Architecture logicielle pour l'adaptation distribuée : Application à la réplication de données

Mohamed Zouari

▶ To cite this version:

Mohamed Zouari. Architecture logicielle pour l'adaptation distribuée : Application à la réplication de données. Génie logiciel [cs.SE]. Université Rennes 1, 2011. Français. NNT : . tel-00652046

HAL Id: tel-00652046
https://theses.hal.science/tel-00652046

Submitted on 15 Dec 2011

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.

N° d'ordre : 4334 ANNÉE 2011

THÈSE / UNIVERSITÉ DE RENNES 1
sous le sceau de l'Université Européenne de Bretagne

pour le grade de

DOCTEUR DE L'UNIVERSITÉ DE RENNES 1

Mention : Informatique

Ecole doctorale Matisse

présentée par

Mohamed ZOUARI

préparée à l'unité de recherche IRISA
IRISA/MYRIADS
IFSIC

Intitulé de la thèse : Architecture logicielle pour l'adaptation distribuée : application à la réplication de données

Thèse soutenue à Rennes le 28 juin 2011

devant le jury composé de :

Philippe ROOSE
Maître de Conférences HdR, Université de Pau et des Pays de l'Adour / rapporteur

Lionel SEINTURIER
Professeur, Université Lille 1 / rapporteur

Khalil DRIRA
Directeur de Recherche, LAAS-CNRS / examinateur

Jean-Louis PAZAT
Professeur, Institut National des Sciences Appliquées de Rennes / examinateur

Antoine BEUGNARD
Professeur, Télécom Bretagne / invité

Françoise ANDRÉ
Professeur, Université de Rennes 1 / directrice de thèse

Maria-Teresa SEGARRA
Maître de Conférences, Télécom Bretagne / encadrant de thèse

Remerciements

Mes remerciements s'adressent d'abord aux membres de mon jury qui m'ont fait l'honneur de participer à ma soutenance de thèse. Je remercie particulièrement les rapporteurs M. Philippe Roose et M. Lionel Seinturier pour l'intérêt qu'ils ont montré pour ce travail et leurs remarques constructives. Merci à M. Antoine Beugnard, M. Khalil Drira et M. Jean-Louis Pazat d'avoir accepté d'examiner mon travail.

J'exprime mes profonds remerciements à ma directrice de thèse Françoise André et à mon encadrant Maria-Teresa Segarra sans qui rien de tout cela ne serait arrivé. Merci pour leur encadrement, leur soutien et leurs conseils tout au long de la thèse.

Je tiens à remercier les responsables de l'équipe MYRIADS (ex PARIS), Mme Christine Morin et M. Thierry Priol, de m'avoir accueilli dans l'équipe. Merci pour les excellentes conditions de travail qu'ils m'ont offertes pendant les trois années de thèse.

Je remercie l'ensemble des membres de l'équipe MYRIADS et mes amis dans les équipes de recherche KERDATA, ADEPT et ATNET pour les bons moments passés ensemble à l'IRISA/INRIA Rennes - Bretagne Atlantique.

Je remercie également Mme Annie Gravey le responsable du département informatique de Télécom Bretagne et les membres du départent de m'avoir accueilli pendant les derniers mois de mon travail de thèse à Brest.

Je tiens à adresser ma gratitude à ma famille (mes parents, ma sœur et mon frère) qui m'a toujours soutenu.

Finalement, un grand merci à tous mes amis qui m'ont aidé de près ou de loin à finaliser mon travail de thèse.

Résumé

L'adaptation dynamique permet de modifier une application en cours d'exécution en fonction des fluctuations de son environnement et des changements des exigences des utilisateurs. De nombreux travaux ont proposé des méthodes et mécanismes pour adapter une application centralisée. Mais, le cas des applications distribuées a été beaucoup moins abordé. En particulier, la distribution du système d'adaptation lui-même est très peu envisagée. Nous nous intéressons à ce contexte, en proposant un système d'adaptation comme un ensemble d'entités logicielles coopérantes. Chaque entité contrôle l'adaptation d'un groupe de composants de l'application et peut coordonner ses activités avec ses homologues afin de prendre en compte les dépendances entre les composants applicatifs. L'ensemble des entités doit être capable de prendre des décisions d'adaptation cohérentes de manière collective en résolvant les conflits éventuels. De plus, il peut être nécessaire de coordonner le contrôle des modifications de l'application répartie réalisées en parallèle. Quelques travaux ont abordé ces sujets sans toutefois proposer des mécanismes de coordination flexibles et réutilisables associés à des facilités pour spécialiser le système d'adaptation en fonction de l'application cible. Afin de pallier ces manques, nous proposons dans cette thèse une approche visant à définir une architecture logicielle à base de composants pour permettre la gestion distribuée et coordonnée de l'adaptation dynamique d'applications. Nous définissons un modèle d'architecture logicielle de systèmes d'adaptation qui permet la variabilité des configurations du système et qui inclut des mécanismes spécialisables pour assurer la coordination. Un expert en adaptation de l'application explore l'espace composé des configurations du système d'adaptation pouvant être définies et détermine une solution appropriée. Il exprime cette solution sous la forme d'une description de l'architecture logicielle du système d'adaptation souhaité et la fournit à une fabrique qui se charge de créer effectivement le système.

Nous avons choisi la gestion de données répliquées pour illustrer un domaine d'application de notre approche d'adaptation. Nous définissons un modèle d'architecture logicielle d'un système de réplication de données à base de composants qui structure les fonctions d'un tel système. L'apport du modèle proposé est de supporter une grande variabilité de configurations possibles, de permettre la spécialisation du système de réplication initial en fonction des besoins et de permettre l'adaptation dynamique fine du système. La spécialisation du modèle s'effectue par un expert en réplication de données. Elle reflète la solution de réplication envisagée par l'expert pour l'application qui manipule les données et inclut des interfaces de contrôle pour observer et agir sur les composants constituant le système de réplication. Le modèle ainsi spécialisé est fourni ensuite à une fabrique qui l'utilise pour réifier le système. Enfin, ce dernier est associé à un système d'adaptation pour le rendre auto-adaptable.

Pour valider notre approche, nous avons développé un prototype pour la construction de systèmes d'adaptation distribués d'une part et de systèmes de réplication auto-adaptables d'autre part. Le prototype, qui se base sur le modèle de composants Fractal, nous a permis de mener quelques expérimentations d'adaptation distribuée sur un système de réplication de données en milieu médical pour le suivi d'un patient à domicile.

Mots-clés : gestion distribuée d'adaptation dynamique, mécanismes de coordination de décision et d'exécution d'adaptation, architecture logicielle, réplication de données, composant logiciel.

Abstract

Dynamic adaptation allows the modification of an application during its execution, according to fluctuations in its environment and changes in users' requirements. Several studies have proposed methods and mechanisms to adapt a centralized application. But the case of distributed applications has not been addressed substantially. In particular, the distribution of the adaptation system itself has been rarely considered. We address this context by proposing an adaptation system as a set of cooperative software entities. Each entity controls the adaptation of a group of the application components and may coordinate its activities with its counterparts in order to take into account the dependencies between these components. These entities must be able to make collective and consistent decisions regarding adaptation and to solve the potential conflicts that may arise among them. In addition, it may be necessary to coordinate the control of the parallel modifications of the distributed application. Some works have addressed these issues without proposing reusable and flexible coordination mechanisms associated with facilities to specialize the adaptation system according to the targeted application. To overcome these problems, we propose an approach in this thesis to define a component-based software architecture supporting the distributed and coordinated management of dynamic adaptation. We define a software architectural model of adaptation systems that allows the variability of the system configurations and includes coordination mechanisms that can be specialized. An adaptation and application expert explores the space composed of the alternative adaptation system configurations that can be defined and determines an appropriate solution. He formulates this solution as a description of the software architecture of the required adaptation system and provides it to a factory that is responsible for concretely creating the system.

We have chosen the management of replicated data to illustrate an application domain for our adaptation approach. We define a software architectural model of component-based replication systems that structures the functions of the system. The contribution of the proposed model is to support a wide variability of possible configurations, while enabling the specialization of the initial replication system according to the needs and to allow fine-grained dynamic adaptation of the system. The model specialization is performed by an expert in data replication. The specialized model is then provided to a factory that uses it to reify the system. Finally, the replication system is associated with an adaptation system in order to make it self-adaptable.

