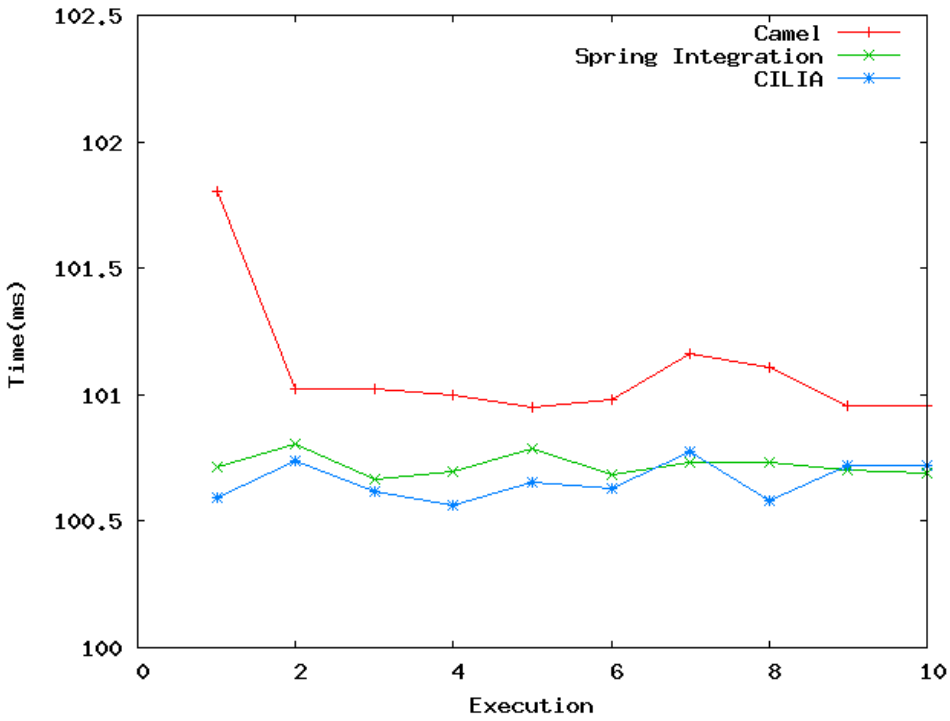

FIGURE 7.9 – Overhead de Camel, Spring Integration et CILIA

6 Synthèse

Dans ce chapitre nous avons proposé une validation qualitative du framework de médiation CILIA. En premier lieu, nous avons détaillé l'utilisation de CILIA dans divers projets nationaux et Européens. Le résultat obtenu lors de l'utilisation de CILIA est très satisfaisant. Certains participants ont utilisé CILIA et ont aussi contribué à l'amélioration en donnant des retours et des suggestions, comme l'ajout de certains composants de médiation et adaptateurs.

Concernant la qualité de code de CILIA, le framework est d'une bonne qualité, en considérant que ce projet est principalement un projet de recherche. L'outil SONAR a donné une bonne note, plus de 81% du code est conforme au style prédéfini. Cette qualité, ainsi que le fait que notre proposition a été entièrement implémentée, nous a permis de participer aux divers projets mentionnés précédemment.

Finalement, nous avons mis en place des essais du framework afin d'examiner son comportement et, plus précisément, son overhead, ainsi que des comparatifs avec d'autres frameworks d'intégration. Nous avons constaté que l'overhead généré par CILIA est négligeable, et qu'il y a un comportement similaire, voire meilleur, que les autres frameworks analysés. Néanmoins il est important de noter que les autres frameworks sont plus complexes, voire plus robustes, et qu'ils

fournissent divers options de configuration qui peuvent dégrader un peu la performance. En plus ils fournissent un large catalogue de composants prêts à l'usage. Cependant nous voulions montrer que CILIA est capable de fournir les mêmes capacités de base pour enchainer des composants afin de réaliser une solution d'intégration. Par ailleurs, CILIA fournit des capacités modulaires, dynamiques et de reconfiguration ce qui est sans aucun doute un manque dans les frameworks d'intégration existants.

8

Conclusions & Perspectives

Sommaire

1 **Conclusions** 194

2 **Perspectives de recherche** 197

3 **Perspectives de travail** 199

Dans les chapitres précédents, nous avons présenté CILIA un modèle à composants spécifique à la médiation et le framework d'exécution associé qui permet de réaliser des reconfigurations à l'exécution. CILIA est un modèle inspiré par des patrons d'intégration, ainsi, la structure des médiateurs permet de définir des composants qui répondent aux patrons d'intégration. CILIA est intégralement implémenté et il est actuellement utilisé comme brique logicielle dans des projets internes, mais aussi dans des projets nationaux et européens (MEDICAL, ASPIRE, OSAMI...). Dans ce chapitre nous faisons une synthèse des propositions faites dans cette thèse.

