HAL
open science

UNE APPROCHE À COMPOSANT POUR L'ORCHESTRATION DE SERVICES À LARGE ÉCHELLE

Virginie Legrand Contes

To cite this version:

Virginie Legrand Contes. UNE APPROCHE À COMPOSANT POUR L'ORCHESTRATION DE SERVICES À LARGE ÉCHELLE. Architectures Matérielles [cs.AR]. Université Nice Sophia Antipolis, 2011. Français. NNT : . tel-00710427

HAL Id: tel-00710427

https://theses.hal.science/tel-00710427v1

Submitted on 20 Jun 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 NICE – SOPHIA ANTIPOLIS
ÉCOLE DOCTORALE STIC
SCIENCES ET TECHNOLOGIES DE L'INFORMATION ET DE LA COMMUNICATION

THÈSE

pour obtenir le titre de

Docteur ès Sciences

de l'Université de Nice – Sophia Antipolis Mention Informatique

présentée et soutenue le 15 décembre 2011 par

Virginie LEGRAND CONTES

UNE APPROCHE À COMPOSANT POUR L'ORCHESTRATION DE SERVICES À LARGE ÉCHELLE.

Thèse co-dirigée par Françoise BAUDE et Philippe MERLE

Devant le Jury composé de :

Johann MONTAGNAT	CNRS, France	*Président*
Didier DONSEZ	Université Joseph Fourier, Grenoble, France	*Rapporteur*
François CHAROY	Université Henri Poincaré, Nancy, France	*Rapporteur*
Nicolas SALATGÉ	Petals Links, Toulouse, France	*Examinateur*
Françoise BAUDE	Université de Nice-Sophia Antipolis, France	*Directrice*
Philippe MERLE	LIFL, Lille, France	*Co-directeur*

à mes deux petits ~~monstres~~, Antoine et Julie,

à Arnaud,

à ma famille.

Remerciements

Si il y a une dizaine d'années on m'avait dit que j'aurais réalisé une thèse, je ne l'aurais pas cru. Le parcours n'a pas été simple,avec des angoisses et des peurs de ne pas y arriver. Heureusement j'ai été soutenue, aidée et remotivée pour achever tout cela. Aussi j'aimerais remercier :

Françoise Baude pour m'avoir fait prendre goût à la recherche et pour m'avoir guidée tout au long de ces dernières années.

Philippe Merle pour m'avoir co-encadrée. Les discussions que nous avons eues ont toujours été enrichissantes.

Les membres du jury, Didier Donsez et Francois Charoy qui m'ont fait l'honneur de rapporter le manuscrit mais aussi Johann Montagnat et Nicolas Salatgé pour être venus m'écouter le jour J.

L'équipe OASIS et ses "permanents", Eric, Ludovic, Fabrice, Denis pour m'avoir supportée pendant 7 ans.

Les anciens et les nouveaux d'OASIS, ingénieurs, doctorants, stagiaires, apprentis. Chaque rencontre a été une expérience enrichissante et parfois s'est transformée en belle amitié.

Les assistantes d'OASIS, Claire, Christiane, Patricia et Sylvie.

Bouba, Petit Ours Brun, Baby Einstein, T'Choupi et Mickey pour avoir occupé mes enfants quand c'était nécessaire.

Mes amis, de plus ou moins longue date, mais toujours présents.

Ma famille. Mes parents pour m'avoir soutenue tout au long de ma scolarité et permis notamment de travailler "au vert" à la campagne. Mon père pour m'avoir fait comprendre que la règle de trois était la base de tout (oui oui !) et pour avoir passé du temps, lorsque j'avais des problèmes à résoudre, à m'écouter (peut être sans rien y comprendre) et à me mettre sur la bonne voie. Mes grands-parents pour m'avoir accueillie à un moment de ma scolarité.

Mes enfants pour avoir fait de moi une maman comblée et fière. Ils ont suivi dès le début ce travail de thèse et ils ont grandit avec.

Mon mari, Arnaud sans qui tout ça n'aurait pas pu se faire. C'est lui qui m'a motivée pour faire le grand saut et qui m'a soutenue avec patience (ou pas) chaque jour.

Une page se tourne, et c'est grâce à vous tous ...

– Virginie

Table des matières

1 Introduction **17**
1.1 Contexte . . . 17
1.1.1 L´Entreprise Inter-Organisationnelle . . . 17
1.1.2 Des processus métiers basés sur l'Architecture Orientée Services . . . 19
1.2 Problématique : Orchestration dynamique et décentralisée des processus inter-organisationnels . . . 20
1.2.1 Processus métiers distribués à travers le réseau . . . 20
1.2.2 Bénéfices d'une approche distribuée pour l'exécution d'un processus métier . . . 21
1.2.3 Contraintes liées à la décentralisation des processus métiers . . . 22
1.3 Contribution de la thèse . . . 23
1.4 Structure du document . . . 24

2 Contexte : Les architectures orientées services **27**
2.1 Services et Architectures Orientées Services . . . 28
2.1.1 Services : De multiples définitions . . . 28
2.1.2 Le paradigme de programmation basée sur les services (SOC) . . . 29
2.1.3 Les Architectures Orientées Services . . . 31
2.1.3.1 La SOA étendue . . . 32
2.1.4 Mise en œuvre d'une SOA et plateformes à services . . . 33
2.1.4.1 Services Web . . . 33
2.1.4.2 Les services RESTful . . . 35
2.1.4.3 De la Solution "maison" . . . 36
2.1.4.4 . . . à l' ESB : Un environnement d'intégration basé sur les Services Web . . . 37
2.1.4.5 La fédération du bus à services distribué de la plateforme SOA4ALL 39
2.1.5 Conclusions sur l'approche à services . . . 40
2.2 Composition des services . . . 40
2.2.1 Cycle de vie d'une composition de services . . . 41
2.2.2 Le contrôle de la composition . . . 42
2.2.3 Composition dans le temps . . . 42
2.2.4 Composition structurelle : Les composants orientés services . . . 44
2.2.4.1 L'approche à composants . . . 44
2.2.4.2 Approches et Objectifs des composants orientés services . . . 46
2.2.4.3 Service Component Architecture (SCA) . . . 47
2.2.4.4 iPOJO . . . 50
2.2.4.5 Mashups et Web 2.0 . . . 51

2.2.5 Conclusions sur la composition de services 51
2.3 Synthèse du chapitre 51

3 Des systèmes d'orchestration centralisés aux systèmes d'orchestration répartie 53
3.1 Système de gestion de Workflow 54
3.2 Composition centralisée avec WS-BPEL 56
3.2.1 Déploiement 57
3.2.2 Dynamicité et distribution 57
3.2.3 Implémentations de WS-BPEL 57
3.2.4 Synthèse 58
3.3 Systèmes d'exécution de Workflows Dynamiques et Distribués 58
3.3.1 Exotica/FMQM [AMA+95] 59
3.3.2 SCENE/secse [CDM06] 60
3.3.3 JOPERA [PA05b] 62
3.3.4 SELF-SERV [SDM02] 64
3.3.5 FOCAS [Ped09] 66
3.3.6 WISE [LASS00] 67
3.3.7 Crossflow [GAHL00] 69
3.3.8 NIÑOS [GYJ11] 71
3.3.9 SwinDew [YYR06] 73
3.3.10 Taverna [HWS+06, OAF+04] 74
3.3.11 KEPLER [ABJ+04] [LAB+06] 75
3.4 Synthèse de l'état de l'art 76
3.5 Conclusion du chapitre 78

4 Positionnement de notre travail 81
4.1 Choix d'une approche basée sur les composants 81
4.1.1 Approche à composants 82
4.1.1.1 Structure hiérarchique 83
4.1.1.2 Reconfiguration dynamique 84
4.2 Notre proposition : Conception d'une orchestration décentralisée et dynamique en utilisant une approche à composants 84
4.2.1 Positionnement de notre approche 84
4.2.2 Vue d'ensemble de notre méthodologie 87
4.3 Conclusion 88

5 Exécution d'orchestrations réparties grâce aux composants 89
5.1 Définition d'un fragment 90
5.1.1 Le fragment, élément de base 90
5.1.2 Composition de Fragments 92
5.2 Contrôle du cycle de vie du fragment 96
5.2.1 Génération du Fragment 96
5.2.2 Interfaces de contrôle 97
5.2.3 Déploiement d'un Fragment 98
5.2.4 Exécution d'un fragment unitaire 99
5.2.5 Administration et Supervision d'un sous-processus 101
5.2.6 Reconfiguration dynamique du Fragment 102

5.3 Projection d'une orchestration décentralisée sur un composant hiérarchique distribué . . . 103
5.3.1 Projection de l'orchestration distribuée . . . 103
5.3.2 Gestion de la complexité du déploiement de l'orchestration distribuée . 104
5.3.3 Exécution d'un fragment composite . . . 105
5.3.4 Gestion de la dynamicité dans une orchestration distribuée . . . 106
5.3.5 Administration et supervision de l'orchestration globale . . . 107
5.4 Conclusion . . . 107

6 Mise en Œuvre de la solution 109
6.1 Contexte Technologique . . . 109
6.1.1 WS-BPEL . . . 110
6.1.2 Fractal . . . 115
6.1.3 GCM : Grid Component Model . . . 118
6.1.4 Le support pour le déploiement distribué . . . 118
6.1.4.1 Le support pour les communications collectives . . . 119
6.1.4.2 Les interfaces *gathercast* . . . 119
6.1.4.3 Les interfaces *multicast* . . . 119
6.1.4.4 Le support pour les préoccupations non-fonctionnelles . . . 120
6.1.4.5 Séparation entre les préoccupations Fonctionnelles (F) et Non-Fonctionnelles (NF) . . . 120
6.1.4.6 Les interfaces NF . . . 120
6.1.4.7 Notation et liaisons Non-Fonctionnelles . . . 121
6.1.4.8 Contrôleurs standards . . . 122
6.1.4.9 Reconfiguration . . . 122
6.1.5 GCM/ProActive . . . 123
6.1.5.1 Objets Actifs et communications asynchrones . . . 123
6.1.5.2 Communications asynchrones dans les composants GCM/ProActive . . . 126
6.1.5.3 Instrumentation des composant GCM/ProActive . . . 127
6.1.5.4 Marquage des messages dans GCM/ProActive . . . 129
6.1.5.5 Fonctionnalités additionnelles . . . 129
6.2 Implémentation de notre solution . . . 130
6.2.1 Choix techniques . . . 130
6.2.2 Implémentation d'un Fragment Unitaire . . . 131
6.2.2.1 Structure du Fragment Unitaire . . . 131
6.2.2.2 Déploiement et génération du fragment . . . 135
6.2.2.3 Exécution d'un Fragment unitaire . . . 136
6.2.2.4 Supervision et Administration du fragment . . . 138
6.2.2.5 Reconfiguration Dynamique . . . 139
6.2.3 Projection d'une orchestration distribuée sur un composant composite 139
6.2.3.1 Définition d'une orchestration décentralisée avec WS-BPEL . . 139
6.2.3.2 Le gestionnaire de fragment composite . . . 141
6.2.3.3 Projection d'une orchestration décentralisée sur un composant GCM . . . 141
6.2.3.4 Exécution du fragment composite . . . 142
6.2.3.5 Reconfiguration dynamique du fragment composite . . . 144
6.3 Évaluation des performances du modèle à composants . . . 144
6.4 Conclusion . . . 145

7 Mise en pratique **149**
7.1 Introduction 149
7.1.1 La norme OSGi [The07] 149
7.1.2 Problématique du déploiement à large échelle 150
7.1.3 Gestion d'un parc de passerelles OSGi via une approche à services . . 152
7.2 Application de notre solution 154
7.2.1 Une orchestration distribuée et dynamique pour gérer un parc de passerelles 154
7.2.2 Configuration de la partie dynamique: ajout des plans de déploiement à l'orchestration globale 157
7.3 Conclusions 158

8 Conclusions et perspectives **159**
8.1 Bilan de notre contribution 159
8.2 Perspectives 161

Bibliographie **163**

Table des figures

1.1 Un exemple d'entreprise inter-organisationnelle 18
1.2 Un processus distribué sur plusieurs sites 21

2.1 Les Rôles dans une Architecture Orientée Services 31
2.2 Architecture Orientée Service Etendue [PTDL07] 33
2.3 Une ressource RESTful et l'utilisation des opérations HTTP permettant de la manipuler 35
2.4 Un exemple de ressource REST 36
2.5 Une implémentation de la SOA avec un ESB 37
2.6 Les différentes étapes d'une interaction au sein d'un ESB [Ley05] 38
2.7 L'architecture globale de SOA4All 40
2.8 Deux visions de compositions de services dans le temps 44
2.9 Structure d'un composant 45
2.10 Un composite SCA incluant deux composants SCA 48

3.1 Exécution centralisée d'un processus composé de 4 sous-processus (SP1, SP2, SP3, SP4) 55
3.2 Exécution décentralisée d'un processus composé de 4 sous-processus (SP1, SP2, SP3 et SP4). 56
3.3 Distribution d'un processus sur plusieurs nœuds avec trois instances actives dans Exotica/FMQM [AMA+95] 60
3.4 L´architecture de la plateforme SCENE 61
3.5 L'architecture logique du moteur d'exécution distribué JOPERA 63
3.6 Architecture de SELF-SERV [BDS05] 65
3.7 Architecture d'un noeud FOCAS 67
3.8 Le canevas d'exécution WISE [LASS00] 68
3.9 L'exécution d'un service dans l'architecture CrossFlow [GAHL00] 70
3.10 Crossflow : Exemple d´un consommateur qui externalise deux de ses activités (D+ et E+) 71
3.11 L'architecture de distribution d'un workflow dans NIÑOS 72
3.12 L'architecture du système d'exécution décentralisé de workflow SwinDew [YYR06] 73

4.1 La composition dans l'espace combinée à la composition dans le temps . . . 85
4.2 Un exemple de composant spatio-temporel (correspondant au processus distribué de la Figure 1.2) . . . 86
4.3 Phases de notre méthodologie . . . 87

5.1 Représentation fonctionnelle d'un Fragment . . . 91
5.2 Structure d'un fragment d'orchestration . . . 91
5.3 Les composants *prox*ies permettant d'intercepter les appels vers le moteur d'exécution et vers les services externes . . . 93
5.4 Un exemple de composition de fragments . . . 93
5.5 Dépendances temporelles multiples . . . 94
5.6 Composition Hiérarchique des fragments . . . 95
5.7 Le modèle de fragment . . . 96
5.8 Composition du Gestionnaire de Fragment . . . 98
5.9 Un exemple d´exécutions multiples d´un même processus . . . 100
5.10 Les différents niveaux d'exécution impliqués dans une orchestration décentralisée . . . 105
5.11 Exemple d'une orchestration distribuée projetée sur un fragment composite . . . 108

6.1 Anatomie d'un processus WS-BPEL . . . 111
6.2 Exemple graphique d'un processus WS-BPEL correspondant au listing 6.1 . . . 112
6.3 Éléments du modèle à composants Fractal . . . 116
6.4 Le modèle d'un composant Fractal et son implémentation avec Julia . . . 117
6.5 Les Interfaces *multicast* et *gathercast* de GCM . . . 119
6.6 Les éléments d'une application GCM incluant des composants Non-Fonctionnels dans la membrane . . . 122
6.7 L'architecture à Méta-Objet . . . 123
6.8 Séquence d'un appel asynchrone sur un objet actif . . . 125
6.9 L'architecture à Meta-Objet pour un composant GCM . . . 127
6.10 Séquence d'un appel asynchrone dans les composants GCM/-ProActive . . . 128
6.11 Le flux d'information dans un composant GCM/ProActive . . . 128
6.12 L'implémentation du fragment en GCM : Le Gestionnaire de fragment est inséré dans la membrane du composant GCM 133
6.13 Le composant composite Non-Fonctionnel correspondant au Gestionnaire de Fragment Unitaire . . . 134
6.14 Le composant "englobant" le moteur d'orchestration . . . 134
6.15 Le composant proxy . . . 135
6.16 Le composant GCM responsable du déploiement du Fragment . 136
6.17 Le composant d'exécution NF d'un fragment unitaire . . . 137

6.18 Le composant permettant d'accéder au processus métier WS-BPEL 138
6.19 Le composant d'administration et de supervision 138
6.20 Le composant de reconfiguration dynamique 139
6.21 Utilisation des interfaces GCM *multicast* pour l'implémentation du gestionnaire d'un fragment composite. 141
6.22 Interfaces multicast et gathercast pour l'exécution parallèle des fragments 143
6.23 Un exemple de fragment composite implémenté avec GCM . . . 147

7.1 Exploitation des passerelles à distance 150
7.2 Les étapes à suivre pour la réalisation du déploiement d'une appication sur un ensemble de passerelles 152
7.3 Le service d'obtention de l'infrastructure, représenté avec la notation SCA 152
7.4 Structure du parc de passerelles, distribué sur plusieurs domaines 153
7.6 Le service de calcul de plan de déploiement 154
7.5 Le domaine d'administration du parc de passerelles 154
7.7 L'orchestration distribuée correspondant au processus de déploiement 155
7.8 Le service d'installation des plans de déploiement 156

Liste des tableaux

2.1 Récapitulatif des différents concepts de SCA 49

3.1 Caractéristiques de l'exécution des processus métiers WS-BPEL 58
3.2 Classification des différents systèmes d'exécution de workflows 79

5.1 Types d'évènements proposés par l'interface de supervision d'un fragment 102

6.1 Activités composites du langage WS-BPEL 113
6.2 Activités composites du langage WS-BPEL 114
6.3 Les notifications générées par GCM/ProActive 129
6.5 Le surcoût induit par notre modèle sur l'exécution d'un processus 145
6.4 Le surcoût induit par le modèle à composant dans l'appel à un service depuis un *proxy* 145

Listings

5.1 Dépendances temporelles d'un Fragment 94
5.2 Interface de déploiement d'un Fragment 99
5.3 Interface d'administration d'un fragment 101
5.4 L´Interface de supervision du fragment 102
5.5 Interface de contrôle de la dynamicité 103
5.6 L´ interface de reconfiguration dynamique d´un fragment composite 106
6.1 Un exemple simple de processus WS-BPEL 112
6.2 Création d´ un objet actif avec ProActive 124
6.3 L´ interface du composant Processus permettant le declenchement du workflow 138
6.4 Exemple de définition d'une orchestration décentralisée en WS-BPEL 140
6.5 Définition des patrons d'invocation parallèle et de jointure correspondant à la Figure 6.22. 143
7.1 Un exemple de plan de déploiement unitaire 151

L'important n'est pas de convaincre,
mais de donner à réfléchir.

Bernard Werber, Extrait de *Le Père de nos pères.*

Chapitre 1

Introduction

Contenu du chapitre :

1.1 Contexte 17
1.1.1 L´Entreprise Inter-Organisationnelle 17
1.1.2 Des processus métiers basés sur l'Architecture Orientée Services 19
1.2 Problématique : Orchestration dynamique et décentralisée des processus inter-organisationnels 20
1.2.1 Processus métiers distribués à travers le réseau 20
1.2.2 Bénéfices d'une approche distribuée pour l'exécution d'un processus métier 21
1.2.3 Contraintes liées à la décentralisation des processus métiers 22
1.3 Contribution de la thèse 23
1.4 Structure du document 24

1.1 Contexte

1.1.1 L´Entreprise Inter-Organisationnelle

Le contexte économique de ces dernières années et l'évolution continuelle des technologies de l'information incitent de plus en plus les entreprises à entreprendre de profondes mutations en adoptant de nouvelles structures d'organisation, les rendant plus concurrentielles, plus modulaires et plus aptes à s'adapter aux changements technologiques et d'organisations, ainsi qu'à recourir à des activités externes [BAGS02]. En effet, des restructurations telles que divisions, fusions ou acquisitions sont nécessaires à leur évolution. Les entreprises ont besoin d'activités périphériques et non métier, nécessitant des ressources spécifiques que d'autres entreprises détiennent. Ces réorganisations amènent à la création d'*entreprises inter-organisationnelles*.

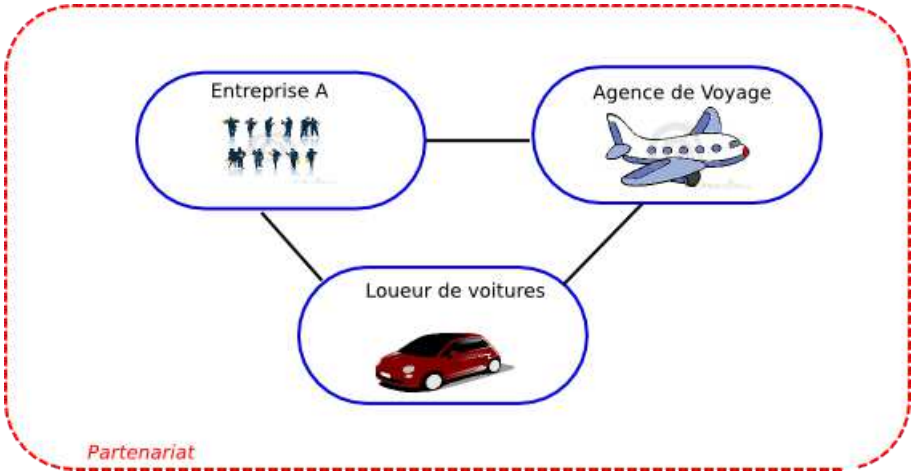


FIG. 1.1 – Un exemple d'entreprise inter-organisationnelle

Définition [*entreprise inter-organisationnelle*] : Une entreprise est dite inter-organisationnelle si elle construit son offre (de produits ou de services) en s'appuyant délibérément et durablement sur d'autres entreprises dont elle mobilise les ressources et les compétences, tout en partageant des ressources à travers le monde, essentiellement par le biais de réseaux de communications tels que l'Internet.

Le *Système d'Information* (SI) de l'entreprise est donc amené à utiliser des services extérieurs à celle-ci : les entreprises se développent par delà leurs frontières, en cherchant "dehors" des compétences et des savoir-faire que le "dedans" n'est plus apte à fournir de manière efficiente [PP06]. On assiste ainsi à une évolution flagrante des technologies de l'information et de la communication : les entreprises dépassent leurs frontières afin d'établir des relations inter-entreprises et des partenariats et afin de fournir des services à valeur ajoutée à leurs utilisateurs. Les infrastructures des Systèmes d'Information sont conçues en prenant en compte de nouvelles offres telles que les *clouds* [VRMCL08] hybrides fusionnant des ressources internes à des services à haute valeur ajoutée fournis par des entreprises extérieures. L'exemple le plus marquant de cette évolution se réalise au niveau de la gestion des parcs de serveurs de calcul : les entreprises délaissent leurs coûteux *datacenters*[1] privés pour des solutions virtualisées externes telles que Amazon EC2 [2] ou Windows Azure [3]. La raison principale de cette évolution est une maîtrise des coûts au plus juste, minimisant les investissements matériels en amont.

La façon d'organiser le travail dans ces entreprises est un facteur majeur de réussite, de performance et de compétitivité : le fonctionnement de l'entreprise requiert une planification rigoureuse des activités sous peine de ne pouvoir satisfaire les demandes des clients.

La Figure 1.1illustre la réalisation d'un partenariat entre une entreprise, une agence de voyages et un loueur de véhicules. Afin de se focaliser sur les activités propre à son métier, externalise le traitement des réservations de voyages et de véhicules à deux entreprises

[1] centres de données
[2] http ://aws.amazon.com/fr/ec2/
[3] http ://www.microsoft.com/france/windows-azure/

externes, créant ainsi un partenariat.

1.1.2 Des processus métiers basés sur l'Architecture Orientée Services

Ces dernières années, l'Architecture Orientée Services (AOS ou SOA[4]) est devenue l'architecture standard adoptée par de nombreuses entreprises. La SOA n'est pas seulement une mode technologique mais aussi **une approche de conception** qui vise à organiser le parc informatique existant de manière à transformer un environnement hétérogène composé de systèmes et d'applications complexes distribués, en un réseau de ressources intégrées, simplifiées et offrant une souplesse maximale [Coi08]. **La SOA définit aussi et approvisionne l'infrastructure**, ce qui permet aux différentes applications d'échanger des données et de participer à des processus métiers.

> **Définition [*Processus métier*] :** *Un processus métier, ou workflow*[a]**,** est un ensemble d'activités qui s'enchaînent de manière chronologique pour atteindre un objectif, généralement délivrer un produit ou un service, dans le contexte d'une organisation de travail. De façon plus pratique, le *workflow* décrit le circuit de validation, les tâches à accomplir entre les différents acteurs d'un processus, les délais, les modes de validation, et fournit à chacun des acteurs les informations nécessaires pour la réalisation de sa tâche. Il permet généralement un suivi et identifie les acteurs en précisant leur rôle et la manière de le remplir au mieux

[a] La composition des services a plusieurs similarités avec la technologie des workflows. Les deux ont pour but de spécifier le processus métier par la composition des entités autonomes et de forte granularité. Leur différence réside dans la nature de l'entité. Dans le cas des *workflows*, les entités sont des applications conventionnelles, dans celui des services, les entités sont uniquement des services. Dans la suite de ce manuscrit, par abus de langage, nous utiliserons le terme de workflow pour désigner aussi une composition de services ordonnée dans le temps.

Si une application ou un client requièrent des fonctionnalités et qu'aucun service n'est apte à les fournir seul, la ***composition de services*** permet de combiner des services existants afin de répondre aux besoins métiers de cette application ou de ce client, créant ainsi un nouveau processus métier. Même si un processus peut invoquer des services distribués et localisés sur des serveurs différents, la logique d'enchaînement des activités de ce processus, est en général centralisée. Ainsi, il existe un point central qui est en charge de gérer les activités du processus ainsi que leur ordre d'exécution. C'est ce point central qui gère le ***flux de données***, c'est-à-dire les données échangées (variables globales, résultats d'invocation de services, ou encore résultats intermédiaires. . .).

Le contrôle du flux de données consiste à spécifier et à configurer les interactions d'un ensemble de composants logiciels en considérant la propagation de l'information dans une composition qui les utilise.

Dans le cadre de la SOA, **les services sont en général faiblement couplés**, de façon à ce qu'ils puissent être modifiés, remplacés ou retirés rapidement sans impacter le processus global, obéissant à des formats d'échange standardisés. La méthodologie de la SOA et sa standardisation permettent aux entreprises qui s'y conforment d'avoir accès au bouquet de services offert par d'autres entreprises.

Grâce aux infrastructures de type SOA et aux processus métiers, les entreprises automatisent les interactions Business-2-Business (B2B) ainsi que Business-2-Consumer

[4] pour Service Oriented Architecture

(B2C). À l'intérieur et au-delà de ses frontières, l'entreprise doit pouvoir connecter les processus, les personnes et les informations. Si il a été conçu selon le principe de la SOA, le SI peut être vu comme un ensemble de services coopérant pour mener à bien les actions relatives au métier de l'entreprise, services qui peuvent interagir entre eux par le biais de réseaux publics du type Internet.

1.2 Problématique : Orchestration dynamique et décentralisée des processus inter-organisationnels

1.2.1 Processus métiers distribués à travers le réseau

Dans le cadre des entreprises virtuelles, le SI devient lui-même *inter-organisationnel*, les processus métiers, et donc les services s'exécutant dans ce SI deviennent eux aussi inter-organisationnels.

> **Définition [*Système d'Information Inter-Organisationnel*]** : Un *Système d'Information Inter-Organisationnel (SIIO)* a pour fonction particulière de supporter des processus qui traversent les frontières d'une organisation. Il peut ainsi supporter des processus interconnectés entre deux organisations[AD02].

Les processus métiers, répartis à différents endroits s'appellent alors des ***processus inter-organisationnels***. Le but d'un processus inter-organisationnel est de supporter la coopération entre des processus métiers, potentiellement hétérogènes, s'exécutant dans des organisations différentes, et cela pour atteindre un but commun. Le processus peut également contenir des parties qui peuvent être **actives en parallèle et non uniquement séquentiellement**. Chaque partenaire peut modifier son processus dans la mesure où il ne modifie pas les protocoles de communication (connexion, communication, échange de données) établis dans le partenariat [AD02].

La Figure 1.2 montre l'exemple d'un processus d'organisation de voyage s'exécutant sur les différents sites impliqués dans le partenariat illustré dans la Figure 1.1. La fonctionnalité du processus global, disponible via un portail accessible par les employés d'une entreprise, est de fournir les disponibilités en terme de voyage, d'hébergement et de véhicule. Le processus est divisé en cinq sous-processus qui s'exécutent sur les sites de chaque partenaire impliqué dans le processus. Par exemple, le processus *SP3* s'exécute sur le site de l'agence de voyage et fait usage d'un service qui vérifie la disponibilité d'un vol à une date donnée et d'un service qui vérifie la disponibilité d'une chambre d'hôtel. Ces deux services ne sont accessibles uniquement depuis l'agence de voyage, ce qui justifie la présence du sous-processus *SP3* sur le site de cette agence. Les sous-processus de cet exemple coopèrent, en s'échangeant les données applicatives nécessaires (ici une date de départ, une date de retour, et une localisation), afin de réaliser le but du processus qui est de renvoyer un compte-rendu de disponibilité de voyage.

Dans le contexte de la SOA, les concepteurs sont amenés à composer des processus métiers, formant ainsi de plus larges processus ; on peut alors parler de *megaprogramming* [PA05a]. Cette approche de composition peut être caractérisée comme une *approche ascendante*, qui consiste à définir un processus commun réunissant plusieurs partenaires : à partir d'un ensemble de compositions de services donné, on peut concevoir une composition globale.

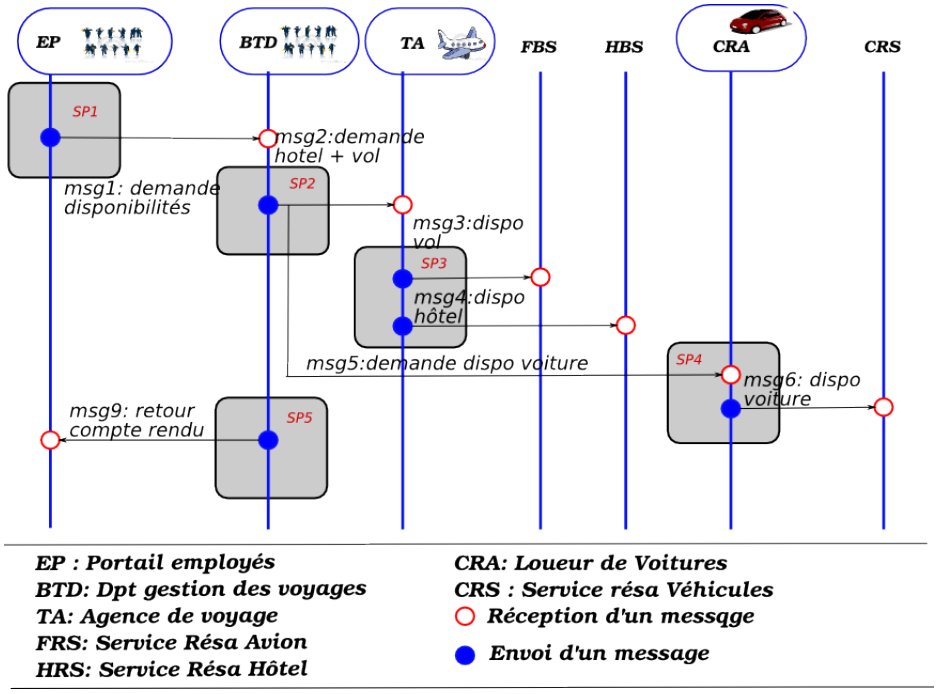


FIG. 1.2 – Un processus distribué sur plusieurs sites

Inversement, pour des raisons de gain de performance ou encore de sécurité (par exemple, la non transmission de données en dehors des limites de l'entreprise) [YG07], on peut aussi adopter une *approche descendante* qui consiste à définir un processus métier commun réunissant les différents partenaires et d'établir les connexions entre-eux : une composition de services globale est partitionnée afin d'en obtenir des *fragments* que l'on va exécuter de manière distribuée au travers des différents participants. De nombreux travaux ont été menés dans le but de décentraliser une orchestration de services en produisant un ensemble de sous-orchestrations collaborant entre-elles[YG07, NCS04, BMM05].

Cette thèse se place dans le contexte de l'exécution des orchestrations de processus métiers décentralisés et répartis à large échelle. Une partie de notre travail s'inscrit dans le contexte du projet FP7 NESSI IP SOA4ALL [soa], qui a pour objectif général d'offrir une architecture orientée services capable de *"supporter des millions de services composables dynamiquement, à l'échelle de l'Internet, avec l'ambition de réussite comparable qu'a dorénavant le WEB"*. Nous nous plaçons donc dans un contexte où il existe potentiellement un très grand nombre de services, de processus, et d'utilisateurs.

Se préoccuper de l'aspect décentralisé des processus métiers requiert de bien cerner les bénéfices et contraintes, ce que nous abordons dans la suite de cette section.

1.2.2 Bénéfices d'une approche distribuée pour l'exécution d'un processus métier

Plusieurs raisons peuvent pousser à adopter une approche distribuée et dynamique pour exécuter un processus métier [CCMN04] :

- **Éviter les goulots d'étranglement** : Les moteurs d'orchestration, tels que ceux qui orchestrent les processus décrits en WS-BPEL [AAA+06], agissent comme des coordinateurs centraux pour toutes les interactions parmi les services impliqués dans un processus. Tandis que cette approche donne un contrôle complet sur l'orchestration à une seule entité (qui peut par exemple, surveiller et administrer le processus), elle mène la plupart du temps à des communications inefficaces, les résultats intermédiaires étant renvoyés vers la localisation du moteur centralisé, ce qui peut constituer un goulot d'étranglement.
- **Placer les fragments de processus au plus près des services**, de manière à réduire les transmissions de données et les communications ou tout simplement de manière à pouvoir l'utiliser, en se plaçant, par exemple, dans un domaine sécurisé.
- **Gérer efficacement la charge** : L'exécution de l'orchestration peut générer une charge importante résultant du nombre d'utilisateurs, donc du nombre d'instances de processus à créer. Cette charge de travail peut affecter les serveurs d'orchestration ainsi que le réseau de communication.
- **Ré-utiliser les compositions existantes :** Les processus métiers sont distribués par nature dans le cas où ils sont inter-organisationnels : le processus global est alors une composition de processus métiers définis et s'exécutant localement dans chaque entreprise partenaire.

1.2.3 Contraintes liées à la décentralisation des processus métiers

Pour bénéficier de ces avantages, le système qui exécute ces orchestrations doit gérer et obéir à des contraintes :

- **L'hétérogénéité des processus et de leur système d'exécution** : Dans une entreprise virtuelle, la difficulté de l'intégration des différents systèmes de chacun des partenaires d'une entreprise virtuelle vient en grande partie du fait que ces systèmes sont hétérogènes. Le système inter-organisationnel global doit donc pouvoir gérer différents moteurs d'exécution de processus métiers. Bien que ce type d'hétérogénéité ait plus de chance d'exister entre différentes entreprises (dans le cadre d'un partenariat, par exemple) il n'est pas rare de le trouver à l'intérieur d'une même entreprise, particulièrement si celle-ci a fusionné avec d'autres.
- **La dynamicité des processus métiers :** Un des challenges émergeant de la gestion des processus métiers est la dynamicité au cours de leur exécution. La dynamicité est une propriété qui permet qu'une nouvelle activité dans un processus puisse être définie et déployée, ou qu'une activité existante soit modifiée ou retirée. La dynamicité est aussi à prendre en compte lorsque le processus est décentralisé, par exemple lorsque l'on doit remplacer un sous-processus par un autre. Un concepteur ou administrateur de processus métiers a besoin de rendre possible la mise-à-jour (automatique ou non) des processus sans pour autant arrêter le système entier. Cette mise-à-jour peut concerner par exemple un changement de service externe pendant l'exécution du processus ou le remplacement d'un fragment du processus par un autre (une nouvelle version d'un fragment corrigeant un mauvais comportement par exemple). Dans le cas de processus s'exécutant dans le cadre de transactions de longue durée (s'étalant sur des mois), il est primordial de pouvoir effectuer une reconfiguration de ce processus sans arrêter le système entier et sans recommencer l'exécution du processus entier. Idéalement, on pourrait s'attendre à ce que ces modifications dynamiques se réalisent de manière automatique.
- **Gérer le flux de données entre processus :** Les différents fragments d'un proces-

sus inter-organisationnel se transmettent le flux de données, propageant les données fonctionnelles traitées par le processus global. Le flux de données, par exemple, peut contenir des données concernant l'initiateur du processus (par exemple des données personnelles provenant d'un formulaire). Il est alors important de pouvoir transmettre ces données de fragment en fragment afin de réduire les communications, et les transferts de données, sans pour autant faire appel au point central qu'est l'orchestrateur : les données doivent être récupérées directement là où elles sont produites et traitées là où elles sont nécessaires.

Définition[*Flux de données*] : Ensemble des informations et données utiles à l'exécution d'une instance de processus circulant d'un point à un autre, dans un ordre chronologique.

- **Administrer et surveiller les processus métiers distribués** : En général, les processus métiers, distribués ou non, une fois exécutés, sont analysés et améliorés par leur concepteur dans le but d'améliorer la Qualité de Service (QoS pour Quality of Service). Dans le cadre des processus métiers distribués, il est important de les analyser dans leur globalité par rapport à leur exécution. Par exemple, un concepteur peut vouloir connaître l'endroit où un processus échoue, ou encore le temps de réponse d'un service donné.
  Ce type d'analyse peut être difficile à réaliser et être fastidieux sur un processus décentralisé au travers de plusieurs entreprises et la plupart du temps ces données sont collectées manuellement sur chaque système. Dans un système inter-organisationnel, le concepteur a besoin de manipuler une vue globale et unifiée des processus métiers. Ce besoin d'administration et de surveillance est intimement couplé au besoin d'agilité et de flexibilité du service, la fonction de surveillance du processus étant un des facteurs pouvant déclencher une reconfiguration de processus.
- **Gérer la complexité du déploiement de l'ensemble des parties du workflow** : Avant d'être exécuté, le processus décentralisé va être associé à plusieurs nœuds d'exécution, et donc à plusieurs moteurs d'exécution des processus. Il faut alors, une fois le processus global défini, pouvoir gérer la complexité du déploiement de l'ensemble des sous-workflows et de leur moteur d'exécution.

1.3 Contribution de la thèse

Dans cette thèse, nous nous intéressons à la résolution des problèmes liés à l'exécution distribuée et dynamique des orchestrations de services, et ce, en adoptant une approche à composants.

L'ingénierie basée sur les composants (CBSE[5]) et les architectures basées sur les services font partie des styles d'architectures les plus mises en avant au sein des entreprises de nos jours. Dans la littérature, ces deux approches apparaissent souvent comme rivales. Cependant, comme l'évoquent [CCC+07], nous pensons également qu'il est bénéfique de marier ces deux approches qui apparaissent complémentaires, chacune d'entre elles apportant des bénéfices à l'autre : les composants sont plus maniables et reconfigurables que les services, qui eux prônent l'approche SOA. Le composant bénéficie de propriétés non-fonctionnelles et s'exécute dans un environnement contrôlé, ce qui facilite l'adaptation de l'orchestration, tout en bénéficiant de propriétés comme le couplage faible et l'interopérabilité.

[5]Component-Based Software Engineering

Nous mettons en avant dans cette thèse **une vision "composant distribué" correspondant à une orchestration de services**. Cette vision nous permettra de manipuler la vue globale distribuée des processus métiers inter-organisationnels, tout en prenant en compte les contraintes énoncées précédemment.

Dans ce but, la contribution de cette thèse se décline en :

- Un **mécanisme de projection d'une orchestration de services sur un composant**. Nous prenons comme support de base à nos travaux la spécification GCM[6][BCD+09], une extension distribuée pour la grille du modèle à composants Fractal[BCL+]. Cette spécification est suffisamment générique pour être utilisée dans le cadre d'applications à base de services. Nous utilisons GCM pour modéliser un workflow par un ensemble de composants connectés entre-eux par une relation temporelle. Ceci reflète la traduction d'un workflow dans un composant où les préoccupations fonctionnelles (la logique métier) et non-fonctionnelles (la gestion de l'exécution temporelle ou les mécanismes de surveillance) sont séparées.
- Une **vue unifiée et globale d'une orchestration distribuée.** L'approche que nous présentons est indépendante de tout langage d'orchestration en particulier et suffisamment générique pour être appliquée à différentes technologies. Il est alors possible avec notre solution de faire collaborer plusieurs moteurs d'orchestration pour atteindre un but (dans une entreprise virtuelle par exemple). Nous modélisons cette collaboration (fédération ou partenariat...) par un composant composite GCM, permettant ainsi d'en obtenir une vue globale, sans se soucier de la localisation des différentes parties de l'orchestration.
- Un **environnement d'exécution d'orchestrations distribuées et dynamiques.** Les composants GCM sont distribués par nature et le modèle fournit une structure hiérarchique, un composant pouvant inclure d'autres composants dans sa partie fonctionnelle. Par ailleurs, utiliser GCM permet d'obtenir un support d'exécution embarqué dans le composant : le composant est lui-même le support d'exécution du workflow, le moteur d'interprétation de workflow étant embarqué dans sa partie non-fonctionnelle. D'un point de vue technique, notre solution s'appuie sur l'intergiciel GCM/ProActive, qui nous fournit un support pour la reconfiguration dynamique, et qui fournit un mécanisme de déploiement des composants distribués.
- Une **évaluation de notre proposition basée** sur un cas d'utilisation concret de déploiement d'applications sur un parc multi-domaines de passerelles OSGi, basé sur le concept de *plan de déploiement*, que l'on peut traduire par un processus métier.

1.4 Structure du document

En comptabilisant cette introduction, ce rapport de thèse est composé de 7 chapitres :

- Le **chapitre 2** présente le contexte de la thèse et les principes de base sur les architectures orientées services ainsi que sur les systèmes et langages d'exécution d'orchestration de service. Nous introduisons les notions de services et d'architectures à services. Nous présentons ensuite les compositions de services en nous concentrant sur deux modèles de composition particuliers que sont la composition temporelle et la composition structurelle.
- Le **chapitre 3** présente l'approche des orchestrations distribuées et dynamiques, les problématiques liées, et les différents systèmes existants pour la mise en œuvre de ce type d'orchestration. Nous présentons une comparaison entre les différentes fonc-

[6] Grid Component Model

tionnalités qu'ils fournissent et nous les classifions selon différents critères que nous aurons définis auparavant.
- Le **chapitre** 4, présente une synthèse de l'état de l'art et positionne notre travail par rapport à cette synthèse. Nous décrivons ici les problèmes que nous nous sommes employés à résoudre dans le cadre de ce travail.
- Le **chapitre 5** présente le concept de projection d'une orchestration distribuée vers un composant, qui constitue la base de notre contribution. Nous explicitons ici les notions de composition dans le temps et composition dans l'espace et montrons comment, grâce à la combinaison de ces deux concepts, nous pouvons obtenir une composition répartie et dynamique.
- Le **chapitre 6** présente l'implémentation de notre modèle, basé sur GCM/ProActive, en détaillant les éléments nécessaires à la mise en œuvre d'une projection d'orchestration de services sur un composant distribué.
- Le **chapitre** 7 présente des évaluations expérimentales que nous avons réalisées avec notre implémentation, et décrit un cas d'utilisation de notre environnement d'exécution. Nous y décrivons les expérimentations que nous avons réalisées dans le cadre de déploiement d'applications sur un parc multi-domaines de passerelles OSGi.

Finalement, nous terminons par le **chapitre 8** qui conclut sur le travail réalisé pendant cette thèse, tout en décrivant les perspectives d'évolution du modèle.

Rendre service de tout son pouvoir, de toutes ses forces, il n'est pas de plus noble tâche sur la terre.

Sophocle, Extrait de *Oedipe Roi.*

Chapitre 2

Contexte : Les architectures orientées services

Contenu du chapitre :

2.1 Services et Architectures Orientées Services . . . 28
2.1.1 Services : De multiples définitions . . . 28
2.1.2 Le paradigme de programmation basée sur les services (SOC) . . . 29
2.1.3 Les Architectures Orientées Services . . . 31
2.1.4 Mise en œuvre d'une SOA et plateformes à services . . . 33
2.1.5 Conclusions sur l'approche à services . . . 40
2.2 Composition des services . . . 40
2.2.1 Cycle de vie d'une composition de services . . . 41
2.2.2 Le contrôle de la composition . . . 42
2.2.3 Composition dans le temps . . . 42
2.2.4 Composition structurelle : Les composants orientés services . . . 44
2.2.5 Conclusions sur la composition de services . . . 51
2.3 Synthèse du chapitre . . . 51

Le but de ce second chapitre est de présenter les différents concepts généraux qui sont en relation avec le contexte scientifique dans lequel s´inscrit cette thèse. Derrière le terme d'orchestration de services se cachent de nombreux concepts, notamment ceux de *services* et d'approche à service (en anglais SOC acronyme de *Service Oriented Computing*). L'approche à services est un paradigme informatique dont le but est l'augmentation de la modularité des applications composées d'entités logicielles que l'on appelle *services* et la réduction du couplage entre ces mêmes entités logicielles.

La section 2.1 présente les services ainsi que les principes de base des architectures orientées services. Après avoir mis en avant les éléments caractéristiques de cette approche, nous étudierons dans la section 2.2, deux types de compositions de services, à savoir la composition dans le temps et la composition structurelle. Notre objectif n'est pas ici de faire un état de l'art exhaustif sur l'approche à composant, mais plutôt d'en

introduire les concepts généraux, afin de pouvoir appréhender la compréhension de la spécification SCA[1] [SCA07a] dont l'objectif est de réaliser une approche à services au travers du paradigme de composant. Finalement, nous concluons le chapitre par une synthèse des différentes approches qui soulignera leurs implications dans ce travail de thèse.

2.1 Services et Architectures Orientées Services

2.1.1 Services : De multiples définitions

Le terme de service est largement utilisé dans la vie courante. La définition de base, donnée par le dictionnaire [RR01], est la suivante :

"Ensemble organisé d'activités destinées à remplir un besoin." [RR01]

Ce terme de service prend des significations différentes selon le contexte. Ainsi, dans la vie de tous les jours, on l'utilisera pour désigner une action que l'on accomplit pour le compte d'une personne tierce afin de lui apporter de l'aide. Dans le contexte de l'entreprise ou de l'industrie, un service représente une prestation que l'on met à disposition des clients pour satisfaire un besoin. Particulièrement, dans le contexte d'applications logicielles, le service est utilisé pour désigner un type d'application qui s'exécute en arrière-plan pour fournir un support. Les services fournissent des fonctionnalités métiers, comme par exemple une application dédiée aux voyages d'affaire, ou aux crédits. Cela diffère fortement des fonctionnalités orientées-technologie comme interroger ou mettre à jour une table dans une base de données.

Le domaine de l'ingénierie logicielle est un domaine en perpétuelle évolution. Ainsi, les architectures à objets [Tay98] et les architectures à composants [SGM02] sont des approches qui ont conduit à l'apparition de l'approche à services. Bien que la programmation orientée service ait hérité de plusieurs concepts de la programmation par objets, elle diffère de façon importante de ses ancêtres : les objets et les composants sont par exemple dépendants du langage dans lequel ils sont écrits, et sont détruits localement après leur utilisation, alors que les services ne le sont pas. L'approche à service cherche à fournir un niveau d'abstraction supérieur à ceux des approches à objets et à composants : elle encapsule des fonctionnalités et permet la ré-utilisation de services déjà existants.

[Pap03] en donne la définition suivante :

"Services are self-describing, platform-agnostic computational elements that support rapid, low-cost composition of distributed applications" [Pap03]

Il met en avant qu'un service est une entité logicielle utilisable grâce à son auto-description. Le service doit alors fournir tous les éléments nécessaires à son utilisation : le consommateur du service n'a pas besoin de connaitre les détails d'implémentation du service ainsi que de son support d'exécution. Inversement, le service ne connait rien de ses utilisateurs ni de leur contexte. Cela montre une propriété importante des services, facilitant le *couplage faible* et l'intégration possible du service dans une application à large échelle. De façon complémentaire, le consortium OASIS[2], en charge de la normalisation et de la standardisation des applications sur Internet, donne également une définition du service :

[1] Service Component Architecture

[2] Organization for the Advancement of Structured Information Standards , Consortium à but non-lucratif qui pilote le développement et l'adoption de standards pour les applications internet : http ://www.oasis-open.org/home/index.php.

> *"A service is a mechanism to **enable access to one or more capabilities**, where the access is provided by using a **prescribed interface** and is exercised consistent with constraints and policies as specified by the service composition. A service is **accessed by means of a service interface** where the interface comprises the specifics of how to access the underlying capabilities. There are no constraints on what constitutes the underlying capability or how access is implemented by the service provider. **A service is opaque** in that its implementation is typically hidden from the service consumer except for*
> *(1) the information and behavior models exposed through the service interface and*
> *(2) the information required by service consumers to determine whether a given service is appropriate for their needs."* *[MLM+06]*

Ainsi d'après cette définition un service permet l'accès à des fonctionnalités par le biais de la spécification d'une interface, tout en respectant un ensemble de contraintes et de politiques d'accès. Bieber et Carpenter [BC01] insistent sur le fait qu'un service ait un comportement défini par un contrat, et dont l'utilisation est conditionnée par ce contrat : ce service est vu comme une boîte noire, l'implémentation étant cachée à l'utilisateur, la seule information qui lui est mise à disposition étant la description de l'action du service. L'utilisateur du service ne doit connaître que les détails qui lui sont utiles pour l'utilisation du service par rapport à ses besoins. L'utilisation de service est souvent conditionnée à la négociation d'un accord entre l'entité fournisseur et l'entité consommateur : avant d'utiliser un service, il est nécessaire de connaitre ses capacités fonctionnelles et ses capacités non-fonctionnelles.

Suite à ces multiples définitions, nous retiendrons dans cette thèse qu'un service est :

> *Une entité logicielle qui apporte une fonctionnalité métier, qui est autonome et indépendante de toute plateforme, accessible par une interface, qui peut être publiée, découverte, composée, orchestrée dans le but de construire des applications collaboratives, potentiellement à large-échelle.*

2.1.2 Le paradigme de programmation basée sur les services (SOC[3])

Bien que n'étant pas une démarche nouvelle (elle est apparue dans les années 1990 avec l'apparition des architectures client/serveur), l'idée d'une approche à services est de plus en plus adoptée par les entreprises pour construire des architectures adaptées aux serveurs d'application, capables d'exposer des services interopérables (patrimoniaux, progiciels) et de collaboration entre les entreprises (services métiers). Ces dernières années, les entreprises ont fait progresser cette approche pour répondre à des besoins de temps de réduction de développement des applications et à des besoins de flexibilité. L'approche à services est le paradigme architectural qui utilise le concept de service comme élément fondamental pour définir et développer des applications.

Pour construire une solution logicielle basée sur le SOC, nous avons besoin de respecter les principes de l'approche à services [Erl].

- ***Une description standardisée des services*** : La seule information partagée entre les différentes parties est la ***description de service*** dont le but principal est de fournir une spécification des fonctionnalités offertes par le service. Celle-ci définit l'ensemble des signatures des opérations proposées par le service. L'interface est un contrat entre

[3] acronyme anglais de Service Oriented Computing

le fournisseur et le consommateur, définie indépendamment de l'implémentation. Ce principe est certainement le pilier des architectures orientées services, permettant d'appliquer d'autres principes, comme le couplage faible par exemple. Les opérations des services sont définies comme des ensembles de messages. Ces messages spécifient les données à échanger et les décrivent d'une façon indépendante de la plate-forme et du langage. Les services échangent des données dans un format standardisé, contrairement aux applications objets et composants.

- ***Le couplage faible*** : Le couplage faible est une notion clé de l'approche à services. Cela permet à un service de changer (localisation, implémentation, fournisseur) sans générer d'impact sur les autres entités de l'architecture. Les services ainsi implémentés permettent de maintenir d'autres propriétés de l'approche à services telles que la ré-utilisation ou l'autonomie. Seule la description, ne contenant que l'information nécessaire utile à l'utilisation du service est partagée entre fournisseur et consommateur, est obtenue par l'inter-médiation du registre de services. En outre cette propriété permet aux compositions de services de s'organiser sous la forme d'orchestrations (nous aborderons les notions de composition de services et d'orchestration dans la section suivante), autorisant une plus grande flexibilité dans le choix des services à composer ainsi que dans l' ordonnancement de leur composition.
- ***L'abstraction :*** Ce principe met l'accent sur la nécessité de cacher le plus possible les détails d'un service. Cela permet ainsi d'établir et de préserver la relation de couplage faible.
- ***Ré-utilisabilité :*** Un service est une brique logicielle réutilisable. Elle est réalisée en distribuant la logique de l'application au travers des services de façon à ce que chaque service puisse potentiellement être utilisé par plus d'un consommateur. Grâce à cette propriété les services dans une SOA sont ***composables.*** Ils peuvent ainsi être regroupés dans des services composites qui coordonnent les échanges de données entre les services impliqués.
- ***Autonomie :*** Les services ne contrôlent que la logique métier qu'ils encapsulent.
- ***Sans État*** : Bien qu'il n'est pas interdit que les services aient un état, il est conseillé qu'ils respectent la propriété d'***idempotence*** : l'exécution d'un service ne doit pas dépendre d'un état antérieur, ce qui n'empêche pas le service d'interagir avec des données persistantes qui peuvent influencer son comportement. Les services sont sans état : ils ne maintiennent pas d'état spécifique à une exécution.
- ***Découverte*** : Il est nécessaire de se reposer sur un mécanisme qui permet à un utilisateur de trouver des services répondant à ses besoins.
- ***Composabilité :*** Basé sur le principe de ré-utilisabilité, le principe de composabilité représente la faculté pour les services d'être regroupés dans des services composites qui coordonnent un échange de données entre-eux.
- ***L'interopérabilité :*** L'interopérabilité des services est facilement réalisable comme les services interagissent entre-eux à travers les interfaces qui sont indépendantes vis-à-vis des plateformes et des langages d'exécution.
- ***La Liaison retardée*** : À la différence des modèles à composants où l'architecture et les liaisons sont définies à l'avance, l'approche à services permet de réaliser la liaison entre entités logicielle au moment de l'exécution. L'environnement créé par cette architecture est dit dynamique : les descriptions peuvent être publiées ou retirées à tout moment. La liaison d'un consommateur vers un fournisseur peut donc varier selon l'exécution. Le consommateur peut invoquer des services implantés différemment d'une exécution à une autre, et même au cours de la même exécution. La liaison retardée vers un service est une propriété importante de la SOA qui rend l'architecture

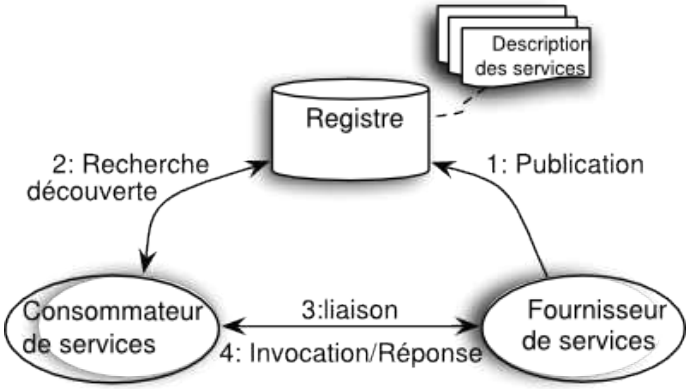


FIG. 2.1 – Les Rôles dans une Architecture Orientée Services

cliente re-configurable à l'exécution.

Comme on peut le constater, la plupart de ces principes sont intimement liés : par exemple, si l'on garde en tête le principe d'autonomie lorsque l'on divise la logique de l'application en services, on obtient des entités logicielles ré-utilisables, composables et faiblement couplées, et qui pourront ainsi être utilisées de nouveau dans des projets futurs. Inversement, si on ne respecte pas au moins un principe de la SOA, il sera difficile d'assurer les autres. Par exemple, si on ignore le principe d'idempotence d'un service au moment de la conception, on obtiendra des briques logicielles moins réutilisables et moins composables pour construire une application.

2.1.3 Les Architectures Orientées Services

La SOA est une façon de bâtir des solutions à base de services. Ce style d'architecture est une approche évolutive qui contribue à mettre en place un système d'information flexible, assurant ainsi une interopérabilité intrinsèque et une meilleure agilité pour l'entreprise. Il est difficile de trouver une définition universelle des Architectures Orientées Services. Nous trouvons dans la littérature différentes définitions telles que :

- Celle publiée par le Gartner Group : "*A service-oriented architecture is a style of* ***multi-tier computing*** *that helps organizations share logic and data among multiple applications and usage modes.*" [SN96]
- Celle donnée par le Consortium OASIS : "*Service Oriented Architecture is a paradigm for organizing and utilizing distributed capabilities that may be under the* ***control of different ownership domains****. It provides a uniform means to* ***offer****,* ***discover****,* ***interact*** *with and use capabilities to produce desired effects consistent with measurable preconditions and expectations.*" [MLM+06]
- Celle donnée par Dodani et al : *"SOA enables flexible integration of applications and resources by: (1) representing every application or resource as a service with standardized interface, (2) enabling the service to exchange structured information(messages, documents, "business objects") and (3) coordinating and mediating between the services to ensure they can be invoked, used and changed effectively.*" [Dod04]
- Celle donnée par Microsoft : "*The* ***policies****,* ***practices****,* ***framework****s 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.*" [SW04]

En résumé, ces définitions s'accordent à dire que les architectures à services respectent un protocole bien défini entre différents rôles, comme illustré sur la Figure 2.1.