To validate our approach, we developed a prototype for the construction of distributed adaptation systems and replication systems. The prototype based on Fractal component model has enabled us to make experimentations on the distributed adaptation of a data replication system, used in a medical environment dedicated to remotely monitor patients at home.

Keywords : distributed management of dynamic adaptation, coordination mechanisms for adaptation decision making and execution, software architecture, data replication, software component.

Table des matières

1 Introduction **9**
1.1 Contexte et problématique . . . 9
1.2 Objectifs et contributions . . . 10
1.3 Organisation du manuscrit . . . 11

2 Adaptation d'applications distribuées **13**
2.1 Introduction . . . 13
2.2 Principes de l'adaptation . . . 13
2.2.1 Notion d'adaptation . . . 13
2.2.2 Moments d'adaptation . . . 14
2.2.3 Types d'adaptation . . . 14
2.2.4 Phases d'un processus d'adaptation . . . 15
2.3 Centralisation/distribution de la gestion d'adaptation d'entités multiples . . 16
2.3.1 Gestion centralisée d'adaptation . . . 17
2.3.2 Limites de la solution centralisée pour les applications distribuées . . 19
2.3.3 Enjeux de la gestion distribuée de l'adaptation . . . 19
2.3.4 Coordination pour la gestion distribuée de l'adaptation . . . 20
2.3.5 Étude de travaux existants avec gestion distribuée de l'adaptation . 22
2.3.6 Synthèse . . . 29
2.4 Spécialisation des infrastructures d'adaptation . . . 30
2.4.1 Spécialisation de comportement . . . 30
2.4.2 Spécialisation de la répartition des mécanismes d'adaptation . . . 31
2.4.3 Synthèse . . . 32
2.5 Conclusion . . . 32

3 Adaptabilité des systèmes de réplication **35**
3.1 Introduction . . . 35
3.2 La gestion du placement de répliques . . . 35
3.2.1 Terminologie . . . 36
3.2.2 Le problème de placement de répliques . . . 36
3.2.3 Adaptation et heuristiques . . . 37
3.2.4 Synthèse . . . 38
3.3 La sélection de répliques . . . 38
3.3.1 Stratégies de sélection de répliques . . . 38
3.3.2 Adaptation et stratégies de sélection de répliques . . . 39
3.3.3 Synthèse . . . 40
3.4 La gestion de la cohérence . . . 40
3.4.1 Protocoles de gestion de la cohérence . . . 41
3.4.2 Adaptation et protocoles de gestion de la cohérence . . . 44

3.4.3 Synthèse 47
3.5 Conclusion 47

4 Modèle d'architecture pour l'adaptation distribuée et coordonnée 49
4.1 Introduction 49
4.2 Notations et terminologie 49
4.2.1 Composant logiciel et modèle architectural 49
4.2.2 Modélisation d'une architecture logicielle 50
4.2.3 Description d'un modèle architectural 52
4.3 Principes de construction d'applications auto-adaptables 54
4.4 Fonctions pour l'adaptation dynamique distribuée 55
4.4.1 Observation et reconfiguration 56
4.4.2 Gestion du contexte et d'adaptation 56
4.4.3 Coordination 57
4.5 Modèle architectural pour la gestion distribuée de l'adaptation dynamique . 57
4.5.1 Types de composants pour la gestion de l'adaptation 57
4.5.2 Distribution de la gestion de l'adaptation dynamique 58
4.5.3 Modèle architectural du gestionnaire de contexte 61
4.5.4 Modèle architectural du gestionnaire d'adaptation 62
4.5.5 Spécialisation de la gestion d'adaptation 64
4.5.6 Coordination des activités de gestionnaires d'adaptation 65
4.5.7 Protocole de négociation de stratégies d'adaptation 74
4.6 Conclusion 77

5 Modèle d'architecture pour des systèmes de réplication de données adaptables 79
5.1 Introduction 79
5.2 Cas d'étude : gestion de données répliquées pour des services de télémédecine 80
5.2.1 Scénario d'utilisation : services de télémédecine 80
5.2.2 Système de réplication de données auto-adaptable pour les services de télémédecine 81
5.3 Principes de développement de systèmes de réplication auto-adaptables . . . 82
5.4 Fonctions d'un système de réplication de données 83
5.4.1 La gestion du placement des répliques 83
5.4.2 L'interrogation des données 83
5.4.3 Le contrôle d'accès aux répliques 84
5.4.4 La manipulation des répliques 84
5.4.5 La propagation des mises à jour 84
5.4.6 Le contrôle de la concurrence 85
5.5 Modèle architectural d'un système de réplication 85
5.5.1 Types de composants du modèle 85
5.5.2 Points de variation du modèle architectural 88
5.6 Cas d'étude : exemple de spécialisation du modèle architectural 93
5.6.1 Répartition de composants d'un système de réplication 93
5.6.2 Comportement du système 94
5.6.3 Spécialisation d'un système d'adaptation 98
5.7 Conclusion 99

6 Implémentation et expérimentation 101
6.1 Introduction 101
6.2 Le modèle de composants Fractal et les outils associés 101
6.2.1 Le modèle de composants Fractal 101
6.2.2 Julia : une implémentation de Fractal 103
6.2.3 Fractal ADL : le langage de description d'architecture de Fractal . . 104
6.2.4 Fractal RMI 106
6.3 Notre prototype 106
6.3.1 Implémentation du modèle architectural d'adaptation 107
6.3.2 Implémentation du modèle architectural de réplication 111
6.3.3 Fabrique pour construire des systèmes de réplication auto-adaptables 119
6.4 Évaluation de performances 123
6.4.1 Plateforme expérimentale 123
6.4.2 Comparaison des approches de gestion d'adaptation : centralisée et distribuée 124
6.4.3 Gains et coûts de l'adaptation de la gestion de données répliquées . . 126
6.5 Conclusion 127

7 Conclusion et perspectives 129
7.1 Résumé des contributions 129
7.2 Perspectives 130

Table des figures

2.1 Structure d'un composant adaptable basé sur [CHS01] 23
2.2 Deux composants composites connectés dans deux instances K-Component différentes [DC04] 27
2.3 Caractéristiques des travaux étudiés 29

4.1 Description d'un modèle architectural pour une famille d'applications 51
4.2 Notation graphique des connecteurs UML 2.3 51
4.3 Notation graphique d'un composant logiciel 52
4.4 Multiplicités possibles d'un type de composant 53
4.5 Notation graphique d'un type de composant 54
4.6 Principes de construction d'un système d'adaptation 55
4.7 Fonctions pour l'adaptation dynamique distribuée 56
4.8 Exemple d'un composant auto-adaptable 58
4.9 Modèle architectural pour une gestion distribuée d'adaptation 59
4.10 Exemple de distribution de la gestion d'adaptation dynamique 60
4.11 Modèle architectural d'un gestionnaire de contexte 61
4.12 Modèle architectural d'un gestionnaire d'adaptation 63
4.13 Classe UML représentant un type de stratégies d'adaptation d'un composant de placement de répliques 64
4.14 Classe UML représentant un plan d'adaptation de stratégie de placement de répliques 65
4.15 Modèle architectural pour des décideurs coopératifs 68
4.16 Classe UML et l'objet associé définissant une collection d'actions ordonnées d'un plan d'adaptation coordonné 71
4.17 Modèle architectural pour des exécuteurs coopératifs 72
4.18 Exemple de règles de coordination d'exécution de plans 74
4.19 Classe UML représentant les attributs d'un contrat négocié 75
4.20 Diagramme de séquence de négociation entre deux gestionnaires d'adaptation 76

5.1 Environnement d'exécution des services de télémédecine considérés 81
5.2 Approche pour construire un système de réplication auto-adaptable 82
5.3 Fonctions d'un système de réplication 84
5.4 Modèle architectural d'un système de réplication de données 85
5.5 Exemple de classification de données et utilisateurs 89
5.6 Structure interne d'un composant primitif à comportement variable 91
5.7 Contraintes sur l'instanciation de composants d'un système de réplication . 92
5.8 Exemple de répartition des composants d'un système de réplication 94
5.9 Diagramme de séquence de réplication d'une donnée 95
5.10 Diagramme de séquence d'accès à une donnée 96