Nous sommes conscients qu'il reste des questions à résoudre. Dans ce chapitre nous détaillons des perspectives de recherche autour de CILIA. Notamment, nous avons identifié trois perspectives de recherche principales. Celles ci concerneront la définition de langages de haut niveau d'abstraction pour définir des applications de médiation, les reconfigurations automatiques, et finalement la réalisation de la validation des applications lors de la reconfiguration afin d'assurer le fonctionnement correct des applications. De plus, nous avons identifié des perspectives de travail technique autour de CILIA. Celles-ci concerneront l'implémentation d'outils de développement et l'implémentation d'outils de monitoring, et finalement, l'enrichissement de la base de composants prêts à l'usage.

1 Conclusions

Cette section met en avant les contributions de ce manuscrit. Elle résumera les différents points abordés tout au long de cette thèse.

1.1 Contexte

La problématique d'intégration n'est pas récente. Les industriels de l'informatique et la communauté scientifique a proposé depuis des années des solutions qui adressent principalement les problèmes récurrents dans les domaines des Systèmes d'Information. Ainsi, des solutions de type middleware ont été utilisé comme un moyen de communication permettant de découpler les diverses applications à intégrer. Et plus récemment, des solutions de type EAI (*Enterprise Application Integration*) et ESB (*Enterprise Service Bus*) qui se basent sur des EIP *(Enterprise Integration Patterns*), ces derniers correspondent à la connaissance acquis par des experts d'intégration dans ce domaine.

Parallèlement, la technologie évolue. Cette évolution implique qu'il est de plus en plus possible de fournir aux personnes des services numériques via une multitude d'équipements électroniques tels que des ordinateurs, des téléphones portables, des tablettes... Cela ouvre des possibilités au développement de services innovants, que nous appelons nouveaux domaines d'applications. Ces domaines sont aussi variés que la santé à domicile, la domotique, le divertissement, ou encore, le transport et la gestion de l'énergie.

La mise en place de ces nouveaux services numériques requière la création et la maintenance d'infrastructures d'intégration complexes. Il s'agit plus précisément d'intégrer des éléments physiques, souvent mobiles, et des Systèmes d'Information. Nous pouvons constater que les besoins d'intégration ont évolué. Au départ ils étaient uniquement requis par les industriels et maintenant ils sont devenus indispensables dès lors que l'on souhaite fournir des services plus riches aux personnes. De plus, les nouveaux systèmes doivent prendre en considération l'évolution continue [BN06] et le dynamisme des acteurs. Le développement d'applications n'est plus statique, et nous devons prendre des décisions de conception même à l'exécution.

1.2 Exigences

Les nouveaux domaines d'applications ont des caractéristiques particulières qui sont intrinsèques à ces domaines, ces caractéristiques sont principalement :

- Grande hétérogénéité : les équipements physiques viennent avec des protocoles de communication, souvent standards, mais aussi souvent *ad-hoc*. De plus, il y a souvent une hétérogénéité au niveau syntaxe et sémantique.
- Distribution : une application (dans ces nouveaux domaines) est souvent distribuée sur divers équipements, allant des équipements personnels, aux réseaux des capteurs, mais aussi des grands systèmes distants dans une Infrastructure IT.

- Mobilité : certains dispositifs sont mobiles, ainsi que les usagers de ces applications. Certaines applications doivent donc être conscientes de la mobilité des acteurs, et prendre des décisions prenant cet aspect en considération.
- Information contextuelle : certaines applications requièrent de l'information contextuelle pour prendre des actions. Cette information liée au contexte peut être obtenue par divers moyens, comme des dispositifs (souvent capteurs), mais elle peut aussi être une information issue de l'application elle-même.
- Une rapide évolution : la technologie avance rapidement. Nous sommes arrivés à un point tel que les systèmes logiciels doivent prendre en considération l'évolution de la technologie, ainsi que la rapide adoption de ces dispositifs par le grand public.

A ces caractéristiques s'ajoutent des défis technologiques importants aux systèmes tels que :

- Un grand besoin d'intégration : l'intégration n'est pas seulement un problème des systèmes d'information. Les nouveaux domaines d'application font face à des défis d'intégration importants. Cela afin de réussir l'interopérabilité entre d'un part la grande diversité de dispositifs existantes (capteurs, GPS. . .), et d'autre part les briques logicielles souvent embarquées dans d'autres dispositifs (téléphone portable, tablette, mini-PC,..), comme des briques logicielles dans les infrastructures IT.
- Gestion du dynamisme : le dynamisme est une caractéristique majeure de ces domaines. Les usagers, ainsi que certains dispositifs, sont mobiles, ce qui oblige les concepteurs d'applications à prendre en compte cette capacité.
- Gestion autonomique : la gestion autonomique est un des défis importants de nos jours. En fait, il s'agit d'incorporer la connaissance des experts au sein d'éléments logiciels chargés de gérer eux-mêmes les tâches d'administration. Ainsi, les tâches de configuration et paramétrage sont réalisées par cet élément logiciel. Ces capacités sont appliquées, par exemple, lorsque certaines informations de contexte évoluent ou changent au cours de temps.