Dans un premier temps, (1) l'entité qui veut offrir une capacité, le *fournisseur de services* publie une description de service dans une autre entité appelée *registre de services* (ou annuaire). Le registre de services stocke des descriptions de services et permet d'effectuer des recherches sur ces dernières. Dans un second temps, (2) une entité qui veut *utiliser* un service, le *consommateur de services*, interroge le registre de service et *découvre*, via le registre, les services disponibles qui correspondent à sa requête. Grâce à cette information, (3) le consommateur de service peut *trouver* le fournisseur de services, s'y connecter et initier (4) une communication. Le registre de services agit comme un courtier de services qui permet aux consommateurs et fournisseurs de se mettre en relation.

2.1.3.1 La SOA étendue

La SOA de base n'apporte pas de solutions sur les aspects transverses tels que l'administration ou l'orchestration des services. [Pap03] propose une possible organisation d'une SOA étendue (ESOA[4]) en plusieurs niveaux, illustrée sur la Figure 2.2. Cette organisation comprend trois niveaux :

- ***Les services de base*** : C'est le niveau le plus bas, qui comprend la base de la SOA, dont les mécanismes ont été décrits dans les sections précédentes. Ce niveau prend en charge les mécanismes de base tels que ceux qui permettent la publication, la découverte et les interactions avec les services.
- ***Les services composés*** : C'est le niveau dans lequel les services sont composés à partir des services de base du niveau le plus inférieur. Dans ce niveau apparait un nouveau rôle, celui d'agrégateur de services. Un agrégateur de services est un fournisseur de services qui consolide et compose les services qui sont fournis par d'autres fournisseurs de services, en créant un service à valeur ajoutée. Les services résultant de cette composition peuvent être considérés comme des services à part entière et être agrégés dans d'autres compositions ou être utilisés directement et publiés dans le but d'être utilisés par des consommateurs. Les agrégateurs de services deviennent alors à leur tour fournisseurs de services, offrant une description du service nouvellement créé.
- ***Les services administrés*** : Ce niveau offre la possibilité aux services (qu'ils soient composés ou non) d'être administrés au travers d'un environnement distribué et surtout hétérogène. Afin d'optimiser le rendement de leurs applications, les entreprises ont besoin d'obtenir des métriques concernant leurs applications (performances minimales et maximales, moyennes, état global, état de chaque service).

L'environnement étendu de la SOA peut comprendre aussi des mécanismes additionnels, assurant la prise en charge des mécanismes non-fonctionnels tels que la sécurité, la gestion des transactions ou encore la qualité de services. Ces mécanismes non-fonctionnels servent de base à l'établissement d'un contrat d'utilisation du service. Ce contrat spécifie non seulement la fonctionnalité fournie par le service, mais aussi des aspects non-fonctionnels mais contractuels qui définiront par exemple le tarif du service en fonction du niveau de performance minimal qui peut être basé sur le temps de réponse garanti du service.

[4] Extended SOA

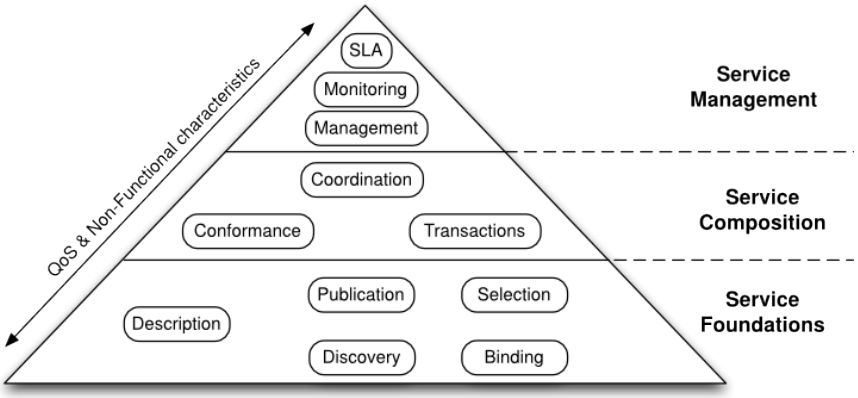


FIG. 2.2 – Architecture Orientée Service Etendue [PTDL07]

2.1.4 Mise en œuvre d'une SOA et plateformes à services

Dans le but d'assurer la réalisation d'une SOA, un système doit mettre en œuvre un environnement d'exécution et d'intégration des services, capable de gérer les interactions entre les différents rôles définis précédemment. Il est important de noter que la SOA, qui fournit une méthodologie pour construire des solutions orientées services, n'est pas liée à une technologie en particulier. Plusieurs implémentations de l'approche à services existent actuellement pour mettre en œuvre une SOA, telles que : OSGi (services co-localisés sur une même machine virtuelle), Jini, UPnP (services répartis dans des réseaux ad-hoc). Cependant, la technologie la plus utilisée et la plus complète en terme d'environnement, qu'est celle des Services Web, est celle que nous avons expérimenté au cours de cette thèse et que nous présentons dans la suite.

Dans la suite de cette section, nous présentons les concepts des Services Web, utilisés dans les travaux de cette thèse pour mettre en œuvre notre approche, ainsi que la méthodologie des services RESTful. Nous détaillerons ensuite le fonctionnement d'un ESB (Enterprise Service Bus) qui permet d'intégrer les Services Web. Nous terminons cette section en introduisant la fédération de bus à services développée dans le projet SOA4LL.

2.1.4.1 Services Web

Connue sous le terme de "Web Service-Oriented Architecture" (WSOA), la technologie des Services Web concrétise l'approche à service et fournit une manière efficace de partager la logique applicative au travers de plusieurs machines hébergeant des environnements d'exécution hétérogènes. Pour réaliser cela, les Services Web utilisent SOAP, WSDL, XML Schema et d'autres technologies basées sur XML, fournissant ainsi une approche standard permettant de franchir la barrière de l'hétérogénéité des plateformes et des langages. Grâce à cette technologie, il devient possible de délivrer et de consommer des services logiciels sur Internet. Un Service Web est une application logicielle rendue disponible à travers Internet qui peut être décrite (grâce à WSDL), publiée et découverte (grâce à UDDI), et invoquée (grâce à SOAP). Elle peut être configurée en utilisant des standards reposant sur XML. Afin d'être atteignable, un Service Web possède une adresse (URI).

Le W3C définit un Service Web comme suit :

> *"A web service is a software system designed to support interoperable machine-to-machine interaction over a network. It has an interface described in a machine-processable format(specifically WSDL). Other systems interact with the Web Service in a manner prescribed by its description using SOAP-messages, typically conveyed using HTTP with an XML serialization in conjunction with other Web-related standards".*

Communication Un Service Web est, du point de vue d'un client, une boite noire, cachant les détails de son implémentation et exposant un ou plusieurs ports qui sont les points d'accès au service. Chaque port contient un ensemble d'opérations qui fournissent et réalisent la fonctionnalité du service. Il reçoit une requête de la part d'un client, requête qu'il traite avant de renvoyer la réponse. La communication avec un Service Web peut se faire en utilisant deux modes de communication différents [PD04] :

1. *synchrone* : Lors de l'utilisation d'un mode synchrone, les clients envoient leur requête comme un appel de méthode et sont bloqués avant de continuer leur exécution tant que la réponse du service n'est pas reçue. Un exemple d'utilisation possible de ce mode de communication serait de demander un prix pour un produit donné ou encore de demander les prévisions météorologiques d'un lieu donné.
2. *asynchrone* : Lors de l'utilisation d'un mode asynchrone, les clients envoient au service un message (appelé aussi document) plutôt qu'un ensemble de paramètres. Une fois le message reçu, le service le traite. Le client ne reste pas bloqué durant son exécution ; la réponse pouvant revenir après un long délai, dans le cas d'un ordre de commande, par exemple.

Le concept des Services Web repose sur quatre technologies dans le but d'implémenter une SOA : (1) XSD pour décrire les structures de données, (2) WSDL pour décrire les interfaces des services, (3) SOAP pour communiquer et (4) UDDI pour publier et découvrir des services.

Décrire les données : XSD XSD[5] est un métalangage XML utilisé pour formaliser la structure d'un document XML. Dans le contexte des Services Web, il est utilisé pour spécifier les structures de données échangées entre services.

Décrire les fonctionnalités avec WSDL [6] La clé de la réalisation d'un couplage faible entre fournisseur et consommateur est la séparation de la description d'un service de son implémentation. En décrivant dans un format standard les fonctionnalités du service, WSDL [CCMW01], Web Service Description Language, permet de réaliser l'abstraction des différents langages de programmation utilisés pour implémenter un service, le consommateur du service ne devant pas connaître ces détails d'implémentation. Pouvant être considéré comme une évolution du langage IDL de CORBA [EAS08], WSDL est le langage utilisé pour décrire de manière standard les interfaces des Services Web.

WSDL est utilisé pour : (1) Décrire la fonctionnalité du service en terme d'opérations disponibles, (2) spécifier la façon dont on peut utiliser le service en terme de protocoles de communication et (3) indiquer la localisation du service. Par contre, les propriétés non-fonctionnelles d'un service ne sont pas prises en compte. Il existe pour cela des mécanismes proposant des extensions tels que WS-Policy [VOH+07].

[5] XML Schema Description Language
[6] Web Service Description Language

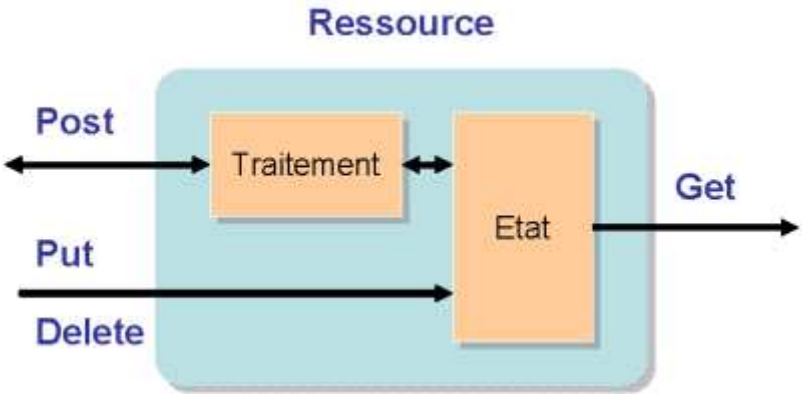


FIG. 2.3 – Une ressource RESTful et l'utilisation des opérations HTTP permettant de la manipuler

Communiquer avec SOAP SOAP[7] [GHM+02] est le protocole d'échange de messages permettant d'interagir avec les Services Web. Ces messages sont envoyés sous la forme de documents XML structurés et sont transportés sur le réseau en utilisant des protocoles standards. Le protocole de transport le plus utilisé est HTTP qui a pour avantage d'être utilisé largement sur Internet, ainsi que d'être indépendant de la plateforme

Les messages sont structurés dans un document XML et sont composés d'une enveloppe obligatoire contenant une entête optionnelle et un corps obligatoire. L'entête permet d'ajouter des extensions à un message, comme par exemple des propriétés transactionnelles ou encore des propriétés de routage du message. Le corps du message contient l'information nécessaire à l'appel du service (l'opération à invoquer et les paramètres à lui passer).

Publier avec UDDI UDDI[8] [CHvR+04] est un standard qui supporte la publication et la découverte de services dans une WSOA. Ce standard spécifie comment le composant d'annuaire doit être mis en place pour la plateforme des Services Web. Un registre UDDI publie les interfaces des Services Web destinées aux consommateurs de services, en se basant sur les descriptions de services spécifiées en WSDL. Indépendant de la plateforme d'exécution des services, il contient des informations concernant les fournisseurs de Services Web ainsi que des méta-données à propos des services. UDDI spécifie plusieurs API pour interagir avec le registre, demander ou publier un service.

2.1.4.2 Les services RESTful

Un des modèles d'implantation des services se repose sur l'architecture REST(REpresentational State Transfer) [Fie00]. Ce style d'architecture fait largement usage du protocole HTTP couramment utilisé par les internautes. Il a notamment pour particularité de chaque requête est sans état, dans le sens où elle ne dépend pas d'un état antérieur de l'interaction et qu'elle repose uniquement sur les données transmises, lesquelles peuvent faire référence à des ressources accessibles par une URL.

[7]Simple Object Access Protocol
[8]Universal Description Discovery and Integration

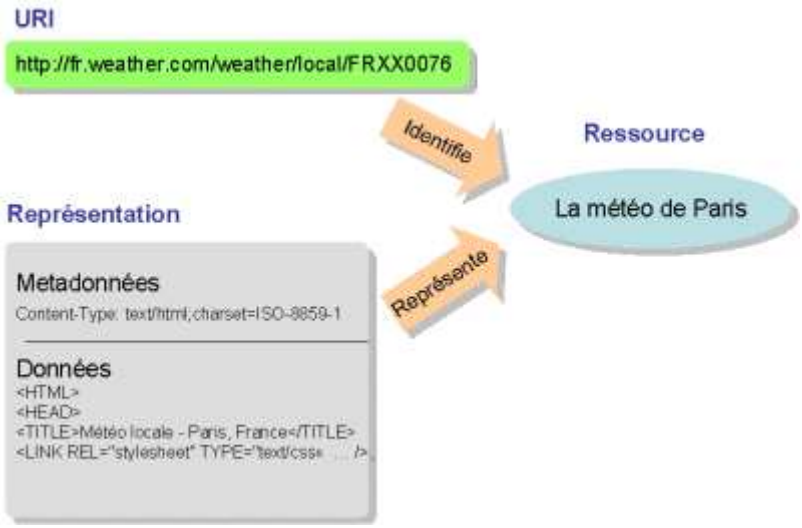


FIG. 2.4 – Un exemple de ressource REST

Dans cette architecture, un client lit ou modifie une ressource en accédant à une représentation de cette ressource. Une ressource est un objet nommable qui peut évoluer avec le temps.

Bien que non standard, la méthodologie REST repose sur les standards du web tels que :

- **URI**[9] comme syntaxe universelle pour accéder aux ressources
- **HTTP ou HTTPS**, protocole sans état, comprenant un nombre minimal d'opérations (GET, POST, ...). Ces opérations, comme montré dans la Figure 2.3, permettent de manipuler la ressource ou d'en obtenir sa représentation.
- Des **liens hypermédias** dans les documents XML et XHTML pour représenter à la fois le contenu des informations et la transition entre les états de l'application
- Les **types MIME** comme text/html, image/jpeg ... pour la représentation des ressources.

La Figure 2.4 illustre le concept de la ressource. Ici la ressource "La météo de Paris" est identifiée par une URI (`http ://fr.weather.com/weather/local/FRXX0076`) et elle est représentée d'une part, par des métadonnées qui décrivent le contenu et d'autre part par la représentation de la ressource qui est au format HTML.

2.1.4.3 De la Solution "maison" . . .

Il existe de nombreuses manières de mettre en place une SOA, la plupart étant des solutions ad-hoc et propriétaires. En effet, la SOA étant une méthodologie pour mettre en place une architecture, il n'existe pas d'implémentation de référence d'un environnement pour l'exécution des services. Il est ainsi possible de construire des solutions ad-hoc simples, comme par exemple avec Apache Axis[10]. Ces solutions permettent de déployer un service web manuellement et facilement sur un serveur, afin de le rendre accessible aux potentiels consommateurs. Du point de vue du client, il existe des générateurs qui per-

[9] Uniform Resource Identifier
[10] http ://www.axis2.org

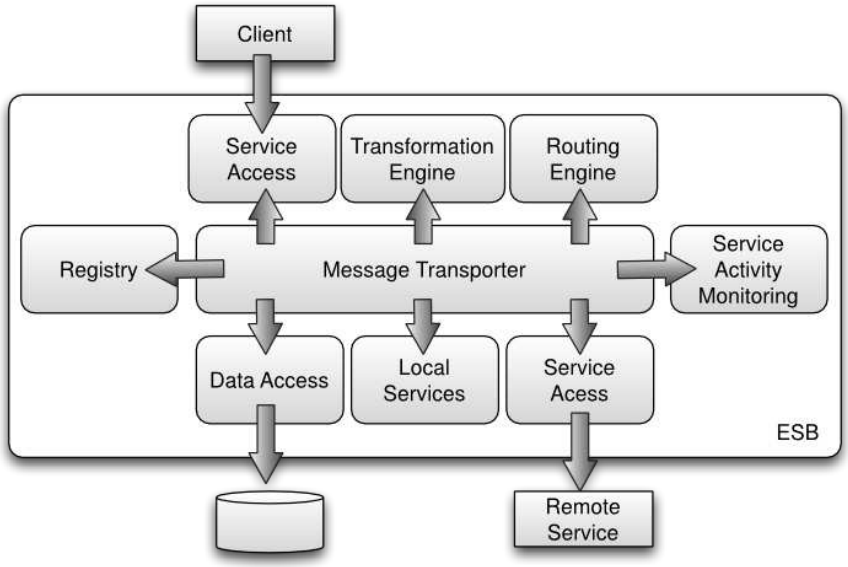


FIG. 2.5 – Une implémentation de la SOA avec un ESB

mettent, à partir de la description WSDL, de générer le code logiciel permettant d'accéder à distance au service. Le code logiciel utilise alors des librairies qui permettent, au moment de l'exécution, d'envoyer le message SOAP adéquat au service en fonction des paramètres d'utilisation.

2.1.4.4 . . . à l' ESB : Un environnement d'intégration basé sur les Services Web

Dans une architecture à services, les fonctionnalités offertes par les différents fournisseurs de services peuvent être découvertes et assemblées dans des applications. L'intégration des services n'est pas instantanée car les services sont rarement compatibles et les développeurs sont souvent confrontés aux différences des protocoles de communication ou des formats des données. Pour répondre à ces problématiques, une solution plus intégrée dont l'utilisation se répand de plus en plus dans les entreprises est celle de l'ESB[11] ou bus à services.

Comme le montre la Figure 2.5, un bus à services fournit un environnement pour l'exécution, l'intégration, le déploiement et la gestion des Services Web [Cha04] et une chaîne de médiation entre les consommateurs et les fournisseurs des services [HTL05]. Il est basé sur un bus de communication (au centre de la Figure 2.5, le *Message Transporter*) pour l'interaction de services [HLP05] et offre la possibilité de publier des applications comme des Services Web, ainsi que de les composer. Les fonctionnalités fournies par un ESB sont les suivantes :

- **Un bus de communication** qui supporte différents types de communication et qui offre des propriétés de qualité de service sur la communication, comme la sécurité, la médiation, la persistance, le routage et le support des transactions,
- Des mécanismes permettant d'**exposer des applications et des sources de données hétérogènes comme des services** avec une interface WSDL,

[11] Enterprise Service Bus

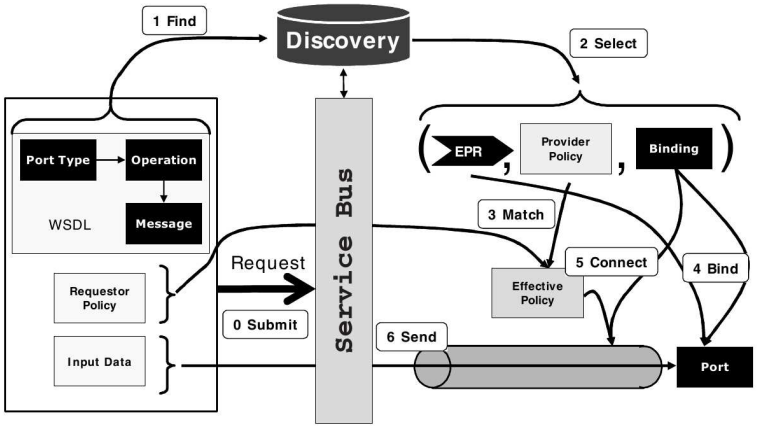


FIG. 2.6 – Les différentes étapes d'une interaction au sein d'un ESB [Ley05]

- Des **mécanismes de sélection** utilisant les descriptions de services pour découvrir et sélectionner les services,
- Des **systèmes d'administration et de monitoring** des services déployés.

Le bus de communication d'un ESB supporte plusieurs types de communication qui sont impliqués dans une architecture à services [KAB+04] :

1. assurer d'abord le transport des requêtes des consommateurs de services vers leurs fournisseurs,
2. garantir la livraison des messages envoyés par des applications vers d'autres applications,
3. permettre la communication basée sur les évènements : les applications gèrent et consomment des événements indépendamment les unes des autres.

Médiation La *médiation* est pour l'ESB le moyen de garantir la réalisation de la liaison entre un consommateur de service et un fournisseur même s'ils ne sont pas compatibles. Par exemple, si le format de message échangé entre le fournisseur et le consommateur est différent, la médiation peut transformer le format du message envoyé pour qu'il corresponde au format reconnu par le fournisseur de service. Si le fournisseur de service demande des message cryptés, l'ESB peut réaliser le cryptage des messages qu'il transmet. Les mécanismes de médiation peuvent être nombreux : filtrage des messages, agrégation, routage des requêtes de services en fonction du contenu des messages compatibles avec les politiques fonctionnelles et non-fonctionnelles du consommateur de service, transformation, etc . . .

La Figure 2.6 présente les différentes étapes de l'utilisation d'un service dans un ESB. Une application (la partie de gauche sur la Figure 2.6) qui nécessite d'utiliser les fonctionnalités d'un service connait uniquement la description WSDL d'un service et ne connait pas d'autres détails relatifs à l'implémentation du service. Elle envoie une requête, contenant les paramètres nécessaires à l'appel du service, à travers le bus ESB qui se charge de sélectionner le fournisseur de service adéquat (étape 0 sur la Figure 2.6). La fonctionnalité principale d'un bus à services est la *virtualisation*. Comme tous les services accessibles via le bus à services sont décrits par une interface WSDL, le bus cache à l'utilisateur les détails d'implémentation de ce service. Quand l'utilisateur veut effectuer une requête, il se

réfère à l'interface et le bus à service sélectionnera une des implémentations disponibles correspondant à l'interface (étape 2 sur le Figure 2.6) .

Pour raffiner la sélection d'un fournisseur de service afin de mieux répondre aux besoins fonctionnels et non-fonctionnels de l´application cliente, des politiques peuvent être ajoutées à la requête du client, respectivement à une implémentation de service. Ces politiques décrivent des propriétés fonctionnelles et non-fonctionnelles, comme le comportement transactionnel, des aspects relatifs à la sécurité, aux coûts, etc... L'ESB utilise les politiques associées à la requête pour réduire le nombre de fournisseurs de services candidats pour réaliser la fonctionnalité nécessaire. Cette médiation permet la mise en place d'un contrat de service qui formalise les termes de l'interaction entre ces deux parties. Après avoir sélectionné une implémentation de services, le bus réalise la liaison entre les deux parties et la requête est transmise au service sélectionné.

En pratique, un ESB permet de réaliser une connectivité faiblement couplée des services et peut gérer les liaisons entre services ainsi que de supporter les aspects non-fonctionnels comme les transactions, la sécurité, la supervision des métriques de performances, la reconfiguration dynamique et la découverte.

2.1.4.5 La fédération du bus à services distribué de la plateforme SOA4ALL

SOA4All [soa] a pour objectif de contribuer à la création d'une solution complète à large échelle dans un environnement où des milliards d'entités exposent et consomment des services dans ce qu'on appelle le *nuage de services* (situé au centre de la Figure 2.7). Le *nuage de services* doit être capable de couvrir Internet tout entier, afin de permettre aux utilisateurs finaux d'invoquer et de coordonner des services qui sont potentiellement situés à n'importe quel endroit dans le monde. C'est dans ce contexte que des bus à services sont fédérés afin de fournir une architecture qui puisse supporter des millions de services ainsi que l'exécution de milliers de compositions impliquant des sous-ensembles de millions de services. A cela, s'ajoutent des millions d'utilisateurs qui accèdent de façon concurrente aux services, et qui les déploient au travers du *Studio SOA4All*, comme le montre la Figure 2.7.

Dans une architecture d'ESB distribuée, les artefacts logiciels peuvent être hébergés sur différents nœuds (que l'on appelle *conteneurs*) et les moteurs de services peuvent être répliqués ou distribués sur ces conteneurs. Cette architecture constitue une base pour un bus à services distribué, ou DSB[12] et fournit une vision globale et unifiée du bus. Les implémentations des bus à services distribués sont développées pour couvrir un ensemble limité de ressources, en général située dans un même domaine administratif ou une même entreprise. La solution proposée par SOA4All consiste à fédérer ces bus distribués, en étendant l'implémentation du DSB PEtals[13], qui est un bus à services distribué dont l'implémentation est basée sur le modèle à composants Fractal et fournie par le Consortium OW2.

Le principe de la fédération de bus à services distribués de SOA4ALL constitue le cœur de la plateforme SOA4All [BFH+10]. L'implémentation de cet élément crucial est une évolution du bus à services distribué PEtals, dans le but d'obtenir un environnement d'exécution à large échelle, soit à l'échelle d'Internet, offrant ainsi un moyen complètement transparent, pour accéder, composer et déployer des services sur Internet.

[12]Distributed Service Bus
[13]http ://petals.ow2.org

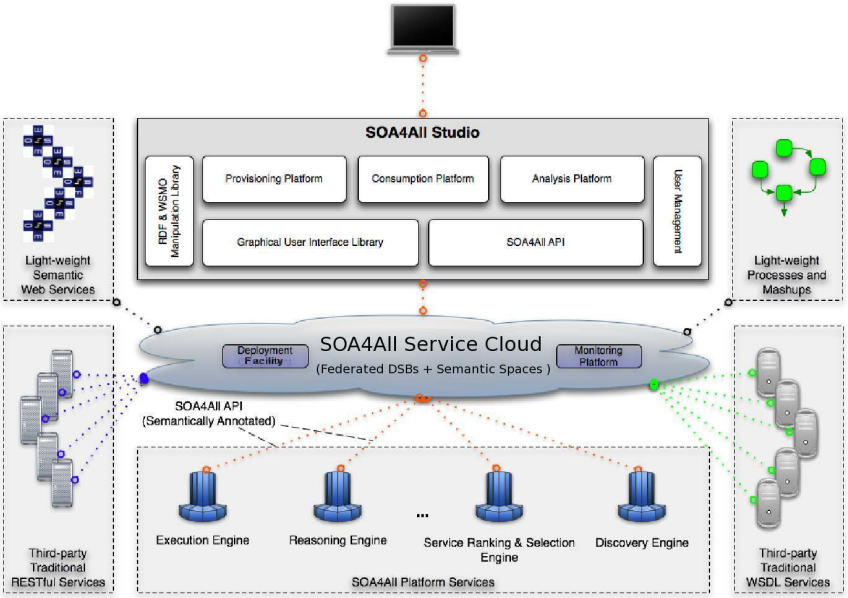


FIG. 2.7 – L'architecture globale de SOA4All

2.1.5 Conclusions sur l'approche à services

Nous venons de présenter l'approche basée sur les services, qui permet le développement d'applications à partir du concept central du service logiciel. Grâce notamment à la propriété de *couplage faible*, le service répond à des problématiques d'interopérabilité entre applications hétérogènes.

Dans le cadre de la programmation orientée service, un service peut être défini comme une entité fonctionnelle auto-contenue, auto-décrite, indépendante des plateformes, et pouvant être décrite, publiée, découverte, invoquée, composée à l'aide de protocoles standards. Basés sur des standards, les Services Web constituent une base technologique pour mettre en œuvre l'approche à services sur le Web. Nous avons présenté des infrastructures d'intégration de services, à savoir les bus à services, dont le bus à services distribué et fédéré de la plateforme SOA4ALL, développées autour des standards des Services Web (WSDL, UDDI, HTTP/SOAP . . .). Les mécanismes de médiation d'un ESB garantissent la réussite de l'intégration entre services et fournissent aussi une base solide pour la gestion de l'exécution des services.

Dans la section suivante, nous mettons en évidence le besoin de composer ces services qui permet l'intégration et la réutilisation des services dans les applications orientées services.

2.2 Composition des services

La section précédente a introduit les approches à base de services et les Architectures Orientées Services (SOA). Un des défis de la SOA ces dernières années, est l'intégration des services pour la fourniture de nouveaux services personnalisés et enrichis. Si une application ou un client requièrent des fonctionnalités et qu'aucun service n'est apte à les

fournir seul, le mécanisme de *composition de services* permet de combiner des services existants afin de répondre aux besoins de cette application ou de ce client. Dans le cadre de la SOA appliquée aux Services Web, la composition de Services Web est le processus par lequel un Service Web est créé par le biais de la combinaison d'autres Services Web. Les services ainsi utilisés sont cachés et ré-utilisés par le service composite, formant ainsi une application distribuée complexe.

La composition de services est le mécanisme qui permet l'intégration des services dans une application [BDDZ05]. Par exemple, la manipulation d'un ordre d'achat est la composition des processus chargés de calculer le prix final de la commande, de choisir un moyen de livraison, et de planifier la production et l'expédition de la commande.

Le résultat d'une composition de services peut être une application ou un autre service nommé *service composite [ACKM04].* Cette propriété entraine le fait que la composition de services peut être récursive ou hiérarchique, c'est-à-dire que des services atomiques ou composites peuvent être intégrés pour implémenter la logique d'autres services composites [KL03].

2.2.1 Cycle de vie d'une composition de services

La composition de services est un processus de raffinement permettant de passer d'une spécification abstraite de la composition vers une description exécutable. Les étapes pour la création d'une composition de services ont été définies par [YP04] :

1. **Une phase de définition abstraite**. Une phase de définition permettant de spécifier d'une manière abstraite la composition de services. Dans cette phase, il faut identifier en premier la fonctionnalité devant être fournie par la composition ainsi que la fonctionnalité devant être apportée par les différents participants. Finalement, les interactions entre les participants sont spécifiées.
2. **Une phase de planification** servant à déterminer comment et à quel moment les services seront exécutés. Dans cette phase, la conformité et la compatibilité des services doivent être vérifiées.
3. **Une phase de construction** fournissant une composition concrète et sans ambigüités, prête à s'exécuter. Uniquement les services potentiellement disponibles à l'exécution restent à déterminer.
4. **Une phase d'exécution** implémentant les liaisons avec les services disponibles et ensuite exécutant la composition.

En d'autres termes, le cycle de vie du processus de composition de services consiste d'abord à définir un service composite abstrait en fonction des descriptions des services requis (par exemple, les interfaces ou spécifications), ensuite à découvrir et à sélectionner les services fournissant ces interfaces parmi ceux qui sont disponibles, et enfin à réaliser les liaisons entre ces services sélectionnés. La sélection de services peut se faire à la phase de conception du composite ou pendant l'exécution. On parlera respectivement de sélection *statique* ou *retardée.*

Définition [*Composition statique*] : Une composition est dite *statique* si les artéfacts participant à la composition sont sélectionnés avant la phase d'exécution.

Définition [*Composition retardée*] : Une composition est dite *retardée,* si les artéfacts participant à la composition peuvent être sélectionnés et liés à la phase d'exécution. En général, une fois les liaisons effectuées, elle restent figées.

Définition [*Composition Dynamique*] : Une composition est dite *dynamique*, si les artéfacts participant à la composition peuvent apparaître et/ou disparaître à tout instant pendant l'exécution. De nouveaux artéfacts peuvent être sélectionnés et/ou de nouvelles liaisons peuvent être réalisées.

Le processus de composition de services est une activité de longue durée, chacune des phases définies auparavant pouvant être elle-même divisée en sous-tâches. L'identification des fournisseurs de services répondant à certains besoins fonctionnels est seulement un exemple de ce type de tâche. Les développeurs se retrouvent rarement dans la situation idéale où tous les services nécessaires à la réalisation d'une composition de services sont compatibles. Dans la majorité des cas, afin de remplir correctement leur tâche, ils doivent résoudre les différentes incompatibilités des services participants, comme par exemple celle des types de données d'entrées et de sortie.

2.2.2 Le contrôle de la composition

Une composition de services assemble les fonctionnalités des services pour réaliser une fonctionnalité plus complexe. Un aspect important dans la réalisation d'une composition de services est la spécification de la logique de coordination des services impliqués. Parmi les techniques de composition de services on peut distinguer deux moyens de réaliser le contrôle d'une composition de services [FS05] :

- ***La composition définie dans le temps***. Dans ce cas, la composition de services est réalisée en spécifiant la logique de coordination des services par un *workflow*. Un *workflow* est généralement représenté par un graphe orienté d'activités et un flot de contrôle qui donne l'ordre d'exécution des activités. Chaque activité invoque la fonctionnalité fournie par un service. Cette description est réalisée en utilisant un langage spécifique qui sera ensuite interprété par un moteur d'exécution spécifique afin de réaliser l'invocation des services impliqués, le routage des évènements et la gestion des erreurs.
- ***La composition structurelle - ou par assemblage***. La spécification de la composition est réalisée en indiquant clairement les composants qui fournissent les services nécessaires et leurs interactions. Chaque composant déclare les interfaces qu'il fournit et celles qu'il requiert. L'assemblage de services se traduit par un assemblage de composants pour lesquels la paire service fourni/service requis correspond. La logique de coordination de la collaboration des services, qui définit la manière dont la composition de services fonctionne, est spécifiée par le développeur et livrée avec l'assemblage. Par exemple, la logique de coordination peut être réalisée par une classe Java.

Dans la suite, nous décrivons en détail ces deux types de composition de services.

2.2.3 Composition dans le temps

La *composition de services dans le temps* décrit au travers d'un processus le comportement de la composition des services. Un processus est composé d'un ensemble d'activités et un flot de contrôle. Pour une composition de services, chaque activité composante correspond à l'invocation sur un service et le flot de contrôle définit les conditions de l'exécution de ces activités, c'est-à-dire dire l'ordre d'invocation des services participants et les différentes interactions entre ces services. L'approche de composition dans le temps reprend

des idées déjà mises en pratique par les technologies de *workflows*, mais en ajoutant les propriétés de l'approche à services [VDT03].

Un service composite joue le rôle d'un *coordonnateur de services*. Un avantage de cette approche est le fait qu'elle rend explicite la logique de contrôle de la composition. De plus, cette logique est externe par rapport aux services composés car elle est décrite en dehors de ces services.

Le modèle de composition dans le temps est généralement représenté par un graphe, où chaque nœud correspond à l'invocation d'une opération d'un service, et les arcs servent à exprimer l'ordre d'enchaînement des opérations. La spécification de la composition est faite dans un langage qui est interprété par un moteur d'exécution. La responsabilité du moteur est d'invoquer les services dans l'ordre spécifié, de réaliser le routage des données, de maintenir et gérer l'état des activités et du composite, ainsi que de gérer les situations d'exception. Deux approches, illustrées dans la Figure 2.8 existent pour ce type de composition [Pel03] :

- **L'*orchestration de services*** (Figure 2.8 (a)) décrit la vision centralisée d'une composition de services.
  Une orchestration décrit l'interaction des services impliqués dans la composition, les messages qu'ils échangent et l'ordre et la manière par laquelle les services interagissent entre eux (par exemple en séquence, en parallèle, ou sous certaines conditions). Une orchestration représente généralement, pour un fournisseur de services, la réalisation du processus métier qu'il expose comme un service, processus qui peut être de longue durée et avoir un comportement transactionnel. Ces interactions peuvent être inter-application et/ou inter-organisations, et résulter en un modèle d'exécution transactionnel de longue durée. Le fournisseur du service détient l'unique point de contrôle de l'orchestration.
  L'orchestration offre une vision centralisée de la logique de coordination d'une composition. L'exécution de l'orchestration de services est contrôlée par une entité centrale – appelée *moteur d'exécution* qui gère l'invocation des différents services intervenant dans la composition selon la logique définie par le processus. Plusieurs langages d'orchestration pour les Services Web existent actuellement. Parmi ceux-ci nous pouvons citer le langage XPDL (XML Process Definition Language), le langage APEL [AE97] et langage WS-BPEL[14] que nous détaillerons plus loin et qui est le plus utilisé de nos jours.
- **La chorégraphie de services** (Figure 2.8 (b)) décrit d'un point de vue global, la manière dont un ensemble de services collabore. Contrairement à l'orchestration, la chorégraphie ne centralise pas le contrôle de l'invocation des services mais elle présente plutôt la vision globale de tous les participants relativement à la manière dont ils doivent collaborer à la réalisation d'un but commun. La logique de la collaboration est cette fois-ci répartie entre les services concernés. Dans une chorégraphie, chaque participant impliqué joue un rôle attribué. La chorégraphie décrit d'une manière globale les échanges de messages (les *conversation*s), les règles auxquelles sont soumises les interactions des différentes parties, respectivement la manière dont les différents services se coordonnent afin de remplir le but de la composition de services.
  Une chorégraphie ne décrit aucune autre action interne d'un service participant qui n'a pas d'effet visible à l'extérieur (telles que des traitements internes ou des transformations de données par exemple) [BDO05]. Elle se concentre sur la séquence visible des échanges de messages entre les participants et présente une vision globale externe de la manière dont les participants interagissent. L'intérêt principal d'une

[14] Web Services Description Language

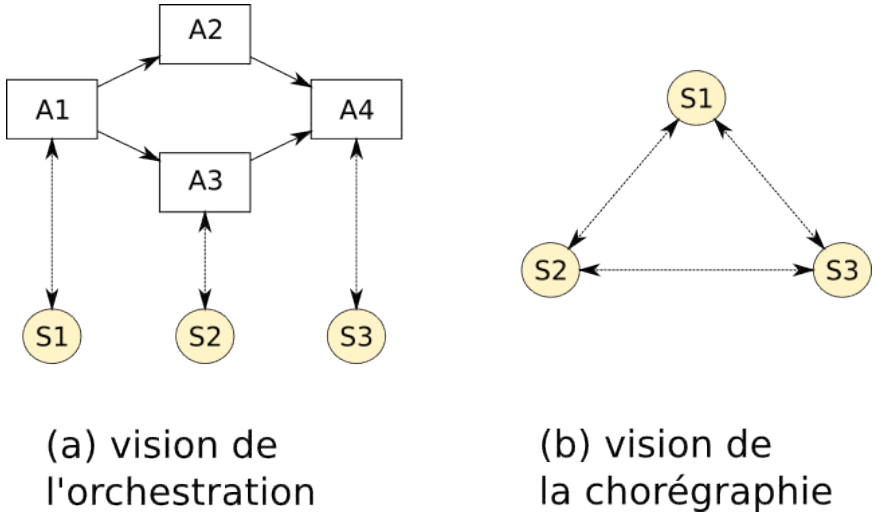


FIG. 2.8 – Deux visions de compositions de services dans le temps

chorégraphie est de vérifier qu'à l'exécution tous les échanges de messages entre les partenaires sont réalisés conformément à une spécification.
Plusieurs propositions de langages pour la chorégraphie de services existent, parmi lesquels, WS-CDL, WSCI (Web Services Collaboration Interface), et ebXML(Electronic Business using eXtensible Markup Language). WS-CDL (Web Service Choreography Description Language) est un langage destiné à la modélisation de chorégraphies de services réalisée par le consortium W3C.

Les notions d'orchestration et de chorégraphie de services sont actuellement utilisées majoritairement pour les compositions de Services Web. Elles peuvent être utilisées en conjonction pour permettre l'intégration des systèmes des différentes entreprises [DZD06]. Ainsi la chorégraphie modélise la collaboration entre les différents partenaires impliqués et l'orchestration est utilisée en interne par chaque partenaire pour modéliser la réalisation de la fonctionnalité rendue disponible par celui-ci.

2.2.4 Composition structurelle : Les composants orientés services

Dans la composition structurelle, les composants qui fournissent les services sont clairement identifiés, chaque composant définit explicitement ses interfaces fournies comme celles requises, la composition est spécifiée comme l'assemblage des composants. Elle exprime des liens entre couples d'interfaces fournies et requises par deux composants. Le formalisme utilisé pour décrire cet assemblage dépend de l'approche suivie. Il est connu en général comme langage de description d'architecture (ADL pour Architecture Description Language).

La logique de contrôle, exprimant comment et à quel moment les opérations des services composés doivent être invoqués, est implicite et répartie entre les différents composants.

2.2.4.1 L'approche à composants

La réutilisation a toujours été un défi majeur depuis le début du génie logiciel. C'est pourquoi, de nombreuses approches [Tay98] [BME+07] [SGM02] [WS01]ont été propo-

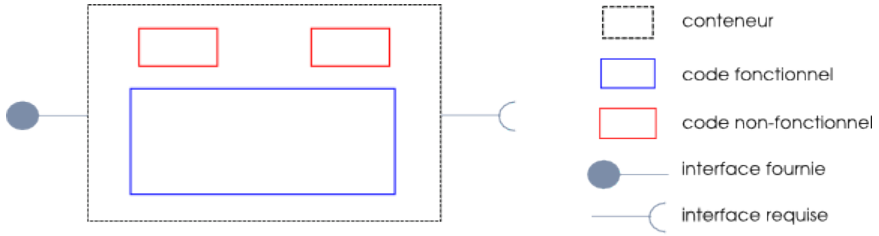


FIG. 2.9 – Structure d'un composant

sées pour apporter des solutions à cette problématique de réutilisation. Ainsi, l'approche à composant a été définie pour améliorer la réutilisation du code des applications logicielles. Pour ce faire, cette approche promeut la construction d'une application à partir d'un ensemble de briques logicielles bien définies et indépendantes, appelées composants [SGM02, WS01].

L'objectif de l'approche à base de composants est d'améliorer les limitations de son prédécesseur en l'occurrence l'approche orientée objet [Tay98, BME+07]. Ainsi, elle cherche à améliorer la réutilisation grâce au développement d'applications par assemblage de composants, à fournir des mécanismes permettant aux développeurs de se concentrer uniquement sur les besoins métiers de l'application, à faciliter la maintenance, l'évolution et l'administration des applications logicielles. En effet, l'approche à composants propose d'encapsuler les données et les fonctionnalités de base d'une application à l'intérieur d'un ensemble d'éléments appelés des composants. Les fonctionnalités fournies par un composant sont uniquement visibles et accessibles à travers ses interfaces publiques. Une séparation claire est effectuée entre ces interfaces et le code réalisant leur implémentation. De ce fait, lors de l'assemblage des composants constituant une application, des connexions seront établies entre ses composants en termes d'interfaces pour spécifier l'architecture de l'application. La personne (c'est-à-dire l'assembleur) réalisant cet assemblage ne connaît pas la structure interne des composants qu'il utilise du fait que selon son point de vue ces derniers sont des « boites noires ». Comme montré dans la Figure 2.9, le composant est vu comme une boite noire qui offre une fonctionnalité via ses interfaces fournies.

Bien qu'il existe beaucoup de définitions du concept de composant, nous retiendrons la suivante :

> **Définition [*composant*] :** Un composant est une entité logicielle qui peut être considérée comme une boite noire qui expose uniquement des interfaces fournies et des interfaces requises. Les composants sont interconnectés par des liaisons entre les interfaces.

A l'heure actuelle, il n'existe pas encore de véritable consensus sur la définition du concept de composant mais les définitions suivantes sont les plus citées et acceptées dans la littérature :

> *"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]*

Dans la définition proposée par Szyperski [SGM02], nous pouvons identifier les caractéristiques suivantes d'un composant :

- Un composant est une unité de composition. Il peut donc être assemblé dans une application par un tiers.
- Un composant possède des interfaces bien définies permettant de spécifier les fonctionnalités fournies par le composant.
- Un composant peut exprimer des dépendances explicites vers d'autres composants en vue de spécifier ses besoins.
- Un composant peut être une unité de déploiement et il peut être déployé de manière indépendante.

Séparation des préoccupations La séparation des préoccupations entre les aspects fonctionnels (en bleu dans la Figure 2.9) et non fonctionnels (en rouge dans la Figure 2.9) tels que la sécurité, les transactions ou la distribution d'un composant fait partie des principes de base de l'approche orientée composant. Le développeur d'un composant réalise seulement le code correspondant à la logique applicative liée aux fonctionnalités offertes par le composant, et les propriétés non fonctionnelles sont gérées et assurées par la plateforme d'exécution associée. Plusieurs plates-formes à composant proposent des mécanismes pour gérer un ensemble de propriétés non fonctionnelles. Généralement, le concept de conteneur est utilisé pour réaliser la gestion de ces aspects non fonctionnels. L'ensemble des propriétés non fonctionnelles est extensible dans la mesure où le développeur peut en définir d'autres en écrivant le code associé.

L'approche orientée composant possède plusieurs avantages dans la mesure où :

- Elle favorise la réutilisation de code grâce au développement d'applications par assemblage de briques logicielles préexistantes.
- Elle facilite le développement incrémental d'applications parce que certains composants d'une application donnée peuvent être développés puis intégrés plus tard.
- Elle effectue une séparation claire entre les aspects fonctionnels et non fonctionnels d'une application en fournissant des mécanismes permettant aux développeurs de se concentrer uniquement sur les besoins de l'application, les aspects non fonctionnels associés au composant étant gérés par la plateforme d'exécution du composant (en général, par un conteneur).
- Elle améliore l'extensibilité, l'évolution et la maintenance des applications puisque ces dernières ne sont plus monolithiques.
- Elle effectue une séparation claire entre les tâches des développeurs et des assembleurs.Dans l'intention de réduire les délais et les coûts de réalisation des logiciels, l'approche à composant propose d'assembler des composants préexistants et réutilisables.

2.2.4.2 Approches et Objectifs des composants orientés services

Les sections précédentes (2.1 et 2.2.4.1) ont présenté l'approche à services et l'approche à composants. Nous nous concentrons maintenant sur la combinaison de ces deux approches, constituant une approche de ***composants orientés services***. Un modèle à composants orienté services combine les avantages de l'approche à composant et de l'approche à services. Dans un tel modèle, une spécification de services est implantée par le biais d'un composant. Les dépendances d'un composant sont exprimées en termes des spécifications de services et elles sont résolues en utilisant le modèle d'interaction de l'approche

à services. De cette façon, dans une application, un composant peut être remplacé par n'importe quel autre composant pourvu qu'il respecte la même spécification de service. En utilisant l'approche à composants orientée services, nous pouvons bénéficier d'une part d'un modèle de développement simple et d'une description de la composition comme le préconise l'approche à composants, et d'autre part d'un faible couplage et de la liaison retardée, caractéristiques apportées par l'approche à services.

2.2.4.3 Service Component Architecture (SCA)

La combinaison des approches à services et approches à composants est basée sur le principe que les services sont caractérisés par un contrat et que les composants implémentent un contrat. Le modèle structuré d'assemblage de SCA permet de développer une application basée sur les services en suivant une approche orientée composant, facilitant la réutilisation des services, avec une indépendance vis-à-vis de leur technologie d'implémentation ou de leur protocoles de communication. Les composants implémentent un contrat : ainsi, les services fournissent des fonctionnalités qui peuvent être réutilisées, ce qui est aussi le cas pour un composant. De plus, un composant peut être complètement remplacé par un autre qui suit le même contrat. Cette convergence entre les besoins de la SOA et les fonctionnalités fournies par les solutions basées sur les composants a donné naissance à l'initiative appelée Service Component Architecture (SCA). SCA est un ensemble de spécifications industrielles qui permettent la simplification du développement d'applications à base de services grâce à :

- Une spécification d'un **modèle d'assemblage** qui simplifie la composition et le développement de services métiers, indépendamment des langages d'implémentation et des plateformes d'exécution. L'objectif de cette spécification est de décrire comment les services fournis par ces composants sont assemblés afin de construire l'application [Cha07]. L'assemblage d'une application SCA peut être décrit en utilisant un fichier ADL, appelé *SCDL pour Service Component Definition Language.*
- Une spécification d'un **modèle de programmation par composant** simple destiné à l'implémentation des services métiers.. L'objectif de cette spécification est de spécifier un ensemble de composants (unités de composition) formant une application. Une fonction métier est fournie comme un ensemble de services qui sont assemblés ensembles pour créer un une solution orientée service. Les applications sont construites comme un ensemble de services, appelées applications composites. Elle peuvent inclure des services créés spécifiquement pour l'application et aussi des fonctions métier existant dans des applications externes, réutilisées dans la composition.

Les unités de composition de base, les *composants SCA*, sont définies de façon indépendante à une quelconque technologie, ce qui permet la séparation de l'implémentation des services et des protocoles de communication de la description de l'application, ainsi que l'utilisation de technologies hétérogènes pour implémenter les services. L'implémentation d'un *Composant SCA* peut être réalisée en utilisant n'importe quelle technologie, par exemple des langages traditionnels tels que Java, C++, des langages d'orchestration tels que BPEL ou des langages de scripts tels que PHP ou Javascript.

Concepts clés Un *Composant SCA* est une instance d'une implémentation de services qui a été configurée au travers d'une *description SCDL (Service Component Description Language)*. Une implémentation est le code écrit par un développeur pour remplir certaines fonctions, comme par exemple, une classe Java ou un processus WS-BPEL. Un *Composant SCA* englobe l'implémentation d'un service et le rend disponible au travers d'interfaces

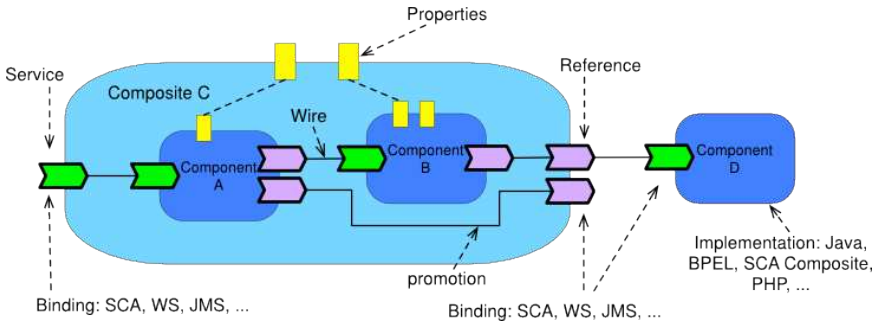


FIG. 2.10 – Un composite SCA incluant deux composants SCA

appelées *Services SCA*. Un service SCA est de ce fait le point d'accès à la fonctionnalité fournie par le composant SCA. Par ailleurs, un composant SCA exprime les dépendances sur les autres services grâce à des *Références SCA*. Une référence est en fait un service qu'un composant SCA peut appeler. Les *Services SCA* et les *Références SCA* sont définies en terme de groupes d'opérations dans une interface. Les *Composants SCA* fournissent un mécanisme pour configurer une implémentation de façon externe, au travers des *Politiques SCA* (*SCA Policies*) [OSO07a]. Une propriété SCA est une assertion qui contrôle ou contraint des propriétés non-fonctionnelles sur les applications et qui doit être appliquée par le support d'exécution. Cette capacité a été ajoutée principalement pour inclure des propriétés relatives à l'authentification, la confidentialité, l'intégrité. la fiabilité des messages, et les transactions.

La spécification SCA supporte aussi SDO(Service Data Objects [15]), qui est un standard permettant de représenter et de manipuler des données indépendamment de leur structure réelle, ce qui constitue un énorme avantage pour SCA puisque facilitant la composition de services hétérogènes sans se préoccuper de la structure des données réelles.

Le tableau 2.2.4.3 ainsi que la Figure 2.10 présentent un récapitulatif de ces concepts.

Réalisation de la composition La composition est réalisée en connectant des composants SCA au travers de leurs interfaces : les services SCA et les références SCA sont connectées par des *Connecteurs SCA (SCA Wires)*. Le mécanisme par lequel les clients peuvent appeler et utiliser le service peut être spécifié en utilisant les *Liaisons SCA (SCA bindings)*. Les types de liaison, peuvent se traduire par le fait que, par exemple, les sont services exposés en tant que Services Web, en tant que services JMS ou d'autres services SCA. SCA fournit un mécanisme d'extensibilité où des types de liaisons additionnels peuvent être définis.

Un modèle de composition hiérarchique La composition peut être réalisée de façon hiérarchique. SCA définit un modèle hiérarchique qui se traduit par les composites SCA. Les composites SCA permettent d'assembler les éléments SCA dans des groupements logiques contenant un ensemble de composants SCA, service SCA, références SCA et liens SCA qui les interconnectent. Les composites SCA peuvent être vus comme des composants SCA ordinaires, ce qui signifie qu'ils peuvent être connectés aux autres composants SCA qui peuvent être aussi des composites, et qu'il peuvent être inclus dans d'autres compo-

[15] http ://www.osoa.org/display/Main/Service+Data+Objects+Home

CONCEPTS		DÉFINITION
Service et Liaison (binding)		Un service est la définition de fonctionnalités métiers. Elle comprend, entre autre, une interface Java ou WSDL composée de plusieurs opérations et un ensemble de liaisons, c'est à dire les différents mécanismes d'accès à utiliser pour appeler le service.
Implémentation		Une implémentation est le code réalisant zéro, un ou plusieurs services. Elle peut être écrite avec plusieurs langages comme Java, C++ ou WS-BPEL.
Composant		Un composant est constitué au plus d'une implémentation qui est configurée dans un fichier de configuration décrit avec le langage SCDL. La configuration d'une implémentation consiste à affecter des valeurs aux propriétés éditables qu'elle définit
Politiques		Dans SCA, les propriétés sont typées et elles peuvent être de type simple ou complexe, par exemple les types définis dans XML Schema.
Référence		Les composants SCA exposent leurs fonctionnalités métiers en termes de services et peuvent aussi utiliser ou dépendre d'autres services. Ces dépendances sont appelées des références.
Connecteur		Le concepts de connecteur ou "*wire*" permet l'établissement des liaisons entre les références et les services.
Composite		Un *composite SCA* est un assemblage de composants SCA.

TAB. 2.1 – Récapitulatif des différents concepts de SCA

sites SCA. Les services SCA et les références SCA utilisés par un composite sont exposés grâce à la *promotion* de leurs interfaces.

Implémentation des supports d'exécution pour les spécifications SCA Il existe différentes implémentations de supports d'exécution pour les composants SCA. Un support d'exécution pour SCA fournit les moyens pour créer, déployer et exécuter une application basée sur les spécifications SCA. Parmi les différentes implémentations nous pouvons citer IBM WebSphere Application Server [IBM], Fabric3 [fab], Apache Tuscany [tus], Paremus service Fabric [par] et FraSCAti [SMF+09].

Inconvénients de l'approche SCA L'inconvénient majeur de SCA est le manque de support pour l'évolution dynamique. Le fichier SCDL est conçu de façon statique, au moment de la conception, et ne peut être modifié durant l'exécution de l'application. De ce fait, les modifications dans la composition de l'application à services nécessitent de stopper l'application entière, et de la relancer pour utiliser la nouvelle composition. La reconfiguration dynamique et les autres aspects non-fonctionnels ne sont pas traités par les spécifications SCA, c'est à l'implémentation du support d'exécution de fournir de telles fonctionnalités. Il existe cependant un exemple d'une telle plateforme qui fournit un support pour la reconfiguration, FraSCAti [SMF+09]. La particularité de FraSCAti est qu'il offre un ensemble de primitives permettant d'observer et d'adapter dynamiquement les composants SCA. FraSCAti permet aussi de gérer des propriétés non-fonctionnelles au niveau des composants. Grâce aux mécanismes d'adaptation proposés, il est alors possible d'introduire ou d'enlever voire de mettre à jour de telles propriétés.

2.2.4.4 iPOJO

IPOJO[16] [Esc08], est un modèle à composants orienté services, permettant de simplifier le développement d'applications OSGi [The07]. Il fournit une infrastructure de développement et d'exécution d'applications dynamiques basées sur les services OSGi. En combinant l'approche à composants et l'approche à services, iPOJO permet d'introduire des caractéristiques de dynamismes au sein des applications OSGi. iPOJO utilise la notion de conteneur afin de dispenser au développeur de connaître les détails techniques des mécanismes de base offerts par la plateforme OSGi.

En se servant du concept de *conteneur*, le modèle iPOJO permet de séparer les préoccupations Non-fonctionnelles du code métier. Un composant iPOJO est constitué de 3 éléments :

- **Le code métier** : un objet POJO[17] qui implémente les services fournis par le composant
- **Le conteneur**, responsable des interactions du composant avec la plateforme. Le conteneur d'un composant iPOJO prend en charge des aspects techniques comme l'enregistrement de services dans l'annuaire, la découverte et sélection de services et la gestion des dépendances d'un composant.
- **Un ensemble d'aspects Non-Fonctionnels**. Un conteneur iPOJO est constitué de plusieurs *handlers*, chacun d'entre eux prenant en compte un aspect non-fonctionnel du composant.Il est possible dans iPOJO d'ajouter de nouveaux *handlers* au conteneur d'un composant pour fournir d'autres aspects non-fonctionnels non pris en

[16]Acronyme pour injected Plain Old Java Objects
[17]Plain Old Java Object

compte par les *handlers* fournis par défaut. Les aspects Non-fonctionnels sont injectés dans le conteneur, et mis en œuvre grâce à une description XML ou des annotations Java, caractérisant le conteneur. La composition des *handlers* ne peut pas être modifiée pendant l'exécution.

2.2.4.5 Mashups et Web 2.0

L'émergence du web 2.0 et de ses technologies associées (REST, Ajax[18], RSS[19]) a mis en avant le concepts des *mashups* dans le but de produire des pages web dynamiques qui peuvent être composées. Un *mashup* est une application composite qui combine du contenu ou des services provenant de plusieurs applications hétérogènes. Dans le cas de sites web, le principe est d'agréger du contenu provenant d'autres sites, afin de créer un site nouveau. Le *mashup* intervient ainsi au niveau de l'interface utilisateur, construite à partir d'éléments ré-utilisables, les données contenues dans une page étant organisées de manière componentisée. De plus, avec ces composants web, les consommateurs peuvent eux-mêmes accomplir la logique métier dans le navigateur.