5.11 Diagramme de séquence de gestion de cohérence suite à une lecture/écriture d'une donnée . . . 97
5.12 Exemple de répartition des composants d'un système d'adaptation . . . 98

6.1 Représentation graphique d'un composite *HelloWorld* . . . 102
6.2 Julia : l'implantation de référence en Java du modèle Fractal . . . 103
6.3 Exemple de description de composants en Fractal ADL . . . 105
6.4 Architecture de l'usine de Fractal ADL . . . 106
6.5 Description des types d'interfaces fonctionnelles d'un composant de type négociateur . . . 109
6.6 Définition d'un composant de type négociateur en Fractal ADL . . . 110
6.7 Spécialisation du comportement d'un négociateur . . . 111
6.8 Exemple d'une règle de négociation d'adaptation du protocole de cohérence 112
6.9 Interface fournie par un contrôleur de configuration . . . 113
6.10 Description de la partie de contrôle d'un gestionnaire de placement . . . 114
6.11 Interface fournie par un contrôleur d'observation . . . 115
6.12 Interface fournie par un contrôleur de modification . . . 116
6.13 Opérations primitives d'intercession dans Fractal . . . 117
6.14 Opérations fournies par un contrôleur de configuration pour la classification 118
6.15 Opérations de base pour construire un système d'adaptation . . . 120
6.16 Opérations de base pour construire un système de réplication . . . 121
6.17 Architecture de la fabrique . . . 122
6.18 Temps de prise de décision avec une configuration centralisée vs. distribuée . 125
6.19 Temps d'exécution avec une configuration centralisée vs. distribuée . . . 126

Chapitre 1

Introduction

1.1 Contexte et problématique

Les environnements d'exécution des applications sont soumis généralement à des variations importantes. En effet, la disponibilité des ressources informatiques varie dans le temps et dans l'espace, par exemple la connexion réseau disponible pour un utilisateur mobile peut se rompre. Les conditions d'utilisation d'un service peuvent également varier. Ainsi, les utilisateurs peuvent avoir des terminaux hétérogènes ayant des capacités différentes pour interagir avec une application. De plus, certains services ou ressources peuvent devenir non disponibles en raison des défaillances du réseau d'interconnexion et des machines. Par ailleurs, les utilisateurs ont aussi des profils et des préférences différents et leurs exigences envers les services fournis peuvent évoluer au cours du temps. Il est donc fortement souhaitable que les applications prennent en compte les variations du contexte d'exécution pour assurer la continuité des services et satisfaire au mieux les exigences de leurs utilisateurs.

Les travaux dans le domaine de l'adaptation dynamique visent à permettre la modification d'une application en cours d'exécution. Certains de ces travaux ont permis de structurer les fonctionnalités nécessaires pour la gestion de l'adaptation. Généralement, le système d'adaptation surveille le contexte d'exécution, prend la décision d'adaptation suite à un changement significatif et modifie l'application en conséquence. En particulier, de nombreuses approches se sont intéressées à l'adaptation d'applications à base de composants logiciels. En effet, le développement fondé sur les composants assure la modularité et favorise la dynamicité de l'application par l'ajout, la suppression et le remplacement des composants applicatifs pour s'adapter. De plus, certaines approches ont défini des canevas logiciels pour faciliter et guider la conception et la mise en œuvre d'un système d'adaptation.

Dans ce travail, nous visons des applications distribuées à base de composants. De ce fait, chacune de leurs fonctionnalités est mise en place par un ensemble de composants qui peuvent être distribués à travers un réseau. Dans ce contexte, les fonctionnalités sont différentes entre elles notamment en termes de quantité et de types de ressources requises et d'exigences qu'elles doivent satisfaire (disponibilité, temps d'exécution, qualité de la réponse...). De plus, l'environnement d'exécution d'une application distribuée peut être très hétérogène et encore plus variable que celui d'une infrastructure centralisée. Ainsi, au sein d'une fonctionnalité un groupe de composants peut être sensible à des informations contextuelles particulières et nécessiter des adaptations différentes des autres. Par exemple, un premier groupe de composants peut être sensible aux préférences de l'utilisateur et un deuxième groupe à la quantité des ressources disponible sur les machines qui les hébergent.

L'adaptation du premier peut consister à changer les comportements des composants alors que celle du deuxième peut être la migration de certains composants. Il est donc intéressant de distribuer la gestion de l'adaptation dynamique sur plusieurs entités logicielles de sorte que chacune soit responsable du contrôle de l'adaptation d'un groupe de composants de l'application. Cependant, cette distribution rend nécessaire la coordination des activités de ces entités lorsque le choix ou l'application d'une adaptation nécessite de prendre en compte des dépendances entre les composants applicatifs comme le taux d'utilisation des ressources partagées.

Certaines approches ont suivi ce principe pour adapter des applications distribuées. Elles ont défini des systèmes d'adaptation distribués et elles ont distribué certaines fonctions pour la gestion de l'adaptation. Cependant, ces travaux sont souvent très partiels et résolvent des cas particuliers. Par exemple, nous constatons que les systèmes existants ne proposent pas de techniques pour prendre des décisions d'adaptation de façon distribuée et pour résoudre les conflits éventuels. Par exemple, suite à des changements du contexte d'exécution, des choix de comportements incompatibles des composants peuvent être faites. De plus, ils n'offrent pas de mécanismes génériques et réutilisables pour coordonner le contrôle distribué des modifications des composants applicatifs.

En ce qui concerne la construction de systèmes d'adaptation, la mise en place de leurs composants est souvent une tâche manuelle. Cette tâche peut être complexe en fonction du niveau d'abstraction utilisé pour leur construction. Elle est d'autant plus complexe que les composants peuvent être distribués. Par ailleurs, la variabilité des configurations des systèmes d'adaptation distribués est un aspect peu pris en compte. Les solutions actuelles facilitent la spécialisation du comportement du système d'adaptation en suivant une approche orientée politique pour l'activité de prise de décision d'adaptation. Mais, la spécialisation ne concerne pas les mécanismes de coordination. De plus, à notre connaissance, aucun travail ne s'est consacré à la prise en compte de la spécialisation de la structure et de la distribution d'un système d'adaptation. Le travail de mise en place manuelle d'un système d'adaptation distribué reste donc considérable.

Un système de réplication de données est un exemple d'application distribuée qui peut bénéficier des apports de l'adaptation dynamique. Les solutions de réplication existantes sont généralement spécifiques à des applications particulières. Cependant, certaines approches supportent la flexibilité et l'adaptabilité des systèmes de réplication. Des canevas et des systèmes spécialisables ont ainsi été définis afin de faciliter la construction de tels systèmes. Ils reposent principalement sur le paramétrage pour configurer un système de réplication ou sur l'injection de traitements spécifiques en fonction des besoins applicatifs. L'adaptation dynamique de la gestion de la réplication a été abordée dans certains travaux. Cependant, la prise en compte du contexte d'exécution est réduite et la variabilité des configurations d'un système de réplication est limitée. De plus, il existe souvent un entrelacement des préoccupations de réplication avec celles d'adaptation ce qui diminue la généricité des solutions et nuit à leur réutilisabilité.

1.2 Objectifs et contributions

Le sujet de cette thèse est d'étudier les spécificités de la gestion de l'adaptation dynamique dans le contexte des applications distribuées. L'étude réalisée nous a permis d'identifier un ensemble de besoins et de défis de la distribution de la gestion d'adaptation. Nous avons comme premier objectif de proposer une approche définissant une architecture logicielle qui permet la distribution et la coordination des activités nécessaires à la gestion de l'adaptation de composants applicatifs distribués.