On voit ici que la reconfiguration est un point important lorsqu'on souhaite construire des solutions d'intégration où le dynamisme est important ou lorsqu'on souhaite construire des applications qui seront automatiquement reconfigurées par des gestionnaires.

1.3 Contribution

La problématique qu'aborde cette thèse est la construction de solutions d'intégration qui soient flexibles et reconfigurables à l'exécution. Pour accomplir nos objectifs, nous proposons une solution qui :

- prend en considération les principes du génie logiciel (abstraction, modularité, séparation de préoccupations, anticipation aux évolutions . . .),
- fournit une approche de développement pour des applications d'intégration,
- fournit un environnement d'exécution qui permet de réaliser des reconfigurations structurelles à l'exécution.

Les principes de génie logiciel sont un point clé de l'approche. L'intérêt principal est d'être capable de réaliser des reconfigurations à l'exécution, cependant, c'est depuis la phase de conception que nous pouvons construire des solutions avec les caractéristiques indispensables qui permettront réaliser de telles reconfigurations.

Afin de remplir les exigences présentées dans ce travail, nous avons proposé un modèle à composant spécifique à la médiation appelé CILIA. Comme tous les modèles à composants, CILIA est composé d'un modèle des composants et d'un modèle d'assemblage, ainsi que d'un framework d'exécution qui permet l'exécution des composants. De plus, nous avons explicité le cycle de vie de développement et d'exécution des applications CILIA.

La définition de notre modèle de composants a été influencée par l'étude des EIP. Cela a notamment guidé la structuration des médiateurs, qui sont des entités réalisant trois tâches d'intégration (selon le comparatif effectué sur les EIP).

Nous avons défini un langage d'un haut niveau d'abstraction qui permet la définition des nouveaux médiateurs, ainsi que leur assemblage. Dans notre implémentation de référence, nous avons fait un parseur de fichiers XML. Ces fichiers permettent d'exprimer des chaînes de médiation dans ce langage de haut niveau.

Parallèlement, nous avons défini un environnement d'exécution divisé en deux couches principales : un « méta-level », qui exprime les structures des chaînes de médiation et un « base-level » qui exprime les instances de composants de médiation. Ceci nous permet de découpler l'exécution des structures des applications à l'exécution. Nous avons défini une API du « méta-level » qui permet de réaliser des reconfigurations des applications de médiation à l'exécution.

Notre proposition offre des caractéristiques uniques qui permettent la construction de solutions d'intégration où le dynamisme et la prise en compte des évolutions sont importants. Ainsi, nous pouvons mettre en avant :

- La modularité. Les composants de médiation, ainsi que ses constituants, peuvent être développés séparément en divers modules. De même, les chaînes de médiation sont des entités indépendantes des composants développés.
- La séparation des préoccupations. Cette caractéristique est un des piliers de CILIA. De fait, la structure des médiateurs (scheduler, processor et dispatcher) permet de clairement séparer différents aspects de l'intégration.
- La flexibilité. CILIA permet de construire des applications flexibles, néanmoins, cette flexibilité est limitée aux capacités de reconfiguration des applications construites.
- L'embarquabilité. Nous avons choisi de construire une framework d'exécution qui soit facile à embarquer, cependant il y a des contraintes à prendre en considération. L'implémentation de référence peut être exécutée sur des plateformes qui supportent OSGi R4 et, par conséquence, qui contiennent Java. De plus, nous avons implémenté, afin de valider notre proposition, un framework d'exécution minimaliste pour exécuter des applications CILIA dans une plateforme Java JME CLDC 1.1.

Ces caractéristiques de CILIA offrent des avantages importants qui sont indispensables à des nouveaux domaines d'application. En particulier :

- Nous facilitons la conception de solutions d'intégration grâce à un modèle de médiateur et de chaînes de médiation basé sur des EIP.
- Nous proposons un modèle de développement ouvert. CILIA est un projet « *open source* ». De plus, il y a des points d'extension bien définis : comme les connecteurs internes/externes, les constituants de médiateurs, entre autres.
- Nous permettons de réaliser des évolutions dynamiques. CILIA propose des mécanismes pour faire des mises à jour des applications. Pour actualiser le code existant, ou pour modifier l'architecture de la chaîne de médiation, par exemple pour fournir une nouvelle fonctionnalité.
- Nous facilitons la maintenance. La modularité et la séparation de préoccupations facilitent la maintenance, cependant c'est de la responsabilité de l'architecte de modéliser les solutions d'intégration avec ces caractéristiques.

2 Perspectives de recherche

Le modèle à composant proposé dans ce manuscrit permet la création de solutions d'intégration modulaires, évolutives et dynamiques. Nous considérons que ce travail peut être enrichi suivant plusieurs axes de recherche. Nous présentons ici trois axes que nous considérons intéressants et importants : un langage de haut niveau d'abstraction, des mécanismes pour valider l'exactitude des architectures à l'exécution et enfin, des mécanismes pour réaliser des reconfigurations automatiques.