Différents participants interviennent dans une application *mashup,* à savoir :
- le fournisseur de contenu, proposant les APIs nécessaire à la manipulation du contennu
- Le site d'hébergement du mashup
- Le navigateur web du consommateur du mashup.

2.2.5 Conclusions sur la composition de services

Nous venons de présenter le principe de la composition de services, qui est au cœur de nos travaux de thèse.

Dans un premier temps, nous avons présenté les concepts fondamentaux de la composition de services.La composition de services permet l'intégration et la réutilisation des services dans diverses applications. Après avoir étudié le cycle de vie d'une composition, nous avons identifié et explicité deux modes de composition, à savoir la composition temporelle et la composition structurelle. La composition dans le temps a été illustrée par les concepts d'orchestration et de chorégraphie.La composition de services structurelle a été présentée et illustrée par le biais des composants orientés services, notamment par le modèle de composition de services SCA, le modèle de composition iPojo et le concept des *mashup Web 2.0*.

2.3 Synthèse du chapitre

Nous avons présenté dans ce chapitre une vision générale de l'approche à services, avec laquelle nous avons travaillé tout au long de cette thèse.

La SOA définit un paradigme de programmation plaçant le service comme concept central. L'architecture SOA propose un cadre conceptuel pour l'utilisation de cette approche, elle spécifie les actions et leur modèle d'interaction pour la construction d'applications à services.

Nous avons mis en évidence la propriété la plus importante de l'approche à services, à savoir le faible couplage existant entre les entités composant une application. Cette propriété permet l'évolution indépendante des différentes parties d'une application. Grâce à

[18]Asynchronous JavaScript and XML
[19]Really Simple Syndication

la séparation entre la fonctionnalité fournie et son implémentation, l'approche à services permet une évolution plus rapide, voire même transparente des systèmes. Une autre propriété intéressante est la possibilité de construire des applications multi-fournisseurs : ainsi, dans ce type d'applications, il n'existe pas une entité centrale d'administration.

Les Services Web proposent la mise à disposition des applications à travers un réseau. Ils sont utilisés aujourd'hui essentiellement pour l'intégration des applications patrimoniales, et les services ainsi exposés sont d'une grosse granularité. Bien que la technologie des Services Web ne soit pas la seule manière de réaliser une SOA, elle est la technologie la plus considérée par l'industrie. Nous avons présenté la technologie de bus à services qui sert de base à la plateforme d'exécution de services du projet SOA4All, permettant l'utilisation de milliond de services.

La contrepartie des avantages de la SOA est la grande complexité de conception des applications orientées services. Dans le but de réduire cette complexité, l'approche à services fournit un cadre de base qui sera utilisé non seulement pour réaliser des interactions entre les différents acteurs, mais aussi dans la construction de services plus complexes qui sont créés à partir de services basiques. Cette opération est connue sous le terme de composition de services. En particulier, nous avons mis en avant le concept d'orchestration de services.

Les notions d'orchestration et de chorégraphie de services sont majoritairement utilisées seulement pour les compositions de Services Web. Elles peuvent être utilisées en conjonction pour permettre l'intégration des systèmes des différentes entreprises [DZD06]. Une différence importante entre l'orchestration et la chorégraphie est que l'orchestration offre une vision centralisée, c'est-à-dire que l'exécution de la composition est toujours contrôlée selon la perspective d'un des partenaires métier. En revanche, la chorégraphie offre une vision globale et plus collaborative de la coordination : elle décrit le rôle que joue chaque participant impliqué dans l'application

Nous avons présenté le modèle structuré d'assemblage de SCA, qui sera utilisé dans la suite pour modéliser notre approche d'orchestration répartie et dynamique. SCA permet de spécifier une application basée sur les services en suivant une approche orientée composant, facilitant la réutilisation des services, avec une indépendance vis-a-vis de leur technologie d'implémentation ou de leur protocoles de communication. Cette séparation rend l'application plus adaptable en fonction des fournisseurs de services hétérogènes. Grâce à SCA il est possible d'utiliser des services distants, sans pour autant se soucier des détails techniques propres aux protocoles d'accès. Nous avons ensuite présenté le modèle iPojo permettant de composer les services OSGi. Nous y avons également abordé le concept des mashup du Web 2.0 permettant la composition des ressources sur le Web.

Le travail de cette thèse se place au cœur des applications à services, le concept clé étant l'orchestration de service distribuée et modifiable dynamiquement pendant son exécution. Dans le chapitre suivant, nous étudions différents systèmes d'exécution d'orchestrations.

Le monde que nous avons créé est le résultat de notre niveau de réflexion, mais les problèmes qu'il engendre ne sauraient être résolus à ce même niveau.

Albert Einstein, Extrait de *Le Prix de l'Excellence.*

Chapitre 3

Des systèmes d'orchestration centralisés aux systèmes d'orchestration répartie

Contenu du chapitre :

3.1 Système de gestion de Workflow . . . 54
3.2 Composition centralisée avec WS-BPEL . . . 56
3.2.1 Déploiement . . . 57
3.2.2 Dynamicité et distribution . . . 57
3.2.3 Implémentations de WS-BPEL . . . 57
3.2.4 Synthèse . . . 58
3.3 Systèmes d'exécution de Workflows Dynamiques et Distribués . . . 58
3.3.1 Exotica/FMQM [AMA+95] . . . 59
3.3.2 SCENE/secse [CDM06] . . . 60
3.3.3 JOPERA [PA05b] . . . 62
3.3.4 SELF-SERV [SDM02] . . . 64
3.3.5 FOCAS [Ped09] . . . 66
3.3.6 WISE [LASS00] . . . 67
3.3.7 Crossflow [GAHL00] . . . 69
3.3.8 NIÑOS [GYJ11] . . . 71
3.3.9 SwinDew [YYR06] . . . 73
3.3.10 Taverna [HWS+06, OAF+04] . . . 74
3.3.11 KEPLER [ABJ+04] [LAB+06] . . . 75
3.4 Synthèse de l'état de l'art . . . 76
3.5 Conclusion du chapitre . . . 78

Dans la Section 2.2.1, nous avons présenté le cycle de vie d'une composition de services. Comme motivé dans la Section 1.2.1, nous nous intéressons dans le cadre de ce travail de thèse à la résolution des problèmes liés à l'exécution décentralisée et dynamique

des orchestrations de services. Ce chapitre présente un état de l'art basé sur différents systèmes d'exécution de compositions de services, prenant en compte la décentralisation et la dynamicité.

Nous commençons par expliciter le fonctionnement d'un moteur d'orchestration dans la Section 3.1.

Nous étudions dans un premier temps les systèmes d'exécution centralisés basés sur le langage WS-BPEL (Section 3.2). Dans un second temps, nous étudions des solutions d'exécution d'orchestrations de services décentralisées et/ou dynamiques (Section 3.3). L'étude de l'état de l'art est guidée par les contraintes liées à l'exécution des processus métiers décentralisés présentés dans la section 1.2.3.

Nous terminons le chapitre par une conclusion de cet état de l'art dans la Section 3.5.

3.1 Système de gestion de Workflow

Dans cette section, nous examinons le fonctionnement d'un moteur d'orchestration, ou moteur de workflow. Cet élément est crucial dans notre travail et nous devons mettre en évidence ses différentes fonctionnalités afin d'identifier les critères discriminants pour notre étude de systèmes d'orchestration décentralisés et dynamiques.

La composition des services présente plusieurs similarités avec la technologie des workflows. Les deux ont pour but de spécifier le processus métier par la composition des entités autonomes et de forte granularité. Leur différence réside dans la nature de l'entité. Dans le cas des workflows, les entités sont des applications conventionnelles, dans le cas des services, les entités sont des services. Les travaux autour de la composition des services se sont basés sur les concepts de base développés dans le contexte des systèmes de gestion de workflows (WfMS [1]).

La WfMC[2] définit les Systèmes de Gestion de Workflow comme :

> **Définition [*Système de Gestion de Workflows*] :** Un Système de Gestion de Workflow est un système complet qui permet de définir, de gérer, et d'exécuter des procédures en exécutant des programmes dont l'ordre d'exécution est pré-défini dans une représentation informatique de la logique de ces procédures [WFM99].

L'exécution d'une composition de services dans le temps se réalise grâce à un moteur de workflow, responsable de la transmission des données entre les différentes entités impliquées dans la composition. Au moment de l'exécution, la définition d'un processus est interprétée par un logiciel qui est responsable de créer et de contrôler les instances du processus, et de planifier les étapes d'activités du processus et d'invoquer les acteurs impliqués. Ce système gère, en plus de l'exécution proprement dite, les définitions de processus et permet de s´interfacer avec des outils d'administration, des outils de suivi, des applications clientes ou d'autres systèmes de gestion de workflow.

Un tel système est capable de charger en mémoire une ou plusieurs définitions de processus de workflow. Sur demande de l'utilisateur, un processus peut être démarré (donc instancié). Le système va suivre le cheminement décrit par le processus et présenter la ou les activités à réaliser aux différents acteurs du workflow. Une entité (le consommateur) doit déclencher le workflow en invoquant un (ou plusieurs) des points d'entrée du workflow.

[1] Workflow Management Systems

[2] La WfMC (Workflow Management Coalition) est une organisation qui tente de standardiser les systèmes de workflows, au niveau des langages et des architectures.

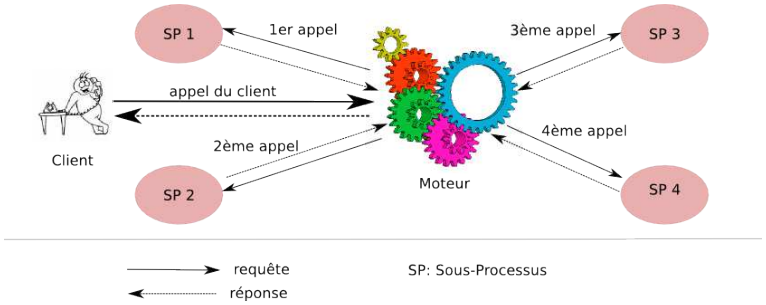


FIG. 3.1 – Exécution centralisée d'un processus composé de 4 sous-processus (SP1, SP2, SP3, SP4)

De nombreux systèmes de gestion de workflows ont été développés et utilisé dans les dernières années, la majorité d'entre eux étant des systèmes commerciaux, tels que *forté Conductor, MQSeries/Workflow, SAP R/#Workflow...* ou des projets Open-source tels que *JBoss, JBPM, Bull Bonita* ou *ActiveBPEL...* Il existe également de nombreux prototypes issus du monde académique tels que *Meteor, ADEPT, SCENE, JOPERA ou SwinDew.*

Dans le fonctionnement d'un moteur de workflow, nous faisons la distinction entre deux modes de fonctionnement, à savoir l'exécution centralisée et l'exécution décentralisée.

- ***L'exécution centralisée*** est réalisée par une seule entité. La composition structurelle et l'orchestration de services utilisent une entité centrale pour exécuter et contrôler la logique de coordination. Celle-ci gère les interactions des différents services intervenant dans la composition de services. A chaque fois qu'une donnée est produite, elle est envoyée au serveur central [CS01]. Dans la Figure 3.1 nous montrons un exemple d'exécution centralisée d'un processus comportant quatre sous-processus SP1, SP2, SP3 et SP4. La requête du client (logiciel ou humain) est transmise à un moteur d'exécution. Ce dernier, d'après un processus défini au préalable, invoque les sous-processus SP1, SP2, SP3 et SP4 selon un ordre d'exécution.

- Dans le cas d'une ***exécution décentralisée***, la responsabilité de la coordination de l'exécution de la composition est partagée entre les différents fournisseurs des services, comme dans le cas des chorégraphies de services. Par exemple, une chorégraphie de services a une exécution distribuée où chaque partie contrôle son exécution mais aussi respecte les règles de collaboration établies [BDSN02]. Dans la Figure 3.2, nous montrons un exemple d'exécution décentralisée d'un processus comportant quatre sous-processus coopérants SP1, SP2, SP3 et SP4. Le client (logiciel ou humain) établit une requête qui est satisfaite par l'exécution des quatre sous-processus. La requête cliente est transmise au premier sous-processus (SP1) qui est exécuté. Le résultat de SP1 est alors transmis à SP2 qui l'utilise comme paramètre d'entrée. Le processus d'exécution de la composition est identique pour SP2 et SP3. Le sous-processus SP4 termine alors le processus et le résultat de son action est transmis au client.

Dans la suite, nous présentons une approche centralisée qu'est WS-BPEL et qui est la plus populaire de nos jours.

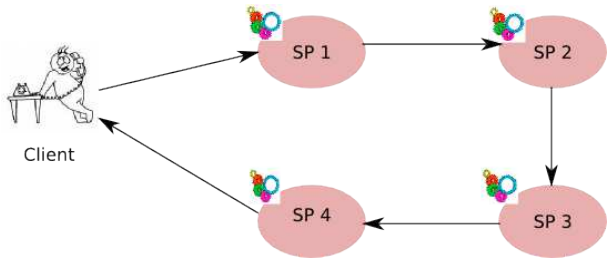


FIG. 3.2 – Exécution décentralisée d'un processus composé de 4 sous-processus (SP1, SP2, SP3 et SP4).

3.2 Composition centralisée avec WS-BPEL

Parmi les langages existants, nous avons choisi de présenter le langage WS-BPEL, la spécification la plus populaire pour l'orchestration des Services Web, WS-BPEL est le résultat d'une collaboration entre IBM, BEA Systems, Microsoft, SAP AG, Siebel Systems.

Conçu comme les autres standards pour les Services Web au dessus du langage XML, WS-BPEL permet la description d'une composition de services au travers d'un processus. Le résultat de cette composition est un Service Web composite dont la description WSDL peut être publiée dans un registre de services UDDI.

La spécification WS-BPEL permet de décrire :

- *un processus abstrait* qui représente une spécification du service composite qui aidera les clients de celui-ci à comprendre la logique du processus exposé. Il spécifie les échanges des messages entre les différents partenaires sans détailler le comportement interne de chaque participant. Cette spécification n'est pas exécutable mais elle peut être utilisée pour produire des implémentations différentes réalisant la même fonctionnalité métier.
- *Un processus exécutable* qui spécifie l'ordre d'exécution des activités, identifie les partenaires impliqués dans le processus (les services concrets) et spécifie le comportement dans le cas d'erreurs ou exceptions à l'exécution.

Concrètement, WS-BPEL est une couche supplémentaire au-dessus de WSDL. Un document WSDL définit les types des messages et des ports qui représentent les opérations proposées par le service défini et les modalités des différentes interactions où il peut intervenir. Ces informations sont utilisées par WS-BPEL pour spécifier le flot des actions à exécuter. Un document WS-BPEL est basé sur XML et peut être exécuté par un moteur d'orchestration qui agit comme un coordinateur central. Le moteur lit le document WS-BPEL et invoquera les différents services dans l'ordre spécifié. Le service composite est lui même présenté comme un Service Web qui peut à son tour être invoqué de la même manière.

Un processus WS-BPEL est constitué de :

1. Une liste de partenaires impliqués dans le processus composite et leurs différents rôles
2. Des mécanismes pour traiter les évènements (*event handlers*), les terminaisons fautives (*fault handlers* et *compensation handler*), les terminaisons normales (*termination handler*),
3. L'activité principale.

3.2.1 Déploiement

La définition du processus peut être accompagnée d'un descripteur de déploiement qui peut aider le moteur d'exécution WS-BPEL à identifier les partenaires impliqués dans la composition de services. Le moteur d'exécution doit réaliser les liaisons avec les services partenaires spécifiés dans le descripteur de déploiement. Toutefois, la réalisation de ces liaisons peut échouer car les fournisseurs de services choisis au préalable peuvent ne plus être disponibles au moment de l'exécution. Dans ce cas, les processus WS-BPEL ne peuvent pas exécuter la composition de services avec succès.

3.2.2 Dynamicité et distribution

WS-BPEL ne propose pas de mécanisme pour gérer la distribution. Celle-ci peut cependant être réalisée en enchainant les processus par des appels et en détournant la notion de lien partenaire (que nous expliciterons dans le Chapitre 6, Section 6.1.1). Les processus ne sont donc pas différenciés des services impliqués dans l'orchestration. La dynamicité quant à elle gérée aussi de façon *ad-hoc* en mettant à jour des variables contenant les adresses des partenaires. En effet, la dynamicité supportée par WS-BPEL consiste à manipuler les points d'entrée des services en faisant un changement de variable classique. Les points d'entrée peuvent être identifiés en utilisant le standard WS-Adressing[3]. L'activité **`receive`** reçoit¸ une donnée qui contient le point d'accès d'un service. Cette donnée remplace le point d'accès du service invoqué par l'activité **`invoke`** prochaine. Le remplacement se fait par l'exécution de l'activité **`assign`**. La gestion de la distribution et de la dynamicité n'est pas séparée du code métier des sous-processus et ne facilite pas la réutilisation de ceux-ci.

3.2.3 Implémentations de WS-BPEL

L'exécution des processus WS-BPEL est réalisée par un moteur d'exécution spécifique. Il n'existe pas de standard pour l'architecture des moteurs WS-BPEL et les applications qui existent sont le résultat d'une interprétation libre de la spécification WS-BPEL faite par les différents concepteurs. Une description détaillée de ces moteurs n'est pas le but de cet état de l'art, toutefois, nous ferons une présentation succincte de quelques moteurs existants ; parmi ceux-ci nous pouvons mentionner :

- **ActiveBPEL**. Ce produit implémente un moteur qui permet d'exécuter des processus qui suivent les spécifications WS-BPEL. Il est écrit en Java et il est également équipé d'une interface graphique pour une conception plus aisée des processus WS-BPEL. Le serveur d'applications inclus dans les outils ActiveBPEL est un serveur sous licence libre. Il est agrémenté par des outils sophistiqués développés de conception et de développement des services. Le moteur est administrable via des Services Web qui sont fournis avec le moteur. Cette interface permet entre autres le déploiement et la gestion des processus et des services exposés par ceux-ci sur le moteur.
- **Apache ODE** est un projet Open Source écrit en Java qui implémente les spécifications WS-BPEL. Il ne fournit pas de support visuel pour la conception des processus.
- **Oracle BPEL Process Manager** est un produit propriétaire développé par Oracle. Il fournit un environnement d'exécution pour les processus WS-BPEL et un outil visuel pour la conception graphique des processus.
- **IBM WebSphere** est une infrastructure complète pour l'intégration d'applicatifs basés sur le Web. Il fournit également un moteur d'exécution pour WS-BPEL

[3]http ://www.w3.org/Submission/ws-addressing/

WS-BPEL	
Paradigme :	Description XML du flot de contrôle et des appels aux services
Distribution :	Non / ad-hoc en modifiant les adresses des partenaires dans le code métier
Dynamicité :	Non / ad-hoc en modifiant les adresses des partenaires dans le code métier
Déploiement :	selon les éditeurs de moteurs d'exécution (descripteur de déploiement XML, archive jar)

TAB. 3.1 – Caractéristiques de l'exécution des processus métiers WS-BPEL

– **Orchestra** est un moteur d'orchestration de Services Web fourni par Bull et qui est conforme à la spécification WS-BPEL. Il permet de spécifier et d'exécuter des processus métiers. Le moteur Orchestra est réalisé sous la forme d'une application J2EE et profite donc des avantages comme la gestion des transactions et de la persistance offerts par les serveurs d'applications. Orchestra fournit un ensemble de fonctionnalités à travers des outils pour la spécification et l'exécution des processus, tels qu'un éditeur et validateur de modèles et une console d'administration. Orchestra intègre un moteur capable d'exécuter les modèles de processus. L'exécution des processus peut se réaliser avec ou sans la gestion de la persistance. De plus, Orchestra offre un mécanisme de reprise sur panne lors de l'exécution de processus, ce qui constitue un grand avantage pour la gestion de l'exécution des processus métiers.

3.2.4 Synthèse

Nous avons présenté dans cette section la composition de services centralisée avec des moteurs d'exécution des processus WS-BPEL.

Le rôle d'un moteur d'exécution WS-BPEL, qui fournit une exécution centralisée des processus, est d'assurer la réalisation des liaisons avec les services partenaires, de réaliser l'invocation des fonctionnalités fournies par ces services, de recevoir les réponses, et, en fonction de la logique du processus, d'invoquer la prochaine activité. La distribution et la dynamicité des processus n'est pas spécifiée mais il existe des solutions *ad-hoc* qui permettent de les réaliser. Bien que propre à chaque solution, le déploiement consiste en général à installer une archive contenant la définition du processus ainsi qu'une description des localisations des Services Web impliqués dans l'orchestration.

Dans le tableau 3.1, nous présentons un récapitulatif des fonctionnalités de WS-BPEL.

3.3 Systèmes d'exécution de Workflows Dynamiques et Distribués

L'approche workflow et les outils de gestion de workflow, bien qu'au cœur des organisations, souffrent des limitations des systèmes centralisés. La gestion de workflow décentralisée est une approche qui n'est pas nouvelle mais qui reste un sujet d'actualité comme

en témoigne la littérature. Elle a pour but de maîtriser les limitations de l'approche classique du workflow. L'idée de base de cette approche est de décentraliser la modélisation et l'exécution d'un processus, ce qui revient concrètement à avoir différentes portions d'un workflow qui s'exécutent en des lieux (moteurs) différents et répartis physiquement.

Dans cette section, nous étudions les travaux issus du monde académique qui proposent des plateformes d'exécution distribuées et/ou dynamiques de services. Pour chacune des plateformes étudiées, nous présentons l'approche et nous les caractérisons selon les critères définis dans la section 1.2.3. Ainsi nous étudierons en particulier, quand cela est possible, pour chaque système :

- **(C1)** : L'hétérogénéité des processus impliqués dans une composition et de leur système d'exécution
- **(C2)** : La prise en compte et la gestion de la dynamicité des processus métiers
- **(C3)** : La gestion du flux distribué de données entre processus
- **(C4)** : L'administration et la supervision des processus métiers distribués
- **(C5)** : Gestion de la complexité du déploiement

3.3.1 Exotica/FMQM [AMA+95]

Présentation : Le système de gestion Exotica/FMQM[4] est développé au centre de recherche IBM Almaden. Le but de ce projet est d'incorporer la gestion des transactions avancées dans un système de gestion de workflow dans le cadre des produits IBM. Le cœur essentiel de Exotica/FMQM est FlowMark qui est un produit antérieur d'IBM. Avec ce dernier, les plateformes spéciales pour l'échange de messages et de mises en queues sont utilisées. Le but de Exotica/FMQM est de diminuer la charge d'une entité centralisée qui peut devenir le goulot d'étranglement pendant l'exécution du processus. La Figure 3.3 montre les différents composants de l'architecture de Exotica/FMQM ainsi que leurs interactions.

(C2) Prise en compte de la dynamicité : Bien qu'Exotica/FMQM permette de définir des processus décentralisés, le détail du partitionnement n'est pas indiqué dans la littérature. Les nœuds d'exécution sont définis également de façon statique.

(C3) : Gestion du flux distribué de données entre processus : Le système de Exotica/FMQM est basé sur la distribution des processus en tenant compte des aspects de passage à l'échelle et la tolérance aux pannes. L'entité qui gère les données du processus est décentralisée avec des échanges de messages persistants entre les nœuds. Chaque nœud dispose d'un tableau de processus qui contient toute l'information statique pour les instances actives. Au cours de l'exécution les informations contenues dans le tableau de processus sont traitées par les processus des activités. Les données de processus sont passées d'un conteneur de sortie d'un nœud vers le conteneur d'entrée d'un autre nœud récepteur. Lorsqu'une instance de processus est démarrée, les messages d'entrée sont envoyés au nœud où la première activité de ce processus est localisée. Chaque nœud d'exécution héberge un gestionnaire de nœud (*node manager*) en charge de communiquer avec le nœud de définition du processus (*process definition node)* qui coordonne les différents sous-processus. De ce fait Exotica/FMQM ne permet pas la décentralisation complète, maintenant une unité de centrale (le gestionnaire de définition de processus), les opérations obéissant au paradigme client/serveur.

[4] FMQM, pour FlowMark on Message Queue Manager

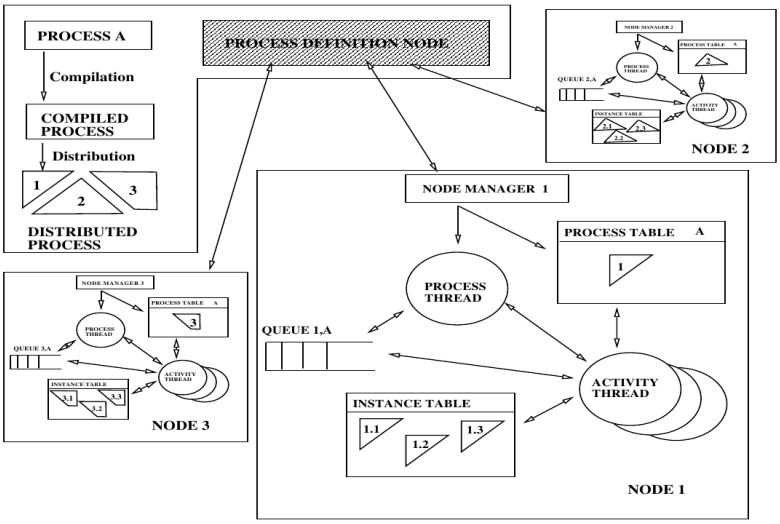


FIG. 3.3 – Distribution d'un processus sur plusieurs nœuds avec trois instances actives dans Exotica/FMQM [AMA+95]

3.3.2 SCENE/secse [CDM06]

Présentation : La plateforme SCENE (*Service Composition Execution Environment*) a été développée dans le cadre du projet IST *SeCSE* (Service Centric Systems Engineering)[5], ayant pour but de fournir des méthodes, outils et plateformes supportant l'ingénierie basée sur les services. Cette plateforme offre un langage de composition qui étend le langage WS-BPEL avec des règles qui sont destinées à guider les reconfigurations des liaisons au cours de l'exécution des compositions de services. L'implémentation de SCENE intègre un moteur d'exécution appelé PXE[6] ainsi qu'un moteur de règles, Drools[7], responsable d'exécuter des règles ECA[8], ainsi qu'un *exécuteur de liaisons (binder)*, responsable d'exécuter les actions de connexion aux services au moment de l'exécution selon les règles définies. L'architecture générale de la plateforme SCENE est montrée sur la Figure 3.4.

Le langage de définition des orchestrations exécutées sur la plateforme SCENE est basé sur deux aspects principaux : une partie correspondant au processus (écrite en WS-BPEL) qui définit la logique métier et une partie définie en terme de règles, qui indique comment l'application doit évoluer pendant son exécution. Le découplage entre le processus et les règles d'évolution permet de séparer clairement les propriétés non-fonctionnelles des propriétés fonctionnelles de l'orchestration. Le code WS-BPEL définit le flot de contrôle et le flot de données circulant au travers de la composition.

(C2) : Gestion de la dynamicité des processus métiers : Le mécanisme de dynamicité proposé par SCENE consiste à s'appuyer sur une règle ECA exprimée dans un langage déterminé et basée sur les propriétés du service (prix, temps de réponse, etc...). Cette requête

[5]http ://secse.eng.it
[6]http ://www.fivesight.com/pxe.html
[7]http ://www.drools.org
[8]Event Condition Action

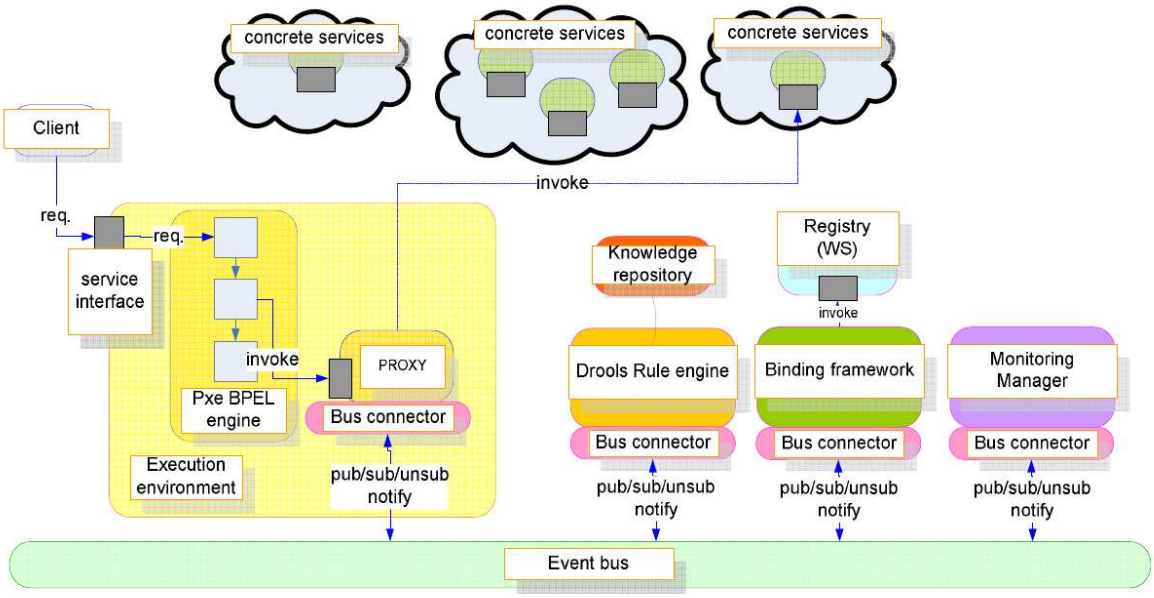


FIG. 3.4 – L´architecture de la plateforme SCENE

fait usage des références aux services. Lors de l'exécution de l'activité d'invocation, la requête est analysée par l'intergiciel supportant l'exécution de l'orchestration afin de trouver les services adéquats. Si lors de la requête plusieurs services sont retournés, une phase de sélection doit être accomplie afin d'en choisir un. De plus, il est possible de reconfigurer dynamiquement la composition lors de la disparition d'un service ou la détérioration de ses propriétés. SCENE supporte ainsi la liaison à l'exécution et la reconfiguration dynamique des compositions de services, cela sans ajouter de concepts dans le langage de composition. De plus, il est possible de changer les services choisis s'ils ne fournissent pas la qualité de service attendue. Ce type d'extension est techniquement réalisé avec le patron de conception de *proxy*[9] ; de cette façon, la modification du moteur d'exécution n'est pas nécessaire, et les composants supportant la nouvelle fonctionnalité sont plus facilement intégrés. L'utilisateur définit du WS-BPEL standard et définit les règles qui sont à appliquer selon l'occurence de certains évènements. Le rôle des *proxies* est de masquer au moteur WS-BPEL la présence du *moteur de règles* et de l'*exécuteur de liaisons*, en charge de gérer les liaisons réelles vers les services. A chaque *proxy* correspond une interface WSDL qui est exposée comme un service abstrait. Au moment de l'exécution quand un service abstrait est appelé à l'intérieur d'un processus, l'appel est transféré au *proxy* associé qui aura la responsabilité de transférer les appels au service concret.

Les composants de SCENE interagissent selon le paradigme *publish-subscribe.* Au moment de l'exécution, quand l'exécution du processus atteint une activité qui invoque un service externe, l'opération qui est réellement appelée est celle d'un *proxy.* Si le *proxy* ne se réfère à aucun service concret, il émet alors un événement qui sera analysé par le moteur de règles qui actionnera la liaison dès qu'un service concret sera disponible. Le contrôle est passé au *proxy* qui invoquera l'opération sur le service connecté, passant ensuite le

[9] *Objet d'interposition*

contrôle au moteur d'exécution de l'orchestration.

La plateforme SCENE a été étendue grâce à l'intégration d'un module permettant de résoudre les incohérences entre les interfaces et les protocoles des services invoqués [CDN08]. Un ensemble d'incohérences possible est défini, ainsi que des listes des stratégies possibles d'adaptation applicables à ces incohérences, exécutables *via* des scripts d'adaptation qui se basent sur l'expression des différences entre le service initial sélectionné pour l'établissement de la liaison (défini au moment de la conception), et les autres services concrets disponibles, candidats à la liaison dynamique.

(C5) : Gestion de la complexité du déploiement : Le déploiement se réalise via un élément appelé *Process Deployer*, qui est destiné à être appelé via l'interface de conception des compositions de services. Lorsque ce module de déploiement est activé, les fichiers WS-BPEL et WSDL correspondant à une orchestration sont modifiés de façon à ce que le moteur invoque les *proxies* plutôt que les réels services impliqués. Par ailleurs, tous les artefacts de code nécessaires au bon déroulement de la composition sont générés (*proxies*, archives de déploiement WS-BPEL). Le code source des *proxies* peut être manuellement modifié si des optimisations spécifiques sont requises.

3.3.3 JOPERA [PA05b]

Présentation : JOPERA est un projet développé au sein de l'Université de Lugano visant à fournir un modèle de composition de services et des outils de spécification, d'exécution et de supervision autonomiques d'orchestrations. Par défaut, il fournit un support pour la composition de Services Web, de méthodes Java et d'applications UNIX. Cependant, JOPERA offre des mécanismes d'extension supportant la composition de nouveaux types de services. JOPERA sépare complétement le flux de données et de contrôle.

Le formalisme de description des orchestrations de services est JVCL (*JOPERA Visual Composition Language).* JOPERA utilise JVCL pour décrire les orchestrations de services. JVCL est un langage graphique basé sur les diagrammes d'activités. Une composition est formée par un ensemble de tâches, de flots de contrôle et de flots de données entre les tâches. Les tâches ne sont pas typées dans JVCL. Une importante caractéristique du langage est qu'il permet de prédéfinir des tâches avec une sémantique associée ; ces tâches prédéfinies peuvent être utilisées dans la définition d'un nouveau processus et adaptées à leur contexte d'exécution. Cette propriété permet de créer une bibliothèque de tâches augmentant la réutilisation. Chaque tâche doit spécifier explicitement ses variables en entrée et ses variables en sortie. De plus, un flot de données est créé pour chaque paire de variables (entrée et sortie), une variable de sortie de tâche étant liée à une variable d'entrée d'une autre tâche, définissant ainsi un flot de données. Une tâche ne peut pas être démarrée tant que toutes ses dépendances de données ne sont pas satisfaites. Une tâche (*task)* peut être, soit une activité (*Activity),* représentant l'appel à un service, soit un sous-processus (*sub-process)* indiquant l'appel d'un autre processus.

L'architecture de JOPERA est montrée dans la Figure 3.5.

(C1) Hétérogénéité des processus impliqués et de leurs systèmes d'exécution : JOPERA permet d'étendre sa plateforme en ajoutant des tâches prédéfinies. Ces tâches indiquent non seulement leur structure, mais aussi leur sémantique, ce qui évite la modification du moteur d'exécution. Par ailleurs, JOPERA permet d'ajouter le support pour de nouvelles technologies à services. Les tâches sont alors définies avec un ensemble de paramètres d'entrée et de sortie ; le mécanisme permet d'indiquer comment ces paramètres

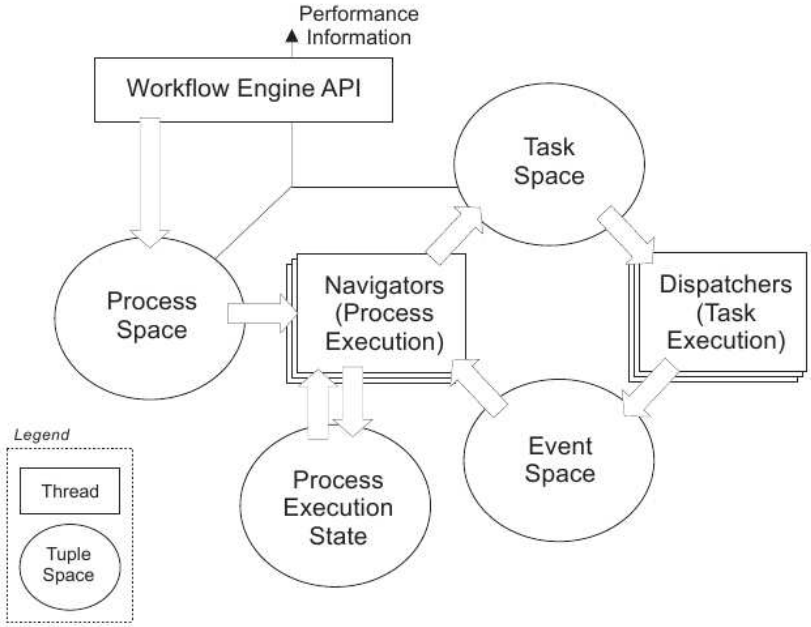


FIG. 3.5 – L'architecture logique du moteur d'exécution distribué JOPERA

seront utilisés par le service accessible *via* une technologie spécifique. JOPERA est aussi un outil permettant de construire des applications distribuées composées de parties hétérogènes.

(C2) Dynamicité des processus métiers : La liaison de services est réalisée à la conception ou au moment l'exécution. Toutes les informations nécessaires à l'instanciation et à l'exécution d'une tâche réalisée sont disponibles au moment de l'exécution selon un mécanisme de *liaison retardée*, principe selon lequel l'implémentation concrète d'un service est localisée le plus tard possible. Un registre de services est embarqué dans l'outil de spécification de l'orchestration, permettant de trouver les services correspondant aux besoins du concepteur. JOPERA utilise un mécanisme de liaison dynamique de services basé sur une activité prédéfinie dans une bibliothèque fournie par le canevas. Par exemple, il est possible de rechercher des services externes dans un registre UDDI et d'importer automatiquement leurs interfaces. Cela est réalisé en traduisant les descriptions WSDL dans la notation graphique JVCL : chaque opération de service est importée comme une activité séparée dont les paramètres d'entrée et de sortie correspondent aux parties des messages de requête et de réponse.

Il est aussi possible de reconfigurer dynamiquement le système, le nombre de *navigateurs* et de *dispatchers* pouvant augmenter ou diminuer sans avoir à arrêter le système complet. Ceci est réalisé *via* une interface de programmation qui permet de contrôler quel *thread* est en train de s'exécuter sur chaque nœud du *cluster*. Néanmoins, la plateforme ne propose pas de mécanisme permettant de gérer les erreurs associées à une portée donnée bien définie, ni pour signaler la remontée d'erreurs dans la logique d'exécution.

(C3) Gestion du flux distribué de données entre processus : L'exécution d'un processus commence avec une requête envoyée au travers de l'interface du moteur. Les processus peuvent être démarrés par les utilisateurs ou invoqués par d'autres processus. Le moteur met en queue les requêtes dans un espace de processus (*process execution space)*. Une nouvelle instance de workflow est alors créée et l'exécution du workflow peut alors commencer. Pour parvenir à cela, le *navigateur* utilise l'état courant de l'exécution d'un processus pour déterminer quelles sont les prochaines tâches qui doivent être invoquées, en se basant sur la complétion des tâches précédentes. Une fois que le navigateur a déterminé qu'une tâche est prête à être invoquée, le *t-uple* correspondant est stocké dans l'*espace des tâches (task execution request space).*

L'invocation des tâches est gérée par un composant *dispatcher.* Le nom de ce composant est dérivé de sa fonction d'exécution des tâches qui envoie des messages vers les fournisseurs de services. Une fois l'exécution de la tâche terminée, le *dispatcher* notifie le *navigateur* grâce à l'*espace des événements,* de façon à ce que le *navigateur* mette à jour l'état d'exécution du processus correspondant et poursuive son exécution.

La raison de séparer l'exécution des tâches de l'exécution globale du processus est basée sur l'observation que ces opérations ont un niveau différent de granularité. Il est attendu pour une tâche que son exécution dure moins longtemps que le temps pris pour le *navigateur* pour la lancer. Les auteurs affirment que grâce à leur approche, la plateforme JOPERA supporte les invocations parallèles de plusieurs tâches appartenant à un même processus. De plus, une tâche s'exécutant lentement n'affectera pas l'exécution des autres processus s'exécutant de façon concurrente, les opérations étant prises en charge par des fils d'exécution différents.

En découplant la navigation dans le processus de l'exécution des tâches, JOPERA est capable de passer à l'échelle selon deux dimensions, à savoir l'infrastructure et le découpage des processus. Dans le cas où une grande capacité d'invocation de tâches est nécessaire, le *dispatcher* est répliqué sur des nœuds multiples afin de gérer l'invocation des tâches concurrentes.

Dans JOPERA, le modèle de services exprime pour chaque service les paramètres d'entrée, les paramètres de sortie et un mécanisme de détection d'erreurs. Ainsi lorsqu'une tâche détecte une erreur, sur l'invocation d'un service par exemple, une autre tâche peut être assignée pour gérer la situation d'exception.

(C4) Administration et supervision des processus métiers distribués : Le moteur d'orchestration est distribué sur un cluster afin de supporter la réplication : le système utilise un contrôleur autonomique qui supervise la charge réelle et l'état du système. Il utilise cette information pour déterminer si le système s'exécute dans une configuration optimale ou si au contraire des actions de reconfiguration doivent être considérées. Par exemple si un pic de message de requêtes est détecté, il y aura plus de nœuds alloués sur le cluster pour les traiter.

3.3.4 SELF-SERV [SDM02]

Présentation : Le projet SELF-SERV (*compoSing wEb accessibLe inFormation and buSiness sERVice),* développé à l'Université de New South Wales, propose une plateforme permettant une composition aisée des services, en tenant compte du caractère dynamique et volatile des services existants au travers du Web. Cette plateforme permet d'exécuter des compositions de services décentralisées selon une approche pair à pair (*peer to peer).* Les compositions sont spécifiées grâce à un langage déclaratif propriétaire basé sur les dia-

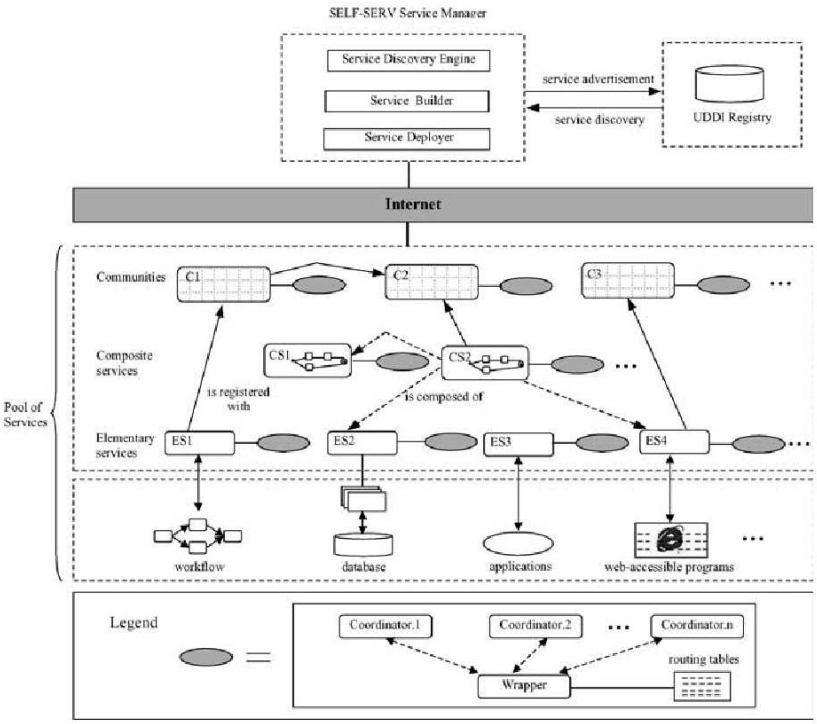


FIG. 3.6 – Architecture de SELF-SERV [BDS05]

grammes à état DSC (*Diagram State Chart).* Le système SELF-SERV est capable de composer des Services Web mais aussi d'autres types de services comme Corba, à condition que ces derniers possèdent une interface décrite en WSDL.

L'architecture générale de SELF-SERV est montrée dans la Figure 3.6.

(C3) Gestion du flux de données distribué entre processus : La coordination des exécutions dans SELF-SERV est distribuée sur l'ensemble des partenaires. Au moment de l'exécution, un modèle d'orchestration basé sur un système pair-à-pair est utilisé pour permettre le passage à l'échelle, le nombre de services à intégrer pouvant être important.L'architecture de SELF-SERV se compose de deux grandes parties, à savoir le gestionnaire de services (*Service Manager)* et le *Pool de services*. Le gestionnaire de services joue le rôle de registre dans l'architecture classique des Services Web et est fortement lié à UDDI. La composition de services est gérée par le *Pool de services* qui fait la distinction entre trois types de services : Les *services élémentaires* (*elementary services* dans la Figure 3.6 ESA, ES2, ES2 et ES4*),* les *services composés* (*composite services)* qui regroupent un ensemble de services (CS1 et CS2 dans la Figure 3.6), et les *communautés de services* (C1, C2 et C3 dans la Figure 3.6) qui représentent un ensemble de services proposant une même activité et permettant de sélectionner le fournisseur le plus adéquat pour une composition de services. Un *service* élémentaire est une application individuelle accessible via le web, et qui ne requiert pas d'autre appel à un Service Web. Un *service composé* agrège plusieurs Services Web. Une *communauté de services* est un ensemble de Services Web

qui offrent une fonctionnalité similaire mais dont les qualités de services sont différentes. L'exécution des services composites est contrôlée par des composants *coordinateurs* qui ont pour but d'initialiser, de contrôler et de superviser l'état de la composition. Les liaisons sont possibles au cours de l'exécution avec les membres d'une communauté. Des tables de routage (*routing tables)*, gèrent la connaissance extraite des diagrammes d'état telle que le flot de contrôle, ce qui permet de déterminer a quel autre coordinateur le contrôle doit être transféré. Cette connaissance permet la planification des *coordinateurs*. Les *coordonnateurs* ont la charge d'initier, de contrôler et de superviser l'état associé et de collaborer avec les autres pairs afin de mener à bien l'exécution.

(C5) Gestion de la complexité du déploiement : La solution SELF-SERV inclut un *déployeur de services* qui génère et déploie les tables de routage pour tous les états du composite décrit par le diagramme d'état. Le *déployeur* ne spécifie donc pas explicitement comment l'orchestration est distribuée entre les nœuds exécutant l'orchestration. Au contraire, un algorithme basé sur les informations des nœuds physiques et la distribution des services est utilisé pour calculer le plan de déploiement.

3.3.5 FOCAS [Ped09]

Présentation : FOCAS (Framework for Orchestration, Composition and Aggregation of Services) est un canevas de composition de services qui se base sur l'approche de l'orchestration. Un des buts de FOCAS est de pouvoir transformer une simple orchestration en une chorégraphie. Il fournit un environnement d'orchestration, de chorégraphie et d'exécution de services. FOCAS utilise l'idée de la séparation des préoccupations pour spécifier un processus. Dans FOCAS, chaque préoccupation est modélisée par un méta-modèle indépendant des autres et l'orchestration de services est réalisée par la composition de ces différents méta-modèles. L'outil FOCAS est extensible du fait que d'autres préoccupations peuvent être ajoutées pour décrire d'autres types d'applications basés sur des processus de manière générale. Ainsi l'outil est capable d'orchestrer plusieurs types de services par exemple des Services Web, des services OSGi, ou des services UPnP. FOCAS sépare le niveau logique dans lequel l'application basée sur les services est définie de façon abstraite du niveau physique dans lequel les services concrets sont sélectionnés dynamiquement et invoqués.

L'outil FOCAS utilise le concept de séparation des préoccupations pour spécifier un processus. Chaque préoccupation est modélisée par un indépendant des autres et l'orchestration des services est réalisée par la composition de ces différents meta-modèles. L'environnement est extensible, c'est-à-dire que des aspects fonctionnels (nouveaux modèles et fonctionnalités) ou non-fonctionnels (par exemple la sécurité ou les transactions) peuvent être ajoutés.

(C2) Gestion de la dynamicité des processus métiers : FOCAS permet de réaliser une correspondance dynamique entre les services abstraits et les services concrets. Cela permet, par exemple, de sélectionner dynamiquement le Service Web réel à invoquer, et de transformer les données applicatives dans le format requis par ce Service Web.

(C3) Gestion du Flux de données entre processus : Dans le but de réaliser les liaisons vers les Services Web impliqués dans une composition de service, FOCAS utilise un mécanisme qui cache la localisation physique des instances de services. L'architecture d'un nœud de FOCAS pour l'exécution distribuée est présentée dans la Figure 3.7. Pour chaque

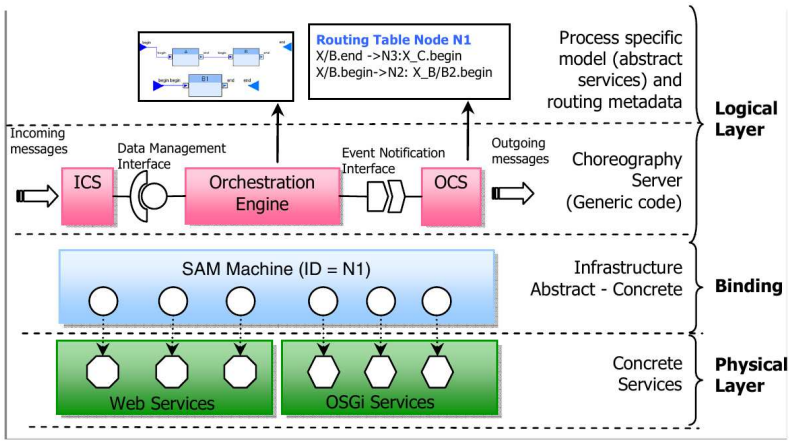


FIG. 3.7 – Architecture d'un noeud FOCAS

nœud utilisé dans l'exécution de l'orchestration, il est installé un serveur de chorégraphie. Ce serveur est composé d'un moteur traditionnel d'orchestration, augmenté de contrôleurs de chorégraphie en charge de router les données.

Le *OCS* (*Output Choregraphy Server*) reçoit des notifications d'événements concernant l'arrivée de données. Pour chaque événement reçu, il vérifie dans une table de routage si un flot de données de chorégraphie est associé à l'évènement. S'il en trouve un, alors il construit un message contenant l'information adéquate et appelle l'*ICS* (*Input Choregraphy Server*) du serveur de chorégraphie destinataire. Le *ICS* reçoit les messages venant d'un *OCS* et réalise l'action requise.

(C4) Administration et supervision des processus métiers distribués : FOCAS inclut un serveur d'administration de chorégraphie qui est associé à chaque serveur de chorégraphie. Ce serveur d'administration interprète un fichier de configuration qui indique ce qui doit être monitoré et où envoyer l'information monitorée ou les informations concernant les éventuelles situations d'erreurs. Il propose des interfaces permettant l'administration et la (re)configuration du serveur.

(C5) Gestion de la complexité du déploiement : Le déploiement d'une application composite est réalisé au moyen de plans de déploiement et d'une description de l'infrastructure physique, une fois les serveurs de chorégraphie installés sur chaque nœud. L'application peut alors être déployée, c'est-à-dire que chaque serveur de chorégraphie reçoit son sous-processus et la table de routage. Les sous-processus sont alors démarrés.

3.3.6 WISE [LASS00]

Présentation : WISE (Workflow based Internet SErvices) est un projet développé à l'ETH de Zurich, avec pour objectif de développer une solution unifiée qui peut être facilement déployée dans les petites et moyennes entreprises, et cela de façon la plus transparente possible. L'architecture de WISE fournit un moyen de définir, mettre en œuvre et superviser les processus métiers d'une entreprise virtuelle ainsi que de gérer la communication entre les différents participants au processus. L'infrastructure fournit (1) un moteur de

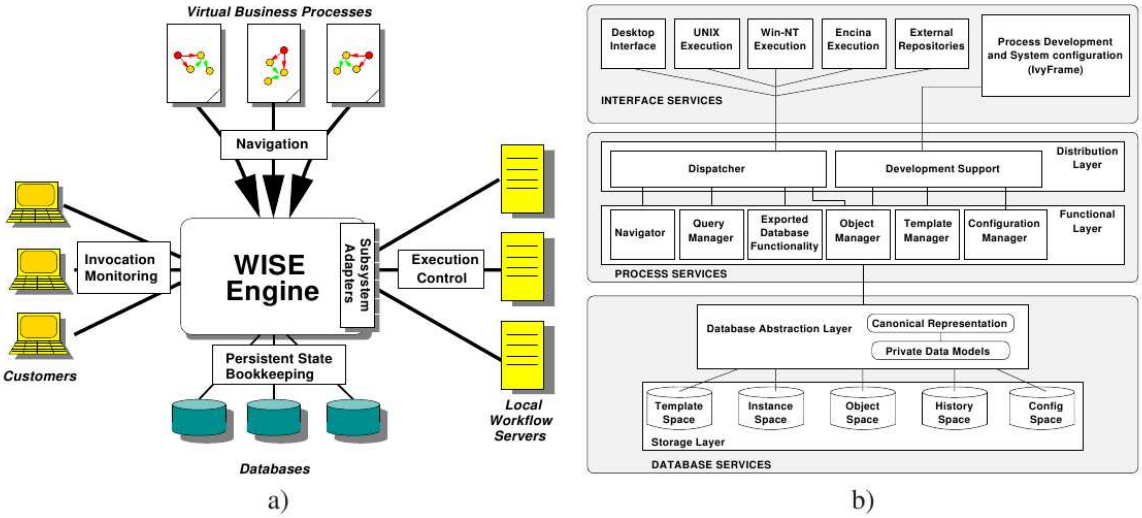


FIG. 3.8 – Le canevas d'exécution WISE [LASS00]

workflow servant de base au système distribué et contrôlant l'exécution des orchestrations de services et (2) un outil de modélisation qui sert à définir et à superviser les processus qui sont qualifiés de *virtuels* (car s'exécutant dans une *entreprise virtuelle)*.

Comme montré dans la Figure 3.8, WISE comprend un moteur d'exécution et un environnement de développement associé au composant d'exécution. WISE est organisé selon trois niveaux (Figure 3.8 (b)) : (1) les services de base de données, (2) les services de processus et (3) les interfaces de services. En particulier le niveau des services de processus contient tous les composants requis pour coordonner et superviser l'exécution d'un processus. Les deux éléments principaux de ce niveau sont le dispatcher et le navigateur. Le dispatcher est en charge de la distribution physique et agit comme un allocateur de ressources pour l'exécution du processus. il détermine sur quel nœud la prochaine étape sera exécutée, localise les nœuds adéquats, gère la charge et gère les opérations de communication avec les systèmes distants. Le navigateur agit comme l'ordonnanceur global, il navigue dans la description du processus stocké en mémoire, établissant ce qui doit être exécuté, ce qui doit être mis en attente, etc... Une fois que le navigateur a décidé quelle était l'étape suivante à effectuer, l'information est passée au dispatcher qui, quant à lui, ordonnance la tâche et l'associe à un nœud d'exécution.

(C1) Hétérogénéité des processus et de leur système d'exécution : WISE inclut un moteur de workflow fournissant la faculté d'exécuter des applications hétérogènes et distribuées. L'exécution d'un *processus virtuel* est réalisée par le moteur WISE (montré sur la Figure 3.8). Durant l'exécution, le moteur interprète la définition du *processus virtuel.* Comme ce processus est construit en utilisant les services d'un ensemble d'entreprises, le moteur de WISE peut être considéré comme un *moteur de moteurs*, dans le sens où il gère les interactions et le flot d'informations entre les systèmes des partenaires dans l'entreprise virtuelle.

(C3) Gestion du flux de données entre processus : Le moteur WISE fournit les fonctionnalités usuelles des systèmes de workflows. Les processus sont stockés de façon persistante et chaque étape de l'exécution est enregistrée dans la base de données du système pour des besoins de supervision ou de reprise sur erreur. Cela permet par exemple, de reprendre l'exécution là où elle a été laissée au lieu de recommencer au début du processus après une défaillance du système. La communication entre sous-processus est basée sur des règles ECA [HA99] et pilotée par le moteur WISE : la communication entre sous-processus et l'échange de données sont basées sur la production d'évènements que le moteur central WISE analyse. Les évènements sont des signaux typés et paramétrés qui peuvent être déclenchés par un sous-processus pour informer les autres sous-processus, qui ont souscrit à ces évènements, de certaines situations produites lors de son exécution, comme par exemple, la fin de l'exécution d'un sous-processus.

(C4) Administration et supervision des processus métiers distribués : WISE fournit les outils nécessaires de supervision qui collectent les données des processus s'exécutant mises à disposition par chaque sous-processus. Ces données comprennent par exemple le temps d'exécution total d'un sous-processus, l'analyse du flot de contrôle, etc... L'idée de base est de capturer l'historique entier de l'exécution pour un usage en temps réel (pour réaliser un équilibrage de charge, des réservations de ressources, ou faire de l'administration du système) ou un usage ultérieur à l'exécution (analyse des données, optimisation).

3.3.7 Crossflow [GAHL00]

Présentation : CrossFlow est un projet Européen ayant pour but de supporter l'externalisation dynamique de services pour les entreprises virtuelles. CrossFlow utilise une approche basée sur des contrats afin de décrire les relations entre les différentes entreprises et permettre ainsi la définition et l'exécution des processus inter-entreprises. Cette externalisation est réalisée de façon dynamique. Le but de CrossFlow est de développer et d'implémenter un mécanisme pour connecter les WFMS d'organisations différentes pour l'exécution de workflows inter-organisationnels.