Notre deuxième objectif est de fournir un ensemble d'outils pour la construction d'un système d'adaptation distribué afin de réduire les efforts de sa conception, de sa spécialisation et de sa mise en œuvre. Cette thèse contribue à la structuration de la gestion distribuée de l'adaptation dynamique, tant sur le plan de l'architecture logicielle que sur celui des techniques de coordination des activités de gestion d'adaptation. Ainsi, nous définissons un modèle d'architecture logicielle de systèmes d'adaptation à base de composants. Les systèmes d'adaptation construits en se basant sur notre modèle sont constitués d'entités logicielles, chacune responsable du contrôle de l'adaptation d'un sous ensemble de composants de l'application. Ces entités pouvant être distribuées, nous définissons un ensemble de mécanismes réutilisables pour la coordination de leurs activités de prise de décision d'adaptation et de modification de l'application. Ces mécanismes implantent des traitements communs qu'il est possible de spécialiser par des politiques externes.

Un expert en adaptation spécialise notre modèle d'architecture logicielle pour définir son système d'adaptation selon les spécificités de l'application à adapter et de son environnement. Pour cela, il fournit une description de l'architecture souhaitée. Puis, une fabrique prend en entrée cette description et notre modèle d'adaptation afin de mettre en place le système d'adaptation souhaité par l'expert et le connecter à l'application cible.

Dans un deuxième axe, nous nous sommes intéressés à la flexibilité et à la variabilité des configurations d'une application distribuée. Ceci a été étudié dans le cadre des systèmes de réplication de données. Notre objectif est d'exprimer et de permettre la variabilité qui peut exister entre des configurations alternatives d'un système de réplication (le comportement, la structure et la distribution). Le choix de la configuration appropriée du système peut être réalisé de manière statique ou dynamique pour adapter le système à son contexte d'exécution.

Dans cette optique, nous avons spécifié un modèle d'architecture logicielle de systèmes de réplication. Ce modèle exprime la variabilité au niveau des composants d'un système de réplication et de leurs connexions ainsi que les interfaces pour surveiller et modifier ces composants. Un expert en réplication spécialise notre modèle sous forme d'une description d'architecture nécessaire pour définir son système de réplication en fonction des besoins. Comme pour le système d'adaptation, la fabrique met en place le système de réplication concret à partir de la description fournie par l'expert et de notre modèle de réplication.

1.3 Organisation du manuscrit

Outre l'introduction, ce rapport de thèse est composé de six chapitres : Deux chapitres introduisent le contexte dans lequel se place cette thèse et dressent un état de l'art des systèmes d'adaptation dynamique d'une part et de l'adaptabilité des systèmes de gestion d'objets répliqués d'autre part. Les trois chapitres suivants présentent notre approche pour construire des systèmes d'adaptation distribués, celle concernant les systèmes de réplication de données auto-adaptables, puis leur mise en œuvre et évaluation. Le dernier chapitre souligne les contributions de cette thèse et introduit les perspectives du travail.

Plus précisément, en ce qui concerne les chapitres centraux :

- Le chapitre 2 *« Adaptation d'applications distribuées »* introduit les principes de l'adaptation et les approches de gestion d'adaptation qui sont en relation avec le contexte scientifique dans lequel se déroule cette thèse. Il met en relief des limites des approches actuelles pour adapter des applications distribuées. Nous concluons ce chapitre en présentant les besoins et les défis de la gestion d'adaptation et de la spécialisation des systèmes d'adaptation dans le cadre des applications distribuées.
- Le chapitre 3 *« Adaptabilité des systèmes de réplication »* présente les principes de la

gestion d'objets répliqués et étudie l'adaptabilité des mécanismes existants. Il précise les avantages et limitations de travaux du domaine par rapport à leur prise en compte de la fluctuation des environnements d'exécution et la variation des exigences des utilisateurs. Ce chapitre montre les besoins d'un modèle de système de réplication générique et flexible et d'améliorer les mécanismes d'adaptation de systèmes de réplication.

- Le chapitre 4 *« Modèle d'architecture pour l'adaptation distribuée et coordonnée »* met en évidence les propriétés de notre proposition pour répondre aux besoins et aux défis d'adaptation des applications distribuées. Dans ce chapitre, nous présentons nos principes pour adapter les applications distribuées. Puis, nous détaillons notre modèle d'architecture logicielle d'adaptation qui définit des mécanismes spécialisables pour gérer l'adaptation dynamique de façon distribuée et coordonnée.
- Le chapitre 5 *« Modèle d'architecture pour des systèmes de réplication de données adaptables »* décrit notre approche pour spécialiser et adapter dynamiquement des systèmes de réplication de données. Dans ce chapitre, nous présentons notre modèle d'architecture de systèmes de réplication et la manière dont il supporte la variabilité de configuration. Nous présentons aussi nos principes pour construire un système de réplication auto-adaptable.
- Le chapitre 6 *« Implémentation et expérimentation »* présente la mise en œuvre de nos modèles d'adaptation dynamique et de réplication ainsi que de la fabrique permettant l'instanciation de systèmes concrets. Nous décrivons également des expérimentations réalisées sur le prototype que nous avons implémenté.

Chapitre 2

Adaptation d'applications distribuées

2.1 Introduction

Les applications distribuées s'exécutent de plus en plus souvent dans des environnements fluctuants comme les environnements mobiles, les clusters de machines et les grilles de processeurs. Elles doivent cependant continuer à s'exécuter quelles que soient les conditions et offrir des services de bonne qualité. Les tâches de reconfiguration pour s'accommoder des changements mènent souvent à des procédures coûteuses du point de vue de l'effort humain et du temps de réalisation. C'est pourquoi il y a un besoin de faciliter ces tâches et de définir des solutions pour les automatiser.

L'état de l'art sur l'adaptation dynamique met en évidence une diversité d'architectures et de mécanismes qui se sont intéressés à différents types d'applications, de technologies et de préoccupations. Ce chapitre s'intéresse aux concepts et aux mécanismes de la gestion de l'adaptabilité des logiciels et aux travaux existant dans ce domaine. Il commence par présenter la notion d'adaptation. Ensuite, il détaille des aspects reliés à la gestion de l'adaptation et analyse un ensemble de travaux qui permettent la gestion distribuée de l'adaptation dynamique. Finalement, les possibilités de spécialisation des infrastructures permettant de construire un système d'adaptation sont étudiées.

2.2 Principes de l'adaptation

2.2.1 Notion d'adaptation

Dans le domaine des systèmes logiciels, une *adaptation* signifie un changement dans le système pour prendre en compte un changement dans son environnemen d'exécution [LR00]. L'*adaptabilité* désigne alors la capacité du système à effectuer de l'adaptation [SC01]. En effet, les administrateurs déploient une même application dans des environnements totalement différents. Les différences incluent les ressources disponibles, les conditions d'utilisation de l'application et les préférences des utilisateurs. Ainsi, il devient indispensable d'adapter l'application afin de garantir une bonne qualité des services qu'elle propose. Par ailleurs, le contexte évolue tout au long de l'exécution de l'application. Par exemple, les changements au niveau de l'environnement d'exécution (manque de ressources, partition du réseau...) risquent d'avoir des conséquences indésirables sur le comportement de l'application qui entraînent une dégradation de la qualité du service ou même une rupture de la continuité du service. Ainsi, l'adaptation dynamique présente une solution appropriée pour résoudre les problèmes pouvant apparaître lors de l'exécution d'une application.

Certains systèmes sont caractérisés par la propriété d'auto-adaptation. Parmi les définitions existantes de *système auto-adaptable*, une est donnée par Oreizy et al. [OGT+99] : Un logiciel auto-adaptable modifie son propre comportement en réponse aux changements dans son environnement d'exécution. L'environnement d'exécution fait référence à tout ce qui peut être observé par le système logiciel comme les entrées de la part d'un utilisateur, les périphériques externes et les capteurs, ou l'instrumentation du programme. Dans une deuxième définition que nous utiliserons par la suite, [Lad00] considère qu'un logiciel auto-adaptable évalue son propre comportement et change celui-ci lorsque l'évaluation indique qu'il n'accomplit pas ce qu'il est destiné à faire, ou quand une meilleure fonctionnalité ou performance est possible.