2.1 Langage de haut niveau d'abstraction

CILIA propose un langage de haut niveau d'abstraction appelé DSCilia. Ce langage est spécifique à la médiation. Cependant, nous envisageons des langages spécifiques aux domaines applicatifs, par exemple pour le domaine de la domotique, de la santé, etc.

Nous proposons comme perspective de recherche d'étudier la construction des applications par objectifs [HSM09], et non plus par des composants de médiation. Par exemple, dans le domaine de la domotique il est intéressant de construire des chaînes de médiation en utilisant des langages de haut niveau qui seront configurés par des utilisateurs, ce que l'on appelle « end-user programming ». Dans ce domaine, des membres de notre équipe, en collaboration avec des experts dans le domaine IHM [1], ont développé un prototype de construction automatique de chaînes de médiation en spécifiant les dispositifs d'interaction, des applications interactives et leurs capacités dans un langage de haut niveau qui permet d'exprimer le modèle d'interaction, néanmoins ce n'est pas encore un langage pour le « end-user programming » à proprement parlé.

Un autre exemple de langage de haut niveau est les « *mashups* ». Où nous pouvons construire des nouveaux services par l'assemblage de services existants, notamment en utilisant des APIs « ouvertes » et des notations graphiques.

1. Interaction Homme-Machine

L'étude dans ce domaine est accompagnée de l'étude de nouvelles approches de développement, de la définition d'un cycle de vie des applications, ainsi que de la spécialisation de composants de médiation et des patrons d'intégration pour des domaines spécifiques.

2.2 Validation à l'exécution

Nous sommes conscients que réaliser des reconfigurations à l'exécution peut occasionner des erreurs avec des conséquences potentiellement importantes. En fait, réaliser ce type d'opération doit être une tâche bien planifiée, bien étudiée et bien effectuée pour minimiser les coûts comme l'arrêt, la reconfiguration ou le redémarrage par des experts.

Nous proposons d'approfondir notre approche dans le domaine de la validation à l'exécution pour vérifier que les opérations à effectuer n'impactent pas les applications en exécution. Ainsi, il est possible de proposer des alternatives pour identifier les problèmes possibles, et maintenir toujours des applications correctes.

L'étude de la validation à l'exécution peut se concentrer sur la définition des architectures de référence, les différentes modes des architectures[HKM06], la surveillance de l'exécution, la résolution de conflits, et l'auto-guérison. La validation à l'exécution permet d'identifier des problèmes et proceder à une reconfiguration automatisé, comme presénté dans [ZPG10].

2.3 Reconfiguration autonomique

Notre proposition fournit un environnement d'exécution qui facilite l'introspection des architectures de médiation à l'exécution. Ainsi, nous fournissons des mécanismes pour modifier ces architectures à l'exécution. Ces reconfigurations doivent être faites par des experts des applications. Néanmoins, certaines applications dans les nouveaux domaines ne sont utilisées que par des utilisateurs finaux et il n'y a pas d'experts in-situ pour réaliser des reconfigurations. Ce constat, parmi d'autres, a entrainé la recherche de solutions qui soient capables de s'autogérer. C'est-à-dire que l'expertise de l'application est codée dans une brique logicielle en charge de réaliser des opérations d'administration (comme la reconfiguration).

Avec ces capacités de reconfiguration autonomique, il est possible par exemple de construire des applications sensibles au contexte, ou simplement des applications où il n'est pas nécessaire d'avoir un ingénieur expert in-situ pour réaliser les tâches d'administration.

Il y a un long chemin à faire dans ce domaine. Cependant, notre proposition offre des interfaces d'inspection et de réaction. On est donc à mi-chemin de solutions d'intégration autonomiques.

3 Perspectives de travail

Parallèlement à ces perspectives de recherche, nous proposons des perspectives de travail plus techniques autour de l'implémentation de référence de CILIA. Ces perspectives de travail permettront d'améliorer le framework ainsi de faciliter le développement des applications de médiation.

Outils de développement

Nous avons implanté un premier prototype d'éditeur graphique de chaînes de médiation. Cependant il y a des aspects à améliorer et à enrichir. Cet outillage permet de définir seulement des chaînes de médiation. Nous proposons d'étendre cet outillage pour qu'il permette plus facilement de décrire les spécifications de composants de médiation et de décrire de même les constituants de médiateurs (*scheduler*, *processor*, *dispatcher*).

Outils de *monitoring*

Le monitoring est une approche qui permet, entre autre, d'identifier le comportement des applications. Ainsi, dans les applications dynamiques, et en général pour les applications d'intégration, il est important de fournir des options de surveillance des applications.

Le monitoring est aussi un besoin lorsque l'on souhaite construire des applications qui seront auto-adaptables. Il est important de connaitre le contexte de l'application, mais aussi de connaitre le flot de l'exécution, qui peut être pertinent lors d'une reconfiguration. Cette connaissance doit être obtenue grâce à des fonctions de monitoring intrinsèques dans le framework.