Le cycle de vie d'une externalisation de service est marqué par trois phases : (1) établissement du contrat, (2) installation de l'infrastructure et (3) exécution du contrat et (4) annulation de l'infrastructure dynamique. :

1. La **phase d'établissement du contrat** permet de déterminer les services à externaliser au niveau processus. Les fournisseurs de service publient des modèles contenant les détails des services au sein d'un moteur de correspondance de services. Le client souhaitant externaliser un service doit contacter ce moteur, afin d'obtenir la liste des modèles de services des fournisseurs qui correspondent à sa requête. À la réception de cette liste, le client choisit un fournisseur avec lequel il établit un contrat.
2. Une fois un contrat établi entre un consommateur et un fournisseur, la seconde phase consiste à **construire dynamiquement une infrastructure,** et cela de façon symétrique, pour l'exécution de services en se basant sur les détails extraits de ce contrat.
3. La troisième phase consiste à **exécuter le contrat.** Le consommateur de service initie l'exécution du service externalisé en contactant le fournisseur.
4. Une fois l'exécution du service terminée, l'**infrastructure dynamiquement construite est abandonnée**.

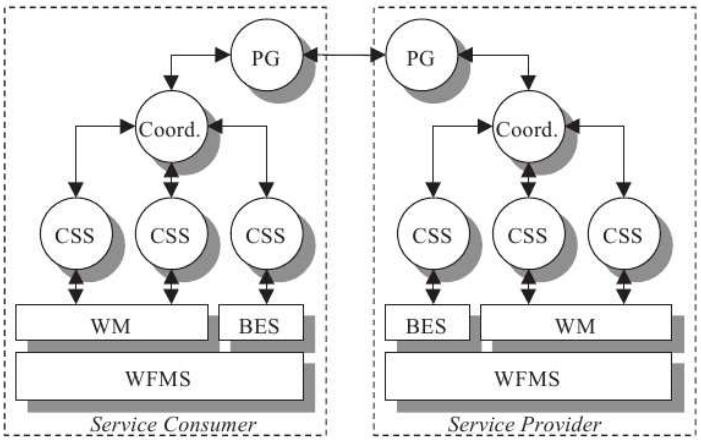


FIG. 3.9 – L'exécution d'un service dans l'architecture CrossFlow [GAHL00]

(C1) Hétérogénéité des processus et de leur système d'exécution : L'approche Crossflow ne permet pas directement de se baser sur les systèmes de gestion de workflows existants. En effet, l'exécution du workflow inter-entreprise est assurée grâce à des passerelles spécialisées configurées a priori et le mécanisme de communication entre ces passerelles est basé sur des interfaces CORBA-IIOP bien définies.

(C2) Gestion de la dynamicité des processus métiers : L'approche CrossFlow est basée sur un paradigme *fournisseur/consommateur* de service dynamique : lorsqu'une entreprise ne peut fournir l'exécution d'une activité d'un de ses processus métiers, et veut qu'un de ses services soit externalisé, alors elle l'externalise elle-même. La décision d'externalisation est prise durant l'exécution du processus nécessitant le service, le fournisseur étant choisi de façon dynamique. La sélection des services se fait par le biais de modèles de contrat dans lequel les offres de services et les requêtes y sont spécifiées.

Cependant, CrossFlow ne considère pas le fait que les fournisseurs puissent joindre et quitter le réseau à tout moment : la dynamicité de l'environnement n'est donc pas prise totalement en compte.

CrossFlow permet l'ajout de types de modules CSS, pouvant être requis par les domaines d'application (par exemple, pour supporter la rémunération automatique ou la gestion de confiance, ou la gestion de la QoS).

(C3) Gestion du flux de données entre sous-processus : L'architecture (montrée dans la Figure 3.9) consiste en trois types de modules : (1) un module coordinateur qui fournit la connexion entre tous les modules sur un site, (2) Une passerelle *proxy (PG)* qui fournit les interfaces externes aux autres organisations et (3) un module de support de la coopération qui fournit des services inter-organisationnels destinés à la gestion du workflow (CSS).

Dans la Figure 3.10, nous donnons l'exemple d'un consommateur qui externalise deux de ses activités (D et E) à un fournisseur externe.

(C4) Administration et Supervision des processus métiers distribués : Durant l'exécution d'un service externalisé, des informations de supervision provenant du fournisseur

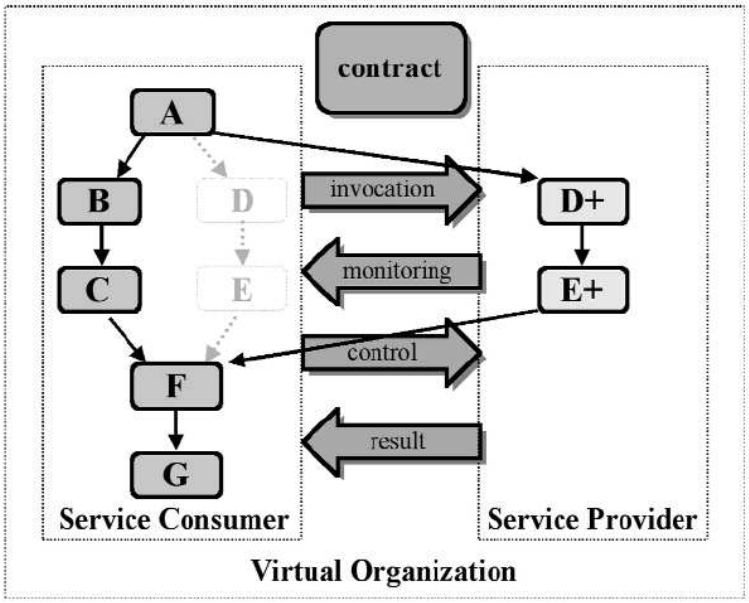


FIG. 3.10 – Crossflow : Exemple d´un consommateur qui externalise deux de ses activités (D+ et E+)

peuvent être envoyées au consommateur, soit par un appel direct, soit par le biais de notifications. Les évènements qui doivent être observés durant l'exécution du service externalisé sont spécifiés dans le contrat.

3.3.8 NIÑOS [GYJ11]

Présentation : NIÑOS est un moteur d'orchestration distribué basé sur les agents dans lequel plusieurs agents légers exécutent une portion d'un workflow global. NIÑOS exploite les fonctionnalités du système PADRES basé sur le paradigme *publish/subscribe*, incluant des capacités de routage basées sur le contenu, du support pour la corrélation d'évènements., des capacités d'équilibrage de charge.

NIÑOS traduit un processus métier décrit en WS-BPEL en un ensemble d'agents collaborant à l'exécution d'un processus global. Ces agents interagissent entre-eux en utilisant le paradigme *publish/subscribe*. Pour simplifier la gestion, NIÑOS permet aux processus d'être déployés et supervisés de façon centralisée.

Comme montré dans la Figure 3.11, l'architecture du système NIÑOS consiste en 4 composants : (1) le réseau de courtage PADRES, (2) les agents d'activité (du type `receive`, `invoke`, `reply`, etc ...), (3) les agents Web Services et (4) un gestionnaire de processus métiers.

Chaque élément d'un processus métier, une activité WS-BPEL par exemple, possède son propre agent d'activité, en fonction du type d'activité, qui est un client *publish/subscribe*. En général, un agent attend que son prédécesseur ait terminé son activité pour exécu-

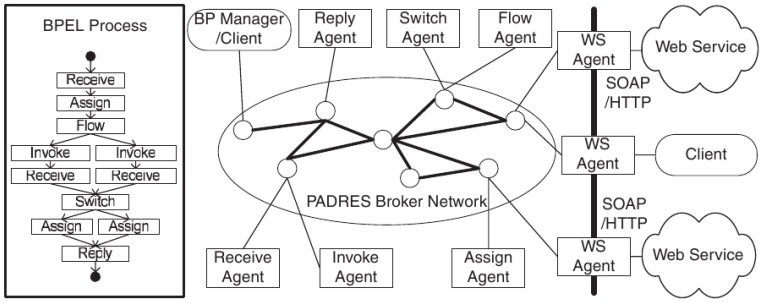


FIG. 3.11 – L'architecture de distribution d'un workflow dans NIÑOS

ter la sienne et finalement invoque les activités suivantes en publiant un événement de complétion. L'exécution d'un processus est guidée par les évènement et est naturellement distribuée.

(C2) Gestion de la dynamicité : Les agents Service Web supportent les partenaires définis statiquement au moment de la conception mais aussi des partenaires déterminés au moment de l'exécution.

(C3) Gestion du flux de données entre processus : Le processus métier déployé peut être invoqué par le biais d'un agent de Service Web qui est en charge de transformer la requête en une requête NIÑOS. La requête de service est un message de publication qui spécifie le processus et l'identificateur de l'instance. L'agent d'activité dans le processus (l'agent `Receive` dans l'exemple donné dans la Figure 3.11) reçoit cette publication et lance l'activité suivante (l'agent `Assign` dans l'exemple donné dans la Figure 3.11). Les agents exécutent leur activité et s'invoquent les uns après les autres en utilisant des messages *publish/subscribe* jusqu'à ce que l'instance du processus se termine.

Tous les agents d'activité peuvent être déployés sur un nœud exécutant un processus de manière centralisée, ou au contraire distribués au travers d'un réseau pour réaliser une exécution distribuée. Il est aussi possible d'effecteur une exécution hybride.

(C4) Administration et supervision des processus métiers : NIÑOS fournit une interface graphique de supervision qui permet de visualiser la topologie du réseau, le routage des messages,, et l'exécution distribuée des processus. La supervision est basée entièrement sur des message *publish/subscribe*.

(C5) Gestion de la complexité du déploiement : Le but du déploiement est d'installer une activité sur un agent en particulier en envoyant les messages, les souscriptions et les informations de l'activité générées lors d'une transformation WS-BPEL vers NIÑOS. NIÑOS permet aux entreprises de collaborer dans le but d'héberger des processus métiers : les systèmes communiquent selon une architecture fédérée basée sur PADRES.

Le gestionnaire de processus métiers qui est aussi un client *publish/subscribe*, transforme les processus métiers en messages *publish/subscribe* pour les agents d'activité, déploie le processus sur les agents disponibles sur le réseau et lance les instances des processus métiers, supervise et contrôle l'exécution. Chaque agent d'activité souscrit à des

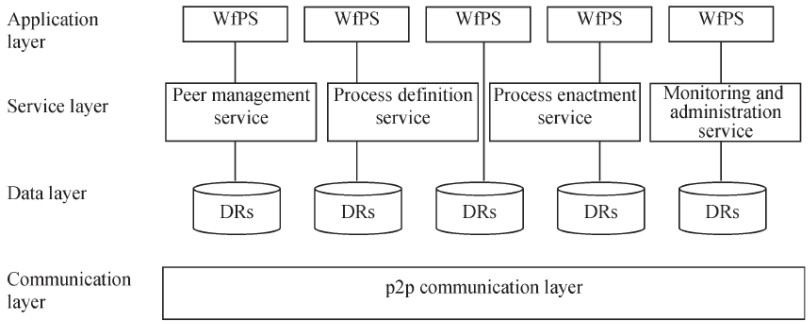


FIG. 3.12 – L'architecture du système d'exécution décentralisé de workflow SwinDew [YYR06]

messages de contrôle d'agent avec un identifiant d'agent unique, permettant au système d'installer une activité sur un agent en particulier.

3.3.9 SwinDew [YYR06]

Présentation : SwinDew (*SwinBurne Decentralized Workflow*) propose un modèle pair-à-pair pour l'exécution d'un workflow. Une définition de processus est découpée en un ensemble de tâches et distribuées sur les pairs.

La Figure 3.12 illustre SwinDew qui est défini sur quatre niveaux. Le niveau du haut, le niveau application, définit la sémantique destinée à l'application pour remplir les fonctions du workflow. Les services au cœur du système de workflow sont fournis par le niveau service qui inclut des services pour la gestion de pairs, pour la définition des processus, pour l'exécution, et pour la gestion de la supervision des processus. Le niveau données (*data layer),* consiste en plusieurs registres de données (*DRs)* qui stockent les informations relatives aux workflows (les définitions de processus et les informations sur les instances). Le niveau communication facilite les communications directes entre les participants du workflow.

(C3) Gestion du flux de données entre processus distribués : SwinDew adopte une structure flexible, faiblement couplée, et plate avec une absence de registre de données centralisé et d'un moteur centralisé pour la coordination. L'élément de base, le *pair,* est composé d'un participant et d'un ensemble de registres de données. Dans la plupart des cas, un pair est localisé sur une machine physique. Chaque pair est capable de communiquer avec les autres pairs directement pour mener à bien ses fonctions. Un pair est une entité autonome qui s'auto-gère et qui est capable de travailler à la fois de façon indépendante et de façon collaborative.

Une instance d'un processus est représentée par un réseau de pairs qui exécutent des tâches dans un ordre donné. L'exécution d'une instance de processus ne repose pas sur un moteur centralisé pour réaliser la coordination entre tâches. Les pairs communiquent entre-eux pour échanger les données applicatives durant l'exécution du processus. Chaque pair possède la connaissance sur la tache dont il est responsable de l'exécution et sur ses prédécesseurs et successeurs. Il est donc capable de déterminer qui lui a envoyé des données ou encore à qui en envoyer.

(C4) Administration et Supervision La supervision de l'exécution consiste au balayage continuel de l'état du processus, et des valeurs réelles des paramètres non-fonctionnels. Les données mises à jour sont récupérées de façon non systématique.

(C5) Gestion de la complexité du déploiement La phase d'instanciation du processus permet de créer les instances des tâches et de déterminer quels seront les pairs qui exécuteront ces tâches au travers de leur collaboration. Toutes les instances de tâches sont créées à des endroits différents.

3.3.10 Taverna [HWS+06, OAF+04]

Présentation : Le système de gestion de workflow Taverna est un gestionnaire de flux de données scientifique open source développé par le consortium ^{my} Grid, en Grande-Bretagne. Il a pour but de développer des intergiciels de haut niveau supportant des expérimentations bio-informatiques. L'objectif de Taverna est d'assister les scientifiques dans le développement et lors de l'exécution des workflows bio-informatiques sur une Grille. Taverna fournit des modèles de données, une interface graphique riche pour la conception de workflows ainsi que FreeFluo [Fre04], un moteur d'exécution de workflows. Le langage utilisé, SCUFL[10], pour décrire et interpréter les workflows est propriétaire.

(C1) Hétérogénéité des processus et de leur système d'exécution : Une fois que le workflow a été défini, il est exécuté par FreeFluo qui est un outil d'orchestration de workflows pour les Services Web supportant un sous-ensemble de WSFL[11] ainsi que du langage SCUFL. La flexibilité de FreeFluo[12] permet de se reposer sur un canevas d'orchestration qui n'est pas lié à un quelconque langage ou une quelconque architecture d'exécution.

(C3) Gestion du flux de données entre processus : Dans Taverna, un workflow consiste en une collection de processeurs connectés par des liens de données, qui établissent les dépendances entre les sorties d'un processeur et les entrées d'un autre. Un *processeur* est une entité de transformation qui accepte un ensemble de données en entrée et produit un ensemble de données en sortie. Pendant l'exécution d'un workflow, chaque *processeur* a un statut d'exécution courant (initialisation, attente, exécution, terminé, erreur ou annulé). Un processeur possède un type qui reflète son activité. Par exemple, il existe un type qui permet d'invoquer un Service Web selon sa description WSDL. Les ports d'entrée et de sortie sont alors déduits du contenu de la description. Un autre type de *processeur* permet l'utilisation de classes Java et de scripts locaux en tant que *processeurs* de workflows, du moment qu'ils exposent leur signature comme des ports d'entrée et des ports de sortie, comme spécifié dans le modèle.

Un workflow peut aussi posséder des données d'entrée et des données de sortie. Un workflow peut être considéré comme étant un *processeur* qui s'exécute et rend le résultat disponible sur son port de sortie.

Hormis ces liens basés sur les données, les processeurs peuvent être connectés selon des liens de coordination, ce qui permet de déclencher l'exécution d'un processeur une fois que celle de son prédécesseur est terminée. Ce niveau de contrôle est requis lorsqu'il existe un processus dont les activités doivent s'exécuter dans un certain ordre et qu'il n'existe pas de dépendances de données entre ces activités.

[10] Simple Conceptual Unified Flow Language
[11] Web Services Flow Language
[12] http ://freefluo.sourceforge.net

Taverna fournit un ensemble de primitives moins élaboré que celui de WS-BPEL. Cependant, le langage est orienté parallélisme, et le moteur d'exécution exploite cela pour permettre un degré de parallélisme élevé entre les processeurs sans pour autant qu'il ait été spécifié par le concepteur du workflow. Au niveau de l'exécution, le moteur fournit un mécanisme de multithreading pour augmenter les capacités de parallélisme. Les utilisateurs peuvent spécifier le nombre d'instances concurrentes qui seront envoyées au processeur en charge d'effectuer un appel parallèle. Le moteur orchestre l'exécution des processeurs de façon consistante avec les dépendances, et gère le flot de données au travers des ports des processeurs. L'exécution globale est conduite par les données, avec un modèle de type *push :* l'exécution d'un processeur est démarrée dès que tous ses données en entrée sont disponibles.

(C4) Administration et supervision des sous-processus : Taverna fournit aussi un environnement qui permet aux utilisateurs de manipuler les workflows, de sélectionner les ressources disponibles, et ensuite d'exécuter les workflows et de les superviser, c'est-à-dire de suivre leur exécution. L'outil permet aussi de parcourir les résultats intermédiaires et finaux obtenus [Tav04].

3.3.11 Kepler [ABJ+04] [LAB+06]

Présentation : Kepler un système de workflow graphique. Kepler est bâti sur le canevas Ptolemy II, développé par l'Université de Californie à Berkeley, qui est une architecture orientée flot de données. Le projet Kepler étend Ptolemy II pour exécuter des workflows scientifiques en ajoutant le support des invocations de Services Web et l'accès aux ressources de Grille. L'approche est basée à la fois sur les ressources et sur les services.

Ptolemy II est un environnement Java pour la conception et la simulation des systèmes concurrents. Le but de Ptolemy II est de permettre la construction de modèles basés sur la composition de composants appelés *acteurs.* Les acteurs sont des encapsulations d'actions particularisées, réalisées sur des jetons d'entrée pour produire des jetons de sortie. Les entrées et les sorties sont connectées au travers de ports disponibles sur les acteurs. Ils fournissent une abstraction commune utilisée pour englober différents types de composants logiciels, incluant des sous-workflows (workflows hiérarchiques), des Services Web ou des Services de Grilles. L'interaction entre les acteurs est définie par un modèle de calcul incluant des continuations, des événements discrets et une synchronisation des flots de données.

(C1) Hétérogénéité des systèmes d'exécution : Kepler supporte l'exécution de tâches spécifiées dans différents langages par le biais de JNI (Java Native Interface), ce qui donne à l'utilisateur la possibilité de ré-utiliser des composants existants et de cibler des outils de calculs en adéquation avec leurs besoins.

(C2) Gestion de la dynamicité des processus métiers : Dans Ptolemy II, il existe de le concept de mutation qui permet de changer la structure du workflow pendant son exécution. Le système d'exécution de Kepler permet de démarrer, mettre en pause et stopper l'exécution d'un workflow.

(C3) Gestion du flux de données : Les processus dans Kepler sont représentés comme des Graphes Acycliques Dirigés (DAGs = Directed Acyclic Graphs) en accord avec un langage appelé MoML (Modeling Markup Language). Ce langage inclut des constructions de données et de contrôle (par exemple des boucles). Kepler permet que les préoccupations concernant la communication soient séparées de la coordination globale du workflow. Le flot de données est alors défini dans un composant séparé appelé *Directeur.* Les acteurs concrets peuvent être hiérarchiquement composés et orchestrés par différents directeurs (qui sont en fait des *ordonnanceurs).*

3.4 Synthèse de l'état de l'art

Dans cette section, nous présentons une synthèse de l'état de l'art en suivant les cinq critères analysés. Le tableau 3.2 présente une classification des différents systèmes présentés. Dans ce tableau, lorsque le critère est respecté, nous le qualifions par le symbole **+**, et nous indiquons, quand cela est pertinent, l'approche sur laquelle il repose. Inversement, lorsqu'il n'est pas respecté, nous le qualifions par le symbole **-**. Si le critère est partiellement respecté, nous indiquons le symbole **~**. Certains critères n'ont pas pu être évalués, en général par manque de précision dans la littérature, nous indiquons alors le symbole **?**.

(C1) Hétérogénéité des processus impliqués dans une composition et de leur système d'exécution Dans une entreprise virtuelle, chaque partenaire dispose de son propre système de gestion de workflow, outils de travail, matériels et infrastructures logicielles et de sa propre organisation interne. Ainsi, il est nécessaire de fournir des mécanismes permettant de supporter l'hétérogénéité des systèmes de gestion de workflows existants des différents partenaires constituant l'entreprise virtuelle afin d'assurer la construction collective des compétences des entreprises à partir des infrastructures déjà existantes.

Dans notre état de l'art nous avons pu constater que quelques solutions permettent cette hétérogénéité. Néanmoins, la plupart des solutions développées pour la gestion de la coopération au sein d'une entreprise virtuelle sont limitées et supposent l'homogénéité des systèmes de gestion de workflows. En effet, la majorité des solutions existantes inclut ses propres systèmes d'exécution et nécessite qu'il soit installé sur chaque lieu d'exécution. Il existe plusieurs manières de gérer l'hétérogénéité des technologies, notamment en utilisant une sémantique particulière et des interfaces prédéfinies (en Corba, ou JNI). Une autre approche consiste à utiliser un *moteur de moteurs*, qui permet de se reposer sur un moteur centralisé qui coordonne les différents sous-processus s'exécutant sur des différents moteurs. Dans le cas des moteurs de workflows scientifiques, tels que Taverna, Kepler ou JOpera, les sous-processus sont englobés dans des tâches pour abstraire les technologies d'exécution, ce qui permet à l'utilisateur de pouvoir les réutiliser, sans pour autant avoir besoin de modifier ni les définitions des sous-processus ni les moteurs de workflows existants.

(C2) Dynamicité des processus métiers L'évolution des architectures basées sur les services a mis en avant le besoin de supporter le développement des systèmes exposant des capacités d'adaptation éventuellement autonomiques, où le comportement des applications évolue au cours de l'exécution. L'adaptation intervient à plusieurs niveaux, et inclut :

- Les changements de l'infrastructure de l'application, où la qualité de services change ou les services deviennent indisponibles.

- Les changements de contexte et de la localisation de l'application, où le workflow doit être à même de remplacer un service par un autre potentiellement différent en terme de paramètres et de propriétés.
- Les changements des utilisateurs, de leurs préférences et des contraintes qui nécessitent une personnalisation de l'application comme un moyen d'adapter le comportement de l'application à un utilisateur en particulier
- Les changements dans les fonctionnalités fournies par les services qui nécessitent de modifier la façon avec laquelle les services sont composés et coordonnés

Dans notre état de l'art, nous avons pu constater que WS-BPEL offre peu de support pour le changement dynamique et ne supporte pas la définition explicite de cette dynamicité. Dans la plupart des systèmes présentés, la liaison se réalise au moment du déploiement en faisant correspondre une définition abstraite de service à un service concret, retardant ainsi le plus tard possible le choix des services participants. Lorsque les assemblages de services sont définis statiquement, ce qui est le cas avec les approches telles que WS-BPEL, il est possible de changer dynamiquement et au moment de l'exécution la connexion à un service, mais cela de façon *ad-hoc*. Ce facteur de dynamicité apparait comme nécessaire pour par exemple, sélectionner le service adéquat en fonction d'une situation donnée (disponibilité d'un service) ou encore de sélectionner un service selon ses performances et les besoins d'un client. Plusieurs propositions provenant du monde académique ont été faites pour traiter ce problème. Parmi elles, nous avons mentionné SCENE, un environnement d'exécution des processus qui est capable de réaliser des liaisons adaptatives. SCENE résout le problème mentionné en permettant l'adaptation des compositions en utilisant des préférences d'adaptation qui accompagnent la définition WS-BPEL et en se basant sur le concept du *proxy*. Le *proxy* fait office d'intercepteur de la requête destinée au service. Il permet soit d'adapter une requête dans un format requis par un service, soit de changer dynamiquement le service ciblé pour satisfaire à des besoins de QoS.

Le changement dynamique de la structure du workflow est peu abordé dans la littérature car il induit plusieurs problèmes au niveau de la définition du workflow. Ce changement revient à remplacer un sous-processus par un autre, ou encore la composition interne d'un sous-processus. La question qui se pose est celle de la modification de la définition lors d'un changement dans la structure du workflow. En fait, plusieurs cas de figure sont à considérer et interviennent à des moments particuliers du cycle de vie du processus : soit les modifications peuvent être propres à l'instance et n'impactent pas la définition du processus, soit les modifications affectent la définition, et dans ce cas, les instances à venir des processus.

(C3) Gestion du flux de données entre processus Afin de supporter les communications inter-organisationnelles, plusieurs mécanismes ont été proposés. Certains, comme Exotica/FMQM reposent sur une entité centralisée pour coordonner l'enchainement des différents sous-processus. Cette approche est donc semi-décentralisée puisque le contrôle est affecté à une seule entité. Le problème de la coopération inter-organisationnelle a été traité en utilisant la notion d'agréments et de contrats pour définir les relations métiers entre les organisations, par exemple avec l'approche CrossFlow.

De notre état de l'art se dégagent deux principaux modes de coordination des sous-processus, à savoir d'une part des communications directes entre sous-processus (avec une approche pair à pair, où le contrôle et les données sont passées de sous-processus en sous-processus, ou encore grâce à des agents responsables d'exécuter le sous-processus et de transmettre les données au suivant), et d'autre part des communications basées sur les évènements (les sous-processus s'abonnent à des évènements produits par les autres

sous-processus, reçoivent les données applicatives et se déclenchent selon ces évènement).

De façon plus générale, nous retiendrons qu'un sous-processus consomme et produit des données. Il reçoit des données en entrée et met à disposition des données en sortie.

(C4) Administration et supervision des processus métiers distribués Dans le cas de processus décentralisés il est fastidieux d'effectuer des opérations d'administration et de supervision. En effet, non seulement la distribution des sous-processus mais aussi l'hétérogénéité des moteurs ne facilite pas la tâche de l'administrateur. Cependant, dans le cadre des processus métiers, l'administration est un aspect important. La plupart des solutions propose des outils et interfaces graphiques afin de visualiser et de manipuler les données. Tout comme pour le critère précédent, nous avons pu observer deux manières de faire, à savoir l'abonnement à des évènements (messages *publish/subscribe)* ou à des communications directes qui consistent à balayer continuellement le processus. Les interfaces d'administration permettent de gérer le cycle de vie du processus métier distribué et de récupérer les résultats applicatifs, intermédiaires ou non, obtenus.

(C5) Gestion de la complexité du déploiement Le but du déploiement dans toutes les différentes solutions présentées dans cet état de l'art est relativement commun. Celui-ci consiste à installer tous les éléments nécessaires (que ce soit des *proxies,* des agents), et/ou à effectuer des opérations (par exemple traduire des messages inter-processus en messages *publish/subscribe)* à l'exécution sur toutes les localisations impliquées dans l'orchestration.

3.5 Conclusion du chapitre

Dans ce chapitre, nous avons présenté un état de l'art sur les systèmes d'exécution de workflows décentralisés et dynamiques. Chacune des approches présentées ci-dessus apporte une ou plusieurs idées intéressantes. Ainsi il est important de tirer les points forts de ces approches afin de les réutiliser et/ou de s'en inspirer. Nous retiendrons :

- Une gestion de l'hétérogénéité des moteurs d'exécution grâce à la définition d'interfaces communes.
- Une gestion de la dynamicité grâce au patron de conception de *proxy.*
- Une communication inter-processus qui consiste, dès la fin d'un sous-processus, à envoyer les données applicatives à son sous-processus suivant.
- La possibilité d'administrer et de superviser les processus métiers distribués, en offrant des interfaces de contrôle.
- La capacité de déployer une définition de processus distribué et de son système d'exécution sur chaque lieu impliqué dans l'entreprise virtuelle.

Nous nous appuierons sur ces points forts pour bâtir notre proposition d'exécution de workflow décentralisés et dynamique que nous présenterons dans le chapitre suivant.

	légende : **+** : respecté **-** : non respecté **~** : respecté	**Ref**	**C1 :Hétéro-généité**	**C2 : Dynamicité**	**C3 : Gestion du flux**	**C4 : Admin et Supervision**	**C5 : Déploiement**
1	**Exotica/FMQM**	[AMA+95]	-	-	~ Messages persistents	?	?
2	**SCENE**	[CDM06]	-	+ proxy	-	?	?
3	**JOpera**	[PA05b]	~	+ Système autonome	+ pair-à-pair	~	?
4	**SELF-SERV**	[SDM02]	-	-	+ pair-à-pair	+	+
5	**FOCAS**	[PE09, PDE09]	-	+	+	+	+
6	**WISE**	[LASS00]	+	-	+	+	?
7	**CrossFlow**	[GMK+09]	+	~	+	-	~
8	**NIÑOS**	[GYJ11]	-	+	+ publish/subscibe	+	+
9	**SwinDew**	[YYR06]	?	-	+ pair-à-pair	+	+
10	**Taverna**	[HWS+06, OAF+04]	+	-	- processeurs - modèle de type *push*	+	?
11	**Kepler**	[ABJ+04, LAB+06]	+	+	+ acteurs (ports d'entrée et de sortie)	?	?

TAB. 3.2 – Classification des différents systèmes d'exécution de workflows

Nous vivons chaque jour dans des environnements virtuels définis par nos idées.

Michael Crichton, Extrait de *Disclosures*.

Chapitre 4

Positionnement de notre travail

Contenu du chapitre :

4.1 Choix d'une approche basée sur les composants 81
4.1.1 Approche à composants 82
4.2 Notre proposition : Conception d'une orchestration décentralisée et dynamique en utilisant une approche à composants 84
4.2.1 Positionnement de notre approche 84
4.2.2 Vue d'ensemble de notre méthodologie 87
4.3 Conclusion 88

Dans les chapitres précédents, nous avons présenté le contexte dans lequel notre contribution s'inscrit ainsi qu'un état de l'art concernant les systèmes d'exécution de workflows décentralisés et/ou dynamiques. Dans ce chapitre, nous positionnons notre contribution. La Section 4.1 justifie le choix d'une approche à composants pour bâtir notre solution d'orchestration décentralisée et dynamique. Ensuite, nous positionnons notre approche basée sur un modèle à composant dans la Section 4.2.

4.1 Choix d'une approche basée sur les composants

Après avoir présenté le contexte dans lequel notre contribution se situe, et l'état de l'art relatif à celle-ci, nous justifions le choix d'une approche basée sur les composants pour répondre à notre objectif général qui est de fournir un environnement d'exécution dynamique d'orchestrations décentralisées qui répond aux cinq critères et donc qui permette de :

- Exécuter des orchestrations décentralisées au travers de plusieurs organisations, et donc de s'adapter à la diversité des systèmes d'exécution
- Exécuter de façon dynamique l'exécution d'un workflow distribué
- Gérer le flux distribué de données entre processus
- Administrer et superviser les processus métiers distribués
- Gérer la complexité du déploiement des processus métiers distribués

Un des aspects cruciaux du positionnement de notre travail est l'utilisation d'un modèle basé sur les composants comme paradigme de composition. Nous avons choisi de nous reposer sur un modèle à composants pour exécuter les orchestrations distribuées. Cela revient naturellement à combiner les workflows et les composants. Les raisons d'un tel choix est que les composants peuvent facilement encapsuler les services, et que les architectures basées sur les composants sont mieux structurées, donc plus facilement adaptables que les solutions purement basées sur les services. Ce point de vue est déjà adopté par SCA où les services requis sont vus comme des composants dans l'architecture résultante. Comparativement à d'autres façons de composer les services ensemble, un composant encapsulant un service rend ses dépendances explicites vers les autres services (qui sont eux aussi des composants). Par conséquent un modèle de composition basé sur les composants permet d'obtenir une vue complète et très structurée de architecture complète qui implique des services. Nous pensons que les modèles de workflows et de composants sont complémentaires et cette complémentarité est prometteuse quant à l'évolution des SOAs. Avec cette approche, la technologie de workflow peut bénéficier de la capacité de reconfiguration des composants et peut être ainsi modifiée durant l'exécution (par exemple quand un service impliqué n'est plus disponible et doit être remplacé).

Nous proposons d'embarquer un workflow dans un composant hiérarchique, distribué, et dynamiquement re-configurable. Cette vue d'un workflow embarqué dans un composant, qui correspond à une composition de service doit être modifiable dynamiquement. Pour cela, nous devons nous reposer sur un modèle à composants qui permette **la reconfiguration, durant l'exécution,** des liaisons entre composants, et de la modification du contenu des composants.

Un modèle à composants hiérarchique permet de construire des composants à partir de plusieurs autres. Cela permet naturellement la réutilisation et **l'architecture orientée service** que l'on peut construire avec ces composants devient **multi-niveaux et hiérarchique**. Si, de plus, les composants à l'intérieur d'un composite peuvent être distribués sur plusieurs machines, un composant composite permet **d'encapsuler une représentation réelle de la localisation physique de chaque service ou sous-processus impliqué.**

Comparé à une architecture mise à plat, une architecture hiérarchique peut être utile pour, par exemple, **délimiter clairement et déléguer les responsabilités d'administration.**

Dans la suite, nous allons maintenant visiter les différentes spécificités d'un modèle à composants, et voir comment elles semblent pertinentes pour supporter l'orchestration de services distribuée satisfaisant les cinq critères.

4.1.1 Approche à composants

Dans ce travail de thèse, l'objectif est de concevoir un canevas dynamique pour la gestion décentralisée d'une orchestration. Ainsi à partir d'un ensemble de sous-processus coopérants entre-eux, nous devons proposer une vue unifiée et globale de cette orchestration. Une idée naturelle est de projeter chaque sous-processus sur un composant. Nous expliquons dans cette section quels sont les avantages d'utiliser une approche à composants pour concevoir et implémenter une telle projection.

Dans le Chapitre 2 , Section 2.2.4.1, nous avons défini l'approche à composants. Comme décrit dans [KM07] et [Gar00], une approche basée sur les composants offre plusieurs avantages :

- **Le niveau d'abstraction** : les architectures logicielles aident à comprendre la conception haut niveau des systèmes logiciels complexes. Elles peuvent exposer les contraintes

haut niveau des systèmes et peuvent guider les décisions architecturales. Un niveau d'abstraction approprié peut faciliter la conception des préoccupations comme les changements dynamiques du système, dans le sens ou l'on peut visualiser les modifications structurelles à appliquer à celui-ci.
- **Ré-utilisation** : Les descriptions architecturales fournissent la capacité de ré-utiliser les composants à plusieurs niveaux : depuis des parties d'un système logiciel jusqu'à de larges composants.
- **Construction** : Les descriptions architecturales guident le processus de développement, montrant les dépendances entre les composants majeurs et en donnant de l'information utile pour les implémenter.
- **Évolution :** Les architectures logicielles aident à comprendre comment un système logiciel peut évoluer et aident à estimer le coût de ces modifications.
- **Analyse :** Un système logiciel conçu avec une approche architecturale peut être analysé selon différents aspects comme la vérification de la cohérence, ou la conformités selon certaines contraintes, etc. . .
- **Possibilité de passage à l'échelle** : Potentiellement, un composant peut être ré-utilisé dans un système plus gros, rendant les approches architecturales appropriées pour les systèmes à large échelle.

En outre, une application conçue avec une approche à composants impose d'avoir une architecture logicielle claire et associée au système à composants. Dans la suite nous explicitons deux propriétés essentielles qu'une approche orientée composant doit posséder à notre sens, à savoir la structure hiérarchique et la reconfiguration dynamique structurelle, si l'on veut réaliser notre objectif.

4.1.1.1 Structure hiérarchique

Le concept de hiérarchie apparaît souvent dans la conception des systèmes à composants. La hiérarchie, ici dans le sens de composition hiérarchique, représente la possibilité de représenter la structure d'un composant complexe (qui peut être appelé *composant composite*) comme la composition de sous-composants de la même nature de ce dernier. Comme la récursion doit s'arrêter à un moment, l'arbre de la composition hiérarchique doit posséder des feuilles qui peuvent être appelée composants primitifs. L'avantage premier est la possibilité de composer ou de décomposer récursivement un composant en plusieurs sous-composants élémentaires.

Ceci représente une fonctionnalité intéressante lorsque l'on conçoit une application entière, spécialement dans le cadre de conception d'une orchestration de globale services dans laquelle nous devons agréger des sous-processus. Composer des composants de façon hiérarchique facilite leur composition, car à chaque niveau de la hiérarchie, le choix de composants possibles est moins important, et aussi plus facilement guidé par un ensemble de buts ou par l'utilisateur final.

La hiérarchie est aussi nécessaire pour le passage à l'échelle de l'administration du système. Administrer un système à large échelle, peut sembler difficile mais une structure hiérarchique peut aider à réduire le nombre de composants à considérer à un niveau hiérarchique donné.

Les composants peuvent être composés de manière ascendante, depuis les composants élémentaires pour former une application haut-niveau, ou de manière descendante : un composant de haut niveau est construit par la composition de composants de plus bas niveau. Ainsi les composant doivent être capables d'être composés hiérarchiquement, dans le sens où un composant dans le graphe peut contenir plusieurs autres composants, la

composition de ces sous-composants formant un système plus complexe.

4.1.1.2 Reconfiguration dynamique

La propriété de reconfiguration dynamique permet de modifier le comportement d'un composant. Cela peut consister à ajouter un nouveau composant à l'intérieur d'un composite, de supprimer un composant ou de changer les connexions entre composants. Ces reconfigurations structurelles représentent un mécanisme puissant et élégant, car elle permettent de modifier le comportement d'un (sous-)système de composant en appliquant des mécanismes génériques.

4.2 Notre proposition : Conception d'une orchestration décentralisée et dynamique en utilisant une approche à composants

4.2.1 Positionnement de notre approche

Un composant peut être implémenté et ensuite publié, afin d'être invoqué, en utilisant différentes technologies. Il peut par exemple être écrit en Java, et être accessible via le protocole Java RMI. Il peut être écrit dans un langage XML, comme WS-BPEL, et accessible via l'utilisation de Services Web ou de protocoles REST. Cette polyvalence concernant la technologie utilisée est un avantage, dans le sens où, dans une SOA construite à base de composants, il sera aisé de mélanger des composants hétérogènes.

Néanmoins, dans ce travail de thèse, les objectifs que nous nous fixons nous mènent à nous concentrer sur le cas où un composant est implémenté grâce à une technologie de workflow (par exemple WS-BPEL). Au moment de l'exécution, un ou plusieurs moteurs de workflows doivent être associés à l'exécution des flots définis dans le composant. Afin de tirer au maximum parti des bénéfices des capacités d'évolution apportées par une approche basée sur les composants, nous considérons aussi le moteur de workflow et chaque dépendance à un service comme un composant dans l'architecture résultante.

Ainsi, d'un point de vue d'une exécution et d'une spécification de service, les composants forment le cadre d'assemblage des services selon une vue spatiale, tandis que les workflows permettent de définir les sous-processus selon une vue temporelle. Cette séparation des rôles correspond aussi à la vue adoptée par la spécification SCA-BPEL [SCA07b]. Curbera [Cur07]renforce l'idée que la combinaison de SCA et BPEL est intéressante : SCA montre la structure d'une application composite tandis que WS-BPEL détermine le flot de contrôle pour chaque opération impliquée dans le service ainsi défini. L'implémentation des processus WS-BPEL dans un composant SCA permet à un composant d'être écrit en WS-BPEL et ainsi d'être déployé et assemblé avec d'autres composants SCA.

En d'autres termes, nous adoptons l'idée que les compositions basées sur les composants couvrent les compositions de services exprimées par le biais de langages d'orchestration.

Nos travaux se placent à l'**intersection entre les modèles de composition dans le temps et les modèles de composition dans l'espace**. La combinaison de ces deux modèles revient à raisonner selon deux dimensions que sont le temps (en terme d'ordre d'exécution) et l'espace (en terme de dépendances fonctionnelles).

L'originalité de notre travail repose sur le mélange de ces deux notions, et donc de leurs avantages : de ces deux modèles de composition, on obtient, comme montré dans la Figure

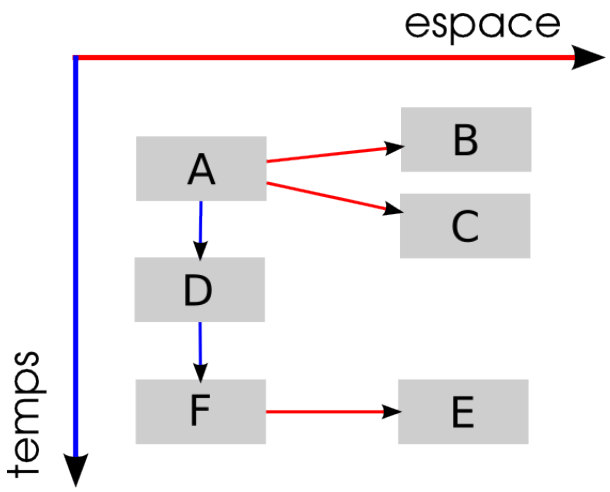


FIG. 4.1 – La composition dans l'espace combinée à la composition dans le temps

4.1, ***un composant défini par deux dimensions : la composition dans le temps et la composition dans l'espace.*** Les orchestrations peuvent bénéficier de la propriété de reconfiguration des composants, et donc être modifiés à l'exécution (par exemple lorsqu'un service n'est plus disponible et doit être remplacé). En nous reposant sur la composition dans l'espace, nous pouvons ainsi représenter les dépendances fonctionnelles qui peuvent exister entre les différents services utilisés dans une orchestration. La Figure 4.1 illustre une telle composition : par exemple, le composant A est lié structurellement au composant B et au composant C. Il est lié dans le temps au composant D : une fois l'exécution du composant A terminée, l'exécution du composant D peut commencer.

Dans le cas d'une composition exprimée par un workflow, il s'agit d'un contrôle des invocations des services externes ; le flot de contrôle de l'agencement des services est *explicite*. Ce flot de contrôle externe explicite d'une composition de services est caractéristique dans les situations dans lesquelles le consommateur des services ne détient pas de contrôle des fournisseurs des services impliqués, comme c'est le cas des Services Web.

Dans le cas d'une composition structurelle, le contrôle de la composition de services est interne et connu seulement par son développeur. De plus, celui-ci détient un peu plus d'informations sur la réalisation des services utilisés, par rapport au réalisateur d'une composition par workflow. Pour spécifier une composition structurelle, l'intégrateur de services doit connaître plus de détails techniques, tels que la vue externe des composants impliqués, définie par les services fournis et les services requis.

La Figure 4.2 illustre notre approche en donnant une vision composant de l'exemple présenté dans la Figure 4.2. A chaque sous-processus impliqué dans le processus global d'organisation de voyage, nous faisons correspondre un composant englobant la logique d'orchestration de ce sous-processus. Les sous-processus sont alors reliés entre-eux par une relation temporelle.

Notre objectif est également de proposer une approche non intrusive, au niveau des définitions, pour la mise en œuvre des compositions de services. En effet, les langages d'orchestration, par exemple WS-BPEL, et les plateformes d'orchestration ont été élaborés avec un effort global de standardisation qui a permis l'émergence des Architectures Orientées Service. Notre approche ne doit donc pas altérer par exemple le langage de définition

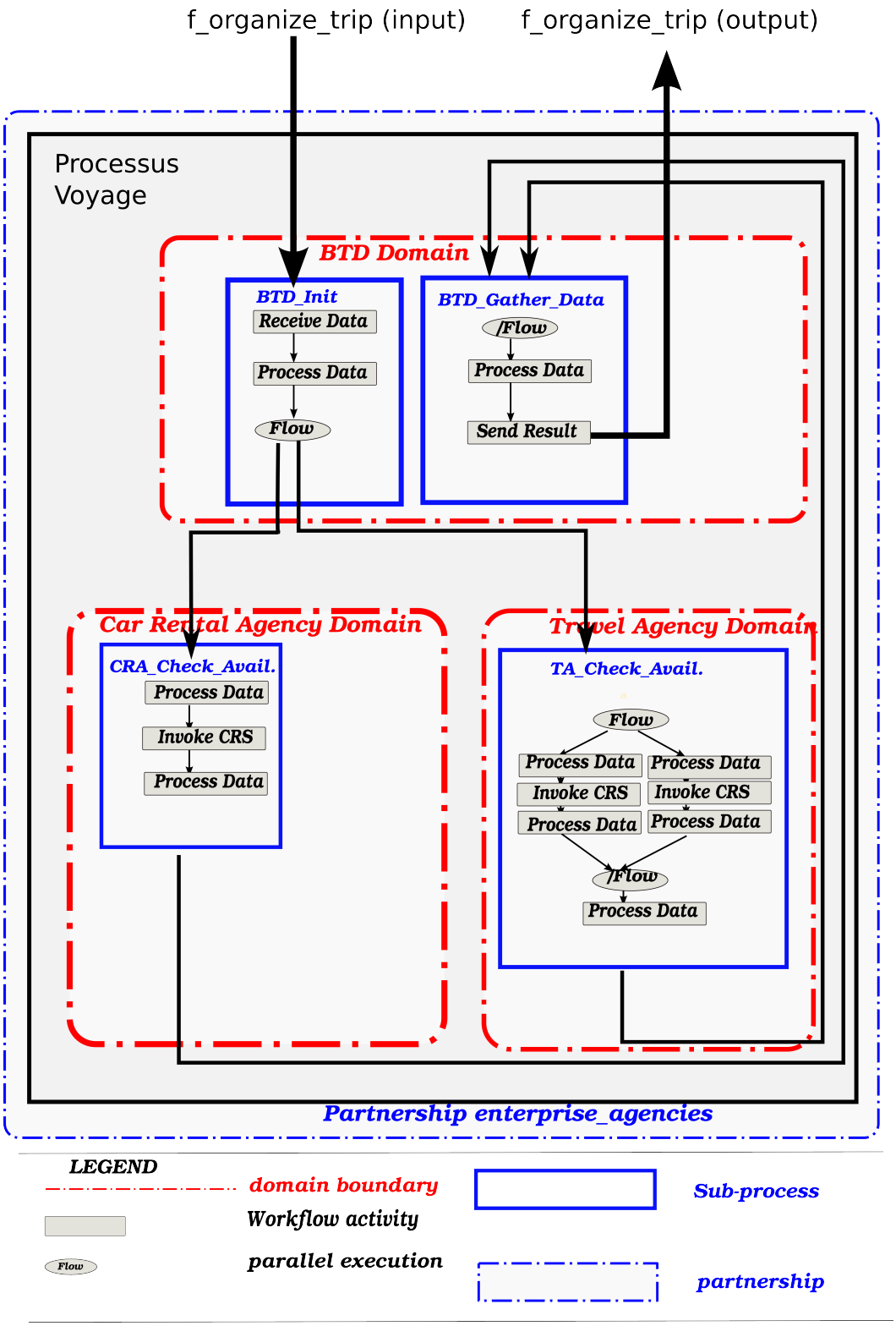


FIG. 4.2 – Un exemple de composant spatio-temporel (correspondant au processus distribué de la Figure 1.2)

ni modifier les plateformes d'orchestration.

Dans la suite, nous donnons une vue d'ensemble des phases couvertes par notre approche.

4.2.2 Vue d'ensemble de notre méthodologie

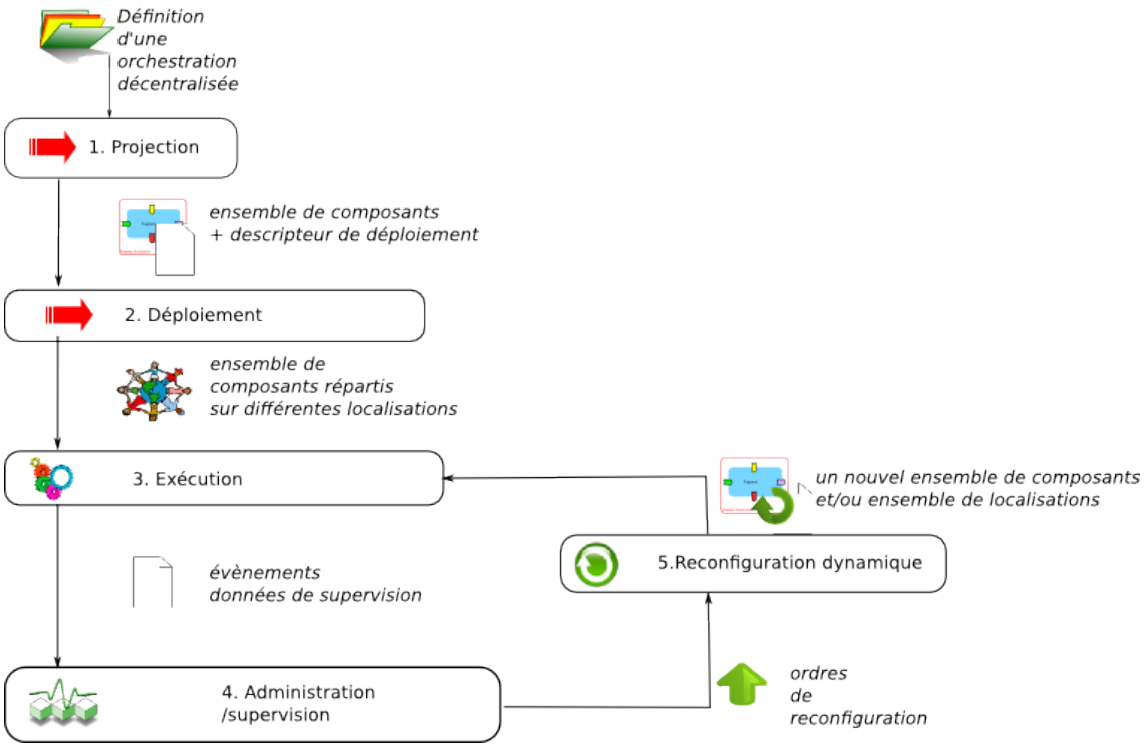


FIG. 4.3 – Phases de notre méthodologie

La fragmentation d'un workflow a une influence importante sur les performances d'exécution, particulièrement sur la communication entre les systèmes de workflows coopérants. De ce fait, la fragmentation doit être anticipée au moment de la conception du workflow distribué. Des besoins formels au niveau des fragments peuvent être nécessaires dans le but d'éviter des inconsistances et d'assurer que le workflow distribué a le même comportement que le workflow non distribué. Dans notre travail, nous considérons que cette étape a été réalisée et qu'un ensemble de fragments collaboratifs cohérent a été produit. Par exemple, les travaux de thèse de Ustun Yildiz [Yil08] présentent un mécanisme de calcul de décomposition d'un processus métier en plusieurs sous-processus coopérants. Cependant, la mise-en-œuvre repose entièrement sur WS-BPEL, que ce soit pour appeler des services fonctionnels que pour effecteur un appel sur le sous-processus suivant impliqué dans le processus métier.

Nous couvrons toutes les phases montrées dans la Figure 4.3. Avec cette méthodologie, nous pouvons représenter une orchestration distribuée grâce à un composant distribué hiérarchique, composé d'un ensemble de composants interagissant entre-eux par une relation temporelle. En effet, nous supposons que le processus a été préalablement fragmenté.

La fragmentation peut avoir été obtenue soit de façon naturelle (selon une approche ascendante ou *top-down*), en composant des sous-processus déjà existants, ou alors soit avec un outil de calcul de distribution de workflow.

Afin de mettre en place l'orchestration distribuée, nous faisons la distinction entre la phase de définition/déploiement et la phase d'exécution/ invocation. Le composant distribué est *projeté* sur un composant composite distribué (étape 1, projection). Pour cela, il faut garantir la bonne coordination des fragments et assurer le passage des données applicatives de sous-processus en sous-processus. De plus, il faut fournir un *descripteur de* déploiement qui va permettre d'installer les définition des sous-processus ainsi que leurs moteurs d'exécution sur un ensemble de nœuds d'exécution. Une fois le processus distribué englobé dans un composant, il faut le déployer sur différents nœuds d'exécution (étape 2, déploiement). .L'exécution peut alors avoir lieu (étape 3, exécution). Le flot de données est alors géré et transmis de sous-composant en sous-composant. L'exécution est supervisée et administrée (étape 4, Administration/supervision). Cette étape de supervision peut faire remonter des données qui guideront une reconfiguration dynamique du processus métier distribué (étape 5), produisant ainsi un nouvel ensemble de composants coopérants.

4.3 Conclusion

Nous avons présenté dans ce chapitre une réponse aux besoins que nous avons identifiés dans notre problématique et que nous avons évalué dans notre état de l'art. A partir de ces besoins, nous avons positionné et justifié notre contribution scientifique, à savoir une approche basée sur un modèle à composants distribués, hiérarchiques et dynamiquement reconfigurables.

Nous présenterons la conception de cette approche dans le chapitre suivant.

Manier des couleurs et des lignes, n'est-ce pas une vraie diplomatie, car la vraie difficulté c'est justement d'accorder tout cela.

Raoul Dufy, Extrait de *Problèmes de la peinture.*

Chapitre 5

Exécution d'orchestrations réparties grâce aux composants

Contenu du chapitre :

5.1 Définition d'un fragment . . . 90
5.1.1 Le fragment, élément de base . . . 90
5.1.2 Composition de Fragments . . . 92
5.2 Contrôle du cycle de vie du fragment . . . 96
5.2.1 Génération du Fragment . . . 96
5.2.2 Interfaces de contrôle . . . 97
5.2.3 Déploiement d'un Fragment . . . 98
5.2.4 Exécution d'un fragment unitaire . . . 99
5.2.5 Administration et Supervision d'un sous-processus . . . 101
5.2.6 Reconfiguration dynamique du Fragment . . . 102
5.3 Projection d'une orchestration décentralisée sur un composant hiérarchique distribué . . . 103
5.3.1 Projection de l'orchestration distribuée . . . 103
5.3.2 Gestion de la complexité du déploiement de l'orchestration distribuée . . . 104
5.3.3 Exécution d'un fragment composite . . . 105
5.3.4 Gestion de la dynamicité dans une orchestration distribuée . . . 106
5.3.5 Administration et supervision de l'orchestration globale . . . 107
5.4 Conclusion . . . 107

Dans les chapitres précédents, nous avons présenté le contexte dans lequel se place notre travail, ainsi que les approches existantes pour l'exécution des orchestrations décentralisées de services. Ce chapitre présente la solution proposée dans cette thèse, à savoir les concepts d'un modèle à composants permettant l'exécution décentralisée et dynamique d'orchestrations de services. La Section 5.1 présente le modèle sur lequel repose notre solution. La contribution se découpe en deux temps : en premier lieu, nous présentons la projection d'un sous-processus coopérant dans la Section 5.2, et en second lieu dans la Section 5.3, nous généraliserons l'approche présentée pour un sous-processus à un ensemble de sous-processus coopérant afin de réaliser une orchestration distribuée.

5.1 Définition d'un fragment

Notre approche consiste à donner à une orchestration distribuée une vue globale et unifiée, la tâche du concepteur et analyste de la composition s'en trouvant ainsi facilitée. Dans cette section, nous introduisons les éléments constitutifs de notre solution ainsi que leurs propriétés. Étant donné que nous nous intéressons à l'exécution des processus décentralisés, notre solution ne couvre pas le partitionnement d'un processus global. Nous supposons en effet que ce partitionnement a déjà eu lieu et a produit un ensemble de *processus coopérants*.

Définition [*processus coopérant*] : Un *processus coopérant* est un processus remplissant une fonction métier et qui possède un ou plusieurs *processus coopérants précédents* et/ou un ou plusieurs *processus coopérants suivants*, permettant aux processus coopérants de s'échanger des données applicatives, globales et nécessaires à l'exécution d'un processus décentralisé.

Nous nous intéressons dans cette section au modèle du fragment qui permet de représenter et de manipuler un sous-processus coopérant.

5.1.1 Le fragment, élément de base

Le *fragment* constitue l'élément de base de notre solution. Il représente la plus petite unité d'exécution manipulable au sein de notre modèle. Notre définition d'un fragment se projette de manière naturelle sur un modèle à composant. C'est pourquoi, dans le but de fournir une approche générique, nous nous basons sur les spécifications SCA pour décrire l'architecture et l'implémentation de notre solution.

Dans la Figure 5.1, nous schématisons la représentation fonctionnelle d'un fragment. Un fragment :

1. propose une fonction d'entrée, en général la fonctionnalité métier qu'elle implémente
2. référence un nombre arbitraire de services
3. peut posséder un fragment précédent et/ou un fragment suivant ; les fragments sont reliés entre-eux par une relation temporelle.
4. s'exécute sur un domaine d'exécution qui représente la localisation physique du fragment.

Un fragment, illustré dans la Figure 5.2, est composé de deux entités complémentaires :

- La définition fonctionnelle du processus coopérant associé au fragment qui sera interprétée par un moteur d'exécution de workflow lors de l'activation du processus suite à une requête entrante. Cette définition peut être écrite sous la forme d'un processus WS-BPEL, mais n'est pas limitée à ce langage, tout dépendra du moteur de workflow utilisé.
- Le gestionnaire de fragment, visible sur la Figure 5.2, qui représente le support d'exécution du fragment. Il a charge de déployer, de contrôler, d'administrer et de superviser l'exécution du processus associé à ce fragment. Il expose des interfaces de contrôle, que nous détaillerons plus loin.