2.2.2 Moments d'adaptation

Le moment de la réalisation de l'adaptation durant le cycle de vie d'un logiciel est un critère important pour classifier les travaux reliés à l'adaptation. Cette adaptation de l'application peut être réalisée *au moment de son développement*. Dans ce cas, les concepteurs adaptent l'application selon l'environnement d'exécution visé et fixent les situations dans lesquelles elle va opérer. L'adaptation peut aussi se faire *au moment du déploiement*. Dans ce cas, un ensemble de tâches sont effectuées en tenant compte de l'environnement cible comme la configuration des composants de l'application et la mise en place des connexions entre eux. Par exemple, différentes compositions peuvent répondre aux exigences des différentes plateformes d'exécution ou types de réseaux. Finalement, l'adaptation peut avoir lieu *au moment de l'exécution* grâce à des mécanismes qui permettent à l'application de s'ajuster automatiquement selon le contexte d'exécution.

En général, deux approches principales ont été identifiées par les chercheurs pour la mise en œuvre de logiciels adaptables, à savoir l'adaptation statique et l'adaptation dynamique [MSKC04]. L'*adaptation statique* se réfère aux méthodes d'adaptation qui ont lieu avant ou pendant le déploiement, tandis que l'*adaptation dynamique* inclut les méthodes qui s'appliquent pendant l'exécution. Dans le cas d'adaptation statique, le changement du comportement exige la recompilation et/ou le redéploiement de l'application ou d'un ensemble de ses composants. L'adaptation dynamique permet de modifier la structure de l'application et de faire évoluer les algorithmes qu'implantent les composants pendant l'exécution et sans avoir à tout redémarrer. L'adaptation dynamique donne donc plus de garanties pour satisfaire les exigences envers l'application de manière continue. Cependant, la gestion de l'adaptation est dans ce cas plus complexe puisque cela nécessite de surveiller constamment l'environnement, de décider « à chaud » de la stratégie d'adaptation et de modifier l'application pendant son fonctionnement.

Les deux types d'adaptation, statique et dynamique, sont intéressants dans le cadre des logiciels distribués. En effet, l'adaptation statique permet de prendre en compte les caractéristiques de nature des environnements d'exécution comme l'utilisation d'un réseau fixe ou mobile. L'adaptation dynamique est indispensable pour traiter les changements du contexte d'exécution comme la montée de charge. Le moment d'adaptation sera un des critères pour classifier les travaux qui ont défini des systèmes de réplication adaptables dans le chapitre suivant. Dans la suite de ce chapitre, nous allons nous intéresser particulièrement aux mécanismes d'adaptation dynamique.

2.2.3 Types d'adaptation

Un autre point de variation concerne les types de modifications supportés par chaque logiciel adaptable. Une *configuration du système à adapter* est définie comme un ensemble

d'entités logicielles reliées les unes aux autres [Weg03]. En effet, chaque système logiciel se décompose en objets, composants, services selon son architecture et la technologie utilisée pour sa mise en œuvre. Chacune de ces entités peut être le sujet d'adaptation (attributs, méthodes, composition...). Dans la suite de ce chapitre, le terme « composant » sera utilisé au sens large et ne cible pas exclusivement les approches orientées composants.

Nous distinguons plusieurs types de modifications possibles de l'application dans les travaux existants [KBC02, AC03]. Généralement, la modification concerne les paramètres de configuration, l'implémentation, les interfaces, la géométrie et la structure de l'application.

- Modification de paramètres [YV02, ZZ03] : Ce type d'adaptation consiste à modifier les valeurs de variables qui influent sur le comportement de l'application sans changer les algorithmes.
- Modification d'implémentation [CHS01, EFDC02, CEM03] : Cette adaptation permet de modifier l'implémentation du composant (le code du composant) sans changer ni les interfaces ni les connexions avec les autres composants de l'application.
- Modification d'interfaces [Hei00] : Chaque composant de l'application fournit des services à travers des interfaces bien définies. L'adaptation d'interfaces consiste à modifier les services fournis par un composant en changeant ses interfaces.
- Modification de géométrie [TS09] : L'adaptation de géométrie modifie la distribution géographique d'une application. En effet, ce type d'adaptation impacte uniquement l'emplacement des composants qui peuvent changer de localisation en migrant d'une machine vers une autre. Cette adaptation est appelée aussi adaptation de localisation ou encore la migration.
- Modification de structure [GCH+04, DC04, LH05, CRD09] : Ce type d'adaptation modifie la topologie de l'application. Le changement porte sur la structure de l'application en terme de composants et de connexions. Les opérations de base pour réaliser les modifications sont : l'ajout d'un composant, la suppression d'un composant, l'ajout d'une connexion et la suppression d'une connexion.

Un système d'adaptation réalise une ou plusieurs modifications pour ajuster l'application selon les circonstances. La faisabilité d'un type particulier d'adaptation dépend notamment de la modularité et de la technologie utilisée par l'application. Un système distribué auto-adaptable qui supporte plusieurs types d'adaptation offre plus de liberté dans le choix du type approprié. Par exemple, pour réduire la consommation de ressources sur un nœud, le système d'adaptation a le choix entre changer l'implémentation d'un composant pour exécuter un algorithme moins complexe ou migrer le composant sur un autre nœud. Les deux types de modifications ont des degrés d'utilité différents et des coûts d'adaptation différents en ce qui concerne le temps et les ressources consommées.

2.2.4 Phases d'un processus d'adaptation

Les phases réalisées par un système d'adaptation ont été notamment présentées par IBM [FLG06] avec le modèle MAPE (*Monitor-Analyse-Plan-Execute*) défini pour suivre les changements de contexte et agir en conséquence. Le terme *processus d'adaptation* dénote l'exécution de toutes ces phases. Nous présentons ci-dessous les quatre étapes de base d'un processus d'adaptation : surveiller, analyser, planifier et exécuter.

- La fonction de surveillance collecte des données sur l'application et son environnement et les corrèle pour pouvoir être analysées. La collecte des données est faite en utilisant les interfaces fournies par les capteurs associés aux entités observables. Ces données peuvent inclure des informations sur l'utilisation des ressources, la topologie de l'application, la configuration des composants... Certaines données sont statiques,

tandis que d'autres changent continuellement pendant l'exécution de l'application. La fonction de surveillance corrèle et filtre les informations jusqu'à ce qu'elle détermine un symptôme qui doit être analysé.

- La fonction d'analyse (ou de décision) raisonne sur les symptômes fournis par la fonction de surveillance et analyse la situation afin de décider si un changement doit être fait. Par exemple, lorsque la fonction d'analyse trouve que le temps de réponse d'un composant ne répond plus aux exigences, elle peut décider de le répliquer afin que la charge soit répartie entre les deux composants. Si un changement est nécessaire, la fonction d'analyse indique à la fonction de planification les transformations qui doivent être apportées à l'application. Dans [GCH+04] le résultat de l'analyse est appelé une « stratégie d'adaptation », expression que nous utiliserons par la suite.
- La fonction de planification crée ou sélectionne une procédure, c'est à dire un ensemble d'actions, pour effectuer la modification souhaitée. Son rôle est d'ordonnancer correctement les actions, certaines devant être faites avant d'autres comme par exemple l'instantiation d'un composant avant l'établissement de ses liens avec d'autres composants de l'application.
- La fonction d'exécution (ou de reconfiguration) applique le plan d'actions précédemment établi. Ces actions sont réalisées en utilisant les interfaces fournies par les effecteurs des éléments à adapter.

Nous nous sommes particulièrement intéressés dans ce travail de thèse à la distribution et à la coordination des activités d'analyse et d'exécution. C'est pourquoi les travaux étudiés dans les sections suivantes sont choisis parmi ceux qui mettent l'accent sur la prise de décision d'adaptation et le contrôle de l'exécution des actions d'adaptation. Peu de travaux dans le cadre de l'adaptation dynamique se sont intéressés à la phase de planification. Dans notre équipe un travail de doctorat est mené sur ce sujet [FAEDGN+10]. Ce point n'est pas détaillé ici. Concernant la phase de surveillance, plusieurs plateformes pour la gestion du contexte d'exécution ont été proposées [DAS01, CFJ03, KMK+03, GPZ04, RCS08]. L'une de ces plateformes pourrait être utilisée pour réaliser cette phase de l'adaptation.