Enrichir la bibliothèque de composants et constituants prêts à l'usage

Actuellement, nous avons une bibliothèque de composants minimale. Un travail technique autour de CILIA est nécessaire pour enrichir cette liste de médiateurs génériques, ainsi que de connecteurs pour des systèmes généralistes ; comme les bases de donnée, des protocoles de communication et de middlewares de messagerie.

Une caractéristique souhaitée par certains partenaires était justement cette bibliothèque d'éléments prêts à l'usage. Nous avons développé des composants basiques, que nous considérons comme essentiels et qui ont permis de valider nos concepts lors de l'usage de CILIA dans des projets. Néanmoins, il y a de travail à réaliser pour améliorer ces composants existants et pour en construire de nouveaux.

BIBLIOGRAPHIE

[AA06] Ali Arsanjani and Abdul Allam. Service-Oriented Modeling and Architecture for Realization of an SOA. *2006 IEEE International Conference on Services Computing (SCC'06)*, (January) :521–521, September 2006.

[AFM05] Marco Aiello, Ganna Frankova, and Daniela Malfatti. What's in an agreement ? an analysis and an extension of ws-agreement. In *Proceedings of the Third international conference on Service-Oriented Computing*, ICSOC'05, pages 424–436, Berlin, Heidelberg, 2005. Springer-Verlag.

[All] OSGi Alliance. The OSGi Architecture. available at `http://www.osgi.org/About/WhatIsOSGi`.

[ALN11] Pierre-Alain Avouac, Philippe Lalanda, and Laurence Nigay. Service-oriented autonomic multimodal interaction in a pervasive environment. In *Proceedings of the 13th international conference on multimodal interfaces - ICMI '11*, page 369, New York, New York, USA, November 2011. ACM Press.

[ANL11] Pierre-Alain Avouac, Laurence Nigay, and Philippe Lalanda. Towards autonomic multimodal interaction. In *Proceedings of the 1st Workshop on Middleware and Architectures for Autonomic and Sustainable Computing - MAASC '11*, pages 25–29, New York, New York, USA, May 2011. ACM Press.

[Bal] Roland Balter. What the ObjectWeb's ESB may look like. available at `https://wiki.ow2.org/ESBi/attach?page=Vision%2FJORAM_ESB.doc`.

[BBB+00] F. Bachmann, L. Bass, C. Buhman, S. Comella-Dorda, Fred Long, J. Robert, Robert Seacord, and K. Wallnau. *Volume II : Technical concepts of component-based software engineering*, volume II. Software Engineering Institute Carnegie Mellon, 2000.

[BCG04] Gordon S. Blair, Geoff Coulson, and Paul Grace. Research directions in reflective middleware : the lancaster experience. In *Proceedings of the 3rd workshop on Adaptive and reflective middleware*, ARM '04, pages 262–267, New York, NY, USA, 2004. ACM.

[BdMAS09] Javier F. Briones, Miguel A. de Miguel, A. Alonso, and Juan Pedro Silva. Quality of Service Composition and Adaptability of Software Architectures. In *2009 IEEE International Symposium on Object/Component/Service-Oriented Real-Time Distributed Computing*, pages 169–173. IEEE, March 2009.

[Ber93] Philip A. Bernstein. Middleware an architecture for distributed system services. *Communications of the ACM*, 39 :86–98, 1993.

[Ber96] Philip A. Bernstein. Middleware : a model for distributed system services. *Commun. ACM*, 39(2) :86–98, February 1996.

[BG05] Luciano Baresi and Sam Guinea. Dynamo : Dynamic monitoring of ws-bpel processes. In *In Proceedings of the International Conference on Service-Oriented Computing (ICSOC'05*, pages 478–483. Springer, 2005.

[BGG04] Luciano Baresi, Carlo Ghezzi, and S. Guinea. Towards self-healing service compositions. *Proceedings of PRISE*, 2004.

[BGP07] Luciano Baresi, S. Guinea, and Liliana Pasquale. Self-healing BPEL processes with Dynamo and the JBoss rule engine. In *International workshop on Engineering of software services for pervasive environments : in conjunction with the 6th ESEC/FSE joint meeting*, number 2, pages 11–20. ACM, 2007.

[BH06] Peter F Brown and Rebekah Metz Booz Allen Hamilton. Reference Model for Service Oriented Architecture 1.0, 2006.

[BH07] André Bottaro and Richard S. Hall. Dynamic contextual service ranking. pages 129–143, March 2007.

[BLE10] J. Bardin, P. Lalanda, and C. Escoffier. Towards an Automatic Integration of Heterogeneous Services and Devices. In *Services Computing Conference (APSCC), 2010 IEEE Asia-Pacific*, pages 171–178. IEEE, 2010.

[BN84] Andrew D. Birrell and Bruce Jay Nelson. Implementing remote procedure calls. *ACM Transactions on Computer Systems*, 2(1) :39–59, February 1984.

[BN06] Luciano Baresi and Elisabetta Di Nitto. Toward Open-World Software : Issues and Challenges. *Computer*, 2006.