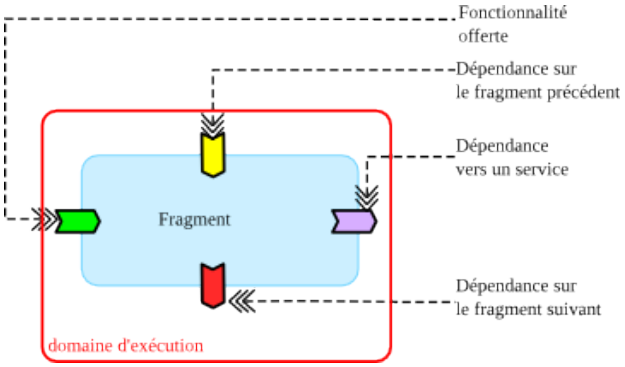


FIG. 5.1 – Représentation fonctionnelle d'un Fragment

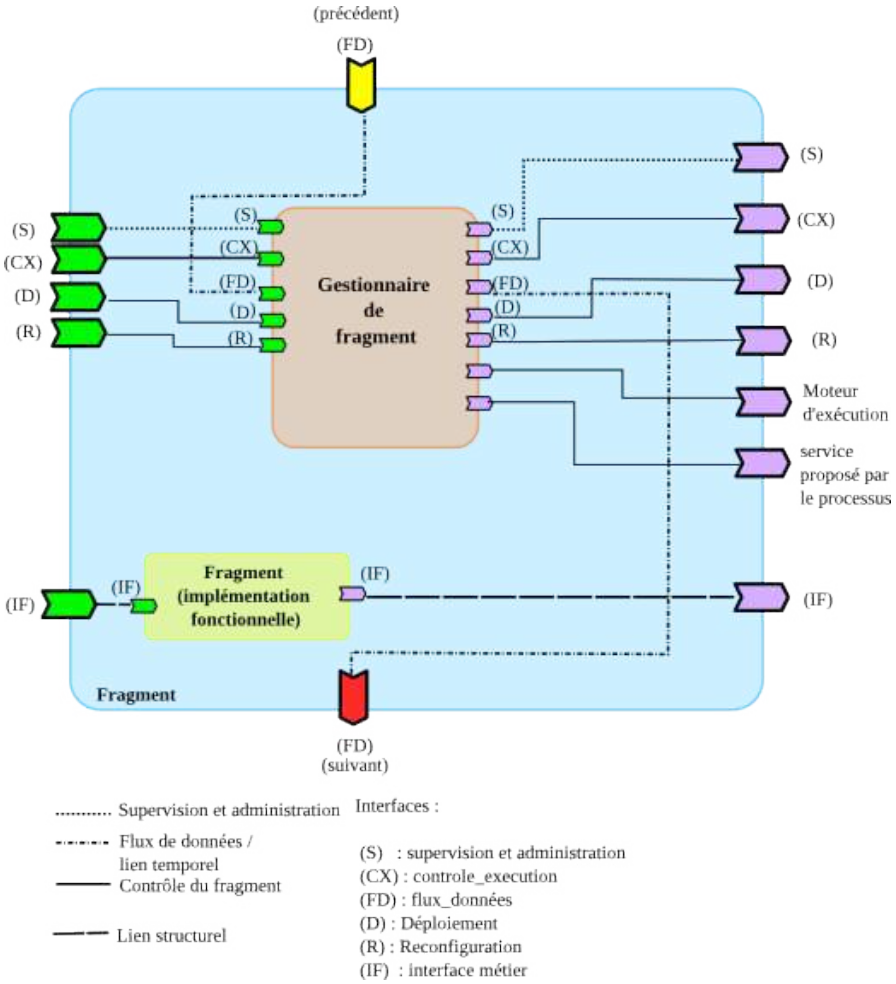


FIG. 5.2 – Structure d'un fragment d'orchestration

Un fragment peut être soit *unitaire*, c'est à dire qu'il représente un processus, soit *composite,* c'est-à-dire qu'il contient un ensemble d'autres fragments, et dans ce cas, il représente une *orchestration décentralisée* ; nous y reviendrons en détails dans la Section 5.3. Notre modèle se veut donc hiérarchique, fournissant ainsi une encapsulation totale du processus global et offrant par la même occasion une vue globale et unifiée du système sous la forme d'un fragment composite unique, facilement manipulable et également ré-utilisable.

Nous allons maintenant voir comment les fragments se composent entre-eux.

5.1.2 Composition de Fragments

Les fragments sont composables entre-eux selon des relations structurelles et temporelles de façon à construire une orchestration. Une telle orchestration est montrée dans la Figure 5.4. Afin de pouvoir réaliser la composition, un *fragment* possède des dépendances, non obligatoires de deux types, à savoir des *dépendances temporelles* et des *dépendances structurelles.* Nous les détaillons ci-après :

Dépendances structurelles Il existe deux sortes d'interfaces structurelles pour un fragment. Ce sont ces dépendances qui permettent de traduire la dimension dans l'espace d'une composition de services. Ces deux types d'interfaces permettent soit de recevoir un appel en entrée, ce qui aura pour effet de déclencher une exécution du sous-processus, soit de faire appel à des services externes. Ces dépendances sont semblables à celles du modèle SCA sur lequel nous nous appuyons.

Un fragment peut exposer sa fonctionnalité métier par le biais d'une interface ou dépendance structurelle d'entrée. Si cette interface est invoquée, une instance du sous-processus est créée, et le sous-processus est alors initié. Dans l'exemple illustré par la Figure 5.4, le fragment T-1 expose sa fonctionnalité et la met à disposition par le biais d'un service exposé.

Les dépendances structurelles de sortie représentent les liens du fragment avec les services externes, impliqués dans la définition du sous-processus lui correspondant. Ces dépendances structurelles peuvent être modifiées de façon dynamique. Ainsi, un service peut être substitué par un autre durant l'exécution. La signature de ces interfaces est dépendante de celle des services impliqués. Un fragment peut dépendre d'autres services, qui peuvent être eux mêmes des fragments qui offrent eux-même une fonctionnalité métier.

Des proxies vers les services impliqués dans l'orchestration Pour chaque dépendance structurelle, il existe dans le fragment un composant *proxy* qui permet l'interception des appels vers et depuis le sous-processus. Comme illustré dans la Figure 5.3, la liaison du moteur d'exécution vers le service S n'est pas directe : un composant *proxy* est inséré entre les deux entités. Le service proposé par le *proxy* (S') est le même mais avec une valeur ajoutée. Nous pouvons ainsi contrôler les accès au service dans le but d'une reconfiguration dynamique ou d'un ajout de fonctionnalité au service, comme par exemple, un accès sécurisé au service si cela est nécessaire. La définition du processus est modifiée en conséquence, ainsi tous les liens vers les services externes sont remplacés par des liens vers les services proposés par les proxies sur ces services externes.

Dépendance temporelles Un fragment se situe dans le temps, c'est-a-dire qu'il peut posséder un fragment précédent et un fragment suivant. Il existe deux sortes d'interfaces temporelles, que l'on peut apparenter à des ports d'entrée et de sortie des données applicatives. Ces relations permettent le passage du flux de données de l'orchestration globale.

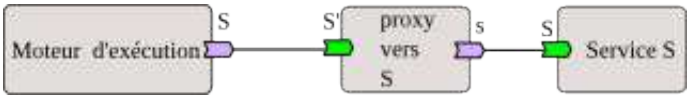


FIG. 5.3 – Les composants *prox*ies permettant d'intercepter les appels vers le moteur d'exécution et vers les services externes

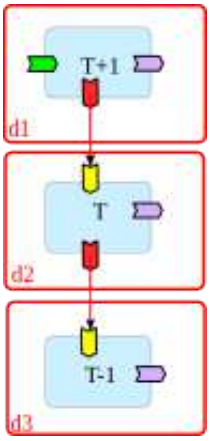


FIG. 5.4 – Un exemple de composition de fragments

Un exemple de composition temporelle est montré dans la Figure 5.4. Un fragment T est composé selon une relation temporelle avec deux autres fragments T-1 et T+1. Le fragment T-1 est le *fragment précédent* du fragment T et le fragment T+1 est le *fragment suivant* du fragment T.

Les *dépendances temporelles d'entrée* permettent à une instance de sous-processus de pouvoir recevoir les données applicatives qui vont lui être nécessaires pour s'exécuter. Ce type de dépendance peut être *simple*, c'est-à-dire que le fragment n'attend des données que d'un seul fragment précédent, permettant d'exécuter les sous-processus en séquence. Au contraire, le type peut être multiple, comme montré dans la Figure 5.5, c'est-à-dire que le fragment attend des données de plusieurs fragments précédents. Cette réception multiple de données peut être apparentée au patron AND-JOIN [vTKB03]. La réception de toutes les données applicatives initie l'exécution du sous-processus.

Les *dépendances temporelles de sortie* sont soit de type *simple*, soit de type *multiple*. Dans le cas d'un type *simple*, le fragment est relié à un seul fragment suivant. Dans le cas d'un type multiple, le fragment est relié à plusieurs fragments suivant afin de leur transmettre des données. Cet envoi multiple de données applicatives peut être apparenté au patron AND-SPLIT [vTKB03], permettant ainsi de pouvoir déclencher une exécution parallèle en invoquant de manière parallèle les sous-processus suivants.

Dans l'exemple illustré par la Figure 5.5, le fragment `T-1` possède une interface temporelle de sortie multiple, c'est-à-dire qu'une fois son exécution terminée, il va diriger le flux de données vers trois autres fragments (`Ta,Tb et Tc)` qui vont alors s'exécuter en parallèle. Une fois ces trois sous-processus terminés, le flux de sortie sera dirigé vers le fragment `T+1` qui s'exécutera une fois les données reçues via son interface temporelle, elle aussi multiple.

L'**interface de contrôle du flot d'exécution** permet de passer les données globales

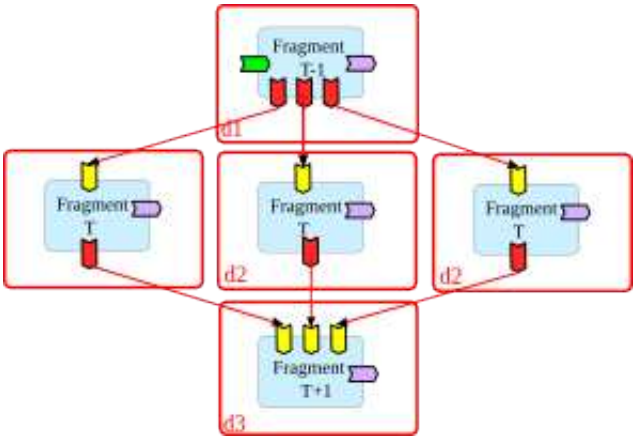


FIG. 5.5 – Dépendances temporelles multiples

lors de l'exécution d'une orchestration. Le flot de contrôle est propre à chaque instance de l'orchestration globale, le gestionnaire de fragment est responsable de l'acheminement des données au fragment. La signature de ces interfaces temporelles exprimée en Java est donnée dans le Listing 5.1.

Listing 5.1 – Dépendances temporelles d'un Fragment

```
/** Interface qui permet de gérer le flux de données applicatives dans un
    Fragment */
public interface Flux_Donnees {

/* Permet de récupérer les données applicatives nécessaires à l'exécution de l'
    instance du processus global dont l'ID est passé en paramètres. Le paramètre
    donnees est une table de correspondance nom_de_donnée -> valeur_de_donnée*/

        public void recup_Donnees (String instanceID,
                                   HashMap <String, Object> donnees);
}
```

Fragments composites Dans le but de représenter une orchestration décentralisée et comme nous le verrons en détails dans la Section 5.3, un fragment peut être composé d'autres fragments. Dans ce cas, le fragment n'est pas associé à un moteur d'exécution de workflow. Le *gestionnaire de fragment* est ici responsable de la bonne coordination des différents sous-fragments, permettant ainsi le passage des données applicatives de sous-processus en sous-processus. La Figure 5.6 illustre cette possibilité de hiérarchie. Le fragment `Comp_Fragment` représentant une orchestration distribuée sur deux domaines

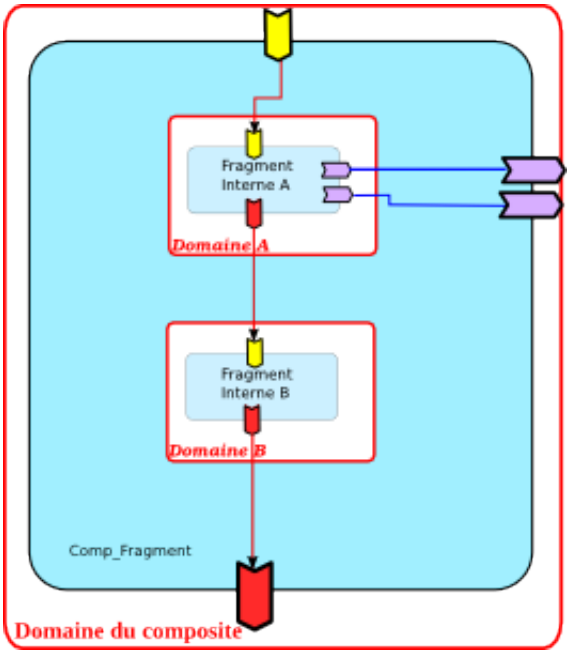


FIG. 5.6 – Composition Hiérarchique des fragments

est composé de deux fragments internes `Fragment Interne A` et `Fragment Interne B`. L'interface de dépendance temporelle du fragment composite est liée de façon interne au premier fragment devant s'exécuter dans la composition, de façon à récupérer les données applicatives envoyées au fragment composite. De manière symétrique, la dépendance temporelle de sortie du dernier fragment interne qui va s'exécuter dans la composition est relié à la dépendance temporelle de sortie du fragment composite. Dans le cas d'un fragment composite, le domaine d'exécution dans lequel se trouve le composite, n'implique que ce composite et non les fragments qui le composent, qui eux, s'exécutent dans leur propre domaine d'exécution.

Nous venons de définir le fragment, élément de base de notre solution, qui permet de projeter la définition d'un sous-processus coopérant sur un composant spatio-temporel. Le modèle UML du Fragment est montré dans la Figure 5.7. Nous allons maintenant nous intéresser dans la suite au cycle de vie du fragment, et en particulier au déploiement et au contrôle de l'exécution d'un fragment.

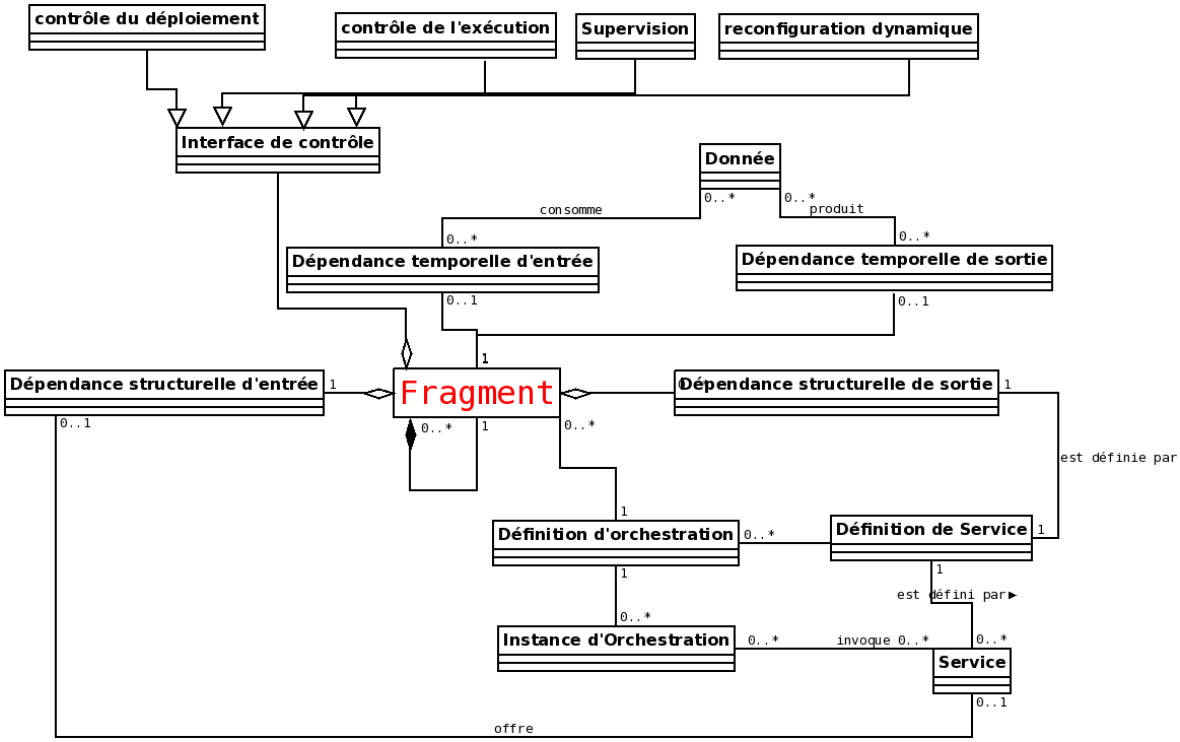


FIG. 5.7 – Le modèle de fragment

5.2 Contrôle du cycle de vie du fragment

Comme nous l'avons explicité dans notre méthodologie dans le chapitre 4, avant de pouvoir l'exécuter et de le contrôler, il faut, à partir de la définition d'un processus coopérant, générer le fragment en réalisant une projection de cette définition sur un composant, et ensuite le déployer. Nous décrivons ces différentes étapes qui contrôlent le cycle de vie du fragment.

5.2.1 Génération du Fragment

Avant de pouvoir le déployer et de le contrôler, il est nécessaire, à partir de la définition d'un processus coopérant, de générer un fragment. La projection d'un processus sur un composant peut être réalisée par le biais d'une analyse statique de la définition du processus, sans impliquer une modification de ce dernier. Nous identifions ici les éléments essentiels à sa génération.

Nous rappelons qu'un processus propose un nombre arbitraire de fonctions métiers (en général une seule, qui déclenche l'exécution de ce processus) et dépend d'un nombre arbitraire de services. De plus, ce processus faisant potentiellement partie d'un processus décentralisé global, il contient dans sa définition un processus suivant et un processus précédent.

Dans un premier temps nous générons un fragment vide du point de vue fonctionnel, pourvu de dépendances temporelles ; celles-ci seront éventuellement liées si le fragment

fait partie d'un fragment composite.

Dans un deuxième temps, nous générons les dépendances structurelles ainsi que leur *proxy* associé. Si le processus expose une ou plusieurs fonctions métiers, alors nous générons pour chacune une interface structurelle d'entrée. Pour chaque service externe impliqué dans la définition du processus, nous générons une dépendance structurelle de sortie.

Ce fragment est également équipé de son *gestionnaire de fragment.* Le gestionnaire de fragment est responsable du contrôle du cycle de vie du fragment et permet via ses interfaces de contrôle de le manipuler une fois généré. Dans la suite, nous explicitons ces interfaces de contrôle.

5.2.2 Interfaces de contrôle

Le fragment englobe un processus, lui procurant des fonctionnalités de contrôle. Ainsi, il est possible d'agir sur ce processus par le biais d'*interfaces de contrôle.* Comme illustré dans la Figure 5.8, le gestionnaire de fragment est composé de quatre composants, **`Exécution`, `Déploiement`, `Admin/Supervision` et `Reconfiguration`** qui permettent le contrôle du cycle de vie du fragment. Le gestionnaire de fragment est relié au moteur d'exécution et au service proposé par le processus.

- L'**interface de contrôle du déploiement (D),** proposée par le composant `Déploiement`, qui permet de déployer la définition du processus sur un moteur donné. Ce dernier peut être, grâce à ce contrôleur, déployé et contrôlé.
- L'**interfaces d'administration (CX),** proposée par le composant `Admin/Supervision`, qui permet de démarrer, d'arrêter ou de suspendre l'exécution du processus associé au fragment mais aussi de contrôler la structure de l'orchestration, en modifiant les liaisons fonctionnelles d'un fragment ou en ajoutant ou supprimant des fragments dans un composite.
- **L'interface de supervision (S)**, proposée par le composant `Admin/Supervision`, qui permet de souscrire à différents types d'évènements qui surviennent lors de l'exécution du fragment.
- L'**interface de reconfiguration (R),** proposée par le composant `Reconfiguration` du fragment qui permet de changer durant l'exécution la définition du processus associé au fragment. Cette reconfiguration peut avoir lieu soit au niveau des liaisons vers un service, soit au niveau de la structure interne du composant, dans le cas d'un composite.

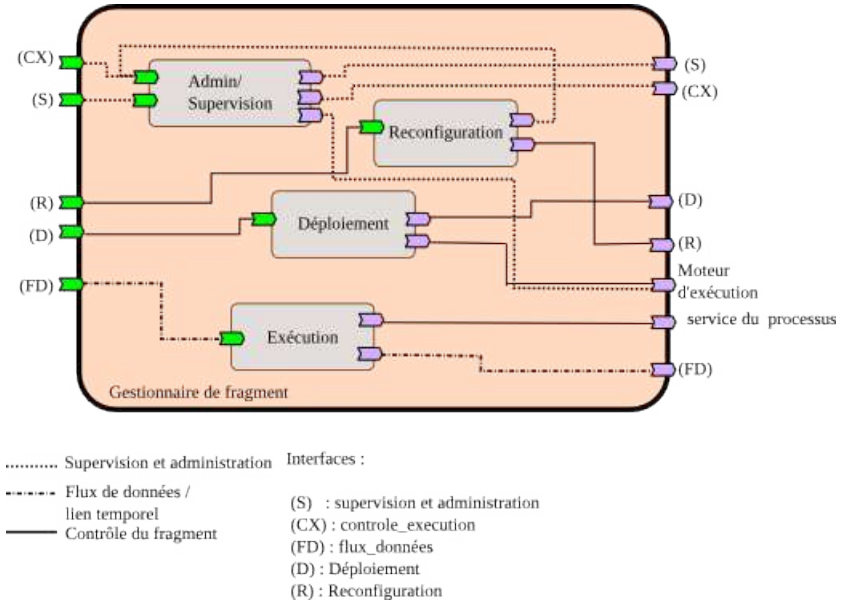


FIG. 5.8 – Composition du Gestionnaire de Fragment

Nous rentrerons dans les détails de ces interfaces de contrôle dans la suite de cette section.

5.2.3 Déploiement d'un Fragment

Nous nous concentrons maintenant sur les besoins de déploiement pour la réalisation de l'exécution d'un Fragment. Le déploiement intervient à plusieurs niveaux : au niveau infrastructure d'exécution même du sous-processus, au niveau du sous-processus, et au niveau du composant.

Déploiement du composant Un fragment est représenté par un composant qui s'exécute sur une localisation donnée ; le déploiement du fragment consiste dans un premier temps à déployer et à installer le composant et son support d'exécution correspondant au fragment à cette localisation. Il est à noter que ce composant n'est pas forcément localisé au même endroit que le moteur d'exécution.

Déploiement de l'infrastructure d'exécution du processus Une fois le fragment installé, il est possible de déployer et installer les éléments qui vont permettre l'exécution du processus. Cette opération d'installation du moteur se réalise grâce au *gestionnaire de déploiement* embarqué dans le gestionnaire de fragment. En particulier, ce gestionnaire propose une interface de gestion du déploiement, `FragmentDeployer`, liée au processus et à son système d'exécution que nous montrons dans le listing 5.2, elle est illustrée dans la Figure 5.8 par l'interface serveur non-fonctionnelle `(D)`. Si le moteur d'exécution n'est pas déjà préalablement déployé, notre mécanisme de déploiement permet de l'installer (via la méthode `deployEngine`()), de le lancer et de le contrôler (via les méthodes `startEngine()` et `stopEngine()`). Dans tous les cas, le mécanisme de déploiement du fragment permet de créer les composants nécessaires pour lier le système d'exécution du processus au

fragment via la méthode `bindEngine()`. Comme nous le montrons dans la Figure 5.8, le gestionnaire de déploiement est lié au moteur d'exécution, permettant ainsi le contrôle de ce dernier.

Déploiement du processus Le dernier élément à déployer avant la mise à disposition du processus est la définition du processus. Toujours grâce à l'interface `FragmentDeployer`, le contrôle du déploiement du processus se réalise par le biais de la méthode `deployProcess`. Le processus est ainsi déployé sur le moteur d'orchestration et prêt à exécuter.

Listing 5.2 – Interface de déploiement d'un Fragment

```
/** Interface qui permet de gérer le  déploiement d'un Fragment et de son
    processus-coopérant associé*/
public interface FragmentDeployer {

/* Déploie et installe le moteur d'exécution du sous-processus */
public void deployEngine(Engine engine);

/* Démarre le moteur d'exécution */
public void startEngine ();

/* Arrête le moteur d'exécution */
public void stopEngine ();

/* Associe le fragment à un moteur d'exécution déjà déployé */
public void bindEngine (Engine engine);

/* Déploie le sous-processus sur le moteur d'exécution associé au fragment */
public void deployProcess (Process process);

/* Retire le sous-processus du moteur d'exécution */
public void undeployProcess (Process process);
}
```

5.2.4 Exécution d'un fragment unitaire

Nous nous intéressons maintenant à l'exécution d'un fragment unitaire, depuis la réception des données applicatives au renvoi du résultat à l'appelant.

Un fragment unitaire représente une définition d'un processus qui fait potentiellement partie d'une orchestration globale. Le fragment est responsable de la gestion des différentes instances de cette partie de l'orchestration. Ainsi, un fragment crée et gère autant d'instances de processus que d'appels via les interfaces qui déclenchent les exécutions, c'est-à-dire les interfaces structurelles d'entrée ou les interfaces temporelles d'entrée. De ce fait, une liste des instances du processus est disponible pour chaque fragment.

Comme explicité auparavant, nous avons pris le parti de représenter un fragment d'orchestration comme un composant SCA enrichi de dépendances temporelles. Ces dépendances temporelles permettent de passer le flux de données d'un fragment à l'autre et sont gérées par un composant non-fonctionnel, le *gestionnaire de fragment.* Le gestionnaire de

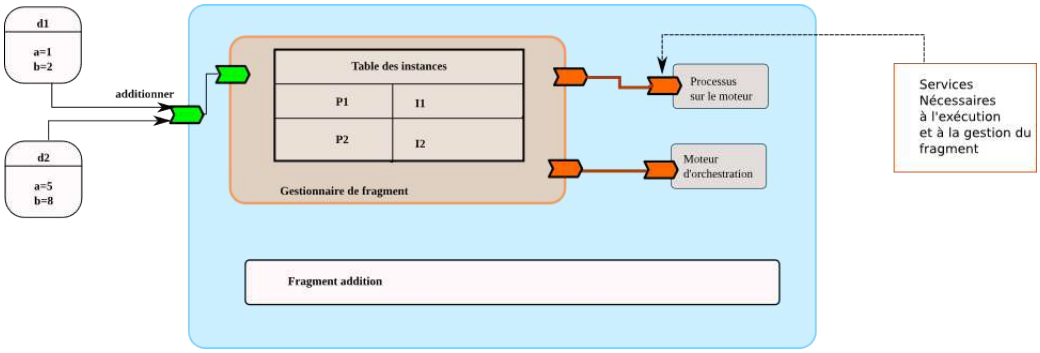


FIG. 5.9 – Un exemple d´exécutions multiples d´un même processus

fragment propose une interface (présentée plus haut dans le Listing 5.1) permettant de recevoir les données applicatives.

L'exécution du fragment se déclenche dès que toutes les données applicatives nécessaire à son exécution sont reçues. Nous listons et détaillons maintenant chacune des étapes de l'exécution d'un fragment unitaire.

L'exécution du fragment se réalise en 5 étapes :

1. **Réception des données applicatives destinées à une instance de processus en particulier :** Les données applicatives associées à une instance de sous-processus sont reçues par le biais de l'interface `FluxDeDonnees` si le fragment possède un processus précédent, ou par le biais de l'interface fonctionnelle du fragment. Ces données vont être traitées par le gestionnaire de fragment qui va les transmettre au moteur d'exécution. Pour chaque appel effectué, le gestionnaire de fragment crée une entrée dans sa table des instances.
2. **Envoi des données applicatives au processus déployé sur le moteur** : Pour chaque appel effectué sur l'interface du fragment, un appel est effectué sur le processus hébergé sur le moteur. Nous rappelons ici que la définition du processus déployé sur le moteur propose un service qui permet de récupérer les données applicatives.
3. **Exécution des processus**. Cette étape se réalise au niveau du moteur d'exécution du sous-processus. Si d'éventuels services externes sont impliqués dans le sous-processus, alors le moteur appelle le service proposé par le *proxy* représentant le service réel.
4. **Récupération des données applicatives.** Une fois le sous-processus exécuté, les données applicatives sont retournées à l'instance gérée par le gestionnaire de fragment, qui va préparer l'envoi de ces données soit au fragment suivant si le fragment en possède un, soit à l'appelant le cas échéant.
5. **Retour des données à l'appelant** si le fragment ne possède pas de fragment suivant**, ou envoi des données au processus suivant, le cas écheant.**

La Figure 5.9 montre un exemple simple d'exécution d'un processus qui additionne deux nombres via le service additionner. Ce fragment est équipé de son gestionnaire de fragment que nous représentons d'une manière simplifiée, et dont nous ne montrons que les liens vers les services nécessaires à l'exécution effective et à la gestion du processus associé au fragment, à savoir le service proposé par le moteur d'exécution ainsi que le service proposé par le processus Nous supposons que deux utilisateurs, P1 et P2, invoquent ce

Listing 5.3 – Interface d'administration d'un fragment

```
1
2  /** Interface qui permet de contrôler le processus coopérant associé à un
       fragment */
3  public interface Orchestration_Controler {
4  /* Arrête l'execution de l'instance du processus  correspondant à l'identifiant
       passé en paramètres */
5  public void stop (long instanceID);
6
7  /* Suspend l'exécution de l'instance du processus  correspondant à l'identifiant
       passé en paramètres */
8  public void suspend (long instanceID);
9
10 /* Reprend l'exécution de l'instance du processus correspondant à l'identifiant
       passé en paramètres */
11 public void resume (long instanceID)
12
13 /* Arrête l'exécution de toutes les instances du processus */
14 public void stopAllInstances ();
15
16 /* Suspend l'exécution de toutes les instances du processus */
17 public void suspend ();
18
19 /* Reprend toutes les instances du processus */
20 public void resume ();
21
22 }
```

service. Pour chaque utilisateur, une entrée est créée dans la table des instance avant de déclencher l'exécution du processus sur le moteur.

5.2.5 Administration et Supervision d'un sous-processus

Le modèle que nous proposons se veut d'être le plus flexible possible vis-à-vis des systèmes d'exécution des sous-processus. Nous pensons que le modèle doit adopter des interfaces simples qui permettent d'abstraire et d'unifier l'ensemble des opérations de gestion du processus. Ainsi, le modèle est indépendant des systèmes d'exécution et permet de changer facilement le support d'exécution associé à un fragment.

Administration du Fragment : Le fragment est équipé d'un contrôleur non-fonctionnel qui permet de gérer le cycle de vie du sous-processus et de ses instances. Ce contrôleur permet d'agir sur l'exécution d'une instance de sous-processus. Ainsi on pourra, par le biais de l'interface de contrôle, arrêter l'exécution d'une instance donnée, la suspendre ou lui faire reprendre le cours de son exécution. L'interface au sens Java de ce contrôleur est montrée dans le Listing 5.3 . Ce contrôleur permet d'avoir en fait un contrôle sur le moteur d'exécution du sous-processus en lui délégant les opérations de gestion du cycle de vie d'un processus donné.

Listing 5.4 – L´Interface de supervision du fragment

```
/** Interface permettant de superviser le fragment */
public interface Supervision {

/* Souscrit à un type d'évènements */
        public subscribeTo (TypeEvenement e, Ecouteur e);

}
```

Supervision du Fragment et gestion des évènements : Dans le but de surveiller et d'analyser le déroulement de l'exécution, le fragment propose une interface de supervision (exposée dans le Listing 5.4). Il est ainsi possible de souscrire en tant qu'Ecouteur à un type d'évènement, lié à une entité du fragment (fragment, processus, moteur, données ...) générés lors de l'exécution d'un fragment. Ces types d'évènements sont décrits dans le Tableau 5.1.

ENTITÉ CONCERNÉE PAR L'ÉVÈNEMENT	TYPE D'ÉVÈNEMENT
Service	Service invoqué Service indisponible Service remplacé
Processus	Processus stoppé Processus suspendu Processus redémarré Processus déployé Processus démarré Fin exécution
Moteur d'exécution	Moteur déployé Moteur arrêté Moteur redémarré Moteur installé
Données	Données applicatives envoyées Données applicatives reçues
Fragment	Fragment reconfiguré Structure changée (ajout d'un sous-fragment) Création d'une nouvelle instance

TAB. 5.1 – Types d'évènements proposés par l'interface de supervision d'un fragment

5.2.6 Reconfiguration dynamique du Fragment

Comme motivé précédemment, l'intérêt d'utiliser un modèle à composants pour englober une orchestration, est de permettre la dynamicité et l'agilité de l'orchestration de manière facile et portable. Ainsi, dans notre modèle nous proposons une gestion dynamique de la structure du fragment. Dans le listing 5.5, nous montrons l'interface qui permet à un service impliqué dans la définition d'un sous-processus d'être changé au cours de l'exécution, par exemple si le premier ne répond pas. La reconfiguration intervient au niveau du fragment, dans lequel nous modifions le composant *proxy* concerné par le service.

Listing 5.5 – Interface de contrôle de la dynamicité

```
/** Interface qui permet de reconfigurer un fragment unitaire*/
public interface ReconfigurationFragment {
        public void replaceReference ( Reference oldRef , Reference newRef);
}
```

Pour effectuer une reconfiguration durant l'exécution d'un ou de plusieurs processus nous procédons de la façon suivante :

1. **Arrêt de l'exécution du proxy.** Les instances de processus utilisant les références sur le service sont suspendues jusqu'à ce que la reconfiguration du proxy ait eu lieu. Ceci est possible notamment grâce à la liaison vers le composant d'administration qui permet de contrôler l'exécution du fragment.
2. **Changement de la référence :** Le proxy est modifié.
3. **Redémarrage du proxy et reprise de l'exécution des instances de processus en cours**

Remarquons que cette modification de l'adresse du service englobé dans le *proxy* impactera toutes les instances en cours et à venir.

Nous venons de détailler le contrôle d'un fragment unitaire, élément de base pour composer une orchestration décentralisée. Dans la suite, nous nous intéressons à la projection d'une orchestration décentralisée sur un fragment composite, son exécution et son contrôle.

5.3 Projection d'une orchestration décentralisée sur un composant hiérarchique distribué

Dans cette section, nous décrivons le mécanisme de projection d'une orchestration décentralisée sur un fragment composite ainsi que les différents concepts liés à l'exécution d'une telle orchestration.

5.3.1 Projection de l'orchestration distribuée

Avant de décrire la projection d'une orchestration décentralisée, intéressons nous tout d'abord à sa définition. En effet, il est important de connaître la manière dont est constituée une telle définition afin de pouvoir la transcrire sur notre modèle de fragment.

La *définition d'une orchestration décentralisée* est représentée par un ensemble de définitions de processus coopérants **reliés entre-eux par une relation temporelle**. La définition d'un processus coopérant est écrite de façon à recevoir les données applicatives par le biais d'une fonction et à les renvoyer une fois que l'exécution du processus est terminée.

Dans notre approche, une orchestration décentralisée est représentée par un fragment composite, contenant un ensemble de fragments, chaque fragment représentant la définition d'un sous-processus compris dans l'orchestration globale. La **projection d'une orchestration décentralisée sur un fragment composite** consiste à générer les différents fragments qui lui correspondent, dans le but de les déployer, en réalisant les liaisons entre ces fragments et ensuite de les exécuter. Cet ensemble de fragments est englobé dans un fragment composite global qui va permettre le contrôle et l'administration de l'orchestration

entière. Ainsi, pour chaque sous-processus impliqué dans l'orchestration, nous générons un fragment comme détaillé dans la Section 5.2.1. À ce stade, nous obtenons un fragment composite, composé d'un ensemble de fragments. Chaque fragment est alors relié à son fragment suivant.

Afin d'obtenir une vue globale de l'orchestration, les interfaces structurelles de chaque fragment sont exportées et promues au niveau du fragment composite global.

Comme nous nous trouvons dans un environnement distribué, la question qui se pose naturellement est celle du passage des données applicatives initiales et celle du retour du résultat à l'appelant : la fonction du processus global est invoquée, ce qui a pour effet de déclencher l'exécution du fragment composite qui va lui-même orchestrer les différents fragments qui le composent. Lorsque l'exécution du processus est terminée, le dernier fragment doit retourner le résultat à l'appelant. Le passage des données en entrée et en sortie pour l'orchestration globale se réalise au niveau du composite global. Pour ce faire, nous définissons deux types particuliers de fragments qui permettent le passage des données depuis et vers le composite, à savoir le *fragment initial* et le *fragment final,* que nous définissons ci-après.

Fragment initial Dans un fragment composite, nous définissons la notion de fragment initial comme suit :

> **Définition [*Fragment initial*]** : Dans la définition d'une orchestration distribuée, le fragment initial est le fragment qui va être exécuté avant tous les autres. C'est à ce fragment que le gestionnaire du fragment composite va transmettre les données applicatives.

Fragment final Dans un fragment composite, nous définissons la notion de fragment final comme suit :

> **Définition [*Fragment final*] :** Dans la définition d'une orchestration distribuée, le fragment final, ou fragment terminal, est le fragment qui va être exécuté après tous les autres. Ce fragment transmet le résultat du processus global au gestionnaire de fragment.

5.3.2 Gestion de la complexité du déploiement de l'orchestration distribuée

Une fois les fragments générés, nous les relions entre eux selon leurs relations temporelles, en réalisant des liaisons entre les interfaces temporelles proposées par les fragments : une ou plusieurs interfaces temporelles de sortie étant reliée à une ou plusieurs interfaces temporelles d'entrée.

Nous supposons ici que nous avons à notre disposition un ensemble de lieux d'exécution qui vont héberger les fragments et un ensemble de lieux d'exécution qui vont supporter l'exécution des processus associés à leur moteur d'orchestration.

1. **Déploiement du fragment composite :** Pour chaque fragment constituant le composite, nous le déployons comme décrit dans la Section 5.2.3. Le fragment composite (avec le gestionnaire de fragment composite) est aussi déployé.

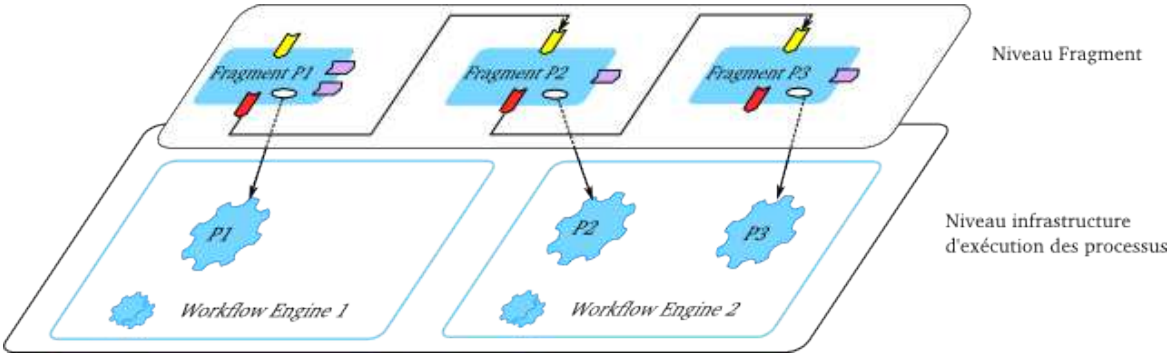


FIG. 5.10 – Les différents niveaux d'exécution impliqués dans une orchestration décentralisée

2. **Établissement des liaisons entre fragments :** A partir de la définition de l'orchestration décentralisée globale, nous identifions les sous-processus initiaux et les sous-processus finaux. Les fragment générés correspondant à ces sous-processus coopérants particuliers sont respectivement reliés à l'interface d'envoi de données interne du composite et à l'interface de réception de données du composite. Dans la Figure 5.11, le fragment final est relié à une interface spéciale du gestionnaire de fragment composite qui reçoit le résultat final de l'orchestration et le transfèrera à l'appelant. Pour chaque fragment, nous identifions, dans la définition de son processus coopérant associé, le processus coopérant suivant qui va nous permettre de réaliser une liaisons vers le fragment suivant correspondant.

La Figure 5.10 nous montre les différents niveaux de déploiement et d'exécution d'une orchestration décentralisée.

5.3.3 Exécution d'un fragment composite

L'exécution d'un fragment composite représente l'enchaînement des différents fragments qui le composent. Les étapes réalisées pour mener à bien cette exécution sont les suivantes :

1. **Réception des données applicatives :** Comme pour l'exécution d'un fragment unitaire (décrite dans la Section 5.2.4), les données applicatives sont reçues par le gestionnaire de fragment du fragment composite qui va créer une instance de processus et lui associer un identifiant unique. Cette instance correspond à l'ensemble des instances et est utilisé pour le passage de données applicatives entre les différents sous-fragments. Ainsi on sait à quelle instance appartiennent ces données et de ce fait, on sait à quel appelant le résultat de l'exécution doit être renvoyé.
2. **Envoi au(x) processus initial(ux) :** Les données applicatives sont alors transmises au(x) processus initial(ux), par le biais d'une liaison interne au composite, qui vont s'exécuter et transmettre les données applicatives aux fragments suivants.
3. **Exécution des fragments intermédiaires :** Les fragments intermédiaires s'exécutent, s'échangeant entre-eux les données applicatives.
4. **Exécution du fragment final qui va renvoyer le résultat à l'appelant**. Le fragment final s'exécute et transmet les données applicatives au gestionnaire du fragment composite, par le biais d'une liaison interne au composite. Le gestionnaire de fragment du

composite reçoit alors le résultat final de l'exécution de l'orchestration et peut alors le renvoyer à l'appelant.

5.3.4 Gestion de la dynamicité dans une orchestration distribuée

Nous avons mentionné dans la Section 5.2.6, que le fragment proposait des opérations de reconfiguration dynamique au niveau structurel, ce qui est le cas aussi pour un fragment composite et donc pour une orchestration distribuée. L'orchestration distribuée introduit la dimension de temps. Ainsi, des modifications au niveau temporel peuvent être nécessaires et impliquer des changements au niveau de la structure interne du fragment composite. Par exemple, pendant l'exécution, un fragment peut être remplacé par un autre, ou un fragment peut être introduit dans la composition, ou encore en être supprimé.

Listing 5.6 – L´ interface de reconfiguration dynamique d´un fragment composite

```
/** Interface de Reconfiguration Dynamique d'un fragment composite */
public interface ReconfigurationDynamiqueComposite extends
    ReconfigurationDynamique {

/* Permet d'ajouter un fragment entre deux fragment existants. Cette
    méthode suppose que les données applicatives entre ces fragments
    soient compatibles */
        public void addFragment (Fragment precedent, Fragment suivant,
            Fragment aInserer);

/* Permet de supprimer le fragment passé en paramètres. Cette méthode
    suppose que les données applicatives circulant entre le fragment
    précédent et le fragment suivant soient compatibles*/
        public void removeFragment (Fragment fragment);

/* Permet de remplacer un fragment par un autre. Cette méthode suppose
    que les données applicatives gérées par le nouveau fragment soient les
     mêmes que celles gérées par l'ancien */
        public void replaceFragment (Fragment ancien, Fragment nouveau);

/* Permet de migrer l'exécution d'un fragment sur un autre noeud d'
    exécution */
        public void moveFragmentTo (ExecutionNode node);

}
```

Pour des raisons de performances ou de tolérance aux pannes, un fragment peut être aussi déplacé sur un autre nœud d'exécution. De même que dans le cas de la reconfiguration d'un *proxy*, toutes les instances en cours et à venir seront impactées par le changement de la structure du fragment composite.

5.3.5 Administration et supervision de l'orchestration globale

Tout au long du processus, il est possible d'administrer le processus et d'obtenir des informations sur l'état de son exécution. Le modèle étant hiérarchique, le contrôle au niveau du fragment composite est transféré aux fragments qui le composent.

L'administration et la supervision de l'orchestration entière sont assurées en utilisant les interfaces de supervision et d'administration du fragment composite d'une part, et de chaque sous-fragment d'autre part. En invoquant l'interface de supervision du composant composite, il est ainsi possible de collecter l'information de supervision agrégée relative à l'exécution du processus global, mais aussi de superviser et d'agir de façon unitaire sur chaque fragment composant l'orchestration. Les requêtes d'administration sont propagées vers les composants interne grâce à la nature hiérarchique de notre modèle. Dans la Figure 5.11, nous pouvons voir que le fragment composite global possède une interface multiple interne qui permet de propager les actions d'administration, telles que les actions de gestion du cycle de vie, soit à tous les sous-fragments, soit à un fragment en particulier.

Nous venons de présenter le mécanisme de projection d'une orchestration décentralisée, constituée de sous-processus coopérants, sur un fragment composite.

5.4 Conclusion

Dans ce chapitre, nous avons présenté la modélisation de notre solution, qui prend en compte les besoins exprimés pour réaliser une composition de services décentralisée et dynamique. Les différents éléments constituant notre solution ont été présentés.

La description d'un composant permettant à une orchestration dynamique de s'exécuter se base sur le modèle SCA que nous avons enrichi afin d'expliciter les relations temporelles entre fragments.

En séparant la logique d'orchestration de l'implémentation métier du composant, nous obtenons une séparation des préoccupations entre le contenu fonctionnel et les activités de gestion de l'orchestration. Un des objectifs que nous nous sommes fixés étant d'être le moins intrusif possible vis-à-vis de l'orchestration globale, les interfaces de gestion d'un fragment sont indépendantes de sa conception fonctionnelle. En utilisant un modèle à composant il est possible d'avoir des implémentations différentes du gestionnaire de fragment. Il est ainsi possible dans une orchestration globale, de coordonner des fragments exécutés par des moteurs d'exécution hétérogènes

Dans le chapitre suivant, nous présentons le contexte technologique qui nous sert de base à la mise en œuvre de notre plateforme d'exécution. Nous présentons en particulier le modèle GCM/ProActive qui permet de créer des composants distribués et re-configurables.

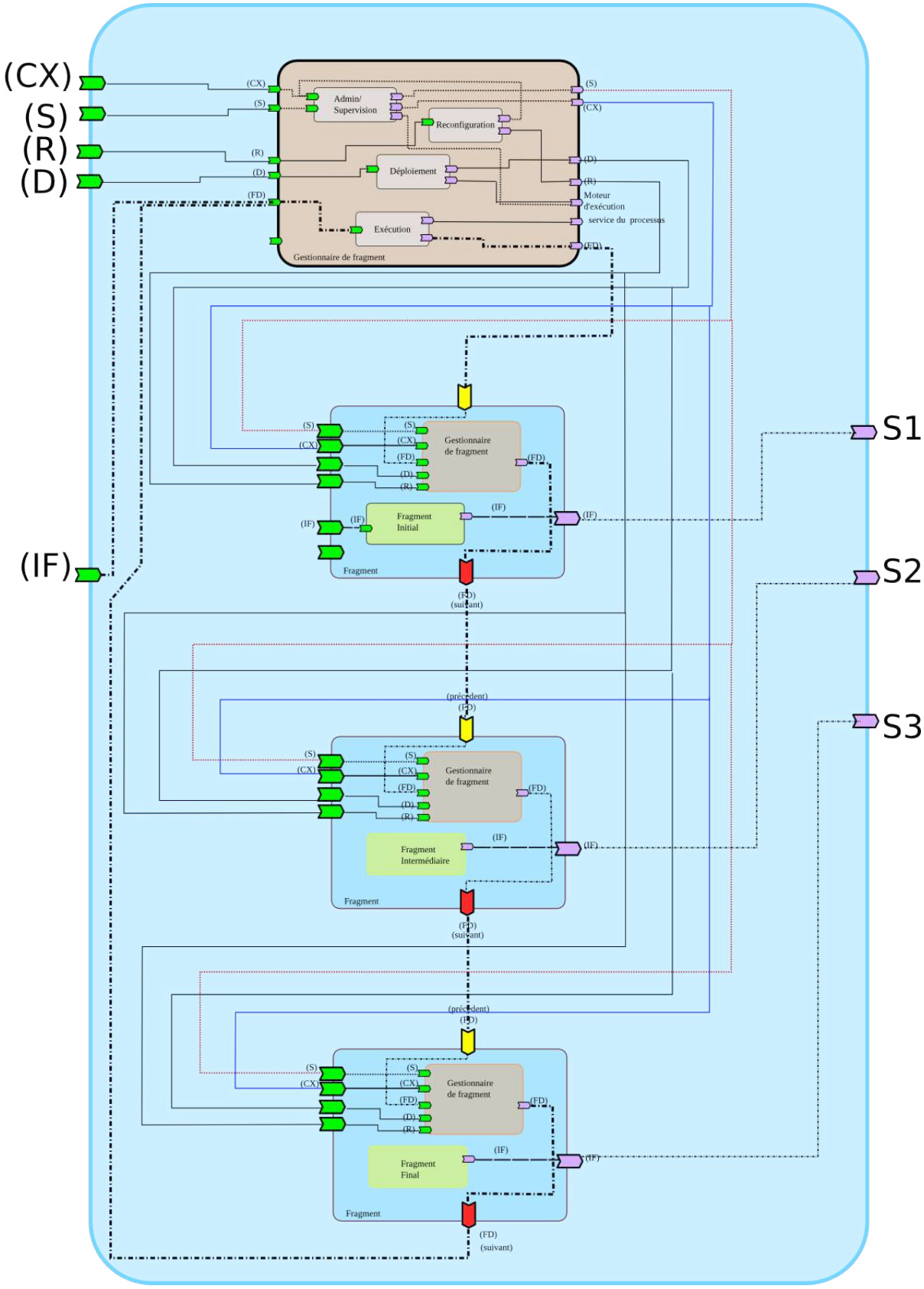


FIG. 5.11 – Exemple d'une orchestration distribuée projetée sur un fragment composite

L'art de diriger consiste à savoir abandonner la baguette pour ne pas gêner l'orchestre.

Herbert von Karajan.

Chapitre 6

Mise en Œuvre de la solution

Contenu du chapitre :

6.1 Contexte Technologique 109
6.1.1 WS-BPEL 110
6.1.2 Fractal 115
6.1.3 GCM : Grid Component Model 118
6.1.4 Le support pour le déploiement distribué 118
6.1.5 GCM/ProActive 123
6.2 Implémentation de notre solution 130
6.2.1 Choix techniques 130
6.2.2 Implémentation d'un Fragment Unitaire 131
6.2.3 Projection d'une orchestration distribuée sur un composant composite 139
6.3 Évaluation des performances du modèle à composants 144
6.4 Conclusion 145

Le Chapitre 5 a présenté la modélisation de notre approche. Ce chapitre présente maintenant la mise en œuvre technique de ce modèle Nous nous basons sur l'intergiciel ProActive/GCM pour supporter cette mise en œuvre afin de démontrer la faisabilité de notre solution. L'implémentation de notre approche permet d'exécuter une orchestration décentralisée définie comme un ensemble de processus WS-BPEL coopérants et englobée dans un composant GCM.

6.1 Contexte Technologique

Cette section présente les détails techniques relatifs à l'implémentation de la plateforme que nous proposons. Dans le chapitre 5, nous avons présenté la modélisation de cette plateforme d'exécution des orchestrations distribuées et dynamiques ; nous allons dans cette section introduire les différents concepts technologiques sur lesquels elle repose. Nous commençons par présenter en détails WS-BPEL que nous avons utilisé pour implémenter

les processus métiers. Nous présenterons ensuite le modèle à composants GCM ainsi que son implémentation de référence qu'est GCM/ProActive.

6.1.1 WS-BPEL

WS-BPEL, que nous avons déjà introduit dans le Chapitre 2, est un langage pour les processus métiers basé sur XML conçu pour orchestrer et composer les Services Web, en se basant sur leur description WSDL. Une orchestration est une collaboration plusieurs services gérée par un tiers, en l'occurrence le moteur d'exécution WS-BPEL.

Le processus WS-BPEL est défini dans un fichier où sont explicités l'enchaînement et la logique des actions qui seront exécutées par le moteur d'orchestration. Ce fichier est véritablement le code source de l'application que constitue le processus, le moteur d'orchestration agissant comme une machine virtuelle capable d'exécuter le code WS-BPEL.

Par ailleurs, un processus WS-BPEL dispose d'une logique d'invocation, celle-ci pouvant être synchrone ou asynchrone. En effet, WS-BPEL fait fortement usage des autres standards liés aux Services Web, comme par exemple, WSDL et SOAP. Le processus WS-BPEL est lui-même accessible en tant que Service Web ; il dispose donc de sa propre description WSDL.

Les processus WS-BPEL peuvent interagir avec les Services Web externes, via le concept du *lien partenaire*[1] , de deux façons :

- Le processus WS-BPEL invoque les opérations des autres Services Web. Pour chaque Service Web impliqué dans le processus, il existe un lien partenaire défini.
- Le processus WS-BPEL reçoit des invocations depuis le client. Un des clients est l'utilisateur de processus WS-BPEL, qui réalise l'invocation initiale. Les autres clients peuvent être des Services Web, par exemple, ceux qui ont été invoqués par le processus, et qui réalisent des rappels (des *callbacks*) pour retourner les réponses. Chaque processus WS-BPEL possède au moins un lien partenaire client nécessaire au déclenchement de son exécution.

Un processus WS-BPEL peut être *synchrone* ou *asynchrone.* Un processus WS-BPEL synchrone bloque le client jusqu'à ce que l'exécution soit terminée et qu'un résultat est retourné à ce client. Un processus asynchrone ne bloque pas le client. Il utilise un *callback* pour retourner le résultat (si il y en a un). En général les processus asynchrones sont utilisés pour définir des actions de longue durée.

Dans la suite de cette section, nous entrons dans les détails de ce langage.

Gestion des données Le processus WS-BPEL possède un état, qui est maintenu par des variables contenant des données. Ces données sont combinées afin de contrôler le comportement du processus métier. Elles sont utilisées dans les expressions et les opérations d'affectation. Les expressions permettent d'ajouter des conditions de transition ou de jointure au flot de contrôle. L'affectation permet de mettre à jour l'état du processus, en copiant les données d'une variable à une autre ou en introduisant de nouvelles données issues de l'évaluation d'expressions.

Activités La structure d'un processus WS-BPEL est composée d'une succession d'éléments du langage WS-BPEL traduisant la logique séquentielle des tâches automatisées. Avec une approche simplifiée, une activité d'un processus WS-BPEL est équivalente à une tâche d'un processus métier sous la forme simple ou structurée. Le type simple représente

[1] partner link

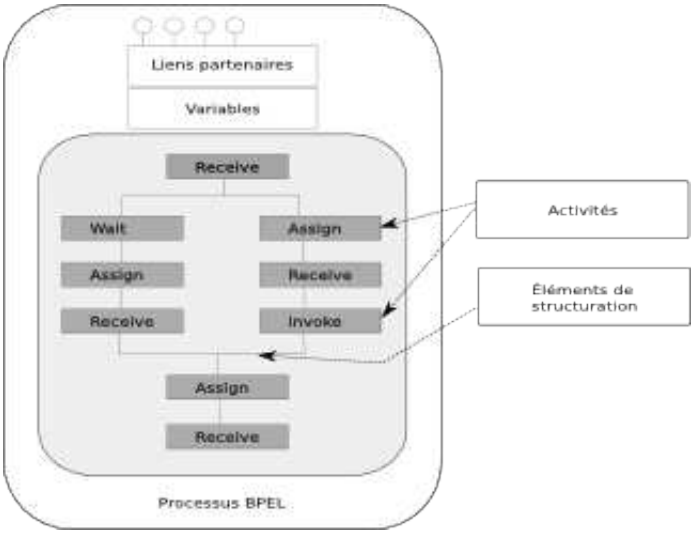


FIG. 6.1 – Anatomie d'un processus WS-BPEL

une opération de base comme l'affectation d'une donnée ou l'envoi d'un message. Le type structuré comporte une succession d'éléments XML illustran tdes comportements logiques spécifiques comme l'itération ou le choix conditionnel. L'ordonnancement des éléments composant la définition d'un processus WS-BPEL suit un schéma initial comme l'illustre la Figure 6.1.

Un workflow WS-BPEL est lui-même constitué d'activités de contrôle, de gestion des données et d'interactions avec des partenaires. Les activités, par essence, proposent des moyens pour le passage des messages (les activités de base, décrites dans le Tableau 6.1) et pour spécifier le parallélisme et la synchronisation (les activités structurées décrites dans le Tableau 6.2).

Les activités structurées décrivent la manière par laquelle un processus métier est créé en composant des activités de base en structures complexes qui permettent d'exprimer le workflow, les structures de contrôle, le flot de données, la gestion des erreurs et des évènements externes et la coordination des messages échangés entre les instances de processus.

Le Listing 6.1, illustré par la Figure 6.2 montre un ensemble minimaliste de code WS-BPEL déclarant deux partenaires (l'interface du composite et un Service Web de météorologie), spécifiant deux variables et déclarant une activité composite de type séquence composée de sous activités simples (**`receive`**, **`invoke`** et **`reply`**) dont le rôle est de recevoir le message client de la composition, de le transmettre au service météo et enfin de renvoyer la réponse au client.

Portées Une portée (ou ***`scope`***) fournit un contexte de comportement pour chaque activité qu'il contient. Une portée inclut obligatoirement une activité structurée primaire qui définit son comportement normal et elle est partagée par l'ensemble des activités qui y sont imbriquées.

En plus des activités, une portée peut contenir des gestionnaires d'erreurs, de compensation, d'événements et de terminaison. Les variables qui y sont définies existent tant que la portée est active. Une portée devient active lorsque ses activités peuvent être exécutées qu'elle contient ont terminé leur exécution.