2.3 Centralisation/distribution de la gestion d'adaptation d'entités multiples

Plusieurs travaux de recherche ont défini des architectures et des méthodes qui servent à doter un système de capacités d'auto-adaptation vis à vis des changements applicatifs et de l'environnement d'exécution. Certaines adaptations du système nécessitent d'appliquer des modifications à plusieurs entités qui le composent et sur plusieurs nœuds du réseau. Dans cette section, nous nous intéressons à ce genre d'adaptations. Nous présentons des solutions existant pour adapter de multiples entités logicielles pendant l'exécution et nous discutons leurs pertinences pour l'adaptation d'applications distribuées. Nous appelons « gestionnaire d'adaptation » l'entité logicielle responsable d'assurer les étapes d'un processus d'adaptation. Certaines approches se basent sur un gestionnaire d'adaptation unique qui contrôle l'adaptation de toute l'application [GCH+04, FHS+06, RLR06, BAP07]. D'autres approches permettent de répartir la gestion de l'adaptation entre plusieurs gestionnaires d'adaptation [CHS01, DC04, BHS09]. Nous allons examiner les principales caractéristiques de ces deux types d'approches et étudier leur adéquation aux systèmes distribués. Nous précisons les caractéristiques reliées aux mécanismes de prise de décision, de planification et du contrôle de l'exécution des actions d'adaptation.

2.3.1 Gestion centralisée d'adaptation

La gestion d'adaptation est centralisée lorsqu'elle s'effectue à l'aide d'un seul gestionnaire d'adaptation qui assure les différentes étapes d'adaptation. Dans la suite, nous décrivons le principe de gestion centralisée de l'adaptation pendant les phases de prise de décision, de planification et d'exécution. Un exemple des travaux existants est détaillé pour chacune de ces phases.

Processus de prise de décision centralisé

Dans plusieurs architectures d'adaptation existantes toutes les prises de décision sont effectuées par un gestionnaire d'adaptation unique [DL03, GCH$^+$04, FHS$^+$06, RLR06, BAP07].

Le processus de prise de décision utilise une vue globale de l'application et son environnement. Cette vue globale permet d'évaluer les états des entités logicielles et les caractéristiques de leurs environnements pour choisir la stratégie d'adaptation appropriée. La vue globale permet aussi de prendre en compte les dépendances entre les différentes entités pour conserver la cohérence de l'application suite à une adaptation. Par exemple, le choix d'algorithme à implémenter par une entité adaptable nécessite la connaissance des algorithmes adoptés par les entités dont elle dépend. Ceci est essentiel pour garantir que les comportements de toutes ces entités soient compatibles entre eux.

Plusieurs techniques ont été utilisées pour la prise de décision comme l'évaluation de règles basées sur le paradigme ECA (Évènement-Condition-Action) [DL03], le couplage d'un modèle de décision Bayésien et un modèle de décision Markovien [PYS98] ou l'utilisation de fonctions d'utilité [GCH$^+$04].

Rainbow [GCH$^+$04] représente un bon exemple de plateforme d'adaptation qui prend en compte en même temps l'état global de l'application et les caractéristiques de son environnement d'exécution. En effet, il s'appuie sur deux types de modèles pour prendre les décisions d'adaptation : le modèle d'architecture et le modèle d'environnement. Le modèle d'architecture reflète les états d'exécution du système. Il prend la forme d'un graphe de composants qui interagissent. Les arcs du graphe sont des connecteurs qui représentent les interactions entre les composants et qui sont généralement réalisées par des intergiciels et des supports de la distribution. Le modèle de l'environnement fournit des informations contextuelles sur le système comme les ressources utilisées et celles disponibles. Par exemple, si des serveurs supplémentaires sont nécessaires pour l'exécution d'un service, le modèle de l'environnement indique quels serveurs sont disponibles. Lorsqu'une meilleure connexion est requise, le modèle d'environnement fournit des informations sur la bande passante disponible sur d'autres chemins de communication. Le choix d'une stratégie d'adaptation se base sur des fonctions d'utilité.

Processus de planification centralisé

Lorsque la gestion de l'adaptation est centralisée, un planificateur unique se charge de fournir le plan d'adaptation. Un plan spécifie les actions à appliquer à un ensemble de composants de l'application afin d'atteindre une nouvelle configuration. Pour déterminer le plan adéquat, le processus de planification utilise des informations concernant ces composants et leur environnement d'exécution. En particulier, ces informations dépendent de la nouvelle configuration à atteindre et elles incluent en général les connexions entre les composants, l'état d'exécution des services et la disponibilité de ressources.

Des solutions ont été définies pour permettre de sélectionner les actions d'adaptation

appropriées [ZYCM04] ou pour déployer les effecteurs nécessaires à l'exécution des plans produits [YRP99].

Nous détaillons l'exemple de l'approche de Zang et al. [ZYCM04] puisqu'elle définit une phase de planification et prend en compte la coéxistance de plusieurs composants logiciels dépendants. Elle se base sur l'analyse de dépendances pour déterminer quels sont les éléments logiciels qui sont touchés par l'adaptation d'un composant. Une relation de dépendance spécifie quels sont les composants qui sont couplés dans leur fonctionnement, ou précise le nombre d'instances requis pour un type de composant. Pour planifier l'adaptation, un graphe d'adaptation (*« safe adaptation graph »* ou SAG) est construit en utilisant ces informations. Ce graphe représente un ensemble de séquences d'actions pouvant produire l'adaptation envisagée. Il décrit les configurations comme des sommets et les actions d'adaptation comme des arcs. Ensuite, l'algorithme de plus court chemin de Dijkstra est appliqué sur le graphe pour trouver une solution réalisable avec un poids minimum, où des coûts fixes (la durée d'adaptation, l'utilisation des ressources...) sont associés aux actions d'adaptation.

Processus de contrôle d'exécution centralisé

Le processus de contrôle d'exécution met en œuvre le plan d'adaptation. Dans un système d'adaptation centralisé, l'exécution des actions d'adaptation est contrôlée d'une manière centralisée par une entité appelée « exécuteur » [OGT+99, MG99, YRP99, GS02, ZYCM04, RLR06]. Comme plusieurs entités de l'application sont adaptables, un groupe d'entités peuvent être modifiées à partir du même plan d'adaptation. Les actions d'adaptation sont réalisées par les effecteurs associés à ces entités. L'exécution des actions doit être coordonnée par l'exécuteur puisque plusieurs entités sont impliquées. Par exemple si une action d'adaptation A pour une entité ne peut pas avoir lieu avant d'avoir appliqué un ensemble d'actions E sur d'autres entités, la coordination consiste à insérer dans le plan un message pour démarrer l'action A à destination de son effecteur après les opérations de réception de messages indiquant que l'ensemble des actions E est terminé. La coordination permet d'exécuter les actions correctement et assure ainsi que l'application atteint un état cohérent à la fin de l'exécution du plan.

Plusieurs travaux ont défini des protocoles pour coordonner des adaptations multiples et atteindre un état cohérent de l'application. Ainsi, Appia [RLR06] définit plusieurs étapes et actions pour reconfigurer un système. C'est est un canevas pour la composition et l'exécution de protocoles de communication. Chaque canal de communication est mis en œuvre par une ou plusieurs piles de protocoles sur chaque nœud. Dans Appia, il est possible de modifier les paramètres du protocole (adaptation paramétrique) comme le délai d'expiration de transmission de message. De plus, le gestionnaire d'adaptation contrôle trois types de changement de piles possibles (adaptation structurelle) : remplacer un seul des protocoles d'une pile par un autre, ajouter/supprimer un protocole unique dans une pile et remplacer une pile par une pile alternative. Lorsqu'un changement dans la composition du protocole d'un ou plusieurs nœuds est nécessaire, le gestionnaire d'adaptation contrôle la reconfiguration grâce à la coordination avec les effecteurs, appelés « agents de reconfiguration », qui s'exécutent sur chaque nœud. Pour assurer la reconfiguration, les agents effectuent un ensemble de tâches : (i) veiller à ce que chaque canal touché par la reconfiguration atteint un état de repos avant la reconfiguration ; (ii) capter l'état qui doit être injecté à la nouvelle configuration ; (iii) déployer les nouvelles piles de protocoles ; (iv) injecter l'état emporté de la configuration précédente et (v) se coordonner avec les agents sur d'autres nœuds via le gestionnaire d'adaptation si nécessaire. Néanmoins, aucune précision de la manière dont cette coordination est réalisée n'est précisée dans Appia.