[BNGG07] Luciano Baresi, Elisabetta Nitto, Carlo Ghezzi, and Sam Guinea. A framework for the deployment of adaptable web service compositions. *Service Oriented Computing and Applications*, 1(1) :75–91, March 2007.

[Cab05] L. Cabral. Mediation of semantic web services in IRS-III. *Proceedings of the First International Workshop on Mediation in Semantic Web Services (MEDIATE 2005)*, pages 1–16, 2005.

[CCMW01] Erik Christensen, Francisco Curbera, Greg Meredith, and Sanjiva Weerawarana. Web Service Definition Language (WSDL) 1.1. Technical report, 2001.

[CH03] Humberto Cervantes and Richard S. Hall. Automating service dependency management in a service-oriented component model. *Proceedings of the 6th Workshop on Component-Based Software Engineering*, 2003.

[Cha99] J Charles. Middleware moves to the forefront. *IEEE Computer*, (May 1999) :17–19, 1999.

[Cha04] D.A. Chappell. *Enterprise service bus*. O'Reilly Media, Inc., 2004.

[Cha07] David Chappell. Introducing SCA. (July), 2007. Avalaible at `http://www.davidchappell.com/articles/Introducing_SCA.pdf`.

[CLB08] Stephanie Chollet, Philippe Lalanda, and Andre Bottaro. Transparently adding security properties to service orchestration. In *Advanced Information Networking and Applications-Workshops, 2008. AINAW 2008. 22nd International Conference on*, pages 1363–1368. IEEE, 2008.

[Cle96] P.C. Clements. A survey of architecture description languages. *Proceedings of the 8th International Workshop on Software Specification and Design*, pages 16–25, 1996.

[CNM06] Massimiliano Colombo, Elisabetta Di Nitto, and Marco Mauri. SCENE : A Service Composition Execution Environment Supporting Dynamic Changes Disciplined Through Rules. pages 191 – 202, 2006.

[CS] James Clark and DeRose Steve. XML Path Language (XPath). Technical report.

[CS02] E. Cerami and S. St Laurent. *Web services essentials*. O'Reilly & Associates, Inc., 2002.

[DEA97] S Dami, J Estublier, and M Amiour. APEL : a Graphical Yet Executable Formalism for Process Modeling. *Architecture*, 5(1) :1–35, 1997.

[DL08] L. Dong and H. Linpeng. A Framework for Ontology-Based Data Integration. In *2008 International Conference on Internet Computing in Science and Engineering*, pages 207–214. IEEE, 2008.

[dRAE+07] Tarcisio da Rocha, Anna-Brith Arntsen, Arne Ketil Eidsvik, Maria Beatriz Felgar de Toledo, and Randi Karlsen. Promoting levels of openness on component-based adaptable middleware. *Proceedings of the 6th international workshop on Adaptive and reflective middleware held at the ACM/IFIP/USENIX International Middleware Conference - ARM '07*, pages 1–6, 2007.

[dro] Drools Engine. available at `http://www.jboss.org/drools`.

[EFGK03] Patrick Th. Eugster, Pascal a. Felber, Rachid Guerraoui, and Anne-Marie Kermarrec. The many faces of publish/subscribe. *ACM Computing Surveys*, 35(2) :114–131, June 2003.

[EHL07] Clement Escoffier, R.S. Richard S Hall, and Philippe Lalanda. iPOJO : An extensible service-oriented component framework. *Computing*, (Scc), 2007.

[FAB06] Marie-Christine Fauvet and Ali Aït-Bachir. An automaton-based approach for web service mediation. *Proceedings of the 2006 conference on Leading the Web in Concurrent Engineering : Next Generation Concurrent Engineering*, pages 47–54, 2006.

[FGCB98] A. Fox, S.D. Gribble, Y. Chawathe, and E.A. Brewer. Adapting to network and client variation using infrastructural proxies : lessons and perspectives. *Personal Communications, IEEE*, 5(4) :10 –19, aug 1998.

[Fou] Apache Foundation. Apache ACE. available at `http://incubator.apache.org/ace/`.

[Fow06] Martin Fowler. Internal DSL, 2006. available at `http://martinfowler.com/bliki/InternalDslStyle.html`.

[Gar93] David Garlan. An introduction to software architecture. *in software engineering and knowledge*, (January), 1993.

[Gho06] S. Ghosh. *Distributed systems : an algorithmic approach*, volume 13. CRC press, 2006.

[Gle05] Rodney Gleghorn. Enterprise application integration : a manager perspective. *IT professional*, 7(December) :17–23, 2005.

[GLV+03] David Gay, P. Levis, R. Von Behren, Matt Welsh, Eric Brewer, and D. Culler. The nesC language : A holistic approach to networked embedded systems. In *Acm Sigplan Notices*, volume 38, pages 1–11. ACM, 2003.

[Hal08] Richard S. Hall. iPOJO : The Simple Life. Technical report, 2008.