Listing 6.1 – Un exemple simple de processus WS-BPEL

```
<process>
        <partnerLinks>
                <partnerLink myRole="proxy" name="proxyPLT"
                                        partnerLinkType="proxyPLT" />
                <partnerLink name="Forecast" partnerLinkType="
                    ForecastLinkType"/>
        </partnerLinks>
        <variables>
                <variable messageType="proxyRequest" name="proxyRequest"/
                    >
                <variable messageType="proxyResponse" name="proxyResponse
                    "/>
        </variables>
        <sequence>
                <receive createInstance="yes" operation="proxy"
                            partnerLink="proxyPLT" portType="ProxyService
                                " variable="proxyRequest"/>
                <invoke inputVariable="proxyRequest" operation="invokeWF"
                     outputVariable="proxyResponse"
                                        partnerLink="Forecast" portType="
                                            invokeWFPT"/>
                <reply operation="proxy" partnerLink="proxyPLT" portType=
                    "ProxyService" variable="proxyResponse"/>
        </sequence>
</process>
```

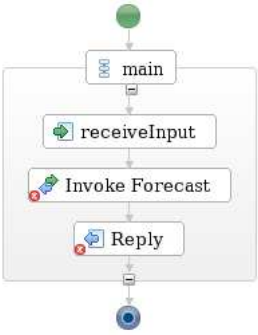


FIG. 6.2 – Exemple graphique d'un processus WS-BPEL correspondant au listing 6.1

ACTIVITÉ	DESCRIPTION
`invoke`	Permet d'appeler le port d'un Service Web partenaire qui doit être préalablement défini. Les activités `invoke` prennent en entrée une variable dont le type est défini dans le WSDL du Service Web associé et une variable de sortie si le processus est synchrone. Cette activité est bloquante dans le cas où l'appel est synchrone.
`receive`	Permet de se mettre en attente de la réception d'un message. L'activité `receive` doit spécifier le partenaire en interaction en précisant son rôle dans le processus, le type de port et l'opération fournie. Le lien partenaire doit avoir été créé au préalable. Il s'agit de la première activité que l'on retrouve dans un flux.
`reply`	Correspond à la réponse adressée au client de la composition de service. Cet élément est utilisé pour répondre à une invocation, ce qui impose qu'il doit obligatoirement suivre un `receive` pour le même partenaire, type de port et opération.
`wait`	Effectue une temporisation
`assign`	Permet d'affecter une valeur à une variable
`throw`	Permet de signaler explicitement une erreur interne et de lancer une exception au cours du déroulement du flux. elle pourra être rattrapée au moyen d'un bloc catch associé à la portée d'exécution dans laquelle l'exception est apparue. A priori, les exceptions à lancer sont des exceptions fonctionnelles, il sera donc nécessaire de définir ces exceptions au préalable. Toute erreur doit avoir un nom unique qui doit être transmis ainsi que les variables contenant des informations sur l'erreur lorsqu'un `throw` est invoqué.
`terminate`	arrête l'instance du processus

TAB. 6.1 – Activités composites du langage WS-BPEL

Gestionnaire de compensation Le but de la compensation est d'annuler les effets des activités exécutées précédemment et qui font partie d'un processus métier qui doit être annulé. S'il est défini, le gestionnaire de compensation (qui est en fait une activité) contient l'activité qui sera effectuée dans le cas où l'utilisateur désire compenser l'activité d'une portée. Le gestionnaire de compensation est installé (c'est-à-dire qu'il est autorisé à s'exécuter) lorsque la portée se termine avec succès, c'est-à-dire lorsqu'aucune erreur ne s'est produite pendant l'exécution et toutes les activités ont été exécutées.

Gestionnaire d'exceptions L'exécution de processus WS-BPEL est généralement soumise à de longues durées avec des périodes d'inactivités, à des échanges de messages asynchrones avec les partenaires et au traitement d'information de l'entreprise. Dans ce contexte, il est nécessaire qu'un processus WS-BPEL puisse disposer d'un moyen de résoudre les problèmes lorsqu'ils surviennent. Le gestionnaire d'exceptions est le mécanisme de détection et de déclenchement d'activités correctives stabilisant la situation. Le méca-

ACTIVITÉ	DESCRIPTION
flow	Exécute un ensemble d'activités de manière concurrente . Il permet de représenter la simultanéité et la synchronisation. Le regroupement de plusieurs activités dans un **flow** permet la modélisation de la concurrence. Un **flow** se termine lorsque toutes les activités qu'il contient sont achevées.
sequence	Exécute séquentiellement un ensemble d'activités. Une activité séquentielle contient une ou plusieurs activités qui sont exécutées dans l'ordre dans lequel elles sont citées dans l'élément de séquence. L'activité de séquence se termine lorsque la dernière activité dans la séquence est terminée.
switch	permet de sélectionner une branche d'activité parmi plusieurs, en tenant compte de conditions. Les différents cas sont examinés dans l'ordre dans lequel ils apparaissent. La première branche dont la condition est vérifiée définit l'activité à accomplir par le **switch**. La condition est exprimée par une condition booléenne. Si aucune condition n'est vérifiée alors c'est l'activité par défaut qui est réalisée, si elle existe, sinon le **switch** s'arrête immédiatement. L'activité se termine quand l'activité sélectionnée est terminée.
while	effectue des itérations jusqu'à satisfaction d'un critère.
pick	bloque le processus jusqu'à ce qu'un évènement spécifique se produise (réception d'un message, alarme temporelle...). Le **pick** est constitué d'un ensemble de branches de la forme évènement-activité où exactement une des branches sera sélectionnée. Une branche est sélectionnée si l'évènement qui lui est associé se produit. Une fois qu'un évènement ait été intercepté, les autres évènements ne sont plus acceptés.
scope	Définit une zone restreinte du processus permettant la définition et la validité de variables, de fautes et de gestionnaires d'exceptions
compensate	Permet d'invoquer un processus de compensation d'exception.

TAB. 6.2 – Activités composites du langage WS-BPEL

nisme de gestion des exceptions est proche de celui supportant le mécanisme de gestion de compensation.La spécification du langage WS-BPEL défini l'élément <catch> représentant un gestionnaire d'exception capable d'exécuter certaines activités associées au type d'exception. L'élément <faultHandlers> regroupe l'ensemble des gestionnaires d'exception.

Corrélations Le concept de la corrélation consiste à identifier l'instance de processus à laquelle un message est destiné. Le comportement d'un processus WS-BPEL dépend de l'historique de ses interactions avec ses partenaires. En réalité, le déroulement d'un processus WS-BPEL correspond à l'exécution de ses instances. La définition du processus WS-BPEL est utilisée comme modèle durant la création de ces instances.L'échange de messages entre les instances de processus BPEL et les partenaires nécessitent un mécanisme de cheminement assurant une communication correcte des informations dans toutes les directions.Un ensemble de corrélation est un mécanisme gardant la trace des messages utilisés durant les interactions entre partenaires et en assurant leur bon cheminement. La corrélation commence à vérifier la concordance avec le destinataire lorsqu'une instance reçoit un message.

Nous venons de présenter la spécification WS-BPEL ainsi que les éléments qui la constituent. Nous utiliserons WS-BPEL pour définir des sous-processus faisant partie d'une orchestration globale. Nous allons, dans la suite nous intéresser à la technologie à composants qui va nous servir de support pour exécuter les orchestrations décentralisées.

6.1.2 Fractal

Le modèle de composants Fractal[2] est un modèle de composants général destiné à mettre en œuvre, déployer, et administrer des systèmes logiciels complexes. Parmi ses fonctionnalités principales, Fractal propose :

- un **modèle de composants hiérarchique**. Les composants Fractal peuvent être des primitives, c'est-à-dire des composants de base unitaires, ou des composants composites qui sont des composants qui peuvent contenir d'autres composants. Cela permet d'offrir une vue uniforme de l'application pour divers niveaux d'abstraction.
- de la **réflexivité**. Les composants Fractal ont des capacités réflexives et sont capables d'introspecter leur structure externe et interne, ce qui permet de leur donner un contrôle sur leur contenu, leurs liaisons et leur cycle de vie.
- des capacités de **Reconfiguration**. Les applications basées sur les composants Fractal peuvent être dynamiquement (re)configurées en modifiant de façon dynamique leurs liaisons et leur contenu.

Un composant Fractal est une entité d'exécution qui est encapsulée, a une identité, et supporte une ou plusieurs *interfaces*. Une interface est le point d'accès à la fonctionnalité du composant et peut avoir un ou deux rôles : les *interfaces serveurs*, qui acceptent les invocations entrantes ; les *interfaces clientes*, qui supportent les invocations sortantes. Les éléments principaux du modèle Fractal, ainsi que la notation habituelle, sont montrés dans la figure 6.3.

Les composants Fractal sont composés d'une *membrane*, qui permet de supporter l'introspection et la reconfiguration des fonctionnalités internes du composant, et d'un *contenu*, qui inclut un ensemble fini d'autres composants (appelés sous-composants) dans le cas d'un composant composite, ou d'une implémentation, dans le cas d'un composant primitif.

[2] http ://fractal.ow2.org

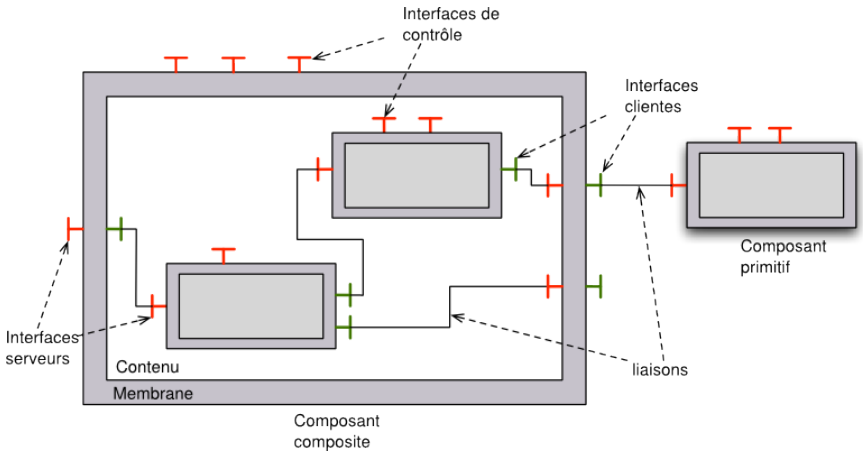


FIG. 6.3 – Éléments du modèle à composants Fractal

La *membrane* d'un composant Fractal comprend des interfaces *externes* et *internes*. Les interfaces internes, qui existent dans le cas d'un composant composite, sont seulement accessibles depuis les sous-composants du composite. Les interfaces externes sont accessibles par les composants externes. La membrane est composée d'un ensemble de contrôleurs, qui permettent de gérer le contenu d'un composant, par exemple en contrôlant le cycle de vie (démarrage et arrêt) du composant, de reconfigurer les liaisons vers d'autres composant, d'ajouter ou d'enlever des composants dans un composite, de modifier les attributs du contenu. Le modèle Fractal est défini comme un système extensible, de façon à créer des contrôleurs personnalisés en fonction des besoins de gestion de l'application. Les tâches de gestion sont disponibles par le biais d'interfaces spéciales, appelées *interfaces de contrôle*.

Un concept important dans Fractal est le mécanisme de liaison, qui permet de construire l'architecture d'une application Fractal. Les liaisons permettent de connecter une interface cliente à une interface serveur, ce qui signifie que les invocations émises par l'interface cliente doivent être acceptées par l'interface serveur. Pour être reliées entre-elles, les interfaces clientes et les interfaces serveur doivent appartenir au même espace d'adressage : les liaisons ne peuvent pas traverser la membrane. Par exemple, dans le cas d'un composite, un composant externe ne peut pas être relié directement à une interface serveur d'un sous-composant. De la même façon, un sous-composant ne peut pas être relié a une interface serveur d'un composant externe à son composite.

En gérant ses liaisons et sa composition, une application Fractal peut être structurellement reconfigurée. Cela signifie que la structure de l'application peut être modifiée en changeant les liaisons d'un composant vers d'autres interfaces serveurs, ou en ajoutant ou enlevant les composants d'un composite. La seule nécessité engendrée par de tels changements est l'arrêt des composants qui vont subir une reconfiguration, cela pour garantir l'intégrité de la composition.

Le type d'un composant Fractal est déterminé par l'ensemble des interfaces qu'il propose. Hormis leur rôle, les interfaces incluent les concepts de *cardinalité* et de contingence. La *cardinalité* peut être fixée à *singleton*, ce qui indique que le composant a exactement une interface du type spécifié, ou elle peut être fixée à *collection*, ce qui indique que le composant possède un nombre arbitraire d'interfaces du type spécifié ; de telles interfaces sont

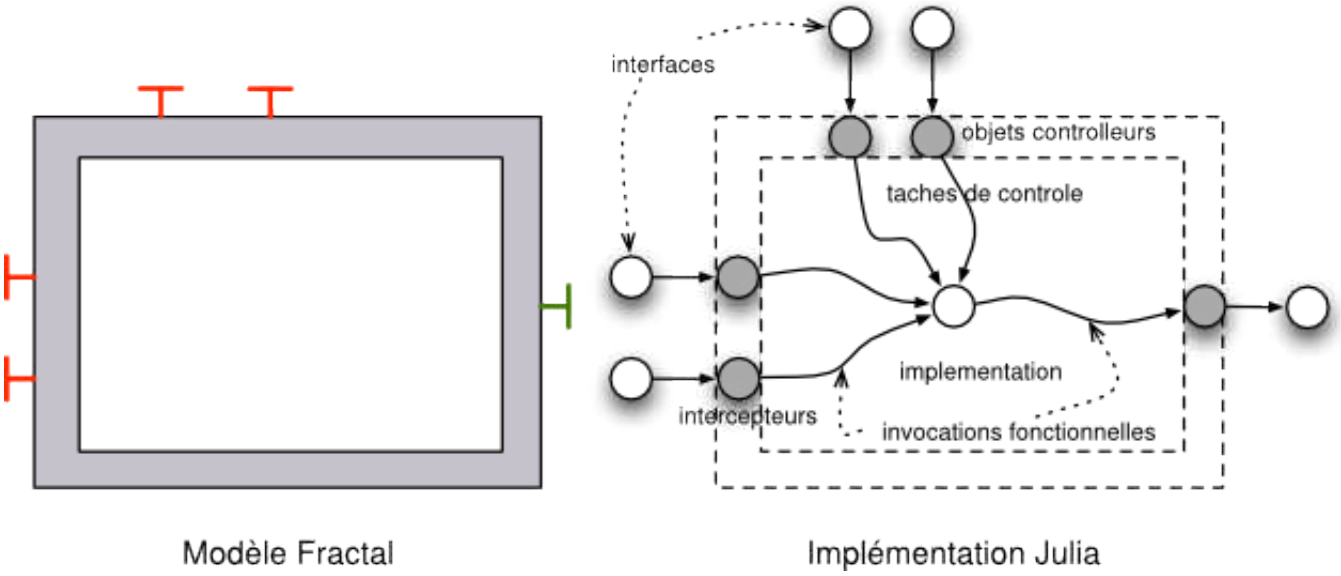


FIG. 6.4 – Le modèle d'un composant Fractal et son implémentation avec Julia

en général créées suite à une requête de création de liaison. La contingence d'une interface correspond au fait qu'une fonctionnalité fournie par une interface peut être disponible ou pas au moment de l'exécution ; ainsi, une interface obligatoire (*mandatory*) est garantie quant à sa disponibilité au moment de l'exécution, ce qui signifie, pour une interface cliente, qu'elle doit être liée avant de démarrer le composant ; une interface optionnelle n'a pas cette garantie, et le composant peut être démarré même si l'interface n'est pas liée.

Julia est une implémentation de référence de Fractal, implémentée en Java. Sans entrer dans les détails de l'implémentation de Julia, il est important de mentionner que Julia repose sur un ensemble d'objets Java pour implémenter le contenu des composants Fractal, leur interfaces et contrôleurs. De plus, des objets optionnels que l'on appelle *intercepteurs* peuvent être ajoutés dans la membrane pour intercepter des invocations fonctionnelles, fournissant ainsi des traitements personnalisés. La membrane d'un composant Julia est un ensemble d'objets Java qui implémentent la logique d'administration du composant. La Figure 6.4montre un composant Fractal générique, et la vue équivalente de ce composant implémentée avec le canevas Julia.

En ce qui concerne l'administration des composants Fractal, la spécification[3] propose des contrôleurs utiles tels que :

- Le contrôleur des attributs (`AttributeController`). Il expose les opérations *getter* et *setter* pour certains attributs configurables du composant.
- Le contrôleur des liaisons (`BindingController`). Il expose une interface *BindingController* qui permet de créer des liaisons, et de les détruire entre une interface cliente du composant vers des interfaces serveurs externes.
- Le contrôleur de contenu (`ContentController`). Il expose une interface qui permet de lister, ajouter et enlever des sous-composants du contenu, dans le cas d'un composant composite.
- Le contrôleur du cycle de vie. (`LifeCycleController`). Il expose une interface `LifeCycleController`

[3]http ://fractal.ow2.org/specification/

qui permet de contrôler l'état du composant.

Fractal a servi de base pour développer des supports d'exécution répondant à la spécification SCA tels que FraSCAti, qui bénéficie des fonctionnalités d'introspection de Fractal et fournit des capacités de reconfiguration aux architectures basées sur la spécification SCA.

Les applications Fractal peuvent être décrites en utilisant l'ADL Fractal[4]. L'ADL Fractal est un langage de description basé sur XML et permet de décrire tous les éléments (composants, liaisons, contrôleurs, implémentations, interfaces) d'une application Fractal, ainsi que leur assemblage.

6.1.3 GCM : Grid Component Model

Le modèle GCM[5], aide à la conception, l'implémentation et l'exécution des applications de grille basées sur les composants. Le modèle GCM a été spécifié dans le cadre du projet CoreGRID, et standardisé par l'Institut Européen des Standards pour les Télécommunications (ETSI, pour European Telecommunications Standard Institute).

La spécification GCM est basée sur le modèle à composants Fractal. De ce fait, les composants GCM partagent beaucoup de caractéristiques avec les composants Fractal, notamment en termes de hiérarchie, de composition, d'interfaces, de liaisons et de capacités de reconfiguration. Cependant, GCM met en avant des considérations qui ne font pas partie de la spécification Fractal :

- Le support pour le **déploiement distribué**
- Un meilleur support pour les **communications collectives**
- Une **séparation** stricte entre les **préoccupations fonctionnelles** et les **préoccupations non-fonctionnelles**.

L'architecture d'une application GCM peut être décrite en utilisant l'ADL GCM, qui étend l'ADL Fractal, incluant les éléments définis par GCM. Dans la suite, nous décrivons les principales fonctionnalités du modèle GCM que nous utiliserons dans la réalisation de notre solution.

6.1.4 Le support pour le déploiement distribué

Une des fonctionnalités ajoutées par GCM par rapport au modèle Fractal est son support générique pour le déploiement distribué, qui est basé sur la notion abstraite du *Nœud Virtuel (VN pour Virtual Node)*. Un Nœud Virtuel est une référence abstraite vers la ressource où le composant sera déployé. Cette abstraction capture les besoins de déploiement et permet de séparer les tâches de conception d'architecture d'une application GCM, de l'approvisionnement des ressources où les composants seront déployés.

Les Nœuds Virtuels possèdent une propriété de cardinalité qui peut être définie comme simple ou multiple. Dans le premier cas, le VN doit correspondre à exactement un nœud sur l'infrastructure physique, tandis que dans le second cas, il peut correspondre à (c'est-à-dire qu'il peut englober) plusieurs nœuds.

L'information concernant les VN est incluse dans l'ADL GCM et décrit une infrastructure virtuelle. Au moment du déploiement, les VNs doivent être associés à une infrastructure physique concrète. L'approvisionnement des ressources physiques peut être délégué à un outil spécialisé. Un exemple d'un tel outil est le Gestionnaire de Ressource ProActive, qui est capable de gérer les nœuds sources situés sur différentes infrastructures.

[4] http ://fractal.ow2.org/fractaladl/
[5] Grid Component Model

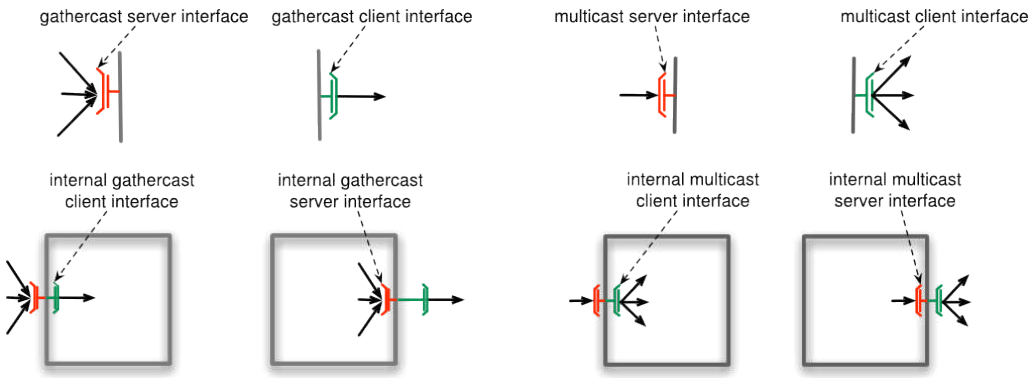


FIG. 6.5 – Les Interfaces *multicast* et *gathercast* de GCM

Dans une situation où le support pour une infrastructure destinée à supporter les applications basées sur les services, et qui repose de plus en plus sur une haute disponibilité des ressources de calcul, un moyen générique de décrire l'infrastructure s'applique parfaitement bien à cette vision, permettant la conception abstraite de l'application par rapport à l'infrastructure existante.

6.1.4.1 Le support pour les communications collectives

Le support pour les communications collectives est amélioré dans le modèle GCM, grâce à l'introduction des interfaces *gathercast* et *multicast*. Le but ici est de fournir des communications *plusieurs-vers-une* et *une-vers-plusieurs* efficaces. Ces interfaces permettent de gérer un groupe d'interfaces comme une seule entité. La notation pour ces deux interfaces est montrée dans la Figure 6.5

6.1.4.2 Les interfaces *gathercast*

Les interfaces *gathercast* permettent d'effectuer des communications *plusieurs-vers-une* (M x 1) et peuvent être utilisées pour effectuer des synchronisations, de la récupération de paramètres, de la réduction et de l'envoi de résultat. Le comportement spécifique pour ces interfaces peut être spécifié en utilisant des politiques d'agrégation.

Les interfaces *gathercast* reçoivent les invocations depuis plusieurs interfaces qui leur sont connectées, rassemblant ainsi tous les paramètres, et les réduisant dans une seule invocation. Quand un résultat est obtenu, l'interface *gathercast* effectue l'envoi correspondant vers les partenaires qui ont effectué l'invocation.

6.1.4.3 Les interfaces *multicast*

Les interfaces *multicast* permettent les communications *une-vers-plusieurs* et peuvent être utilisées pour effectuer des invocations parallèles, l'envoi de paramètres et le rassemblement de résultats. Le comportement spécifique pour ces interfaces peut être configuré en utilisant différents modes d'envoi (*dispatch modes*). Les modes d'envoi peuvent être utilisés pour définir la façon dont les paramètres de l'invocation originelle sera divisée entre les différentes destinations, ou pour choisir l'interface, parmi les interfaces connectées, qui sera choisie pour effectuer l'invocation.

Les interfaces *multicast* transforment une seule invocation en plusieurs invocations parallèles. Quand un résultat est obtenu, les résultats, potentiellement nombreux, peuvent être agrégés et retournés au partenaire qui est à l'origine de l'invocation.

L'existence des interfaces *multicast* et *gathercast* a été développée initialement pour les interfaces fonctionnelles, et permet qu'un composant soit connecté à un nombre indéfini d'interfaces. Dans notre approche, nous utilisons la communication avec les membranes, puisque non-fonctionnelles, de tous les composants impliqués dans l'application, grâce aux interfaces *multicast.*

6.1.4.4 Le support pour les préoccupations non-fonctionnelles

La spécification Fractal, et son implémentation de référence Julia, décrivent les éléments de gestion comme un ensemble d'objets contrôleurs contenus dans la membrane d'un composant Fractal. Ces contrôleurs sont définis de façon statique et sont décrits par le biais de l'ADL Fractal.

Les composants GCM étendent cette notion pour permettre l'introduction des composants Non-Fonctionnels (NF) dans la membrane. Les composants NF ont un but similaire aux contrôleurs, qui est de fournir des fonctionnalités de gestion aux composants GCM, et de prendre en charge des tâches Non-Fonctionnelles.

6.1.4.5 Séparation entre les préoccupations Fonctionnelles (F) et Non-Fonctionnelles (NF)

La motivation pour fournir une membrane "componentisée" aux composants GCM, est de permettre la composition d'activités plus complexes dans la partie contrôle du composant, et de fournir une séparation plus claire des préoccupations entre les tâches fonctionnelles et les tâches non-fonctionnelles. Pour cela, l'ADL GCM peut être divisé et l'architecture componentisée de la membrane peut être décrite dans un fichier séparé de la partie fonctionnelle. Cela permet de développer les deux parties de façon indépendante et de les associer seulement au moment du déploiement, appliquant ainsi la séparation des des préoccupations fonctionnelles et non-fonctionnelles.

Le concept des tâches NF se réfère à toutes les tâches qui ne sont pas en relation avec le but principal (ou but fonctionnel) d'un composant GCM. Les taches NF peuvent être définies comme les tâches qui supportent le but fonctionnel d'un composant. Elles incluent en général la gestion du cycle de vie du composant, et la supervision de l'exécution du but fonctionnel.

6.1.4.6 Les interfaces NF

Dans beaucoup de cas, les activités de gestion peuvent requérir une collaboration significative de plusieurs tâches, qui peuvent être propagées transversalement vers d'autres composants. Par exemple, pour obtenir la mesure de la consommation d'énergie d'une application, il est nécessaire d'agréger la consommation d'énergie de tous les composants individuels inclus dans le composite, ou clients du composite.

GCM inclut des interfaces de contrôle additionnelles que l'on appelle *interfaces non-fonctionnelles*. Contrairement aux composants Fractal qui fournissent seulement des interfaces NF serveurs pour accéder aux tâches NF, GCM propose des interfaces NF clientes. Les interfaces NF clientes peuvent être connectées aux interfaces NF serveurs, dans le but de communiquer avec les membranes des différents composants et permettent d'effectuer des tâches NF collaboratives.

Dans la mise en œuvre de notre solution, nous utiliserons la possibilité d'avoir une vue orientée composant de la membrane des composants GCM, pour fournir une implémentation flexible et modulaire de notre gestionnaire d'orchestration distribuée, décrit dans le chapitre 5.

En ce qui concerne les interfaces internes, GCM permet à la membrane de communiquer avec le contenu, permettant ainsi une communication hiérarchique entre les taches NF collaboratives d'un composite et de ses sous-composants. Dans ce cas, GCM définit des interfaces NF internes. Ces interfaces communiquent avec les composants placés dans le contenu fonctionnel. Elles permettent aussi à ces sous-composants de communiquer avec les composants contenus dans la membrane du composite dans lequel ils sont inclus.

6.1.4.7 Notation et liaisons Non-Fonctionnelles

La figure 6.6 montre un exemple d'une application GCM où les composants NF ont été placés dans la membrane d'un composant composite et d'un composant primitif. Nous pouvons remarquer que GCM permet la coexistence des contrôleurs et des composants NF (parfois appelés composants contrôleurs) dans la membrane. Cependant, les contrôleurs objets ne peuvent pas être modifiés pendant l'exécution, et ne peuvent pas être connectés aux autres interfaces NF, ni communiquer avec elles, comme le feraient d'autres composants NF.

La description dans la Figure 6.6, illustre aussi les connexions possibles entre les interfaces introduites précédemment. Une interface cliente d'un composant NF peut se connecter à une interface NF cliente interne appartenant à la membrane (1). Cette interface NF interne cliente agit comme un point de passage de la membrane vers le contenu. Une interface NF cliente interne peut se connecter à l'interface NF externe d'un sous-composant (2), concrétisant la communication de la membrane vers le contenu. Cela permet de propager les taches d'administration initiées dans la membrane du composite vers les membranes des sous-composants. Un exemple très simple consiste à un signal d'arrêt depuis la membrane du composite qui peut être propagé aux membranes de ses composants internes.

Inversement, la communication peut aussi s'effectuer du contenu vers la membrane. Une interface NF cliente d'un sous-composant peut se connecter à une interface NF serveur interne de son composant parent (3). Cette interface NF serveur agit comme un point de passage du contenu vers la membrane. L'interface NF serveur interne peut se connecter à l'interface serveur d'un composant NF (4), concrétisant ainsi l'accès à la tâche d'administration. Cela permet aux membranes des sous-composants d'accéder aux tâches d'administration dans la membrane de leur composant parent. Par exemple, un composant qui représente une ressource de stockage peut utiliser une interface NF pour communiquer à son parent que la capacité de stockage est presque atteinte, et que la membrane du composite doit prendre une décision.

Les autres éléments montrés dans la figure 6.6 indiquent les connexions entre les interfaces NF serveurs externe d'un composant avec l'interface serveur d'un composant NF dans la membrane (5) ; la connexion entre l'interface cliente d'un composant NF vers une interface cliente NF externe (6) permet la communication avec l'extérieur. Finalement, une interface NF cliente peut être liée à une interface NF serveur d'un autre composant externe (7), permettant la communication entre les membranes de deux composants, et donc la la collaboration entre tâches NF. Par exemple, un gestionnaire de sécurité placé dans la membrane peut propager un certificat de sécurité à tous les composants avec lesquels il maintient une connexion fonctionnelle dans le but de lui-même s'authentifier.

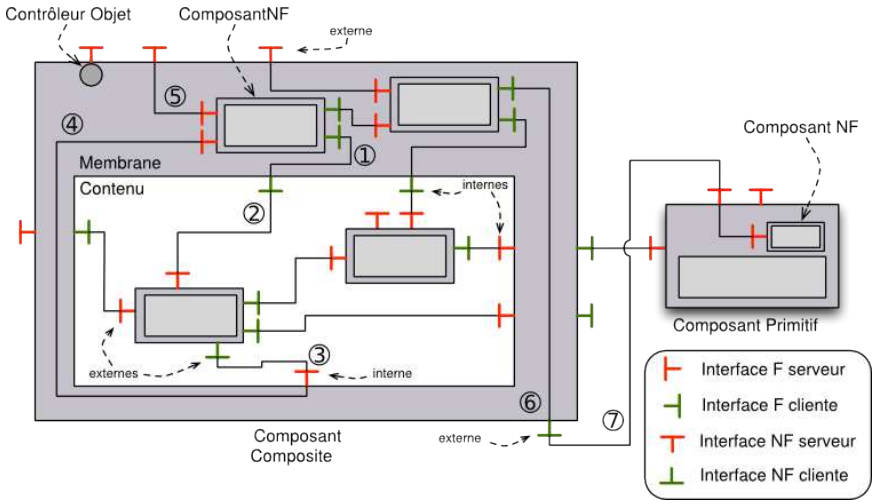


FIG. 6.6 – Les éléments d'une application GCM incluant des composants Non-Fonctionnels dans la membrane

6.1.4.8 Contrôleurs standards

GCM fournit des interfaces correspondant aux contrôleurs basiques mentionnés dans la description de Fractal (Section 6.1.2) et définit des contrôleurs standards additionnels :

- Le ***contrôleur gathercast.*** Il permet la gestion des interfaces *gathercast*, c'est-à-dire la création des interfaces *gathercast* et la gestion de leurs connexions.
- Le ***contrôleur multicast :*** Il permet la gestion des interfaces *multicast*, c'est-à-dire la création des interfaces *multicast* et la gestion de leurs connexions.
- Le ***contrôleur de membrane***. Il permet de gérer la composition et les liaisons des composants NF placés dans la membrane, ainsi que l'ajout ou la suppression des composants de la membrane. Il gère aussi les liaisons arrivant et partant des interface NF internes, permettant la connexion des interfaces NF des sous-composants avec l'activité de la membrane.

6.1.4.9 Reconfiguration

Les capacités basiques de reconfiguration dans GCM, héritées de Fractal, permettent de modifier les liaisons entre les composants GCM, d'ajouter et de supprimer les sous-composants dans un composite. Comme dans le cas d'un composant Fractal, le seul besoin pour exécuter ces reconfigurations structurelles est que le composant soit dans un état stoppé, cela pour préserver l'intégrité de la composition.

Dans le même esprit, GCM permet aussi de reconfigurer structurellement la membrane. En fait, l'ensemble des composants NF dans la membrane peut être juste vu comme une autre application GCM et leurs liaisons et relations de composition peuvent être modifiées si besoin au moment de l'exécution. En ce qui concerne le cycle de vie de l'application, un état intermédiaire a été ajouté, dans lequel les composants GCM peuvent être dans un état *démarré,* mais leur membrane peut être dans un état *démarré* ou dans un état *stoppé.* Lors de l'exécution de reconfigurations sur la composition d'une membrane, il est nécessaire que la membrane soit *stoppée,* ce qui signifie que tous les composants NF à l'intérieur de la membrane d'un composant GCM doivent être arrêtés. préalablement.

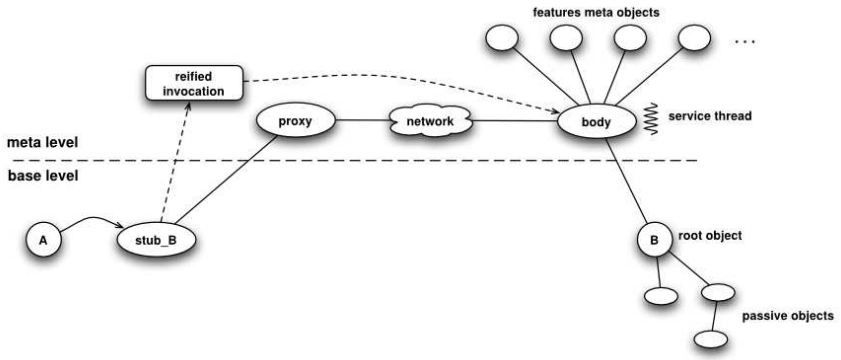


FIG. 6.7 – L'architecture à Méta-Objet

6.1.5 GCM/ProActive

L'implémentation de référence de GCM a été développée au dessus de l'intergiciel ProActive et se nomme GCM/ProActive. GCM/ProActive propose une implémentation des composants basée sur le modèle des *Objets Actifs* qui supporte les communications asynchrones basés sur des objets transparents que l'on appelle *Future.*

La notion basique importante dans GCM/ProActive est que tous les composants (primitifs et composites) sont implémentés par des *Objets Actifs.* Cela implique que les composants ont un seul fil d'exécution, et une queue où les requêtes sont stockées pour être exécutées de façon asynchrone.

Dans la suite, nous décrivons les principales fonctionnalités de GCM/ProActive, que nous utiliserons pour mettre en œuvre notre solution.

6.1.5.1 Objets Actifs et communications asynchrones

Un *Objet Actif* est construit en utilisant un protocole à méta-objet (MOP[6]), qui utilise des techniques d'introspection pour abstraire le niveau de distribution et offrir de l'asynchronisme. L'architecture du MOP est montrée dans la Figure 6.7.

Un Objet Actif est concrètement construit depuis un objet racine (de type B dans la Figure 6.7) qui est un objet normal que l'on caractérise de *passif.* Un objet appelé *body* est attaché à l'objet racine, et ce *body* référence plusieurs méta-objets qui offrent des fonctionnalités spécifiques telles que la tolérance aux pannes, la sécurité, les communications de groupes, etc ... A cela vient s'ajouter un objet appelé *stub.* Il est typé avec un sous-type de celui de l'objet racine. De ce fait, du point de vue de l'appelant, la destination parait comme étant un objet commun du type B.

L'asynchronisme est fourni comme suit et est utilisé comme montré dans le Listing 6.2. Une invocation sur l'objet actif B est en fait une invocation sur l'objet *stub,* qui crée une représentation réifiée de l'invocation contenant la méthode et ses paramètres. Cette invocation réifiée est envoyée à un objet *proxy* qui la transfère au *body* destination, potentiellement à travers le réseau, et place l'invocation réifiée dans la *queue de requêtes* de

[6]pour Meta Object Protocol

Listing 6.2 – Création d´ un objet actif avec ProActive

```
A {
...
// instantiate active object of class B on node2
B b = (B) PAActiveObject.newActive(''B'', constructorParams, node2);
// use active object as a regular object of type B
R r = b.foo();
...
// possible wait-by-necessity
System.out.println(r.printResult());
}
```

l'objet Actif. La *queue de requête* est un des meta-objets référencés par le *body*. Une fois que l'invocation réifiée est placée dans la *queue de requête,* le moment du *rendez-vous* est terminé. Si la méthode invoquée doit retourner un résultat, un objet *Future* est créé du côté de l'appelant et est retourné comme un résultat à l'appelant. Le *Future* est une promesse de réponse qui sera remplie avec la valeur de retour de l'invocation et comprend un *Future Stub* qui est un sous-type du type de retour de l'invocation, et un *Future Proxy* qui référence la valeur réelle retournée. L'objet qui a effectué l'invocation peut continuer à s'exécuter de faç normale. Cependant, si il a besoin de lire la valeur du résultat retourné, l'exécution sera bloquée jusqu'à ce que la valeur soit disponible, ou il continuera de façon transparente si la valeur a déjà été retournée. Cette fonctionnalité, qui est totalement transparente pour l'application, est appelée *attente par nécessité.*

Une fois la requête placée dans la queue de requête de l'objet actif, elle peut être servie de façon asynchrone à tout moment selon la politique de service de l'objet actif (par défaut, une politique FIFO). Une fois que l'objet actif a servi la requête, il contacte de façon transparente l'objet qui a émis la requête (et qui possède un objet *Future* pour la réponse) et met à jour le résultat, débloquant ainsi le fil d'exécution qui attendait par nécessité ce résultat. Cependant, l'appelant peut aussi avoir envoyé des copies de cet objet *Future* comme des réponses à d'autres objets, ou comme des paramètres à des requêtes. Ces copies sont alors mis à jour, toujours de façon transparente. Ce mécanisme est appelé *continuation automatique.*

La Figure 6.8 montre le statut des objets après chaque étape listée dans le Listing **??**. Un objet *a* de type *A* crée un objet actif *b* de type *B* situé sur le nœud "Node 2", en utilisant l'interface de programmation de ProActive. Cela entraine la génération d'un *stub* de type B dans le contexte de *A,* qui représente l'objet actif distant *b,* et d'un *proxy* en charge de transférer les requêtes à l'endroit où est situé l'objet actif. En même temps, l'objet actif *b* est créé sur le "Node 2", avec un *body,* une *queue des requêtes,* et l'objet racine de type *B*(Figure 6.8(a)). Quand un appel est effectué sur l'objet actif *b* (6.8(b)*),* l'appel est réalisé sur le *stub* de type *B*. L'invocation est réifiée par le *proxy* et un *rendez-vous* est initié. Le *proxy* envoie la requête réifiée au *body* de l'objet actif, qui la place dans sa *queue des requêtes*. Après cette étape, le moment du *rendez-vous* est terminé, et l'objet *a* crée un objet *Future* local, initialement vide, qui va recevoir le résultat de l'invocation, *r* (Figure 6.8 (c)). A partir de ce moment, les objets *a* et *b* continuent leurs taches en parallèle. Si *a* nécessite à un moment d'accéder à l'objet *r*, il va utiliser l'objet *FutureStub* correspondant, mais si la valeur n'a pas encore été calculée par *b*, alors le fil d'exécution de *a* bloquera au niveau

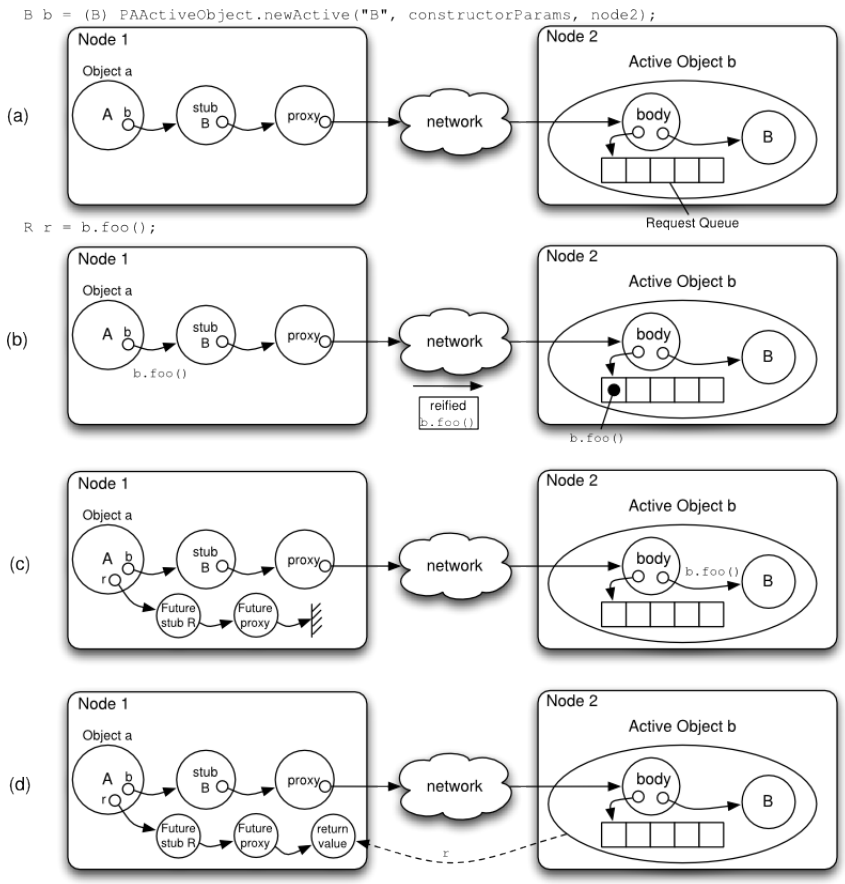


FIG. 6.8 – Séquence d'un appel asynchrone sur un objet actif

du *FutureProxy,* tant que le résultat n'est pas disponible (*attente par nécessité*). De l'autre côté, *b* décide de façon asynchrone de servir la requête et de calculer le résultat *r*. Quand *b* a terminé de servir la requête, le *body* envoie la valeur *r* à l'appelant (*a*) qui le traitera dans le *FutureProxy,* mettant ainsi à jour la valeur (Figure 6.8(d)). A ce moment, les fils d'exécution qui étaient bloqués en attendant le résultat peuvent continuer à s'exécuter.

6.1.5.2 Communications asynchrones dans les composants GCM/ProActive

GCM/ProActive fournit une implémentation de GCM au dessus de l'intergiciel ProActive. Par conséquent, chaque composant GCM/ProActive est réalisé par un objet actif qui est constitué d'un seul fil d'exécution, un *body*, et une queue des requêtes. L'implémentation est basée sur l'architecture MOP, présentée sur la Figure 6.9. La principale différence est que le rôle du *stub* dans l'architecture de base du MOP est prise en charge par deux interfaces :

- L'interface Component, qui expose la nature du composant GCM et qui permet d'introspecter les détails du composant.
- Une interface Interface qui est générée dynamiquement pour chaque interface serveur déclarée par le composant, correspondant aux méthodes fonctionnelles incluses dans l'interface.

Ces éléments agissent comme des représentant locaux d'un composant GCM/ProActive distant, et utilisent le *proxy* pour permettre l'interaction avec le *body* des objets actifs. Le *body* inclut des composants meta-objets additionnels ; parmi eux les contrôleurs mentionnés dans les sections 6.1.2 et 6.1.3.

Une invocation depuis un composant GCM vers un autre prend un chemin comme celui montré dans la Figure 6.10. Le composant GCM *A* est connecté au composant GCM *B* depuis l'interface cliente *aItf* de *A* vers l'interface serveur *bItf* de *B* (cela requiert que les interfaces soient compatibles entre-elles). L'opération de liaison est réalisée en utilisant le *BindingController* de A, et la conséquence est que l'objet actif *A* maintient une référence vers les représentants *Interface* et *Component* du composant *B* (Figure 6.10(a)). Les représentants sont utilisés pour effectuer une opération sur l'interface cliente aItf du composant *A.* L'appel est réifié et transmis comme une invocation à l'interface serveur bItf du composant *B.* L'appel est ensuite stocké dans la queue des requêtes de *B.* Le composant *A* maintient un objet *Future* afin de recevoir le résultat de l'invocation. Le composant *B* traite la requête dans sa queue, et une fois qu'il a terminé de servir celle-ci, il envoie le résultat et met à jour l'objet *Future* dans le composant *A* (Figure 6.10 (b)).

Les composants composite dans GCM/ProActive ne contiennent pas de logique d'implémentation. En revanche, ils agissent comme ses conteneurs vis-à-vis de leur sous-composants. Les composants composites contiennent des objets passifs pour leur interfaces et contrôleurs, et un unique Objet Actif qui traite les invocations reçues par le biais des interfaces serveurs du composite et délégué l'appel au sous-composant correspondant, agissant comme un proxy pour toute entité externe qui veut communiquer avec un sous-composant. De la même manière, l'Objet Actif traite toute les invocations sortantes venant d'un de ses sous-composants vers une entité externe. Dans le but de préserver la propriété d'encapsulation, un sous-composant peut effectuer des invocations sur l'interface serveur de son composant parent, qui transfère la requête au composant externe. Une vue simplifiée des invocations à travers des composants GCM/ProActive est montrée dans la Figure 6.11. L'application est la même que montrée dans la Figure 6.6, cependant, pour des raisons de clarté, nous ne montrons pas les composants non-fonctionnels.

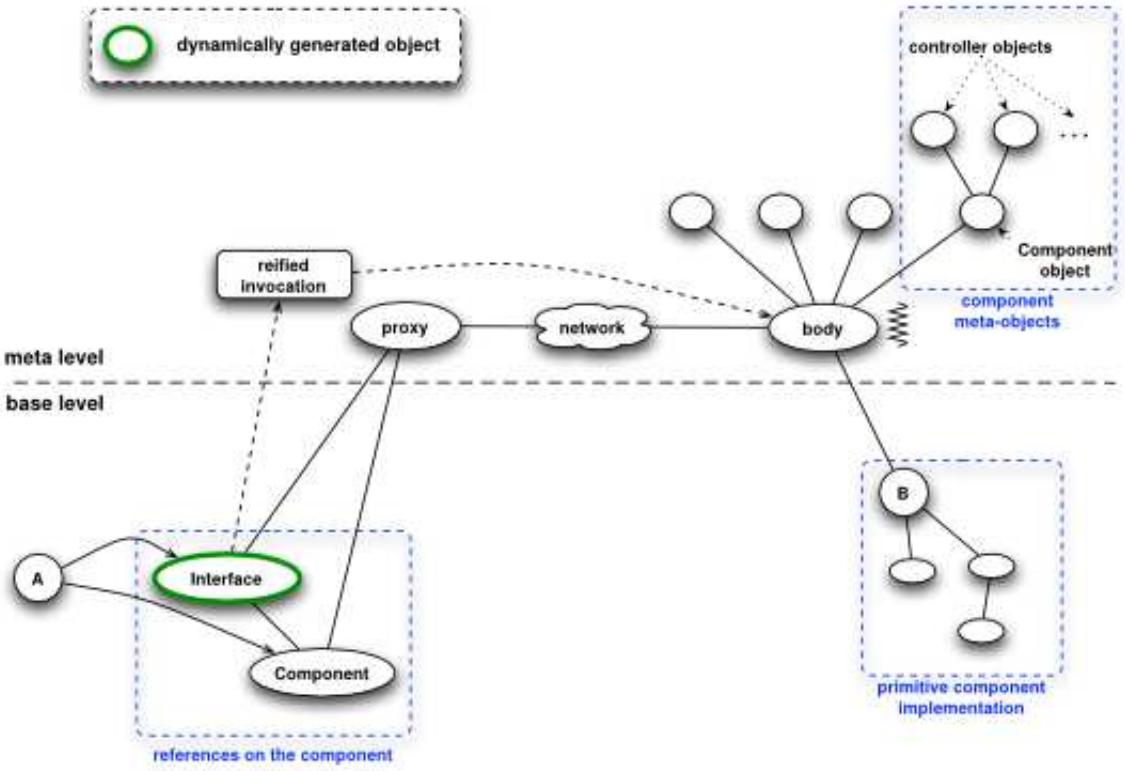


FIG. 6.9 – L'architecture à Meta-Objet pour un composant GCM

Les liaisons sont mises en œuvre comme des références depuis des objets Java qui implémentent la logique d'un composant primitif, vers un objet qui représente l'interface serveur d'un autre composant GCM.

6.1.5.3 Instrumentation des composant GCM/ProActive

L'intergiciel GCM/ProActive est instrumenté grâce à la technologie JMX (Java Management Extension), dans le but de pouvoir fournir des notifications concernant les évènements survenant dans l'intergiciel. GCM/ProActive fournit un *connecteur JMX ProActive* qui permet de se connecter à l'intergiciel et de propager les notifications JMX de façon asynchrone en utilisant la sémantiquee de communication de ProActive [BCL07].

Les *bodies* des Objets Actifs sont instrumentés avec des MBeans (*BodyWrapperMBean*) qui sont utilisés pour générer les notifications JMX. Le tableau 6.3 montre les notifications générées par l'intergiciel GCM/ProActive, qui correspondent au traitement des requêtes et à la gestion des objets *Future*.

Cet ensemble de notifications a été utilisé dans des applications en relation avec l'intergiciel ProActive, comme IC2D, un outil graphique qui permet de superviser et de mesurer les performances des applications ProActive. IC2D écoute les notifications JMX générées par l'*objet BodyWrapperMBean* afin de représenter graphiquement les communications entre les Objets Actifs, et d'obtenir des informations sur les performances, notamment sur le temps passé dans les tâches internes et dans les communications d'une application

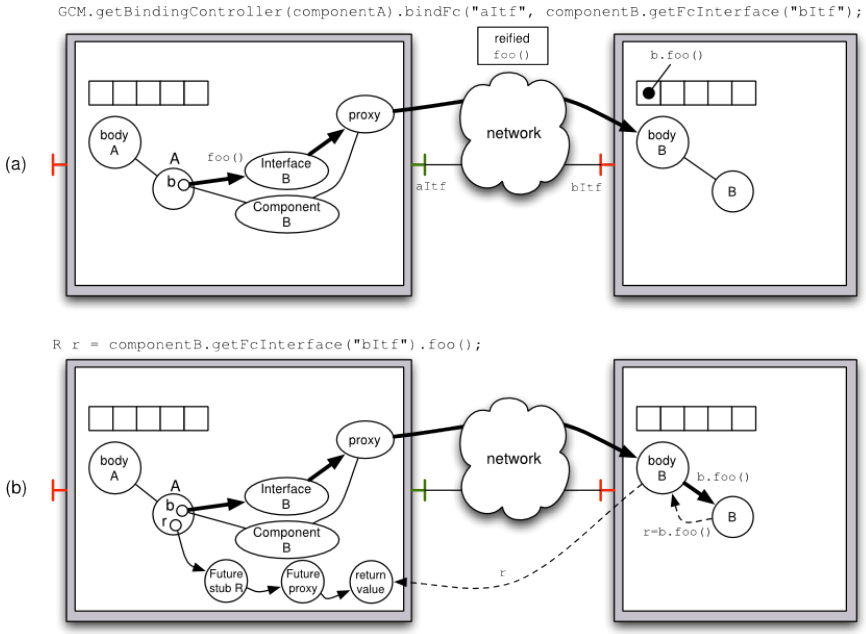


FIG. 6.10 – Séquence d'un appel asynchrone dans les composants GCM/ProActive

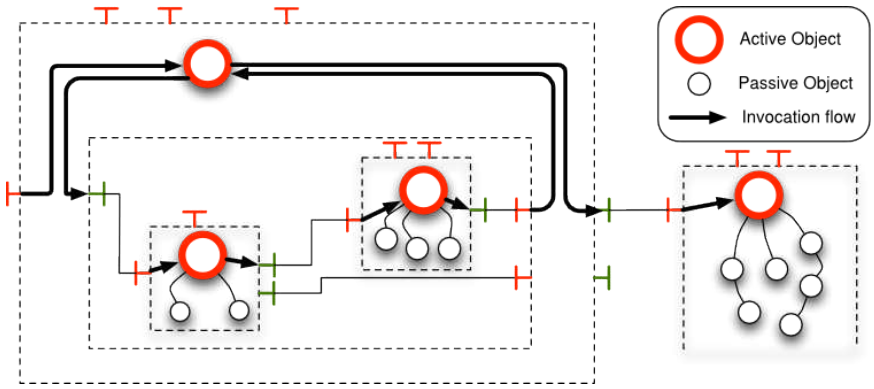


FIG. 6.11 – Le flux d'information dans un composant GCM/ProActive

Nom de la Notification	Évènement correspondant
requestSent	Le composant envoie une requête
requestReceived	Le composant reçoit une requête
servingStarted	Le composant commence à traiter une requête
replySent	Le composant envoie une réponse
replyReceived	Le composant recçoit une réponse
voidRequestServed	Le composant termine de traiter une requête qui ne retourne rien
waitForRequest	Le composant attend de nouvelles requêtes
receivedFutureResult	Le composant reçoit une mise-à-jour pour une valeur dans un Future
waitByNecessity	La requête a été bloquée, attendant la mise-à-jour du Future

TAB. 6.3 – Les notifications générées par GCM/ProActive

basée sur ProActive.

L'information concernant les détails du déploiement peut être obtenue aussi au travers des *MBeans.* Les supports d'exécution ProActive (ProActive Runtimes) et les nœuds sont respectivement instrumentés via un *ProActiveRuntimeWrapperMBean* et un *NodeMBean.* Le premier permet de collecter l'information concernant la machine virtuelle qui héberge l'application ProActive, comme la quantité de mémoire, le nombre de *threads*, le nombre de *bodies*, ou la charge CPU. Le second expose des informations concernant le nœud concret qui héberge l'application, telles que la liste des Objets Actifs ou le nom des Nœud Virtuels.

6.1.5.4 Marquage des messages dans GCM/ProActive

ProActive fournit une interface de programmation qui permet de marquer les messages, en attachant des informations personnalisées, que l'on appelle *tag,* aux messages envoyés entre Objets Actifs. Les *tags* ajoutés à une requête par un Objet Actif peuvent être retrouvés et lus par un autre Objet Actif. L'interface de programmation permet d'identifier individuellement les différents *tags.* De ce fait, un nombre arbitraire de *tags* peut être attaché sans pour autant interférer. L'interface de programmation permet de définir une tâche qui peut être appliquée à un *tag* avant sa propagation, selon un traitement spécifique requis par l'application.

Un exemple très simple de cette fonctionnalité est d'implémenter un compteur comme un *tag* dont la tâche est d'incrémenter une valeur avant chaque propagation. Ce *tag*, indique lors de sa lecture sur chaque Objet Actif cible, la profondeur de l'invocation.

Un autre exemple du marquage des messages dans ProActive est l'identification du flot distribué dans l'invocation d'une opération. Un flot distribué permet de trouver toutes les communications qui sont en relation causale dans une application distribuée. Dans ce cas la tâche associée au *tag* serait de répliquer une valeur d'identification à chaque invocation. Les requêtes qui partagent l'identificateur appartiennent au même flot distribué.

6.1.5.5 Fonctionnalités additionnelles

Emballage de code patrimonial Les composants GCM/ProActive peuvent être utilisés pour englober du code patrimonial existant, permettant ainsi de gérer une application non componentisée dans un environnement de composants. Cette méthode d'emballage a été

développée avec le but de déployer et d'exécuter automatiquement des applications natives écrite dans des langages tels que C, C++, Fortran/MPI, sur des environnements de Grille de façon transparente.

Une fois l'application patrimoniale englobée dans un composant GCM/ProActive, elle peut être déployée et gérée comme les autres composants GCM/ProActive. En utilisant une bibliothèque C/JNI qui inclue des opérations de gestion (init et terminate), des primitives de communication (synchrone/asynchrone send et receive), il est possible de communiquer avec des processus natifs depuis des Objets Actifs.

GCM/SCA La membrane des composants GCM permet d'adapter les caractéristiques non-fonctionnelles des composants. Un exemple a été développé pour l'adaptation de GCM comme une plateforme d'exécution répondant à la spécification SCA. Cette plateforme, appelée SCA/GCM comprend deux contrôleurs additionnels, à savoir le contrôleur *SCAIntentController* et le contrôleur *SCAPropertyController*. Le *SCAIntentController* permet d'attacher des gestionnaires d'*Intents* à une interface serveur ou une interface cliente donnée, avec l'objectif que toutes les invocations entrantes et/ou sortantes soient interceptées par le gestionnaire d'*Intents* avant son traitement normal. Cela permet d'associer les objets implémentant les *Intents SCA* aux composants SCA/GCM. Le *SCAPropertyController* permet d'accéder et de modifier les propriétés sur les composants SCA/GCM. Par ailleurs, GCM/ProActive permet d'exposer les interfaces des composants en tant que Service Web [DGP+07].

Dans les sections précédentes, nous venons de présenter GCM/ProActive qui va nous servir de support d'exécution pour les orchestrations décentralisées. La section suivante présente la mise en œuvre de ce support d'exécution.

6.2 Implémentation de notre solution

Nous avons présenté dans la section précédente le contexte technologique qui supporte notre solution. Dans cette section, après avoir justifié les choix technologiques que nous avons fait pour mette en œuvre le concept du fragment, nous détaillons l'implémentation de notre solution basée sur GCM/ProActive. Nous rappelons que tout au cours de la description de notre solution dans le chapitre 5, nous avions pour objectif de remplir les besoins exprimés dans le chapitre 4, dans le but d'obtenir une solution d'orchestration de services, répartie et flexible, en adoptant une approche à composants.