2.3.2 Limites de la solution centralisée pour les applications distribuées

Quand la gestion d'adaptation est centralisée, le processus de surveillance offre des connaissances concernant le contexte d'exécution global. Ceci est réalisé en mettant à jour périodiquement les informations concernant l'état de l'environnement et de l'application. Cependant, le modèle de description de l'application et de l'environnement devient coûteux à produire et à mettre à jour lorsque l'environnement décentralisé est très fluctuant et de grande taille. Dans un tel cas, l'approche centralisée mène à une surcharge du réseau pour véhiculer les informations vers un serveur central. De plus, l'accès à tous les nœuds n'est pas toujours possible dans certains environnements comme par exemple dans le cas des réseaux mobiles ad-hoc.

Par ailleurs, le contrôle de l'adaptation de plusieurs entités par un gestionnaire d'adaptation unique augmente la complexité des processus de prise de décision et de planification. En effet, le grand nombre d'entités adaptables et le grain fin d'adaptation augmentent le nombre de configurations alternatives à considérer pour la prise de décision. De plus, le nombre de paramètres de contexte à analyser pour décider ou planifier l'adaptation peut être important. Par conséquent, le temps de prise de décision risque d'être insatisfaisant. Enfin, la spécification de la politique d'adaptation et la spécialisation de la fonction de planification deviennent des tâches complexes.

Pour l'adaptation d'un système distribué, l'application de certains plans d'adaptation implique plusieurs composants géographiquement distants. Alors, le contrôle de l'exécution de ces actions nécessite des interactions distantes entre l'exécuteur et les différents effecteurs des composants. Ces interactions augmentent le temps de reconfiguration et la charge réseau.

En conclusion, les approches basées sur une gestion d'adaptation centralisée ne fournissent pas toujours des solutions suffisamment appropriées pour les systèmes distribués surtout quand l'environnement est fortement fluctuant et de grande taille. Le nombre important de configurations alternatives de l'application à considérer est également un problème avec ce type de gestion.

2.3.3 Enjeux de la gestion distribuée de l'adaptation

Dans une approche distribuée, le système d'adaptation est composé d'un ensemble de gestionnaires d'adaptation. Chaque gestionnaire est responsable de l'adaptation d'une ou plusieurs entités logicielles.

Cette approche résout certains problèmes de la centralisation de la gestion de l'adaptation. Il est possible d'exploiter les diverses ressources distantes et d'exécuter en parallèle plusieurs processus d'adaptation distribués ce qui peut améliorer la performance.

La répartition conduit à une démarche exploitant la localité des traitements et la décomposition du problème global en sous problèmes. Dans de pareilles circonstances, chaque gestionnaire d'adaptation surveille un sous ensemble d'entités qui sont dans un espace géographique plus réduit (par exemple la même machine). Ainsi, il dispose d'une vue partielle du contexte d'exécution. Cette vue partielle est moins coûteuse à élaborer et à mettre à jour qu'une vue globale dans le cas de la solution centralisée. De plus, il est possible de placer chaque gestionnaire proche des entités qu'il modifie dans le but de réduire le coût de communication entre l'exécuteur et les effecteurs pendant la phase de reconfiguration.

Enfin, la distribution de la gestion d'adaptation offre de la flexibilité dans la spécialisation des processus d'adaptation. En effet, le développeur a la possibilité d'implémenter chaque gestionnaire d'adaptation de façon différente des autres. Par exemple, un premier groupe de gestionnaires peut inclure un évaluateur de règles spécifiées suivant le paradigme

ECA tandis que les autres gestionnaires peuvent se baser sur des mécanismes d'apprentissage pour la prise de décision.

Toutefois, il faut prendre en compte les dépendances entre les entités logicielles contrôlées par des gestionnaires d'adaptation distincts. Pour cela, il est nécessaire d'introduire des mécanismes de coordination des activités de décision, de planification et d'exécution des différents gestionnaires pour empêcher l'occurrence de configurations incohérentes ou le dysfonctionnement du service suite à une adaptation. Par exemple, les politiques d'adaptation utilisées par les différents gestionnaires doivent être composables de sorte que leur application assure la cohérence du système global. Les mécanismes de coordination constituent un coût supplémentaire en ce qui concerne le temps de réalisation des processus d'adaptation et les ressources consommées par le système d'adaptation. Par exemple, une prise décision collective qui implique plusieurs gestionnaires tarde à converger vers une solution finale lorsqu'elle nécessite un nombre élevé d'interactions distantes entre les gestionnaires impliqués. De plus, un effort de développement supplémentaire s'ajoute pour implémenter les mécanismes de coordination.

Le choix de construction d'un système d'adaptation composé de plusieurs gestionnaires d'adaptation coopérants résulte donc d'un compromis prenant en compte ces différents facteurs. Nous allons étudier, dans la suite, des travaux qui ont suivi une approche distribuée pour la gestion de l'adaptation en nous focalisant sur les mécanismes de coordination qu'ils définissent. Cette étude vise à analyser les techniques utilisées et déterminer leurs avantages et inconvénients en terme de flexibilité et réutilisabilité dans plusieurs domaines d'application.

2.3.4 Coordination pour la gestion distribuée de l'adaptation

Dans ce paragraphe, nous étudions les possibilités de coordination proposées dans des travaux existants. Nous décrivons les formes de coordination possibles au niveau des fonctions de prise de décision, de la planification et de l'exécution et nous montrons les limites des solutions proposées.

Coordination dans la prise de décision distribuée

Par prise de décision distribuée, nous entendons un ensemble d'acteurs qui coopèrent ensemble pour la prise de décision concernant la stratégie d'adaptation appropriée. Ainsi, certains travaux se sont intéressés à la coordination comme une technique qui permet de faire collaborer plusieurs participants pour choisir la stratégie adéquate.

D'une part, la coordination permet d'organiser les participants à une prise de décision distribuée pour déterminer la stratégie d'adaptation. Par exemple, le système Ensemble [VRBH+97] permet à un processus de prise de décision de déclencher un autre sur la couche au-dessus dans une application en couches. Ceci se réalise lorsque le processus initial se trouve incapable de gérer le changement de contexte. Dans le canevas K-Component [DC04], un décideur choisit de déléguer la prise de décision à un autre sur un nœud voisin lorsqu'il estime qu'une meilleure solution pourrait être trouvée.

D'autre part, la coordination permet aussi une forme de coopération pour la résolution de conflits entre plusieurs politiques d'adaptation. Elle permet à un groupe de décideurs de s'accorder en évitant les incohérences et les incertitudes sur les stratégies d'adaptation à appliquer. En effet, un manque de coordination des entités d'un système distribué peut mener à l'interférence entre les comportements adaptatifs des différentes entités ou à des conflits sur l'utilisation des ressources partagées et donc à une contre-performance du système [EFDC02]. L'application d'une stratégie d'adaptation à une entité logicielle doit

convenir aux différentes parties qui sont impactées par l'adaptation. Dans certaines approches le processus de prise de décision est distribué entre les clients utilisant les services d'un composant et l'intergiciel qui l'adapte. Dans ce cadre, l'intergiciel Carisma [CEM03] permet la résolution de conflits en utilisant une méthode basée sur un protocole d'enchères où l'intergiciel joue le rôle du commissaire-priseur, les clients sont les enchérisseurs, et les stratégies que les auteurs appellent politiques, sont les biens mis aux enchères.

L'article [CHS01] présente une approche assez similaire concernant le canevas d'adaptation Cactus où la nécessité d'un processus d'accord dans lequel tous les composants adaptables parviennent à un consensus sur l'action d'adaptation à exécuter est mise en évidence. Cependant, ce canevas n'implémente pas de mécanismes pour atteindre un consensus. Ces mécanismes doivent être définis et mis en œuvre par le développeur.

Nous reviendrons plus en détail sur les plateformes Ensemble, K-Component et Cactus dans le paragraphe 2.3.5.