[HC01] G.T. Heineman and W.T. Councill. *Component-based software engineering : putting the pieces together*, volume 17. Addison-Wesley USA, 2001.

[HCM+05] A. Haller, E. Cimpian, A. Mocan, E. Oren, and C. Bussler. Wsmx-a semantic service-oriented architecture. *2005 IEEE International Conference on Web Services*, 2005.

[HKM06] Dan Hirsch, Jeff Kramer, and Jeff Magee. Modes for software architectures. *Software Architecture*, 2006.

[HM06] Didier Hoareau and Yves Mahéo. Middleware support for the deployment of ubiquitous software components. *Personal and Ubiquitous Computing*, 12(2) :167–178, November 2006.

[HSM09] William Heaven, Daniel Sykes, and Jeff Magee. A case study in goal-driven architectural adaptation. *Software Engineering for Self-*, 2009.

[HT04] Gregor Hohpe and Hsue-shen Tham. Enterprise Integration Patterns with BizTalk Server 2004. *Translator*, (July), 2004.

[HW08] Gregor Hohpe and B Woolf. *Enterprise Integration Patterns*, volume 29. Addison-Wesley, 2008.

[JN08] Jingwen Jin and Klara Nahrstedt. QoS-Aware service management for component-based distributed applications. *ACM Transactions on Internet Technology*, 8(3) :1–31, May 2008.

[JWY+09] Lu Jin, Jian Wu, Jianwei Yin, Ying Li, and Shuiguang Deng. Ontology Alignment Based Service Interface Adaptation. In *2009 IEEE International Conference on Services Computing*, pages 494–497. IEEE, September 2009.

[JWY+10] Lu Jin, Jian Wu, Jianwei Yin, Yin Li, and Shuiguang Deng. Improve Service Interface Adaptation Using Sub-ontology Extraction. In *2010 IEEE International Conference on Services Computing*, pages 170–177. IEEE, July 2010.

[Kaj04] Ejub Kajan. The maturity of open systems for B2B. *ACM SIGecom Exchanges*, 5(2) :34–44, November 2004.

[KCBC02] F. Kon, F. Costa, G. Blair, and R.H. Campbell. The case for reflective middleware. *Communications of the ACM*, 45(6) :33–38, January 2002.

[Kee04] P. Keen, M. and Acharya, A. and Bishop, S. and Hopkins, A. and Milinski, S. and Nott, C. and Robinson, R. and Adams, J. and Verschueren. *Patterns : Implementing an SOA using an enterprise service bus*. International Business Machines Corporation (IBM), 2004.

[KF00] Emre Kiciman and Armando Fox. Using dynamic mediation to integrate COTS entities in a ubiquitous computing environment. *Handheld and Ubiquitous Computing*, 2000.

[Lin00] David S. Linthicum. *Enterprise application integration*. Addison-Wesley Longman Ltd., Essex, UK, UK, 2000.

[LLB+09] Heiko Ludwig, Jim Laredo, Kamal Bhattacharya, Liliana Pasquale, and Bruno Wassermann. REST-based management of loosely coupled services. *Proceedings of the 18th international conference on World wide web - WWW '09*, page 931, 2009.

[LV95] D.C. Luckham and J. Vera. An event-based architecture definition language. *IEEE Transactions on Software Engineering*, 21(9) :717–734, 1995.

[Mar08] Cristina Marin. *Une approche orientée domaine pour la composition de services*. PhD thesis, Université Joseph Fourier, Grenoble, May 2008.

[MKB+04] R. Morrison, G. Kirby, D. Balasubramaniam, K. Mickan, F. Oquendo, S. Cimpan, B. Warboys, B. Snowdon, and R.M. Greenwood. Support for evolving software architectures in the ArchWare ADL. *Proceedings. Fourth Working IEEE/IFIP Conference on Software Architecture (WICSA 2004)*, (d) :69–78, 2004.

[Nau68] Software engineering, report on a conference sponsored by the NATO science committee. October 1968.

[New02] E. Newcomer. *Understanding Web Services : XML, Wsdl, Soap, and UDDI*, volume 3. Addison-Wesley Professional, 2002.

[OAS] OASIS. UDDI Version 3.0.2. available at `http://uddi.org/pubs/uddi_v3.htm`.

[Off05] M. Offermans. Automatically managing service dependencies in OSGi, 2005. available at `http://www.osgi.org/wiki/uploads/Links/AutoManageServiceDependencies_byMOffermans.pdf`.

[Oza10] Gunalp Ozan. *Evaluation de la sureté de fonctionnement des services appliquée aux applications pervasives autonomiques*. Master recherche, Université de Grenoble, 2010.

[Pap03] M.P. Mike P Papazoglou. Service-oriented computing : Concepts, characteristics and directions. *Systems Engineering*, 2003.

[Pas05] J Pasley. How BPEL and SOA Are Changing Web Services Development. *IEEE Internet Computing*, 9(3) :60–67, 2005.

[PD04] Mike Papazoglou and Jean-jacques Dubray. A survey of web service technologies. June 2004.