6.2.1 Choix techniques

Nous justifions maintenant les choix techniques que nous avons faits en utilisant GCM/-ProActive comme support de notre solution, en nous appuyant sur les besoins exprimés dans le chapitre 4

Nous avons choisi d'utiliser le modèle à composants distribué GCM qui nous permet de remplir les objectifs fixés en nous fournissant les fonctionnalités suivantes :

- **Distribution :** par nature, un composant GCM/ProActive est distribué
- **Déploiement :** GCM/ProActive fournit une infrastructure permettant de déployer facilement les composants sur un ensemble de nœuds d'exécution
- **Dynamicité :** La reconfiguration du support d'exécution de nos orchestrations, disponible du fait de l'utilisation de GCM, permet de donner un support approprié pour

insérer et supprimer des éléments dans une application. Nous pouvons donc utiliser GCM pour gérer la flexibilité de nos orchestrations, en terme d'ajout et de disparition des services.
- **Séparation des préoccupations :** GCM/ProActive permet de séparer la logique fonctionnelle de la logique de gestion du composant dans la membrane, Cette membrane étant modulaire, il est ainsi possible de la composer pour fournir un contrôle flexible du composant.

Les fonctionnalités offertes par GCM/ProActive fournissent des solutions techniques permettant d'atteindre les objectifs que nous avons posés dans le chapitre 4, démontrant la validité de la plateforme comme support d'implémentation de notre solution.

Par ailleurs, nous avons fait le choix d'utiliser WS-BPEL pour définir les processus métiers, essentiellement parce que WS-BPEL est le langage le plus courant pour définir les orchestrations de services et qu'il existe de nombreux moteurs d'orchestration supportant le standard WS-BPEL. Cependant ce n'est en rien une limitation technique, il est tout à fait possible de changer de langage de définition de workflow au besoin.

Dans la suite de la section, nous détaillons l'implémentation d'un fragment d'abord unitaire, puis composite. Après avoir détaillé et avant de nous intéresser à l'exécution de ce fragment, nous explicitons la manière dont est projetée une orchestration distribuée, définie par un ensemble de processus WS-BPEL et des liens de coopération, sur le fragment implémenté en GCM.

La solution que nous proposons dans la suite de cette section comprend :
- Une représentation d'un fragment unitaire dont la spécification est décrite dans le chapitre 5.
- Une représentation d'un fragment composite, représentant une orchestration distribuée
- Un mécanisme de projection d'orchestration distribuée sur un fragment composite.

6.2.2 Implémentation d'un Fragment Unitaire

Nous fournissons une représentation du fragment que nous avons introduit dans le Chapitre 5, comme un composants GCM, équipé d'un ensemble de composants non-fonctionnels (NF) dans sa membrane, ce qui lui confère le comportement attendu.

6.2.2.1 Structure du Fragment Unitaire

Dans la Section 6.1.3, nous avons mentionné que dans l'objectif de gérer les aspects non-fonctionnels du composant, le modèle de GCM permettait l'insertion dynamique des composants non-fonctionnels (NF) dans la membrane d'un composant GCM de base, pouvant être reliés aux interfaces non-fonctionnelles du composant. Ainsi, le composant GCM qui implémente un fragment est pourvu d'un ensemble de composants Non-Fonctionnels (NF) qui permettent de le contrôler.

Nous faisons la distinction entre contenu non-fonctionnel et contenu fonctionnel. La Figure 6.12 montre la structure générale d'un composant GCM qui représente un fragment unitaire appelé `Fragment Unitaire`.

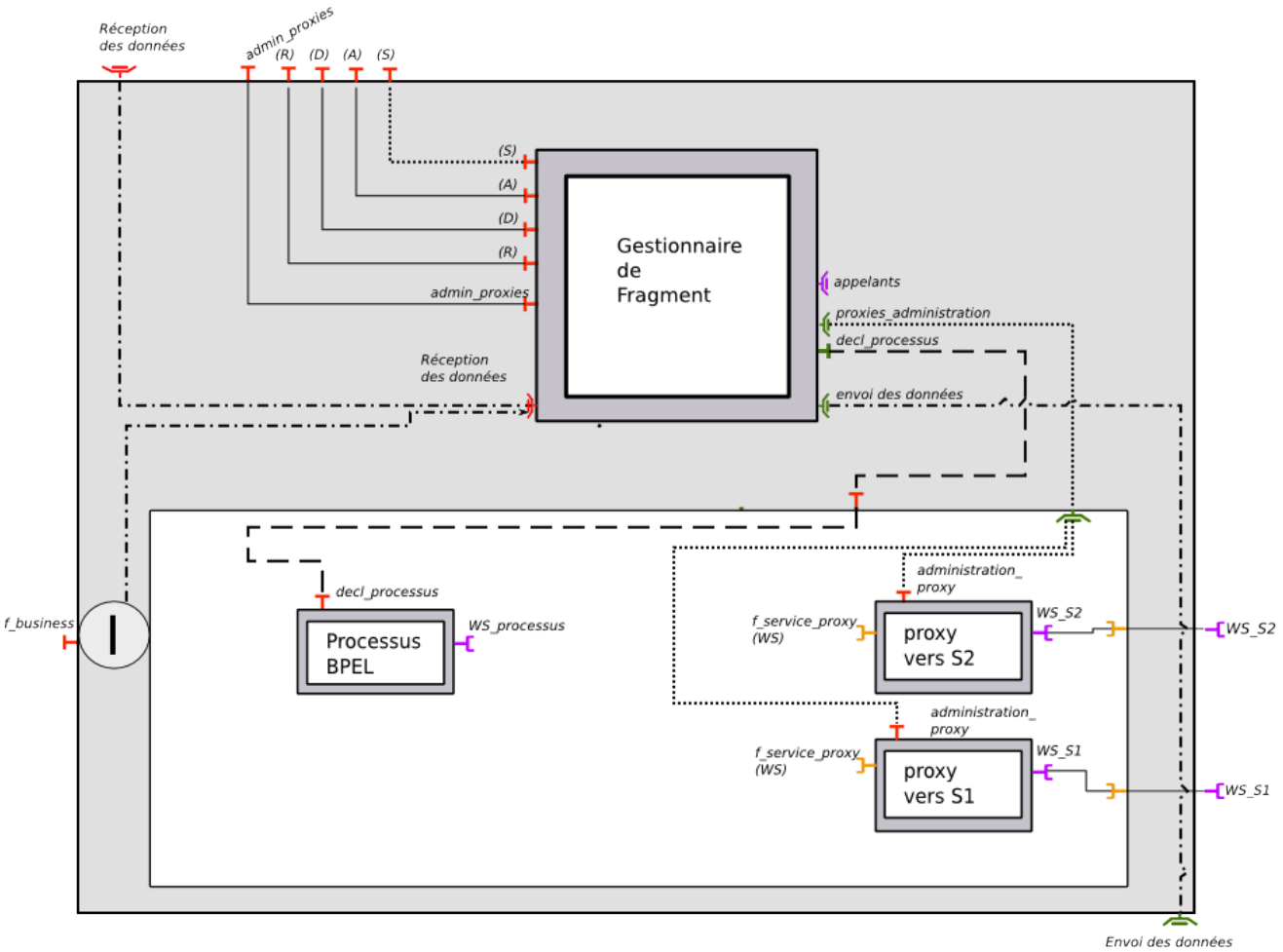
Le contenu fonctionnel comprend tout ce est relatif au processus métier et à son environnement d'exécution, c'est-à-dire qu'il est composé de:
- Un composant permettant d'englober et d'**accéder au processus**, Le processus est défini en WS-BPEL. Il comporte notamment, dans le but de recevoir les données applicatives, une activité `receive` pour la réception de données applicatives et une activité

`reply` pour renvoyer les données applicatives (cf la description détaillée des activités WS-BPEL dans la Section 6.1.1).

– Un ensemble de composants ***proxies* qui permettent d'intercepter les services impliqués dans le processus** ; il y a ainsi autant de proxies dans le contenu fonctionnel que de services impliqués dans le processus métier. Les interfaces fonctionnelles de ces composants proxies sont exposées en tant que Services Web, permettant au moteur d'orchestration WS-BPEL d'y accéder de l'extérieur. Chaque appel du moteur d'orchestration sur le service passe donc par ces *proxies*, permettant de superviser les appels, mais aussi de changer l'adresse du service représenté. Dans la Figure 6.12, nous montrons un exemple de fragment représentant un processus impliquant deux services, S1 et S2 ; la partie fonctionnelle du fragment comprend donc deux composants *proxies* qui interceptent les appels vers les services S1 et S2.

Structure du Gestionnaire de Fragment unitaire Le *Gestionnaire de fragment*, dont le détail est donné dans la Figure 6.13, est implémenté comme un composant composite et est inséré dans la membrane du composant, lui fournissant ainsi les fonctionnalités que nous avons décrites dans le Chapitre 5, notamment les mécanismes de reconfiguration dynamique et d'administration. Les liaisons entre les différentes fonctionnalités reprennent celles montrées dans la Figure 5.8. Cet exemple nous montre le gestionnaire de fragment composé des quatre sous-composants (`Execution`, `Déploiement`, `Reconfiguration` et `Administration/Supervision`) permettant le contrôle d'un fragment unitaire. Le fragment offre cinq interfaces de contrôle supplémentaires par rapport aux interfaces de contrôle basique (telles que celle de la gestion du cycle de vie du composant (`LifeCycle`), ou encore la gestion du Nom (`Name`)), qui sont `Réception des données`, `Supervision (S)`, `Administration (A)`, `Déploiement (D)` et `Reconfiguration (R)` ; elles permettent d'accéder aux fonctionnalités qu'offre un fragment unitaire.

Les dépendances temporelles sont matérialisées par des interfaces GCM NF : le Fragment exporte une interface NF cliente *multicast* qui permet de passer les données applicatives à un ou des éventuels fragments suivants. Symétriquement, afin de recevoir les données applicatives, le fragment propose une interface serveur *gathercast*, permettant de recevoir les données applicatives. L'utilisation de ce type d'interface nous permet, comme explicité dans la Section 6.2.3, d'implémenter les patrons d'invocation parallèle.



(R) Interface de reconfiguration
(D) Interface de déploiement
(A) Interface d'administration
(S) Interface de supervision

Flux de données
Supervision
Appels fonctionnels concernant l'exécution du processus métier

Interface vers un Service Web
Interface exposée en Service Web
Interface cliente GCM
Interface serveur GCM
Interface multicast GCM
Interface gathercast GCM

Figure 6.12: L'implémentation du fragment en GCM : Le Gestionnaire de fragment est inséré dans la membrane du composant GCM

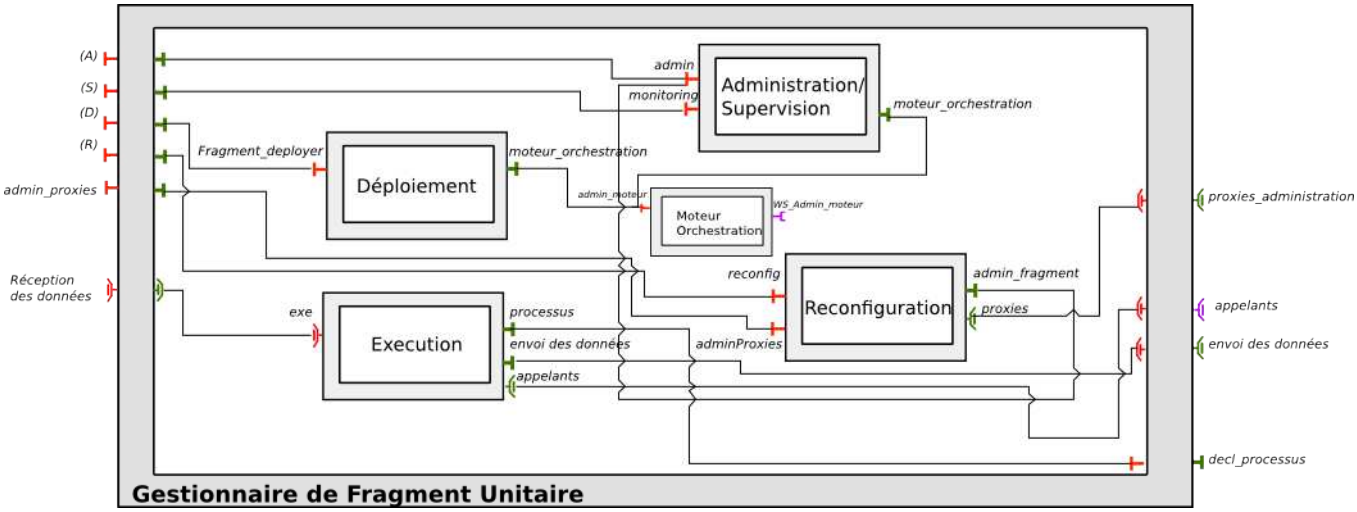


FIG. 6.13 – Le composant composite Non-Fonctionnel correspondant au Gestionnaire de Fragment Unitaire

Le moteur d'orchestration Afin d'exécuter le processus associé au fragment, il est nécessaire de disposer d'un moteur d'orchestration. Le composant `Moteur_orchestration` est illustré sur la Figure 6.14 permettant de représenter la liaison avec un moteur d'orchestration déployé. Le moteur est englobé dans un composant GCM et exporte les interfaces nécessaires à sa gestion (démarrage du moteur, redémarrage, administration ...) montrées dans le Listing 5.2. Ainsi, il est possible pour un composant GCM, comme par exemple le *composant de déploiement*, de contrôler le moteur. Nous avons choisi d'utiliser pour notre implémentation ActiveBPEL fourni par ActiveVOS[7]. Les fonctionnalités de contrôle de ce moteur sont accessibles via un Service Web. Le composant implémtente l'API du Listing 5.2, chaque méthode de cette API permettant de relayer, sous forme d'appel à un Service Web, la fonctionnalité équivalente offerte pour l'administration de ce moteur. Nous fournissons donc pour ce moteur en particulier un composant dont l'implémentation utilise ce Service Web. La liaison vers ce service est réalisée via l'interface `WS_Admin_moteur`. Pour chaque moteur qui serait susceptible d'être utilisé par notre solution, une implémentation spécifique pour l'accès au contrôle du moteur serait donnée. D'autres expérimentations ont été menées avec d'autres moteurs, comme par exemple, Apache ODE[8] ou Timed Automata [BLH+10]

FIG. 6.14 – Le composant "englobant" le moteur d'orchestration

[7]http ://www.activevos.org
[8]http ://ode.apache.org/

Composants *proxies* et reconfiguration dynamique Pour chaque service impliqué dans le processus, un composant *proxy* est généré, ajouté à la partie fonctionnelle du fragment et relié aux composants de reconfiguration dynamique et de supervision, ce qui permet non seulement au composant de reconfiguration de pouvoir changer le point d'accès du service au moment de l'exécution, mais aussi d'obtenir des données de supervision. Ce *proxy* est accessible via une interface qui est exposée en tant que Service Web. C'est ce Service Web qui va être appelé par le moteur d'orchestration, au lieu du service concret. Grâce à ce *proxy*, il est possible de ne réaliser la liaison au service concret qu'au moment de l'exécution, dans un but d'adaptation ou de respect de qualité de service. Si, au moment de son appel, le *proxy* référence un service qui n'est pas disponible, celui-ci génère un évènement auquel un éventuel outil, extérieur au fragment, qui y aura souscrit pourra, via le gestionnaire de reconfiguration, changer le point d'accès du service. Cette situation illustre de façon simple le mécanisme de reconfiguration d'un *proxy*. Grâce à l'utilisation d'un modèle à composants, il est possible d'enrichir les schémas de reconfiguration, par exemple en changeant l'implémentation du *proxy* par celle d'une composition de service sémantiquement équivalent, calculée par un outil externe [BLH+10]. Par ailleurs, le *proxy* permet d'obtenir des métriques concernant l'utilisation du service qu'il référence, telles que le temps de réponse de ce dernier, destinées elles aussi à un outil externe d'analyse de la qualité de service, et qui pourra lui aussi initier une reconfiguration. La structure du *proxy* est montrée sur la Figure 6.15, qui illustre un *proxy* vers un Service Web dont le nom est S1.

Dans la suite de cette section, nous allons détailler l'implémentation des différentes fonctionnalités NF offertes par le fragment. Nous commençons par détailler le déploiement du fragment.

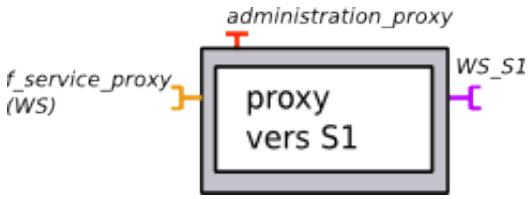


FIG. 6.15 – Le composant proxy

6.2.2.2 Déploiement et génération du fragment

En ce qui concerne le déploiement du fragment, nous utilisons le mécanisme de déploiement de GCM/ProActive, qui permet en fournissant un *descripteur de déploiement* d'installer le support d'exécution du fragment sur un nœud d'exécution donné. Cette étape déploie le composant et les composants NF (le composant composite gestionnaire de fragment) dans sa membrane. À ce moment, le fragment n'est pas encore relié à un quelconque moteur d'orchestration, et il est constitué de l'ensemble minimal de ses composants, c'est-à-dire d'un gestionnaire de fragment, et d'un contenu fonctionnel vide. Ce composant minimal propose une interface métier qui est celle proposée par le processus et qui lui est propre. Ainsi, avant de générer le fragment, cette interface est déduite en analysant la définition du processus WS-BPEL : depuis la description WSDL du service proposé par le processus, il est possible de générer l'interface Java correspondante, grâce à des outils tels que WSDL2Java, proposé par le canevas Apache Axis 2.[9]

[9] http ://axis.apache.org/axis2/java/core/

FIG. 6.16 – Le composant GCM responsable du déploiement du Fragment

Une fois le composant GCM installé, il est démarré. Nous pouvons alors le configurer en suivant les étapes suivantes, grâce au composant `Déploiement` montré dans la Figure 6.16 et qui implémente l'interface du Listing 5.2 :

1. Pour chaque lien partenaire présent dans la description WS-BPEL, nous générons un composant *proxy* qui est inséré dans la partie fonctionnelle du fragment. L'interface du *proxy* est exposée en Service Web. L'adresse du partenaire, présente dans la partie concrète de la définition du processus, est remplacée par celle du Service Web proposé par le *proxy*, de facon à ce que le moteur WS-BPEL puisse appeler le *proxy* à la place du service.
2. Les liaisons depuis l'interface *multicast* `proxies_administration` du gestionnaire de fragment NF vers les composants *proxies* sont établies.
3. Au besoin, le moteur d'orchestration est installé et démarré à distance (de façon programmatique), Il est ensuite englobé dans le composant Moteur (montré dans la Figure 6.14) qui sera relié au gestionnaire de fragment. Il se peut qu'il soit déjà installé et démarré au préalable sur une localisation, auquel cas il faut spécifier au composant représentant le moteur l'adresse à laquelle le Service Web de gestion est disponible, et le lier au gestionnaire de fragment. Comme spécifié précédemment, nous avons fait le choix d'utiliser dans notre implémentation le moteur ActiveBPEL qui permet d'accéder à ses fonctionnalités via des Services Web. La liaison à ce moteur consiste à installer le composant spécifique à ce moteur.
4. La définition du processus (dont la partie contenant les adresses des Services Web a été modifiée dans le but d'appeler les *proxies*) est déployée sur le moteur d'orchestration. Le processus est alors déployé sur le moteur.

6.2.2.3 Exécution d'un Fragment unitaire

Un fragment unitaire peut avoir lieu d'être dans deux cas. Le premier cas est celui où il est un fragment indépendant de toute composition et propose une interface métier. Ainsi, le processus WS-BPEL représenté par un fragment bénéficie des propriétés de reconfiguration dynamique proposées par notre modèle. Le second cas est celui où le fragment unitaire fait partie d'un fragment composite (que nous détaillerons dans la suite) et n'est pas supposé recevoir d'invocation au sens métier. Dans ce cas, il reçoit des données par le biais de son interface de réception des données.

L'exécution du fragment peut donc être déclenchée de deux manières. Soit le fragment reçoit des données applicatives via son interface de réception des données, soit il reçoit un appel fonctionnel (l'appel métier).

Dans le premier cas, les données applicatives sont reçues par l'interface de réception des données et elles sont transmises au processus. Dans le deuxième cas, c'est l'interface métier du fragment, exposée comme un Service Web, qui est invoquée par un appelant extérieur. L'appel est intercepté par un intercepteur GCM (I sur la Figure 6.12) qui transmet

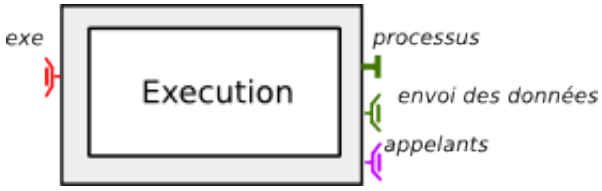


FIG. 6.17 – Le composant d'exécution NF d'un fragment unitaire

la requête, en réifiant l'invocation et en extrayant les paramètres et leurs valeurs, au gestionnaire de fragment (*via* son interface de réception des données), qui sera en charge de transférer l'appel au processus. Un paramètre important de l'invocation de l'interface métier est celui de l'adresse de retour de résultat de l'exécution du processus, qui sert d'identifiant pour un appelant. L'exécution du fragment fonctionne en effet en mode asynchrone et permet lorsque cette exécution est terminée, de renvoyer le résultat à l'appelant, *via* un lien entre le service situé à l'adresse de retour et l'interface cliente *multicast* `appelants`, montrée dans la Figure 6.13 et permettant de sélectionner un élément en particulier. Ainsi, lors d'un appel, un lien Service Web est ajouté à l'interface cliente *multicast* `appelants`. Le composant Execution gère l'exécution des instances du fragment en maintenant une correspondance entre les appelants et les instances de processus. Une fois que le processus correspondant au fragment est exécuté, le composant `Execution` recherche pour l'identifiant correspondant l'adresse de l'appelant pour lui retourner le résultat. Une fois l'exécution du processus terminée, le composant `Execution` renvoie le résultat à l'appelant *via* le sélection de la liaison correspondante dans l'ensemble des liaisons de l'interface cliente *multicast*, qui se transforme en un appel Service Web.

Le composant implémentant l'interface fournie dans le Listing 5.1 est montré dans la Figure 6.17. Remarquons ici que ce moteur possède trois dépendances nécessaires à l'exécution du fragment. Outre les liaisons multiples vers les appelants et la liaison vers le fragment suivant utiles pour transférer les données applicatives, le composant `Execution` est lié au composant `Processus_BPEL`, situé dans la partie fonctionnelle et illustré dans la Figure 6.18. C'est par le biais de cette liaison que le processus va être déclenché. Le composant `Processus_BPEL` permet de transférer l'appel au processus métier par le biais d'une interface cliente Service Web. Pour rappel, le processus effectif WS-BPEL expose un Service Web qui permet de déclencher son activité `receive`. Une fois l'exécution du processus WS-BPEL terminée, le résultat de l'exécution du processus est renvoyé par le biais d'une activité `reply`.

Les étapes de l'exécution d'un fragment sont les suivantes :

1. Dans le cas d'un fragment unitaire et non partie d'un fragment composite, l'invocation de l'interface métier permet de recevoir les données applicatives ainsi que l'adresse de retour du résultat. Dans le cas d'un fragment unitaire faisant partie d'un fragment composite, les données sont reçues par le biais de l'interface de réception des données.
2. Création d'une entrée pour une nouvelle instance par le composant d'exécution et enregistrement dans la table de correspondance d'une représentation de l'instance du processus.
3. L'appel effectif au processus est réalisé par le biais de l'interface NF cliente (générique) `declenchement_processus`, la représentation de l'instance reste bloquée, en attente du résultat de l'exécution du processus sur le moteur d'orchestration. Le détail de cette interface est donné dans le Listing 6.3.

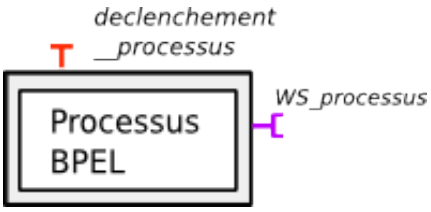


FIG. 6.18 – Le composant permettant d'accéder au processus métier WS-BPEL

Listing 6.3 – L´ interface du composant Processus permettant le declenchement du workflow

```
public interface ControleProcessus {

/** déclenche une nouvelle instance du processus WS-BPEL en lui envoyant
    les paramètres contenus dans la table*/
public void startProcess (HashMap <String, Object> parametres);

}
```

4. L'exécution du processus WS-BPEL est déclenchée par le biais de son activité `receive`. Le moteur d'orchestration invoque les services exposés par les *proxies*, qui à leur tour invoquent le service réel.
5. Le processus WS-BPEL renvoie le résultat, par le biais d'une activité `reply`, à la représentation de l'instance, qui retournera le résultat à l'appelant via l'interface cliente `appelants` dans le cas où le fragment est isolé, ou enverra les données applicatives reçues au fragment suivant présent dans la composition.

6.2.2.4 Supervision et Administration du fragment

Le composant Administration/Supervision montré dans la Figure 6.19 propose une implémentation du Listing 5.3 et du Listing 5.4. Cette implémentation permet d'obtenir des informations concernant l'exécution du fragment et du processus WS-BPEL qu'il englobe, notamment en générant, lorsque cela est pertinent, les évènements que nous avons listés dans le Tableau 5.1. Par ailleurs, les fonctionnalités d'administration du processus (arrêt, reprise, etc ...) sont fournies par le composant `Moteur` relayant les actions par le biais du Service Web de gestion du moteur WS-BPEL.

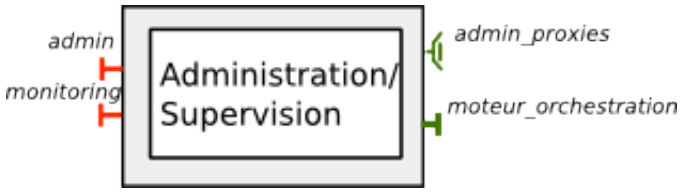


FIG. 6.19 – Le composant d'administration et de supervision

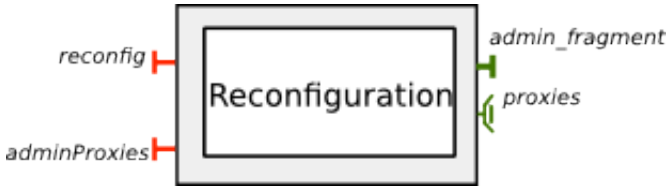


FIG. 6.20 – Le composant de reconfiguration dynamique

6.2.2.5 Reconfiguration Dynamique

Le composant de reconfiguration dynamique, qui permet de changer l'adresse référencée par un *proxy* est montré dans la Figure 6.20. En plus de fournir l'interface proposée dans le Listing 5.5, elle propose une interface de gestion des *proxies*, `admin_proxies`, qui permet de connaître notamment la liste des *proxies* contenus par le fragment, d'en ajouter ou d'en supprimer. Cette interface est exportée au niveau du gestionnaire de fragment, puis au niveau du fragment afin de contrôler hiérarchiquement les *proxies.* Dans le but de contrôler l'ensemble des *proxies* du fragment, elle possède une interface cliente *multicast.* De plus, afin de pouvoir contrôler l'exécution des instances du processus lors d'une reconfiguration, elle possède également une dépendance vers le composant d'administration du fragment, notamment pour stopper ou redémarrer les instances du processus.

Nous venons de décrire la mise en œuvre d'un fragment unitaire grâce au modèle de composant distribué GCM. À partir de ces fragments unitaires, nous pouvons construire une orchestration décentralisée dont nous décrivons la structure dans la suite.

6.2.3 Projection d'une orchestration distribuée sur un composant composite

Dans cette section, nous décrivons comment est réalisée la construction, à partir d'une définition d'une orchestration décentralisée, d'un fragment composite grâce à un composant composite GCM. Nous commençons par détailler la définition d'une orchestration décentralisée avec WS-BPEL. Nous détaillons ensuite, avant d'aborder le mécanisme de projection d'une orchestration décentralisée sur un composant distribué GCM, les éléments de notre solution que nous avons décrits dans le chapitre 5.

6.2.3.1 Définition d'une orchestration décentralisée avec WS-BPEL

Nous proposons ici une possible manière de définir une orchestration décentralisée composée de processus définis en WS-BPEL :

Une orchestration WS-BPEL décentralisée comporte :

- un ensemble de définitions de processus WS-BPEL Ces processus ont été définis de manière à ce qu'ils reçoivent et envoient les données applicatives par le biais d'activités `receive` ou `reply`.
- une définition de l'orchestration globale qui permet de coordonner la collaboration entre processus.

Une telle définition peut être produite par un outil d'analyse et de calcul de décentralisation, comme celui présenté dans [Yil08] ou naturellement en suivant une approche ascendante (bottom-up) et en composant des processus.

Un exemple de définition, d'orchestration décentralisée est donné dans le Listing 6.4. Cette orchestration décentralisée, nommée `Composite`, est composée de trois sous-processus T-1, T, T+1. La définition de l'orchestration consiste à définir la coopération entre sous-processus et à expliciter les paramètres applicatifs qui vont transiter entre-eux. Pour chaque processus, nous identifions ainsi son processus coopérant suivant ainsi que les données qu'il lui envoie. Les processus possèdent un type qui permet de différencier les processus simples, exécutés en séquence, et les processus qui permettent d'implémenter les patrons d'exécution parallèle et de jointure. Ainsi un processus peut être du type `single`, `split` ou `join`. L'utilisation des types `split` et `join` est montrée dans le Listing 6.5. Nous remarquons dans cette définition l'absence de spécification de données applicatives à envoyer au fragment initial. Ce fragment étant particulier, dans le sens où c'est lui qui est relié à l'interface cliente d'envoi des données du gestionnaire de fragment composite, il reçoit les données applicatives reçues au niveau du fragment composite.

Listing 6.4 – Exemple de définition d'une orchestration décentralisée en WS-BPEL

```
<orchestration name="Composite">

<!-- Définition du processus initial de l'orchestration décentralisée -->
        <initial>
                <processus name="T-1" definition="t-1.bpel" type="single
                    ">
                        <data paramName="a" type="int" />
                        <data paramName="b" type="int" />
                        <next> T </next>
                </processus>
        </initial>

<!-- Définition des sous-processus intermédiaires -->
        <intermediary>
                <processus name="T" definition="t.bpel" type="single">
                        <data paramName="a" type="int" />
                        <data paramName="b" type="int" />
                        <data paramName="c" type="int" />
                        <next> T+1 </next>
                </processus>
        </intermediary>

<!-- Définition du processus final de l'orchestration décentralisée -->
        <final>
                <processus name="T+1" definition="t+1.bpel" type="single"
                    >
                        <data paramName="result" type="int"/>
                </processus>
        </final>
</orchestration>
```

Comme détaillé dans la Section 5.3.1, l'orchestration décentralisée comprend un *fragment initial* et un *fragment final*. Ces deux types de fragments sont identifiés dans la définition de l'orchestration décentralisée.

6.2.3.2 Le gestionnaire de fragment composite

La projection d'une orchestration décentralisée consiste à fournir une implémentation du gestionnaire de fragment que nous avons décrit dans la Section 5.3. Le gestionnaire d'un fragment composite a pour particularité de contrôler de façon hiérarchique les sous-fragments. Le contrôle est ainsi propagé aux sous-fragments *via* des interfaces internes clientes *multicast*.

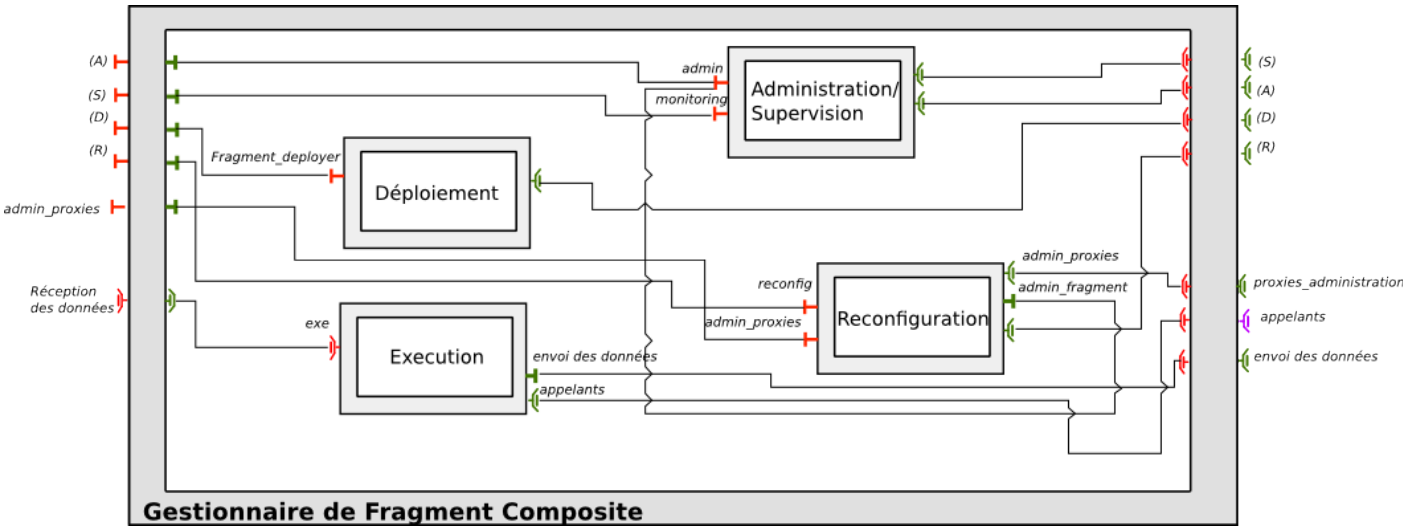


FIG. 6.21 – Utilisation des interfaces GCM *multicast* pour l'implémentation du gestionnaire d'un fragment composite.

La Figure 6.21 montre la structure du gestionnaire de fragment composite. Nous pouvons remarquer que le gestionnaire de fragment composite doit être relié à chaque fragment faisant partie du composite via des interfaces multiples (`(A)`, `(S)`, `(D)` et `(R)`). Ces interfaces de contrôle vont permettre au fragment composite de contrôler en une seule opération les sous-fragments qui le composent. Ainsi les appels aux interfaces `(A)`, `(S)`, `(D)` et `(R)` permettent respectivement d'administrer les sous-fragments, de superviser les sous-fragments, de déployer les sous-fragments et de reconfigurer les sous-fragments. Par exemple, on peut, depuis le fragment composite arrêter les moteurs associés aux sous-fragments appartenant au fragment composite : l'appel à l'interface de déploiement déclenche alors l'appel de toutes les interfaces de déploiement des sous-fragments.

6.2.3.3 Projection d'une orchestration décentralisée sur un composant GCM

Nous rappelons ici que la définition d'une orchestration distribuée comprend un ensemble de descriptions de sous-processus, que nous avons implémentés en WS-BPEL, et d'une description de coordination entre sous-processus ainsi que d'un ensemble de localisations où les fragments vont pouvoir s'exécuter.

Depuis la définition d'une orchestration décentralisée, nous procédons alors comme suivi afin de réaliser la projection sur un composant composite GCM :

1. Le composant GCM représentant un fragment composite est généré, comprenant son gestionnaire de fragment que nous avons détaillé précédemment dans la Section 6.2.3.2. Tout comme pour le gestionnaire de fragment unitaire, l'interface métier du fragment est déduite de la description de l'orchestration. Pour cela, nous analysons la définition du fragment initial car c'est lui qui offre cette fonctionnalité particulière.
2. Pour chaque sous-processus coopérant, nous générons un fragment lui correspondant, comme indiqué dans la Section 6.2.2. En particulier, nous générons le fragment initial et le fragment final, tous deux reliés au gestionnaire de fragment dans la membrane respectivement par le biais de l'interface cliente *multicast* d'envoi de données et par le biais de l'interface serveur *gathercast* de réception de données du fragment composite.
3. Les interfaces multiples de contrôle du gestionnaire de fragment sont reliées à chacune des interfaces de contrôle des sous-fragments.
4. Les interfaces temporelles qui permettent de transmettre les données applicatives sont reliées entre-elles. Dans la description de notre modèle, nous avons mentionné le besoin d'interfaces multiples afin de gérer la réception et l'envoi de données en parallèle. Nous pouvons implémenter ce comportement grâce aux interfaces *multicast* proposées par GCM. Ainsi l'envoi en parallèle de données se réalisera grâce à une interface *multicast* et la réception en parallèle de données se réalisera grâce à une interface *gathercast*. Dans ce dernier cas, l'exécution du fragment concerné par la réception parallèle de données se déclenchera une fois toutes les données reçues. Un exemple d'exécution parallèle de fragments grâce aux interfaces *multicast* et *gathercast* est montré dans la Figure 6.22[10] et dans le Listing 6.5. Dans cet exemple, le comportement du patron AND-SPLIT est réalisé grâce à une interface GCM *multicast,* exposée par le composant `T-1`, qui va permettre de diffuser les données applicatives vers les fragments T1 et T2. Une fois exécutés, T1 et T2 vont envoyer leurs données applicatives vers le fragment T+1 qui réalise une jointure (patron AND-JOIN) grâce à une interface GCM *gathercast.* Dans le cas où l'interface *multicast* (respectivement l'interface *gathercast)* ne comporte qu'une seule liaison, cela permet d'exécuter les sous-fragments exécutés en séquence.

6.2.3.4 Exécution du fragment composite

Comme illustré dans la Figure 6.21, le gestionnaire de fragment composite ne nécessite pas de lien vers un moteur d'exécution WS-BPEL. Cependant, il contient un composant `Execution`, qui permet de gérer les instances propres à chaque appelant. Ainsi, lorsque l'exécution de l'orchestration globale est terminée, le fragment final de l'orchestration décentralisée envoie les données applicatives au gestionnaire de fragment, qui va renvoyer (de la même façon que pour un fragment unitaire) le résultat à l'appelant concerné qui a invoqué le service proposé par le fragment composite et qui propose de recevoir le résultat à une adresse donnée.

La Figure 2.4 illustre une implémentation d'un fragment composite d'après la définition d'orchestration décentralisée donnée dans le Listing 6.4. Les interfaces de contrôle clientes du gestionnaire de ce fragment composite sont de type *multicast* et permettent soit

[10] pour plus de lisibilité nous ne montrons ici que les interfaces nécessaires à l'exécution parallèle

Listing 6.5 – Définition des patrons d'invocation parallèle et de jointure correspondant à la Figure 6.22.

```
...
<processus name="T-1 definition="t-1.bpel" type="split">

        <parallelProcess name="T1" >
                <data paramName="a" type="int" />
        </parallelProcess>
        <parallelProcess name="T2" >
                <data paramName="b" type="string" />
        </parallelProcess>
<processus>

<processus name="T1" definition="t1.bpel" type="single">
        <data paramName="a" type="int" />
        <next> T+1 </next>
</processus>

<processus name="T2" definition="t2.bpel" type="single" >
        <data paramName="b" type="int" />
        <next> T+1 </next>
</processus>

<processus name="T+1" definition="t+1.bpel" type="join" correspondingSplit="T1" >
        <data paramName="c" />
        <next> ... </next>
</processus>
...
```

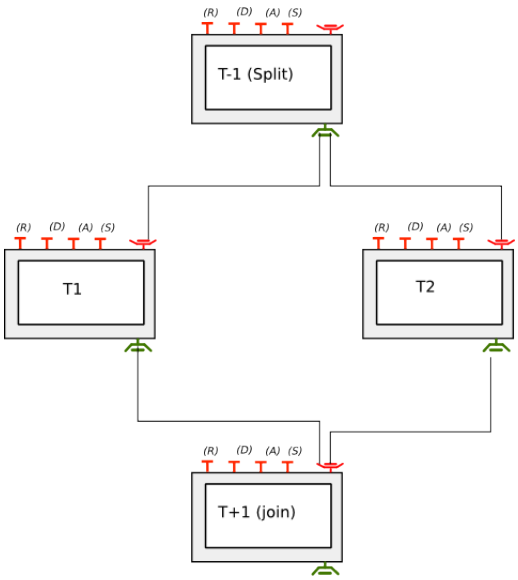


FIG. 6.22 – Interfaces multicast et gathercast pour l'exécution parallèle des fragments

de contrôler les sous-fragments tous ensembles ou un par un, par sélection dynamique d'une des liaisons dans une interface *multicast* (cf les modes d'envoi dans GCM dans la section6.1.4.3). Remarquons que les fragments unitaires composant le fragments composite n'exposent pas d'interface métier car elle n'est pas nécessaire dans ce cas d'utilisation des fragments unitaires.

En ce qui concerne les sous-fragments, ils dépendent chacun d'un ou plusieurs services externes. Les dépendances vers ces services externes ne sont pas exportées au niveau du composite. Nous avons fait ce choix car un sous-fragment peut se trouver dans un domaine d'exécution où un service est accessible. Le composite quant à lui est déployé ailleurs, par exemple, dans le domaine d'exécution de l'appelant. Ce choix est cependant re-négociable en fonction d'une orchestration particulière. Néanmoins, le gestionnaire de fragment composite dispose d'un moyen d'accéder à l'ensemble des services impliqués dans l'orchestration (via son interface *multicast* cliente `proxies_administration`, ce qui permet de traduire l'exportation des interfaces clientes fonctionnelles (vers les services) comme montré dans la Figure 5.6.

6.2.3.5 Reconfiguration dynamique du fragment composite

Afin de reconfigurer dynamiquement la structure du fragment composite, nous fournissons une implémentation de l'interface de reconfiguration dynamique que nous avons détaillée dans la Section 5.3.4 et qui est fournie par le composant `Reconfiguration` montré dans le gestionnaire de fragment composite de la Figure 6.21. Cette implémentation fait usage des propriétés de reconfiguration dynamique proposées par le modèle GCM. Le fragment composite peut ainsi bénéficier des capacités de reconfiguration dynamique au niveau de son contenu (ajout, suppression de sous-fragment) ainsi qu'au niveau des liaisons entre sous-fragments. Par exemple, la méthode `addFragment` proposée dans le Listing 5.3 se traduit concrètement par du code Java utilisant l'API standard de GCM pour ajouter une instance d'un composant fragment (unitaire par exemple) et pour le relier à ses fragments précédents et suivants, ainsi que de relier ses interfaces de contrôle aux interfaces *multicast* du gestionnaire de fragment.

6.3 Évaluation des performances du modèle à composants

Nous nous sommes dans un premier temps intéressés aux performances du modèle à composants GCM/ProActive par rapport à une utilisation directe d'une interface de programmation orientée vers les Services Web, nous avons mesuré les temps de réponse de deux entités essentielles de notre solution, à savoir les temps d'exécution d'un appel vers un service effectué via un *proxy* et un appel vers un processus englobé dans un fragment. Nous avons comparé ces temps aux temps d'exécution d'un appel direct soit vers un service, soit vers le processus.

Afin d'établir au plus juste la performance de ces tests, nous avons utilisé la même interface de programmation orientée Services Web, soit la bibliothèque Axis2 version 1.6.0, . La version de GCM/ProActive est la 5.0.0 , la machine virtuelle java est la JDK OpenJDK 1.6.0_20, 64 bits server, sur une distribution Linux Fedora 14. Les machines sont équipées de processeurs P8600 cadencés à 2,6 Ghz et possédant 4Go de mémoire vive.

Afin d'obtenir une valeur moyenne du surcout, nous avons envoyé un nombre de messages aux services et nous l'avons fait varier. Le service que nous avons utilisé prend en

NB MESSAGES	COMMUNICATIO DIRECTE (S)	COMMUNICATIO AVEC UN PROXY (S)	DIFFÉRENCE (S)	SURCOÛT
10	3,781	4,310	0,529	14%
100	36,284	41,793	55,09	15,18 %
1000	378,932	439972	61,040	16,10 %
2500	1006,785	1171,843	165,058	16,39%
5000	2106,564	2459,561	337352,997	16,7570%
10000	4509,872	5309,083	799,211	17,72%

TAB. 6.5 – Le surcoût induit par notre modèle sur l'exécution d'un processus

paramètres une chaine de caractères et renvoie une chaine de caractères.

NB MESSAGES	COMMUNICATIO DIRECTE (S)	COMMUNICATIO AVEC UN PROXY (S)	DIFFÉRENCE (S)	SURCOÛT
10	4,192	4,652	0,460	10,97 %
100	43,096	47,254	4,158	9,64 %
1000	389,231	424,209	34,978	9 %
2500	967,170	1060,285	93,115	9,62 %
5000	1946,158	2111,804	165,646	8,51 %
10000	4020,0197	3939,71	393,971	9,79 %

TAB. 6.4 – Le surcoût induit par le modèle à composant dans l'appel à un service depuis un *proxy*

Le tableau 6.5 montre les temps obtenus pour chaque exécution selon le nombre de requêtes, ainsi que le surcoût obtenu pour un appel sur un *proxy*. Nous pouvons y constater que le surcoût introduit par un est d'environ 9,5% en moyenne. Ce surcoût est s'explique par les mécanismes inhérents à l'implémentation du composant GCM/ProActive. En effet, afin de pouvoir intercepter les appels de méthodes, ces derniers sont réïfiés, c'est-à-dire que l'appel de méthode est transformé en un objet java, il est ainsi aisé de le manipuler au sein des composants. En plus de la réïfication de l'appel de méthode, une copie profonde des arguments de l'appel est réalisée afin de garantir des propriétés sémantiques du système dont il est inutile de détailler la fonction dans le contexte présent.

Dans le tableau 6.5, nous montrons les temps obtenus pour des appels sur un fragment. Nous pouvons y constater un surcoût moyen de 16 %, en grande majorité dû à l'exécution du gestionnaire de fragment.

6.4 Conclusion

Dans ce chapitre, nous avons décrit les détails de la mise en œuvre de notre solution grâce à l'intergiciel GCM/ProActive.

Cette solution fournit l'infrastructure nécessaire à la création de fragments unitaires et composites, éléments centraux de notre modèle. Nous avons également explicité l'intérêt des mécanismes de *proxy* permettant une reconfiguration dynamique pendant l'exécution d'un processus métier. Ainsi, à partir d'une définition d'une orchestration décentralisée

basée sur un ensemble de définitions WS-BPEL, un composant composite GCM permettant de contrôler cette orchestration est généré. L'implémentation fait usage de la sémantique des applications basées sur les composants GCM/ProActive et supporte le modèle décrit dans le Chapitre 5.

Dans le Chapitre suivant, nous proposons un cas d'utilisation de notre solution basé sur la gestion d'un parc de passerelles OSGi.

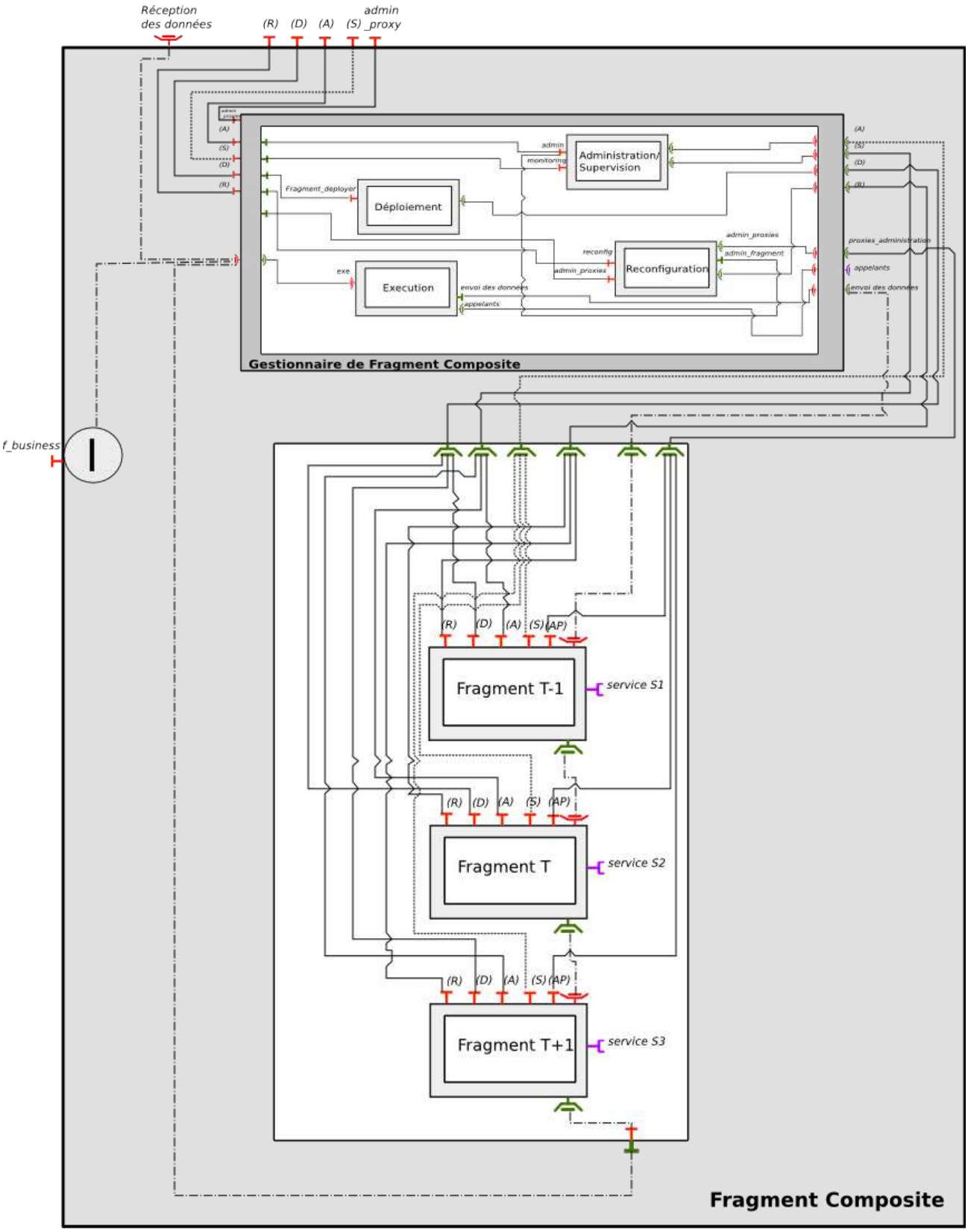


FIG. 6.23 – Un exemple de fragment composite implémenté avec GCM

Lièvres je vous en prie souvenez-vous du jour du fameux jour où la tortue est arrivée avant vous.

Herbert von Karajan, Extrait de *Paroles.*

Chapitre 7

Mise en pratique

Contenu du chapitre :

7.1 Introduction 149
7.1.1 La norme OSGi [The07] 149
7.1.2 Problématique du déploiement à large échelle 150
7.1.3 Gestion d'un parc de passerelles OSGi via une approche à services 152
7.2 Application de notre solution 154
7.2.1 Une orchestration distribuée et dynamique pour gérer un parc de passerelles 154
7.2.2 Configuration de la partie dynamique: ajout des plans de déploiement à l'orchestration globale 157
7.3 Conclusions 158

Dans ce chapitre, nous présentons une mise en pratique du modèle d'exécution dynamique d'orchestration décentralisée que nous avons présenté dans le chapitre 5. Nous nous intéressons ici à l'exécution d'une orchestration décentralisée qui a pour but de gérer et d'administrer un parc de passerelles OSGi.

La Section 7.1 présente le contexte de ce cas d'utilisation, à savoir le déploiement d'applications à large échelle sur des passerelles OSGi. Dans la Section 7.2, nous présentons une application du modèle que nous avons présenté dans le chapitre 5.

7.1 Introduction

Dans cette section, nous présentons brièvement la problématique du déploiement d'application à large échelle sur un parc de passerelles OSGi. Avant cela, nous introduisons la norme OSGi.

7.1.1 La norme OSGi [The07]

L'alliance OSGi™ (connue auparavant sous le nom de *Open Services Gateway initiative)*, fondée en 1999, spécifie un ensemble de standards définissant une plate-forme de services

FIG. 7.1 – Exploitation des passerelles à distance

fondée sur le langage Java et qui peut être gérée de manière distante, comme illustré sur la Figure 7.1. La spécification propose notamment un modèle de cycle de vie d'une application, un annuaire de services, un environnement d'exécution.

Les applications s'exécutant sur une plateforme OSGi sont déployées dans des unités de déploiement appelées *bundles* et qui peuvent être installés, arrêtés, démarrés, mis-à-jour et désinstallés de manière distante, sans nécessiter de redémarrage de la plateforme. Par ailleurs le *bundle* contient les entités nécessaires à l'exécution de l'application, c'est-à-dire non seulement les classes Java, mais aussi les éventuelles ressources et un fichier *manifest* qui permet au *bundle* de s'auto-décrire.

Ces applications sont déployées et exécutées au dessus d'un canevas modulaire écrit en Java qui permet aux services de s'exécuter dans leur propre chargeur de classes (*class loader).* Les interactions entre applications sont définies au travers des services ainsi que par le biais de l'utilisation de paquetages Java exportés par les *bundles.*

L'objectif original se focalisait sur les passerelles de services mais sa mise en application s'est avérée bien plus étendue. Les spécifications sont désormais utilisées plus largement dans des domaines tels que les IDE (par exemple l'IDE Eclipse[1]) ou encore les serveurs d'application.

Dans la suite de cette section, nous abordons la problématique de la gestion du déploiement sur un grand nombre de passerelles OSGi. Ensuite, nous présentons les éléments à notre disposition pour réaliser un déploiement à large échelle sur un parc de passerelles OSGi.

7.1.2 Problématique du déploiement à large échelle

Nous nous plaçons dans le contexte des infrastructures distribuées et hétérogènes que composent les passerelles, par exemple dans les contextes de la domotique, des télécommunications ou industriels. Ces infrastructures sont à la fois distribuées et hétérogènes. De ce fait, les applications exécutées sur les passerelles doivent être configurées selon les caractéristiques propres au matériel ou au contexte d'exécution. Devant le grand nombre de passerelles et d'applications, leur déploiement se doit d'être automatisé, et cela pour faciliter la tâche de la personne en charge du déploiement. Ce que nous entendons par

[1] http ://www.eclipse.org

Listing 7.1 – Un exemple de plan de déploiement unitaire

```
uninstall bundleA, version 0.9
install bundleB, version 1.0
install bundleA, version 1.0
```

déploiement est l'action d'installer une application précise (en terme d'application composée de services, empaquetés dans un ensemble de *bundles*), répondant à une configuration spécifique à chaque passerelle présente dans le parc à gérer, et ceci pour sélectionner et configurer l'application en conséquence, initier le déploiement de cette application et de gérer le cycle de vie complet du logiciel [CFH+98]. Dans le cadre d'un très grand nombre de passerelles à gérer, la solution doit automatiser et paralléliser ces installations afin de cacher la complexité inhérente à la multiplication du nombre de passerelles à gérer.

Les étapes qui doivent être suivies par le mécanisme de déploiement sont montrées dans la Figure 7.2. Le déploiement est initié par la personne en charge de l'administration des passerelles. Il souhaite installer une application donnée, qui peut exister sous différentes implémentations, sur un ensemble de passerelles OSGi. Le déroulement du processus de déploiement se déroule comme suit :

1. **Récupération des données concernant les passerelles**. La première étape consiste à introspecter les passerelles présentes dans le parc, ceci dans le but de connaître leurs caractéristiques (par exemple, la version du système d'exploitation, le type de CPU ou la liste des application déjà déployées), nécessaires pour sélectionner l'application adéquate dans l'étape 2. L'introspection des passerelles peut être réalisée à distance et à large échelle grâce au mécanisme de gestion de passerelles OSGi que nous avons présenté dans [BCL07].

2. **Sélection de l'implémentation.** La sélection de l'implémentation adéquate pour chacune des passerelle présentes dans le parc est réalisée. Selon les caractéristiques récupérées lors de l'étape 1, le processus de sélection fournit un plan de déploiement global destiné à être installé. Ce plan de déploiement représente l'union de plusieurs plans de déploiement unitaires ; un plan de déploiement unitaire contient une séquence d'actions qui doivent être réalisées dans le but d'approvisionner une application (dans notre cas un ensemble de *bundles* fournissant des services) sur un élément physique (la passerelle OSGi) de l'infrastructure. Un exemple de plan de déploiement unitaire est donné dans le Listing 7.1. Ce plan reflète l'installation d'une nouvelle version du `bundleA, version 1.0` qui nécessite la désinstallation du `bundleA, version 0.9` et l'installation du `bundleB, version 1.0`.

3. **Exécution des plans de déploiement.** Une fois la phase de sélection terminée, il faut exécuter les plans de déploiement unitaires qui composent le plan de déploiement global afin de réaliser les actions nécessaires au déploiement de l'application.

Nous venons de présenter la problématique et la méthodologie que nous allons suivre pour réaliser un déploiement à large échelle. Dans la suite, nous nous appuyons sur cette méthodologie pour décrire notre cas cas d'utilisation.

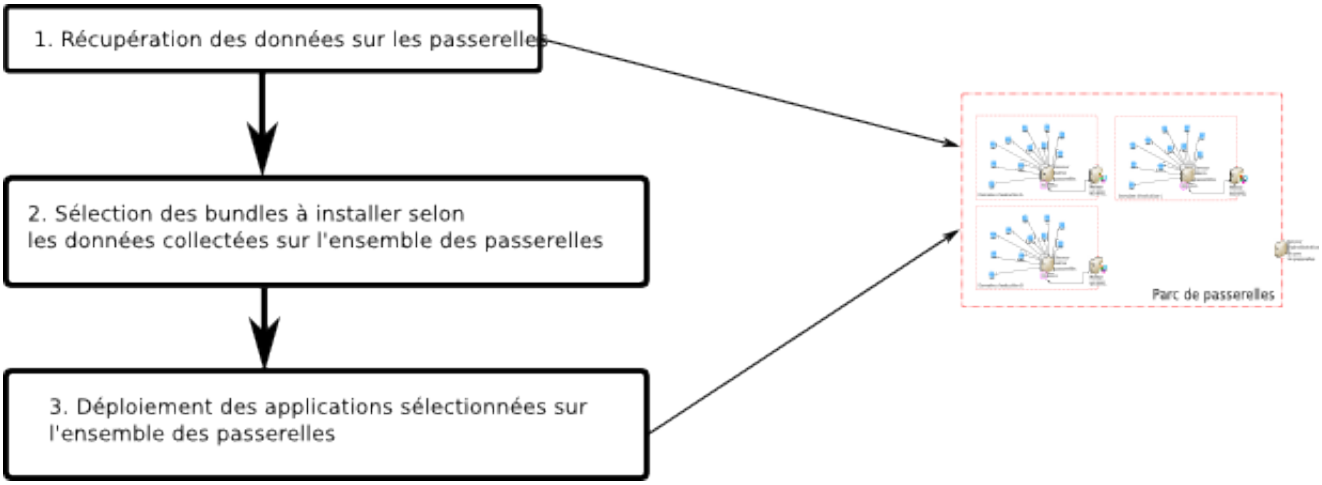


FIG. 7.2 – Les étapes à suivre pour la réalisation du déploiement d'une appication sur un ensemble de passerelles

7.1.3 Gestion d'un parc de passerelles OSGi via une approche à services

Le cas d'utilisation que nous présentons ici, consiste à gérer un processus d'installation d'applications sur un parc constitué d'un grand nombre de passerelles OSGi. Pour ce faire, nous définissons une orchestration décentralisée, basée sur la méthodologie décrite dans la Section 7.1, qui va dans un premier temps récupérer les caractéristiques des passerelles, calculer ensuite les plans de déploiement et finalement installer ces plans de déploiement sur l'ensemble des passerelles, tout en tenant compte de l'hétérogénéité des passerelles.

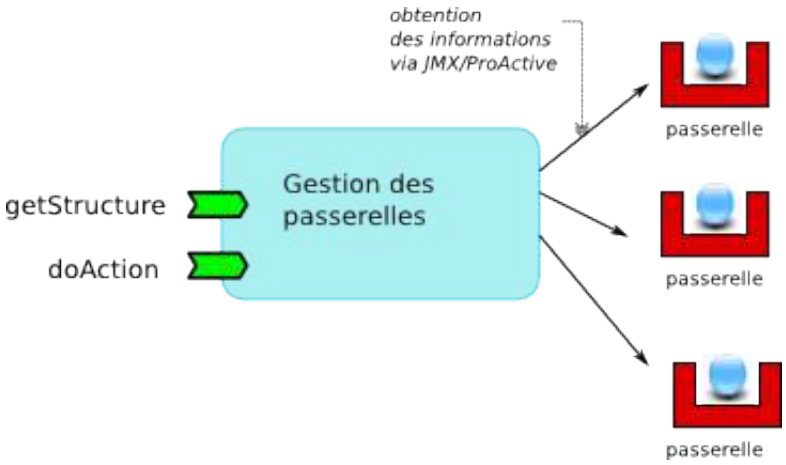


FIG. 7.3 – Le service d'obtention de l'infrastructure, représenté avec la notation SCA

Nous nous sommes fixé pour objectif de gérer un parc de passerelles OSGi, réparti sur trois domaines, `domaineA`, `domaineB` et `domaineC`, qui peuvent être assimilés à des

sous-parcs. L'union de ces trois domaines organisationnels, représente un domaine virtuel nommé `parc_domain`. La structure de ce parc est illustrée sur la Figure 7.4. Chaque parc comprend un ensemble de passerelles potentiellement hétérogènes. Il est géré par un serveur d'administration qui permet notamment de collecter les données des passerelles via un connecteur JMX/ProActive [BCL07]. Le serveur d'administration expose un Service Web, `getStructure` qui permet de récupérer les données de l'ensemble des passerelles. Ce service est montré sur la Figure 7.3.

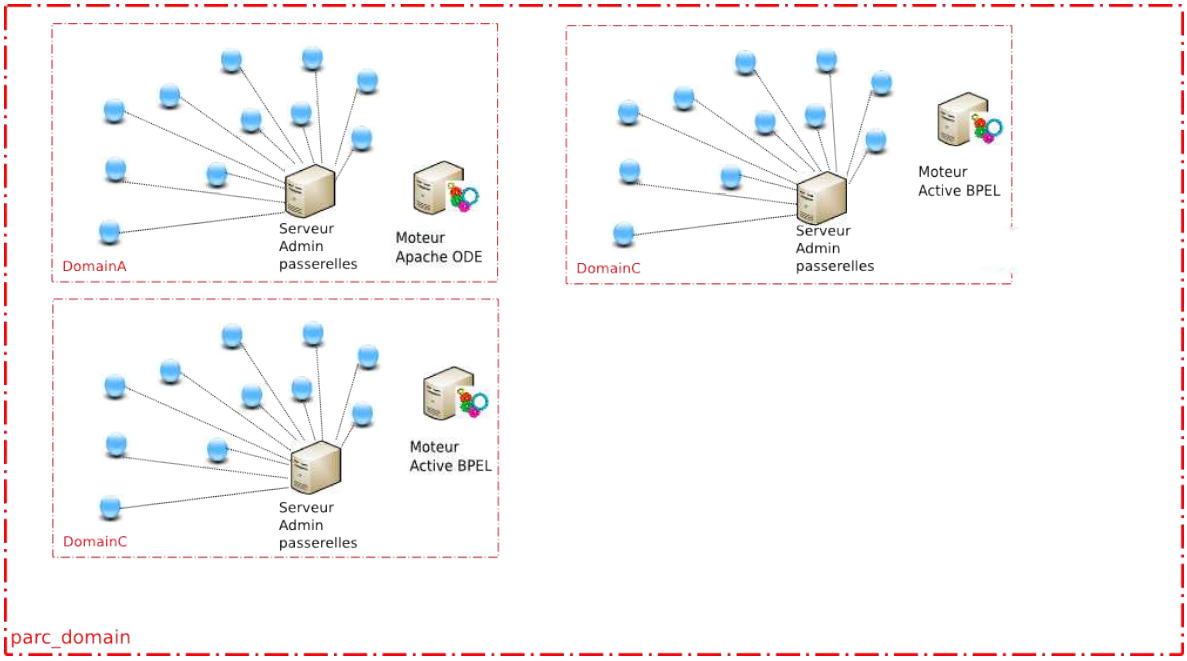


FIG. 7.4 – Structure du parc de passerelles, distribué sur plusieurs domaines

Chaque sous-parc est également pourvu d'un moteur d'orchestration, en l'occurrence ActiveBPEL pour les domaines `domainB` et `domainC` et Apache ODE pour le domaine `domainA`.

L'ensemble du parc est accessible depuis un domaine extérieur, appelé `domain_admin`, à l'intérieur duquel est installé un moteur d´orchestration Apache ODE. Ce domaine d'administration est illustré dans la Figure 7.5. Le serveur d'administration expose un Service Web, montré dans la Figure 7.6, qui permet de calculer les plans de déploiement, selon des caractéristiques données et une application donnée. Il reçoit en entrée une description de la structure du parc et retourne le plan de déploiement global, union des plans de déploiement unitaires destinés à être exécutés pour chaque passerelle.

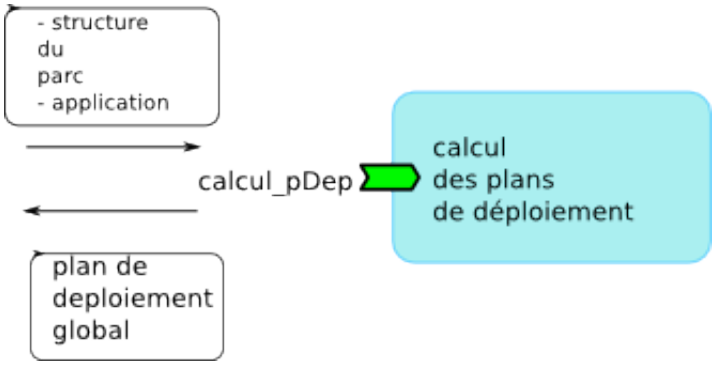


FIG. 7.6 – Le service de calcul de plan de déploiement

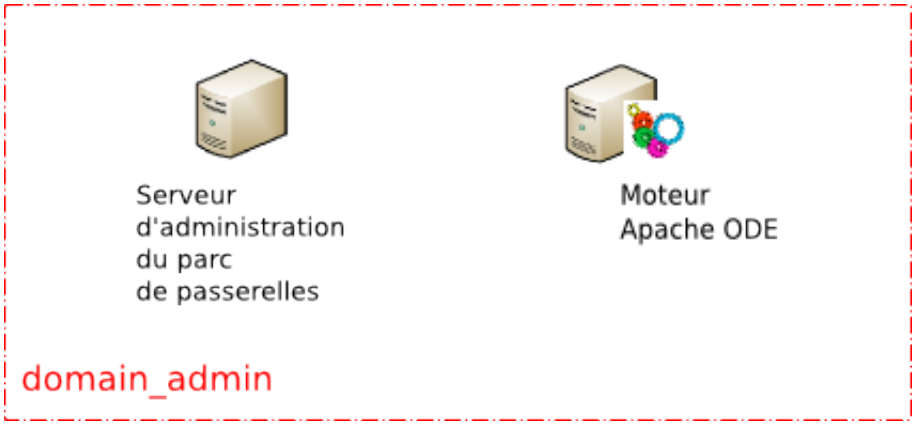


FIG. 7.5 – Le domaine d'administration du parc de passerelles

Dans la section suivante, nous utilisons les éléments que nous venons de décrire, ceci dans le but d'appliquer notre approche basée sur les composants au processus de déploiement que nous pouvons assimiler à une orchestration décentralisée.

7.2 Application de notre solution

L'idée de base pour mettre en œuvre le processus de déploiement est de définir une orchestration distribuée, qui réalise les étapes décrites dans la Section 7.1.

Dans cette Section, nous commençons par décrire cette orchestration décentralisée et nous la modélisons grâce à notre approche à composants. Nous explicitons comment les plans de déploiement unitaires sont gérés de manière dynamique dans le processus global.

7.2.1 Une orchestration distribuée et dynamique pour gérer un parc de passerelles

Le processus de déploiement d'une application sur un parc de passerelles OSGi peut être représenté par une orchestration, composant les différents services que nous avons

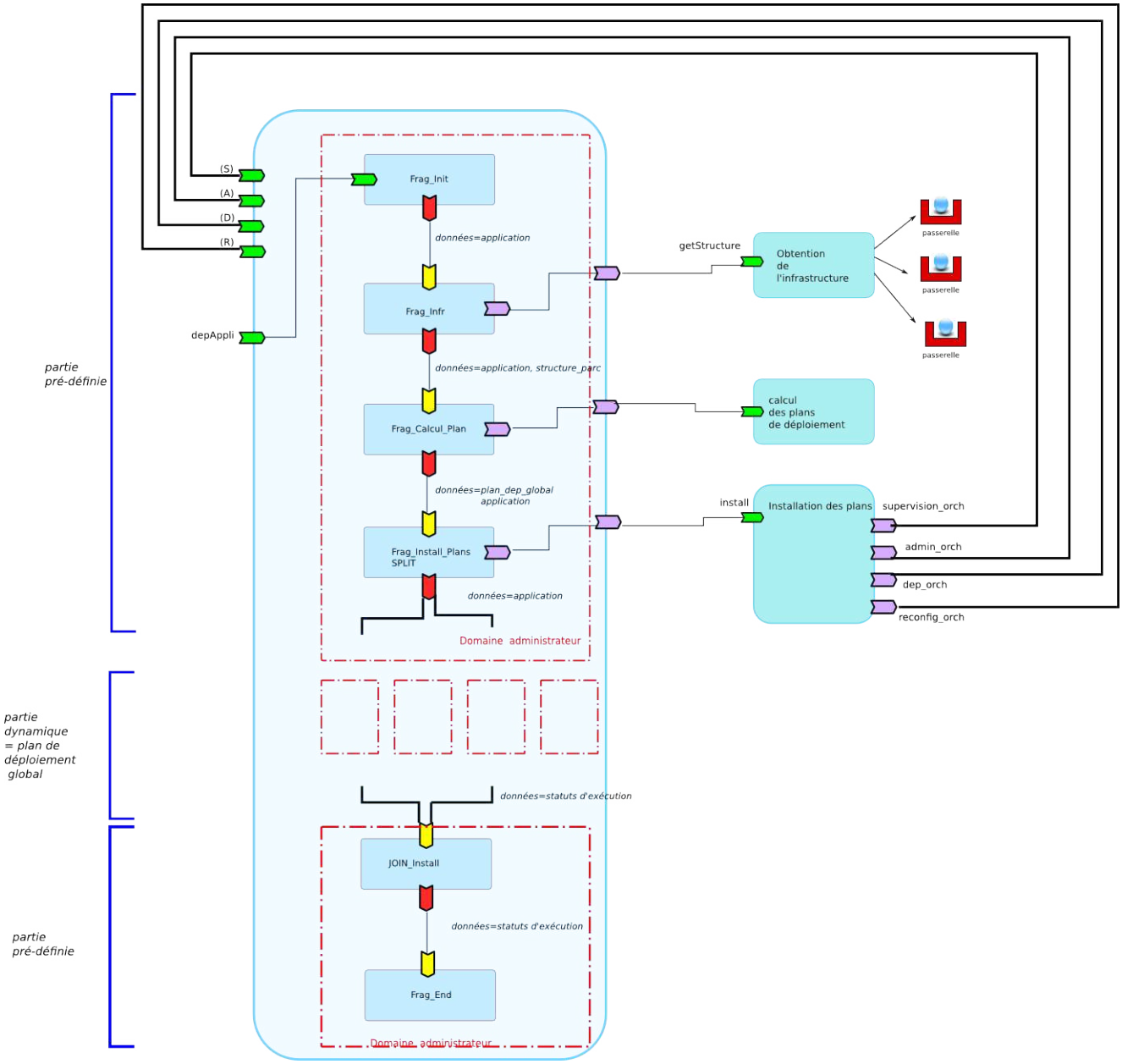


FIG. 7.7 – L'orchestration distribuée correspondant au processus de déploiement

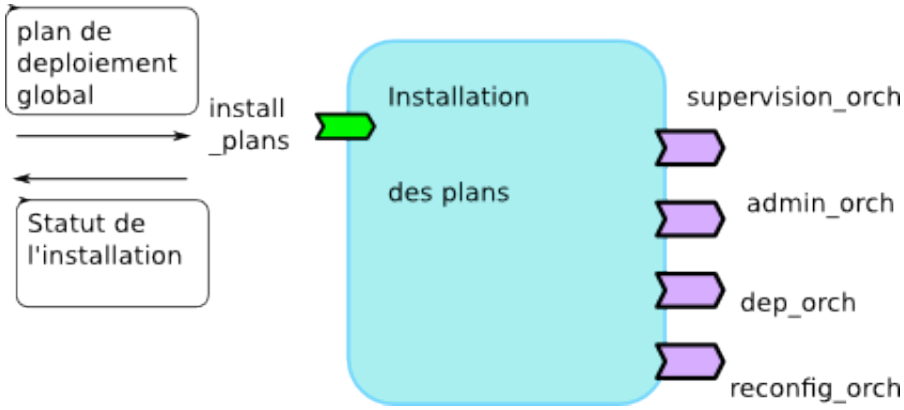


FIG. 7.8 – Le service d'installation des plans de déploiement

décrits précédemment. La Figure 7.7 représente cette orchestration définie comme un fragment composite. Cette orchestration est divisée en deux parties, la partie pré-définie et la partie dynamique. La partie prédéfinie contient les fragments nécessaires au pré-traitement de l'installation effective (c'est-à-dire le calcul des plans de déploiement). La partie dynamique repose quant à elle sur les résultats de la partie prédéfinie, son contenu dépendant notamment du calcul du plan de déploiement global. Le nombre de fragments contenus dans cette partie dynamique est fonction du nombre de passerelles à administrer.

Partie pré-définie La partie pré-définie comprend le fragment initial `Frag_init`, qui va récupérer les données d'appel de l'orchestration globale (le nom de l'application), et se déroule dans le domaine d'administration `domain_admin`. Le fragment suivant, `Frag_infr`, représente un sous-processus qui va récupérer la structure du parc de passerelles, qui sera transmise au fragment suivant `Frag_calc_plan`. Le fragment `Frag_calc_plan` invoque alors le service de calcul de déploiement et passe les données applicatives (le plan de déploiement global) au fragment `Frag_install_plan`.

Partie dynamique La partie dynamique est exécutée une fois que les calculs de pré-installation sont terminés. Pour cela, nous définissons un service en charge de contrôler cette partie dynamique par le biais des interfaces de contrôle du fragment composite représentant l'orchestration. Ainsi, le fragment `Frag_install_plan` invoque un service particulier, montré dans la Figure 7.8. Ainsi il est possible de déployer et d'ajouter des sous-fragments pendant l'exécution du processus global, en invoquant les interfaces de contrôle ((S), (A), (D), (R)) du fragment composite. Ce service, bien qu'externe à la composition globale, a été conçu pour contrôler celle-ci. Il contient donc l'information nécessaire pour effectuer ce contrôle, en particulier, le fragment précédent (`Frag_Install_plan_SPLIT`) et le fragment suivant (`JOIN_Install`) du fragment qu'il devra ajouter.

Par ailleurs, ce service permet via ses liaisons vers les interfaces de contrôle du fragment composite, d'administrer ce même fragment (référence `admin_orch`), de souscrire à des évènements (référence `supervision_orch`). Il a aussi la possibilité de contrôler l'ensemble des moteurs impliqués dans ce fragment composite (référence `dep_orch`), qui peuvent être quant à eux hétérogènes ; dans notre cas, il peut contrôler de manière transparente les moteurs Apache ODE et ActiveBPEL.

Pour des raisons d'administration ou de structuration en cas d'un très grand nombre de passerelles, il est possible d'imaginer une alternative à ce scénario permettant de structurer la partie dynamique en sous-fragments composites reflétant l'organisation de la structure hiérarchique (potentiellement à plusieurs niveaux) en sous-parcs. Un fragment unitaire correspondant au plan de déploiement d'une passerelle serait ajouté dans le sous-fragment composite correspondant au sous-parc où la passerelle est située.

7.2.2 Configuration de la partie dynamique : ajout des plans de déploiement à l'orchestration globale

Pour chaque plan de déploiement contenu dans le plan de déploiement global, on génère une définition WS-BPEL qui représente un plan de déploiement unitaire. Elle comprend une suite d'invocations au Service Web (`doAction`) de gestion présent sur le serveur d'administration des passerelles du domaine dans lequel se situe chaque passerelle, en lui passant en paramètre l'action à effectuer ainsi que la passerelle concernée.

A partir de cette définition, nous générons un fragment lui correspondant. Lors de l'exécution de ce fragment, le service `doAction` sera invoqué autant de fois qu'il y a d'action à effectuer dans le plan, permettant d'installer lcette passerelle là.

Une fois le fragment généré, le service d'installation déploie le fragment sur le serveur d'administration qui gère le sous-parc dans lequel se trouve la passerelle concernée. À l'intérieur du fragment, les étapes du déploiement du fragment sont réalisées, notamment l'encapsulation du moteur WS-BPEL[2]. A ce moment, le fragment n'est pas encore présent dans l'orchestration globale. Il est alors inséré, grâce à l'interface de contrôle de la reconfiguration, dans la partie dynamique (entre le Fragment `Frag_Install_plan_SPLIT` et le fragment `JOIN_Install`). Pour rappel, lors de la reconfiguration dynamique du fragment composite, l'exécution de ses instances est stoppée : le rajout de fragments ne perturbe en aucune manière l'exécution d'une quelconque instance du processus associé au fragment composite.

Le service d'installation traite ainsi tous les plans contenus dans le plan de déploiement global. De ce fait, la partie dynamique contient autant de fragments, qui vont s'exécuter de manière parallèle, que de plans de déploiement unitaires, donc autant que de passerelles OSGi à gérer.

Lorsque la phase d'insertion des sous-fragments est terminée, l'exécution des plans de déploiement est alors initiée en envoyant, à chaque fragment unitaire qui représente un plan de déploiement, les données applicatives (en l'occurence le nom de l'application). Une fois ces fragments exécutés, ils renvoient au fragment `JOIN_Install` l'ensemble des statuts d'exécution ; un statut d'exécution permet de représenter le résultat du déploiement (succès ou erreur).

Remarquons ici que la modification de la structure du fragment composite fait l'hypothèse implicite qu'une seule instance du processus entier s'exécute à la fois, car la partie dynamique est dépendante de l'application à installer. Cette hypothèse implicite, apparemment restrictive, a le mérite de pouvoir garder une cohérence de l'état des passerelles. C'est pourquoi le service d'installation, une fois l'exécution de tous les plans de déploiement effectuées, retire les sous-fragments dans la partie dynamique. Il aura notamment pour cela, souscrit à l'évènement "Fin exécution" pour chaque fragment généré.

[2]Étant donné que le moteur WS-BPEL sera utilisé de nombreuses fois, il est bénéfique de ré-utiliser celui-ci plutôt que de le ré-installer à chaque déploiement de sous-fragment.

7.3 Conclusions

Nous avons présenté dans ce chapitre une mise en pratique de notre solution d'exécution dynamique d'orchestrations décentralisées. Cette mise en pratique repose sur un cas d'utilisation qui consiste à définir une orchestration décentralisée dont la définition évolue au fil de son exécution, et qui permet de déployer des applications sur un parc de passerelles OSGi sur des infrastructures hétérogènes. Cette définition d'orchestration a été projetée sur des composants SCA enrichis de dépendances temporelles, présentés dans le Chapitre 5 et permettent à l'initiateur de l'action de déploiement de gérer de manière aisée le déroulement des plans de déploiements. Nous avons par ailleurs, aussi dans [BL11], illustré notre approche via un processus métier mettant en œuvre plusieurs partenaires dans le cadre de l'organisation d'un voyage d'affaires.

Il faut faire d'abord volontairement, avec plaisir, ce qu'on fait. Le résultat importe peu. On ne le prévoit pas, et on l'apprécie mal. Mais l'auteur s'est satisfait lui-même : c'est toujours ça.

Jules Renard, Extrait de *Journal 1893 - 1898*

Chapitre 8

Conclusions et perspectives

Contenu du chapitre :

8.1 Bilan de notre contribution 159
8.2 Perspectives 161

Ce chapitre présente les conclusions obtenues à partir de notre travail et présente les perspectives de recherche que nous avons pu identifier.

8.1 Bilan de notre contribution

Dans cette thèse, nous avons présenté une plateforme permettant l'exécution décentralisée et dynamique d'une composition de services, en utilisant une approche basée sur les composants. L'objectif majeur de cette thèse étaient de proposer un moyen d'exécuter les orchestrations décentralisées.

Nous nous sommes intéressés à l'exécution des orchestrations décentralisées. Ces orchestrations, pouvant s'exécuter dans un contexte inter-organisationnel, peuvent être le résultat (1) d'une approche descendante, explicite d'un découpage d'une orchestration globale, ou bien (2) le résultat d'une approche ascendante permettant de construire une orchestration répartie à partir d'orchestrations existantes. Nous avons identifié cinq critères qui nous semblent pertinents pour bâtir une solution d'exécution d'orchestration décentralisées, à savoir l'hétérogénéité des processus impliqués dans une composition et de leur système d'exécution (C1), la prise en compte et la gestion de la dynamicité[1] des processus métiers (C2), la gestion du flux distribué de données entre processus (C3), l'administration et la supervision des processus métiers distribués (C4) et la gestion de la complexité du déploiement (C5).

Après avoir présenté dans un premier temps, le contexte des architectures orientées services dans le Chapitre 2, nous avons présenté dans un second temps, dans le Chapitre 3, une étude concernant des systèmes d'exécution d'orchestrations décentralisées, et nous

[1] ou du moins de certains aspects de cette dynamicité

les avons évalués suivant les cinq critères définis auparavant. Cette étude nous a permis de mettre en évidence les forces et les faiblesses de chacun de ces systèmes, mais surtout qu'aucun ne répondait totalement aux cinq critères que nous considérons primordiaux dans un système d'exécution d'orchestrations décentralisées. À partir de ces constatations et en nous inspirant des modèles existants nous avons défini un modèle d'exécution d'orchestrations décentralisées puis nous avons spécifié l'architecture logicielle permettant l'implémentation de notre modèle.

Ainsi, nous avons retenu de l'étude de l'état de l'art que, dans le but d'exécuter des orchestrations décentralisées et cela, de manière dynamique, ces systèmes offraient notamment (1) des mécanismes de gestion de l'hétérogénéité des moteurs d'exécution grâce à la définition d'interfaces communes, (2) une gestion de la dynamicité grâce au patron de conception de *proxy*, (3) une communication inter-processus qui consiste, dès la fin d'un sous-processus, à envoyer les données applicatives à son sous-processus suivant, (4) la possibilité d'administrer et de superviser les processus métiers distribués, en offrant des interfaces de contrôle, ainsi que (5) la capacité de déployer une définition de processus distribué et de son système d'exécution sur chaque lieu impliqué dans l'entreprise virtuelle. Nous avons pris le parti de nous appuyer sur ces concepts et de les utiliser pour bâtir une approche d'exécution orchestration décentralisée basée sur les composants.

Techniquement parlant, enchaîner les sous-processus faisant partie d'une orchestration est une opération réalisable avec tous les moteurs d'orchestration existants. Par exemple, comme c'est le cas dans [Yil08], en utilisant WS-BPEL, les sous-processus coopérants sont enchaînés en utilisant les liens partenaires. Ces mêmes liens sont utilisés pour définir les dépendances fonctionnelles. Cette approche comporte principalement des inconvénients tels que le manque de flexibilité au niveau de la définition induits par l'utilisation de fichiers de description qui sont par nature statiques, ainsi qu'un manque de contrôle de l'orchestration décentralisé : par exemple, une fois l'exécution du processus initiée, il n'est pas possible de réaliser aisément des modifications concernant les liens partenaires. La séparation des préoccupations fonctionnelles (les liens vers les services) et des préoccupations non-fonctionnelles n'est pas non plus réalisée.

Afin de supprimer les limites inhérentes aux modèles analysés, nous avons défini un modèle spatio-temporel basé sur un modèle à composants permettant de séparer la logique d'enchaînement des sous-processus, autrement dit la logique de passage des données applicatives, des appels fonctionnels aux services impliqués dans l'orchestration. La mise en œuvre de cette solution a été réalisée grâce au langage WS-BPEL permettant de définir les sous-processus, ainsi qu'au modèle à composants GCM et à son implémentation de référence GCM/ProActive, nous fournissant un canevas d'exécution distribué et dynamique. Nous avons validé notre approche en réalisant une orchestration décentralisée représentant un processus de déploiement d'une application sur un ensemble de passerelles OSGi.

Les apports de ces travaux de thèse sont :

- Le **modèle de fragment** qui permet d'englober, quand le **fragment est unitaire**, un sous-processus, et quand le **fragment est composite**, une orchestration composée de sous-orchestrations. La composition des sous-fragments dans un fragment composite peut se réaliser en séquence, ou en suivant un patron de réalisation AND-SPLIT/AND-JOIN, permettant d'exécuter plusieurs sous-orchestrations en parallèle.
- Un **mécanisme de projection d'une orchestration distribuée sur un fragment composite.** Un fragment composite étant dans ce cas une composition hiérarchique de fragments et, étant donné qu'il n'existe pas de restriction sur les sous-fragments utilisés, ces derniers peuvent être eux-mêmes des fragments composites ou des fragments unitaires. Nous obtenons ainsi une vue globale de la totalité de l'orchestration

décentralisée sous la forme d'un composant offrant des interfaces de contrôle.
- Une **vue globale et unifiée d'une orchestration décentralisée** que l'on peut **manipuler, déployer, administrer et reconfigurer facilement** lors de son exécution, sous la forme d'un fragment composite unique simplifiant ainsi sa gestion.
- Une **implémentation du modèle de fragment** basée sur le modèle à composants GCM/ProActive.

8.2 Perspectives

L'approche à composants que nous avons présentée dans cette thèse nous permet d'exécuter une orchestration décentralisée et d'en avoir une vue globale. Même si le modèle permet de remplir les objectifs que nous nous sommes fixés dans le Chapitre 1, nous avons tout au long de cette thèse rencontré des situations dans lesquelles des extensions de notre modèle permettraient une réponse encore plus adaptée au besoin des orchestrations décentralisées. Ainsi le modèle peut évoluer de différentes manières que nous listons ci-après :

Vers une "componentisation" plus fine de l'orchestration Dans notre solution, le grain de distribution de l'orchestration est le sous-processus, défini par exemple en WS-BPEL. La propriété de dynamicité peut intervenir à deux niveaux, à savoir au niveau des liaisons structurelles vers les services, et au niveau des liaisons temporelles, entre sous-fragments. Il n'est pas possible, au moment de l'exécution du sous-processus de changer la structure de sa définition. Il serait intéressant de se pencher sur l'utilisation d'un grain plus fin pour décentraliser les orchestrations. L'avantage de représenter le processus, par exemple écrit en WS-BPEL, sous la forme d'un système à composants est de pouvoir reconfigurer une activité ou une portion du processus. Cela reviendrait ainsi à implémenter un moteur d'orchestration, totalement en GCM.

Ajout de structures de contrôle de passage de données Notre solution a proposé une gestion du flot de données, soit en suivant un patron séquence, soit en envoyant les données de manière parallèle aux sous-fragments coopérants suivants. D'autres structures de contrôle pourraient être ajoutées, par exemple, une structure IF, ou OR. Ce qui permettrait de pouvoir définir une coopération plus sophistiquée entre les sous-fragments.

Déployer l'orchestration à la demande La définition d'une orchestration centralisée, est, grâce à notre solution, déployée dans son intégralité. Ainsi, si l'on se base sur la réalisation de la perspective précédente, toute la définition du fragment composite ne serait pas nécessaire à son exécution entière, par exemple, lors de l'exécution d'une structure IF, qui n'exécuterait qu'une seule branche de la définition. La perspective de déployer seulement la branche nécessaire à l'exécution du fragment composite permettrait de ne pas avoir à déployer la définition entière mais de ne déployer que ce qui est nécessaire au fur-et-à-mesure de l'exécution.

Composition des orchestrations sans définir au préalable les liens de coopération Dans notre thèse nous avons présenté des orchestrations décentralisées qui ont été composées de sous-orchestrations qui doivent avoir été préparées (à un degré plus ou moins avancé) pour réaliser la coopération entre fragments (insertion d'opérations de réception

ou de retour de données). Une évolution de notre modèle consisterait à composer des sous-orchestrations sans se soucier de la coopération entre sous-orchestrations. Cela implique la séparation de la logique de la coopération avec la définition du processus totalement indépendante de la coopération.

Amélioration de la gestion du flot de données Par ailleurs, comme présenté dans [TZ06], la gestion du flot de données pourrait être améliorée en incluant un mécanisme s'inspirant du mécanisme de *future*, qui permettrait d'enchaîner l'exécution des sous-fragments. Ainsi, un sous-fragment pourrait commencer à s'exécuter même si il n'a pas reçu toutes les données applicatives, et se mettre automatiquement en attente lors de l'accès effectif à la donnée applicative et ce tant que cette donnée n'est pas disponible. Cela impliquerait notamment de modifier la définition du processus, en ajoutant à des endroits pertinents des activités d'invocation d'un service qui met à disposition la valeur des données lorsque ces dernières sont accédées. Ce service peut se baser sur l'implémentation de GCM/ProActive qui comprend la gestion de communication asynchrone basée sur le mécanisme de *future.* De plus, le service pourrait, lorsque la valeur d'un *future* est disponible, de manière transparente récupérer cette donnée sans attendre un accès à cette dernière, facilitant l'exécution du processus, la réception des données au niveau des interfaces temporelles ne serait ainsi plus bloquante.

Ajout d'un mécanisme de sécurité Notre solution peut être installée et exécutée dans un environnement inter-organisationnel. Ainsi des problèmes de sécurité, en terme de droits d'accès, peuvent apparaître, et cela durant plusieurs phases, notamment celle de l'installation des processus et celle du passage de données applicatives. Un mécanisme de gestion de droits d'accès serait alors nécessaire pour mener à bien le déploiement et l'exécution de l'orchestration. GCM/ProActive propose un mécanisme de sécurité [Con05] basé sur le concept d'entités sécurisées permettant l'interception des communications entrantes et sortantes de ces entités et donc d'en contrôler les accès et le flot de données. Dans notre cas l'entité sécurisée peut se projeter sur le fragment unitaire. De plus, ce modèle étant hiérarchique, il s'intègre parfaitement avec notre définition d'un fragment composite, les règles de sécurité des divers composants se trouvant combinées de manière dynamique lors d'échange de données entre fragments.

Bibliographie

[AAA+06] Alexandre Alves, Assaf Arkin, Sid Askary, Ben Bloch, Francisco Curbera, Yaron Goland, Neelakantan Kartha, Dieter König, Vinkesh Mehta, Satish Thatte, and Others. Web Services Business Process Execution Language version 2.0. Technical Report April, 2006.

[ABJ+04] Ilkay Altintas, Chad Berkley, Efrat Jaeger, Matthew Jones, Betram Ludascher, and Steve Mock. Kepler : An extensible system for design and execution of scientific workflows. In *Scientific and Statistical Database Management, 2004. Proceedings. 16th International Conference on*, pages 423–424. IEEE, 2004.

[ACKM04] Gustavo Alonso, Fabio Casati, Harumi Kuno, and Vijay Machiraju. *Web Services – Concepts, Architectures and Applications*. Data-Centric Systems and Applications. Springer, 2004.

[AD02] Benoit A. Aubert and Aymeric Dussart. Systèmes d'information inter-organisationnels. Technical Report Mars, Rapport Bourgogne, CIRANO, 2002.

[AE97] M. Amiour and J. Estublier. Apel : Un formalisme pour le support à la coopération dans les procédés logiciels. *Génie logiciel*, (46) :71–76, 1997.

[AMA+95] Gustavo Alonso, C Mohan, Divyakant Agrawal, Amr El Abbadi, Roger Gunthor, and Mohan Kamath. Exotica/FMQM : A persistent message-based architecture for distributed workflow management. *IFIP WG8*, 1(August), 1995.

[BAGS02] Rajkumar Buyya, David Abramson, Jonathan Giddy, and Heinz Stockinger. Economic models for resource management and scheduling in Grid computing. *Concurrency and Computation : Practice and Experience*, 14(13-15) :1507–1542, November 2002.

[BC01] Guy Bieber and Jeff Carpenter. Introduction to service-oriented programming (rev 2.1). *OpenWings Whitepaper, April*, pages 1–13, 2001.

[BCD+09] Françoise Baude, Denis Caromel, Cédric Dalmasso, Marco Danelutto, Vladimir Getov, Ludovic Henrio, and Christian Pérez. GCM : A grid extension to Fractal for Autonomous Distributed Components. *Annals of Telecommunications*, 64(1) :5–24, 2009.

[BCL+] Eric Bruneton, Thierry Coupaye, Matthieu Leclercq, Vivien Quéma, and Jean-bernard Stefani. *Software : Practice and Experience*, 36.

[BCL07] Françoise Baude, Virginie Legrand Contes, and Vincent Lestideau. Large-Scale Service Deployment–Application to OSGi. In *proceedings of IARIA 3rd International conference on Autonomic and Autonomous Services (ICAS 2007)*, pages 19–26. IEEE Computer Society, june 2007.

[BDDZ05] Boualem Benatallah, Remco M. Dijkman, Marlon Dumas, and Maamar Zakaria. Service Composition : Concepts, Techniques, Tools and Trends. In Z. Stojanovic and A. Dahanayake, editors, *Service-Oriented Software System Engineering : Challenges and Practices*, pages 48–66. Idea Group Publishing, Hershey, 2005.

[BDO05] Allistair Barros, Marlon Dumas, and Phillipa Oaks. A Critical overview of the Web Services Choreography Description Language. *BPTrends Newsletter*, 3(March) :1–24, 2005.

[BDS05] Boualem Benatallah, Marlon Dumas, and Quan Z. Sheng. Facilitating the rapid development and scalable orchestration of composite web services. *Distributed and Parallel Databases*, 17(1) :5–37, 2005.

[BDSN02] Boualem Benatallah, Marlon Dumas, Q Z Sheng, and Anne H. H. Ngu. Declarative Composition and Peer-to-Peer Provisioning of Dynamic Web Services. In M Dumas, editor, *Proc. 18th International Conference on Data Engineering*, pages 297–308, 2002.

[BFH+10] Françoise Baude, Imen Filali, Fabrice Huet, Virginie Legrand, Elton Mathias, Philippe Merle, Cristian Ruz, Reto Krummenacher, Elena Simperl, Christophe Hammerling, and Others. ESB federation for large-scale SOA. In *Proceedings of the 2010 ACM Symposium on Applied Computing*, pages 2459–2466. ACM, 2010.

[BL11] Françoise Baude and Virginie Legrand. A component-based orchestration management framework for multidomain SOA. In *Integrated Network Management (IM), 2011 IFIP/IEEE International Symposium on*, pages 1156–1163. IEEE, 2011.

[BLH+10] Françoise Baude, Virginie Legrand, Ludovic Henrio, Paul Naoumenko, Heiko Pfeffer, Louay Bassbouss, and David Linner. Mixing Workflows and Components to Support Evolving Services. *International Journal of Adaptive, Resilient and Autonomic Systems (IJARAS)*, 1(4) :60–84, 2010.

[BME+07] Grady Booch, Robert Maksimchuk, Michael Engle, Bobbi Young, Jim Conallen, and Kelli Houston. *Object-oriented analysis and design with applications, third edition*. Addison-Wesley Professional, third edition, 2007.

[BMM05] Luciano Baresi, Andrea Maurino, and Stefano Modafferi. Workflow partitioning in mobile information systems. *Mobile Information Systems*, pages 93–106, 2005.

[CCC+07] Philippe Collet, Thierry Coupaye, Hervé Chang, Lionel Seinturier, and G. Dufrêne. Components and Services : A Marriage of Reason. Technical Report RR-2007-17-FR, Laboratoire d'Informatique de Signaux et Systèmes de Sophia Antipolis - UNSA-CNRS, 2007.

[CCMN04] Girish B. Chafle, Sunil Chandra, Vijay Mann, and Mangala Gowri Nanda. Decentralized orchestration of composite web services. In *Proceedings of the 13th international World Wide Web conference on Alternate track papers & posters*, WWW Alt. '04, pages 134–143, New York, NY, USA, 2004. ACM.

[CCMW01] Erik Christensen, Francisco Curbera, Greg Meredith, and Sanjiva Weerawarana. Web services description language (WSDL) 1.1. http://www.w3.org/TR/wsdl, 2001.

[CDM06] Massimiliano Colombo, Elisabetta Di Nitto, and Marco Mauri. Scene : A service composition execution environment supporting dynamic changes disciplined through rules. In Asit Dan and Winfried Lamersdorf, editors, *Service-Oriented Computing – ICSOC 2006*, volume 4294 of *Lecture Notes in Computer Science*, pages 191–202. Springer Berlin / Heidelberg, 2006.

[CDN08] Luca Cavallaro and Elisabetta Di Nitto. An approach to adapt service requests to actual service interfaces. In *Proceedings of the 2008 international workshop on Software engineering for adaptive and self-managing systems*, SEAMS '08, pages 129–136, New York, NY, USA, 2008. ACM.

[CFH+98] Antonio Carzaniga, Alfonso Fuggetta, Richard S. Hall, André van der Hoek, Dennis Heimbigner, and Alexander L. Wolf. A characterization framework for software deployment technologies. Technical Report CU-CS-857-98, Department of Computer Science, University of Colorado, April 1998.

[Cha04] David Chappell. *Enterprise service bus*. O'Reilly Media, Inc., 2004.

[Cha07] David Chappell. Introducing SCA. http://www.davidchappell.com/articles/, 2007.

[CHvR+04] Luc Clement, Andrew Hately, Claus von Riegen, Tony Rogers, et al. UDDI Version 3.0. 2, UDDI Spec Technical Committee Draft. http://www.uddi.org/pubs/uddi_v3.htm, 2004.

[Coi08] Francois Cointe. À la quête de la SOA. https://profile.microsoft.com/RegSysProfileCenter/wizard.aspx?wizid=454ea1e7-6315-4c8f-9407-2c1bb15986fc\&lcid=1036, 2008.

[Con05] Arnaud Contes. *Une Architecture de Sécurité Hiérarchique, Adaptable et Dynamique pour la Grille* . PhD thesis, University of Nice Sophia-Antipolis, September 2005.

[CS01] Fabio Casati and Ming-Chien Shan. Dynamic and adaptive composition of e-services. *Information Systems*, 26(3) :143–163, May 2001.

[Cur07] Francisco Curbera. Component contracts in service-oriented architectures. *Computer*, 40 :74–80, November 2007.

[DGP+07] Jan Dünnweber, Sergei Gorlatch, Nikos Parlavantzas, Françoise Baude, and Virginie Legrand. Towards automatic creation of web services for grid component composition. In Sergei Gorlatch and Marco Danelutto, editors, *Integrated Research in GRID Computing*, pages 31–42. Springer US, 2007. 10.1007/978-0-387-47658-23.

[Dod04] Mahesh H. Dodani. From Objects to Services : A Journey in Search of Component Reuse Nirvana. *Journal of Object Technology*, 3(8) :49–54, 2004.

[DZD06] Gero Decker, Johannes Maria Zaha, and Marlon Dumas. Execution Semantics for Service Choreographies. *Journal of the American Academy of Dermatology*, 54(6) :1121–1122, June 2006.

[EAS08] Wolfgang Emmerich, Mikio Aoyama, and Joe Sventek. The impact of research on the development of middleware technology. *ACM Transactions on Software Engineering and Methodology*, 17(4) :1–48, August 2008.

[Erl] Thomas Erl. *SOA Principles of Service Design*. Prentice Hall.

[Esc08] Clément Escoffier. *iPOJO : Un modèle à composant à service flexible pour les systèmes dynamiques*. PhD thesis, Université Joseph Fourier, Grenoble, December 2008.

[fab] Fabric3. http://www.fabric3.org/.

[Fie00] R.T. Fielding. *Architectural styles and the design of network-based software architectures*. PhD thesis, University of California, 2000.

[Fre04] Freefluo Overview. http://freefluo.sourceforge.net/, 2004.

[FS05] D. F. Ferguson and M. L. Stockton. Service-oriented architecture : Programming model and product architecture. *IBM Systems Journal*, 44, 2005.

[GAHL00] Paul Grefen, Karl Aberer, Yigal Hoffner, and Heiko Ludwig. Crossflow : Cross-organizational workflow management in dynamic virtual enterprises. *International Journal of Computer Systems Science & Engineering*, 15(5) :277–290, September 2000.

[Gar00] David Garlan. Software architecture : a roadmap. In *Proceedings of the Conference on the Future of Software Engineering*, pages 91–101. ACM, 2000.

[GHM+02] Martin Gudgin, Marc Hadley, Noah Mendelsohn, Jean-Jacques Moreau, Henrik Frystyk Nielsen, Anish Karmarkar, and Yves Lafon. Soap version 1.2 part 1 : Messaging framework. http://www.w3.org/TR/soap12-part1/, 2002.

[GMK+09] Paul Grefen, Nikolay Mehandjiev, Giorgos Kouvas, Georg Weichhart, and Rik Eshuis. Dynamic business network process management in instant virtual enterprises. *Computers in Industry*, 60(2) :86 – 103, 2009.

[GYJ11] Siddarth Ganesan, Young Yoon, and Hans-Arno Jacobsen. NiÑos take five : the management infrastructure for distributed event-driven workflows. In *Proceedings of the 5th ACM international conference on Distributed event-based system*, DEBS '11, pages 195–206, New York, NY, USA, 2011. ACM.

[HA99] Claus Hagen and Gustavo Alonso. Beyond the black box : event-based inter-process communication in process support systems. In *Proceedings. 19th IEEE International Conference on Distributed Computing Systems*, number 303, pages 450–457. IEEE Comput. Soc, 1999.

[HLP05] B Hutchison, P Lambros, and R Phippen. The Enterprise Service Bus : Making service-oriented architecture real. *IBM Systems Journal*, 44(4) :781–797, 2005.

[HTL05] Colomble Hérault, Gaël Thomas, and Philippe Lalanda. Mediation and enterprise service bus : A position paper. In *Proceedings of the First International Workshop on Mediation in Semantic Web Services (MEDIATE 2005)*, pages 67–80. Citeseer, december 2005.

[HWS+06] Duncan Hull, Katy Wolstencroft, Robert Stevens, Carole Goble, Mathew R Pocock, Peter Li, and Tom Oinn. Taverna : a tool for building and running workflows of services. *Nucleic acids research*, 34(Web Server issue) :W729–W732, July 2006.

[IBM] IBM. WebSphere Application Server V7 Feature Pack for Service Component Architecture. http ://www-01.ibm.com/software/webservers/appserv/%20was/featurepacks/sca/. http://www-01.ibm.com/software/webservers/appserv/was/featurepacks/sca/.

[KAB+04] Martin Keen, Amit Acharya, Susan Bishop, Alan Hopkins, Sven Milinski, Chris Nott, Rick Robinson, Jonathan Adams, and Paul Verschueren. *Patterns : Implementing an SOA using an enterprise service bus*. IBM, International Technical Support Organization, july 2004.

[KL03] Rania Khalaf and Frank Leymann. On web services aggregation. In Boualem Benatallah and Ming-Chien Shan, editors, *Technologies for E-Services*, volume 2819 of *Lecture Notes in Computer Science*, pages 1–13. Springer Berlin / Heidelberg, 2003. 10.1007/978-3-540-39406-8-1.

[KM07] Jeff Kramer and Jeff Magee. Self-managed systems : an architectural challenge. In *proceedings ofFuture of Software Engineering (FOSE '07)*, pages 259–268, Los Alamitos, CA, USA, 2007. IEEE Computer Society.

[LAB+06] Bertram Ludäscher, Ilkay Altintas, Chad Berkley, Dan Higgins, Efrat Jaeger, Matthew Jones, Edward A. Lee, Jing Tao, and Yang Zhao. Scientific workflow management and the kepler system. *Concurrency and Computation : Practice and Experience*, 18(10) :1039–1065, 2006.

[LASS00] A. Lazcano, G. Alonso, H. Schuldt, and C. Schuler. The WISE approach to electronic commerce. *International Journal of Computer Systems Science and Engineering, special issue on Flexible Workflow Technology Driving the Networked Economy*, 15(5) :345–358, 2000.

[Ley05] Frank Leymann. The (service) bus : Services penetrate everyday life. In Boualem Benatallah, Fabio Casati, and Paolo Traverso, editors, *Service-Oriented Computing - ICSOC 2005*, volume 3826 of *Lecture Notes in Computer Science*, pages 12–20. Springer Berlin / Heidelberg, 2005. 10.1007/11596141-2.

[MLM+06] C.Matthew MacKenzie, Ken Laskey, Francis McCabe, Peter F. Brown, and Rebekah Metz. Reference model for service oriented architecture 1.0. http://docs.oasis-open.org/soa-rm/v1.0/, 2006.

[NCS04] Mangala Gowri Nanda, Satish Chandra, and Vivek Sarkar. Decentralizing execution of composite web services. In *Proceedings of the 19th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, OOPSLA '04, pages 170–187, New York, NY, USA, 2004. ACM.

[OAF+04] Tom Oinn, Matthew Addis, Justin Ferris, Darren Marvin, Martin Senger, Mark Greenwood, Tim Carver, Kevin Glover, Matthew R Pocock, Anil Wipat, and Peter Li. Taverna : a tool for the composition and enactment of bioinformatics workflows. *Bioinformatics (Oxford, England)*, 20(17) :3045–54, November 2004.

[PA05a] Cesare Pautasso and Gustavo Alonso. From web service composition to megaprogramming. *Technologies for E-Services*, (1820) :39–53, 2005.

[PA05b] Cesare Pautasso and Gustavo Alonso. JOpera : A Toolkit for efficient visual composition of web services. *International Journal of Electronic Commerce*, 9(2) :107–141, january 2005.

[Pap03] Mike P. Papazoglou. Service-Oriented Computing : Concepts, Characteristics and Directions. In *Proc. Fourth International Conference on Web Information Systems Engineering WISE 2003*, pages 3–12, Los Alamitos, CA, USA, 2003. IEEE Computer Society.

[par] The paremus service fabric - a technical overview. http://www.paremus.com/resources/resources-documents/The-Paremus-Service-Fabric---A-Technical-Overview.html.

[PD04] Michael P. Papazoglou and Jean-Jacques Dubray. A survey of web service technologies. Technical Report DIT-04-058, Ingegneria e Scienza dell'Informazione, University of Trento., 2004.

[PDE09] Gabriel Pedraza, Idrissa A Dieng, and Jacky Estublier. FOCAS : An Enginering Environment for Service-Based Applications. In *Proceedings of the the 4th int. conf. on Evaluation of novel approaches to software engineering (ENASE) 6-10 May 2009, Milan, Italie*, 2009.

[PE09] G Pedraza and J Estublier. Distributed Orchestration Versus Choreography : The FOCAS Approach. In *Trustworthy Software Development Processes : International Conference on Software Process, ICSP 2009 Vancouver, Canada, May 16-17, 2009 Proceedings*, page 75. Springer, 2009.

[Ped09] Gabriel Pedraza. Distributed Orchestration Versus Choreography : The FOCAS Approach. *Trustworthy Software Development Processes*, pages 75–86, 2009.

[Pel03] Chris Peltz. Web Services Orchestration and Choreography. *Computer*, 36(10) :46–52, 2003.

[PP06] Gilles Paché and Claude Paraponaris. L'entreprise en réseau : approches inter et intra-organisationnelles. Post-Print halshs-00009555, HAL, 2006.

[PTDL07] Michael P. Papazoglou, Paolo Traverso, Schahram Dustdar, and Frank Leymann. Service-oriented computing : State of the art and research challenges. *Computer*, 40 :38–45, 2007.

[RR01] P Robert and A Rey. *Le grand Robert de la langue francaise*. Dictionnaires Le Robert, 2001.

[SCA07a] Service Component Architecture Assembly Model V1.00. Technical report, SCA Consortium, 2007.

[SCA07b] Service Component Architecture Client and Implementation Model Specification for WS-BPEL. Version 1.00. Technical report, SCA Consortium, 2007.

[SDM02] Quan Z Sheng, Marlon Dumas, and Eileen Oi-yan Mak. SELF-SERV : A Platform for Rapid Composition of Web Services in a Peer-to-Peer Environment. *Proceedings of the 28th VLDB Conference, Hong Kong, China*, pages 1–4, 2002.

[SGM02] Clemens Szyperski, Dominik Gruntz, and Stephan Murer. *Component software : beyond object-oriented programming*. Addison-Wesley Professional, 2002.

[SMF+09] Lionel Seinturier, Philippe Merle, Damien Fournier, Nicolas Dolet, Valerio Schiavoni, and Jean-Bernard Stefani. Reconfigurable SCA Applications with the FraSCAti Platform. In *6th IEEE International Conference on Service Computing (SCC'09)*, pages 268–275, Bangalore, Inde, 2009. IEEE.

[SN96] R W Schulte and Y V Natis. SSA Research Note SPA-401- 068, Service oriented architectures, part 1 & 2, 1996.

[soa] Service Oriented Architecture For All - SOA4All website. www.soa4all.eu.

[SW04] D Sprott and L Wilkes. Understanding service-oriented architecture. *The Architecture Journal*, 1(1) :10–17, 2004.

[Tav04] Taverna User Manual. http://taverna.sourceforge.net/manual/docs.word.html, 2004.

[Tay98] David A. Taylor. *Object technology (2nd ed.) : a manager's guide*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1998.

[The07] The OSGi Alliance. OSGi service platform core specification, release 4.1. http://www.osgi.org/Specifications, 2007.

[tus] Apache Tuscany.

[TZ06] G. Tretola and E. Zimeo. Workflow fine-grained concurrency with automatic continuation. *Proceedings 20th IEEE International Parallel & Distributed Processing Symposium*, page 8 pp., 2006.

[VDT03] W.M.P. Van Der Aalst, Marlon Dumas, and A.H.M. Ter Hofstede. Web service composition languages : old wine in new bottles ? In *Euromicro Conference, 2003. Proceedings. 29th*, pages 298–305. IEEE, 2003.

[VOH+07] Asir S Vedamuthu, David Orchard, Frederick Hirsch, Maryann Hondo, Prasad webMethods Yendluri, Toufic Boubez, and Umit Yalçinalp. Web services policy 1.5-framework. *W3C Recommendation*, 4(September) :1–41, 2007.

[VRMCL08] Luis M. Vaquero, Luis Rodero-Merino, Juan Caceres, and Maik Lindner. A break in the clouds : towards a cloud definition. *SIGCOMM Comput. Commun. Rev.*, 39 :50–55, December 2008.

[vTKB03] W.M.P. van Der Aalst, A.H.M. Ter Hofstede, B Kiepuszewski, and A.P. Barros. Workflow patterns. *Distributed and parallel databases*, 14(1) :5–51, 2003.

[WFM99] WFMC. Workflow Management Coalition Terminology & Glossary. *Management*, (3) :1–65, 1999.

[WS01] Rainer Weinreich and Johannes Sametinger. *Component-based software engineering : putting the pieces together*, chapter Component Models and Component Services : Concepts and Principles., pages 33–48. Addison-Wesley Longman Publishing Co., Inc., 2001.

[YG07] Ustun Yildiz and Claude Godart. Information Flow Control with Decentralized Service Compositions. In *Web Services, 2007. ICWS 2007. IEEE International Conference on*, pages 9–17, July 2007.

[Yil08] Ustun Yildiz. *Decentralisation des procédés métiers : qualité de services et confidentialité*. PhD thesis, Université Henri Poincaré - Nancy 1, 2008.

[YP04] Jian Yang and Mike.P. Papazoglou. Service components for managing the lifecycle of service compositions. *Information Systems*, 29(2) :97 – 125, 2004. The 14th International Conference on Advanced Information Systems Engineering (CAiSE*02).

[YYR06] Jun Yan, Yun Yang, and Gitesh K Raikundalia. Swindew : a p2p-based decentralised workflow management system. *IEEE Transactions on System, Man and Cybernetics - Part A*, 36(5) :922 –935, September 2006.

Une approche à composant pour l'Orchestration de services à large échelle.

Résumé

Cette thèse s'intéresse à l'orchestration de services répartie, résultat (1) d'une approche explicite de découpage d'une orchestration en sous-orchestrations localisées sur des sites physiques distants à des fins de protection de données par exemple, ou (2) d'une approche constructive issue du regroupement d'orchestrations existantes potentiellement hétérogènes, afin de constituer une orchestration globale mais répartie. Les orchestrations de services reflètent des processus métiers, souvent de longue durée, et qui doivent donc pouvoir être adaptables dynamiquement à l'exécution. Cette thèse propose un support d'exécution pour des orchestrations réparties, hétérogènes, dynamiquement reconfigurables, et permettant une administration globale. Une orchestration de services peut être abordée selon ses deux dimensions : temporelle qui reflète l'enchainement des services dans le temps, spatiale qui reflète les services que l'orchestration a besoin d'invoquer afin de s'exécuter. Nous proposons ainsi un nouveau modèle à composants pour les applications orientées services, inspiré en partie de SCA et de SCA/BPEL, mais permettant de représenter ces deux dimensions. Notre approche se fonde sur un modèle de composants logiciels répartis et dynamiquement reconfigurables, et hérite donc des qualités de répartition et de reconfiguration dynamique. Nous décrivons une mise en œuvre au dessus de l'implémentation du modèle "Grid Component Model" sur la plateforme de programmation répartie à objets actifs "ProActive". Nous validons notre approche expérimentalement via une application à services d'installation et d'administration d'un parc de passerelles basées sur OSGi.

A component based approach for large-scale service orchestration

Abstract

This thesis focuses on the distributed orchestration of services, resulting (1) from an explicit decomposition of an orchestration into sub-orchestrations, located on physical remote sites to ensure protection of data for example, or (2) from a constructive approach whereby existing and possibly heterogeneous orchestrations are coupled in order to build a global and still distributed orchestration. Service orchestrations reflect business processes, sometimes lasting long, which thus have to be reconfigurable at runtime. This thesis introduces an execution support for distributed, heterogenous, dynamically reconfigurable orchestrations, while enabling a global management. A service orchestration can be considered along its two associated dimensions : temporal reflecting the chain of service invocations during time, spatial which makes it explicit which services are needed to be invoked in order for the orchestration to take place. We thus promote a new model, based upon a software component approach, for service oriented applications, inspired partly from SCA and SCA/BPEL, but allowing to represent and manipulate these two dimensions at once. Our approach grounds upon a model for software components which are distributed and dynamically adaptable, thus inheriting from these qualities of distribution and dynamic reconfiguration. We describe an implementation of our model using the "Grid Component Model" reference implementation on top of the distributed active objects library named "ProActive". We validate experimentally our approach through a service-based application for the installation and management of a park of OSGi gateways.