[Pel03] Chris Peltz. Web services orchestration and choreography. *Computer*, 36(10) :46–52, 2003.

[PF02] S.R. Ponnekanti and Armando Fox. Sword : A developer toolkit for web service composition. In *Proc. of the Eleventh International World Wide Web Conference, Honolulu, HI*, 2002.

[PLF+01] Shankar Ponnekanti, Brian Lee, Armando Fox, Pat Hanrahan, and Terry Winograd. Icrafter : A service framework for ubiquitous computing environments. *Proceedings of the 3rd international conference on Ubiquitous Computing*, pages 56–75, 2001.

[PPMM11] Diego Perez-Palacin, Raffaela Mirandola, and José Merseguer. Software architecture adaptability metrics for QoS-based self-adaptation. In *Proceedings of the joint ACM SIGSOFT conference – QoSA and ACM SIGSOFT symposium – ISARCS on Quality of software architectures – QoSA and architecting critical systems – ISARCS - QoSA-ISARCS '11*, page 171, New York, New York, USA, June 2011. ACM Press.

[RABA] S. Rigo, G. Araujo, M. Bartholomeu, and R. Azevedo. ArchC : A SystemC-Based Architecture Description Language. *16th Symposium on Computer Architecture and High Performance Computing*, pages 66–73.

[RCBO06] J. Revillard, S. Cimpan, E. Benoit, and F. Oquendo. Intelligent Instrument Design With ArchWare ADL. *Fourth Workshop on Model-Based Development of Computer-Based Systems and Third International Workshop on Model-Based Methodologies for Pervasive and Embedded Software (MBD-MOMPES'06)*, pages 63–74, 2006.

[SD05] Zoran Stojanovic and Ajantha Dahanayake. Service-oriented software system engineering : challenges and practices. 2005.

[SG96] Mary Shaw and David Garlan. *Software Architecture : Perspectives on an Emerging Discipline*, volume 123. Prentice Hall, 1996.

[SGM02] C Szyperski, D Gruntz, and S. Murer. *Component software : beyond object-oriented programming*. Addison-Wesley Professional, 2002.

[SL90] A P Sheth and J A Larson. Federated database systems for managing distributed, heterogeneous, and autonomous databases. *ACM Computing Surveys (CSUR)*, 22(3) :183–236, 1990.

[SMF+09] Lionel Seinturier, Philippe Merle, Damien Fournier, Nicolas Dolet, Valerio Schiavoni, and Jean-Bernard Stefani. Reconfigurable SCA Applications with the FraSCAti Plat-

form. *2009 IEEE International Conference on Services Computing*, pages 268–275, 2009.

[SW04] D. Sprott and L. Wilkes. Understanding service-oriented architecture. *The Architecture Journal*, 1(1) :10–17, 2004.

[SZ00] Clay Spinuzzi and M. Zachry. Genre ecologies : An open-system approach to understanding and constructing documentation. *ACM Journal of Computer Documentation (JCD)*, 24(3) :169–181, 2000.

[TDR08] L. Touseau, D. Donsez, and W. Rudametkin. Towards a sla-based approach to handle service disruptions. In *Services Computing, 2008. SCC'08. IEEE International Conference on*, volume 1, pages 415–422. IEEE, 2008.

[TOHS99] Peri Tarr, H. Ossher, W. Harrison, and S.M. Sutton Jr. N degrees of separation : Multi-dimensional separation of concerns. In *Proceedings of the 21st international conference on Software engineering*, pages 107–119. ACM, 1999.

[Vin06] Steve Vinoski. Advanced message queuing protocol. *IEEE Internet Computing*, 10(6) :87–89, November 2006.

[W3C] W3C. SOAP Specifications (W3C Recommendation 27 April 2007). available at `http://www.w3.org/TR/soap/`.

[W3c07a] W3c. Web service definition language (wsdl) (W3C recommendation 26 june 2007). Technical report, W3C, 2007. available at `http://www.w3.org/TR/wsdl/`.

[W3c07b] W3c. XSL transformations (XSLT) version 2.0 (W3C recommendation 23 january 2007). Technical report, W3C, 2007. available at `http://www.w3.org/TR/xslt20/`.

[WG97] Gio Wiederhold and Michael Genesereth. The conceptual basis for mediation services. *IEEE Expert*, 12(5) :38–47, 1997.

[Wie92] Gio Wiederhold. Mediators in the architecture of future information systems. *Computer*, 25(3) :38–49, 1992.

[Wie07] Gio Wiederhold. Mediators , Concepts and Practice. *Interfaces*, 2007.

[ZDP09] Y. Zhao, J. Dong, and T. Peng. Ontology classification for semantic-web-based software engineering. *IEEE Transactions on Services Computing*, 2(4) :303–317, 2009.

[ZPG10] E. Zahoor, O. Perrin, and C. Godart. Disc : A declarative framework for self-healing web services composition. In *Web Services (ICWS), 2010 IEEE International Conference on*, pages 25 –33, july 2010.