Coordination de processus distribués de planification

Dans une approche de planification distribuée, les activités de planification sont réparties au sein d'un ensemble de gestionnaires d'adaptation autonomes ayant chacun son propre planificateur. La coordination d'un ensemble de processus de planification s'attaque à des situations dans lesquelles un groupe de planificateurs fournissent des plans d'adaptation à appliquer et il existe des dépendances entre des actions spécifiées dans des plans distincts. En effet, les plans d'adaptation peuvent être en conflit en raison des incompatibilités des états des composants du système, de l'ordre d'exécution des actions ou de l'usage des ressources. Par exemple, on peut avoir un plan d'adaptation qui spécifie des actions pour changer le comportement d'un composant alors que ce dernier doit être supprimé d'après un autre plan établi en parallèle. Si tel est le cas, le problème consiste à détecter et résoudre les conflits éventuels entre ces plans pendant la phase de planification afin de pouvoir les exécuter de manière cohérente par la suite. Ainsi, il s'avère nécessaire de raisonner sur les dépendances entre les plans d'adaptation et prévoir des interactions entre leurs exécuteurs pour converger vers une configuration cohérente du système global.

Des problèmes similaires ont été abordés dans le contexte de systèmes multi-agents. Ainsi, nous trouvons dans ce contexte plusieurs solutions au problème de planification distribuée multi-agents.

Par exemple dans l'approche de Georgeff [Geo83], chaque agent construit localement un plan. Ensuite, un agent coordinateur rassemble les plans et les analyse pour identifier les conflits relatifs aux accès simultanés à des ressources partagées. L'agent coordinateur résout les conflits en modifiant les différents plans. La résolution se base sur des travaux relatifs au problème de satisfaction de contraintes dans le but d'empêcher que des portions de plans conflictuelles soient exécutées de façon concurrente.

Durfee et Lesser [DL87] ont proposé une approche appelée « planification partielle globale » (*« Partial Global Planning »* ou PGP). Cette approche ne vise pas à résoudre des conflits entre les plans des différents agents mais son objectif est d'optimiser le temps de calcul par la réduction de l'inactivité des agents et l'élimination des tâches redondantes. Les agents sont organisés de façon hiérarchique. Ils propagent leurs plans à l'agent coordinateur supérieur dans la hiérarchie. Ce coordinateur analyse les différents plans fournis par les agents et identifie les buts distincts des agents à partir de leurs plans locaux. Pour les plans communs, il privilégie ceux qui réalisent de façon concurrente les actions, requièrent le moins de ressources et approuvent ou réfutent la poursuite de certains autres buts. Ensuite, il élimine les plans redondants et communique aux agents leur plan final à exécuter.

Von Martial [Mar92] a proposé un modèle de coordination de plans qui s'appuie sur la résolution de conflits basée sur deux types de relations négatives et positives pouvant exister entre ces plans. Les relations entre les plans entraînent une modification des plans des agents. Les relations négatives (ou conflictuelles) sont celles qui gênent ou empêchent plusieurs actions de s'accomplir simultanément et sont dues en général à des incompatibilités de buts ou des conflits de ressources. Par exemple dans une vente aux enchères, deux agents veulent acquérir un même produit. Les relations positives (ou synergiques) sont celles qui permettent aux actions de bénéficier les unes des autres. Ainsi, la réalisation d'une action par un agent réalise du même coup une action que devait accomplir un autre agent ou favorise la réalisation d'une action par un autre agent. Par exemple X, Y et Z sont dans une pièce dont les fenêtres sont fermées et les stores baissés. X a chaud et Z aimerait avoir de la lumière. Y monte les stores et ouvre une fenêtre [JCD02].

Cependant, les problèmes de planification distribuée ne sont pas traités dans les travaux concernant la gestion distribuée de l'adaptation dynamique. À notre connaissance les systèmes d'adaptation actuels ne supportent pas de formes de coordination entre plusieurs planificateurs. Il est possible d'adapter les approches définies pour les systèmes multi-agents pour fournir des solutions spécifiques utilisables dans le contexte de systèmes d'adaptation distribués.

Coordination de processus distribués de contrôle d'exécution

Pour mettre en œuvre les modifications dans une application, chaque exécuteur interagit avec un ensemble d'effecteurs associés aux entités logicielles pour appliquer les actions d'adaptation. Il contrôle et coordonne ces effecteurs pour appliquer correctement les actions spécifiées dans le plan d'adaptation.

Lorsque plusieurs gestionnaires d'adaptation sont associés à une application, le contrôle d'exécution peut être réparti entre des exécuteurs distincts. Plusieurs plans d'adaptation peuvent agir simultanément sur une topologie distribuée d'entités logicielles. Si le processus de planification a pris en compte la distribution des entités logicielles à adapter et la distribution des exécuteurs, il a pu inclure dans les différents plans les actions de synchronisation et d'échanges de messages d'information nécessaires entre les exécuteurs. Ceux-ci n'ont donc qu'à suivre leurs plans. Dans le cas contraire, les différents exécuteurs doivent décider eux mêmes d'interagir pour échanger des informations et coordonner leurs activités de contrôle d'exécution des plans dépendants.

Certains travaux ont proposé des protocoles qui permettent l'exécution de plans sur plusieurs nœuds d'une manière coordonnée. Chen et al. [CHS01] ont défini le protocole *« graceful adaptation protocol »* dans le canevas Cactus. Ce protocole permet d'appliquer des adaptations d'implémentations de composants en cours d'exécution sans arrêter le déroulement de l'application. Changer une implémentation se déroule en plusieurs étapes. La coordination permet d'achever une étape de la reconfiguration sur tous les composants impliqués dans l'adaptation avant d'appliquer l'étape suivante. Cette approche sera décrite plus en détail dans le paragraphe suivant.

2.3.5 Étude de travaux existants avec gestion distribuée de l'adaptation

Dans ce paragraphe, nous présentons des travaux qui permettent une gestion distribuée de l'adaptation et qui se sont intéressés à l'aspect de coordination des processus d'adaptation. Ces travaux n'utilisent pas de phase de planification aussi nous allons décrire comment ils assurent la prise de décision d'adaptation et l'exécution des plans.

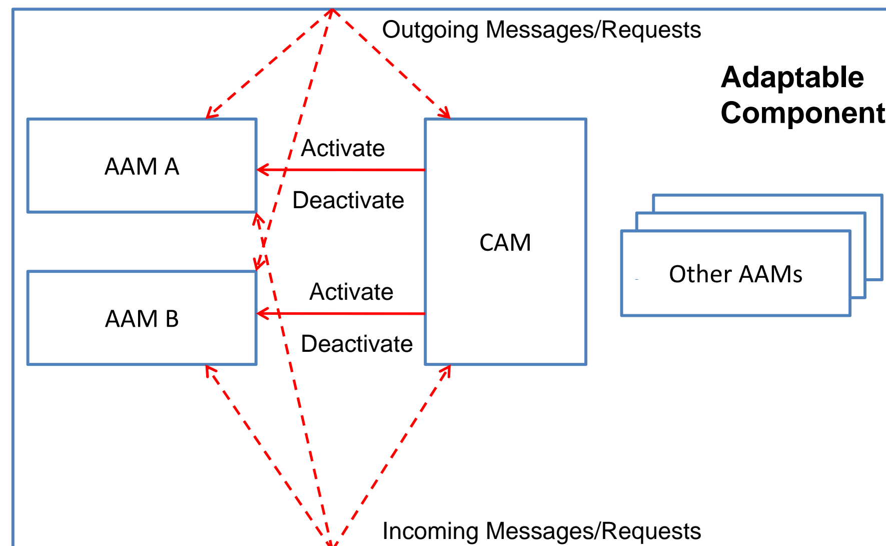

FIGURE 2.1 – Structure d'un composant adaptable basé sur [CHS01]

Cactus

Description. Cactus [CHS01] est un canevas pour construire des services configurables dans des systèmes distribués. Dans Cactus, un nœud est organisé en couches hiérarchiques, où chaque couche inclut plusieurs composants adaptables. Chaque composant adaptable encapsule des alternatives de mise en œuvre d'un service spécifique.

La structure interne d'un composant adaptable (voir figure 2.1) se compose de deux types de modules : le module adaptateur du composant (*« Component Adaptation Module »* ou CAM) et les modules algorithmiques alternatifs conscients de l'adaptation (*« alternative Adaptation-Aware algorithm Modules »* ou AAMs). Chaque AAM présente un algorithme différent qui implémente la fonctionnalité du composant, tandis que le CAM contrôle le comportement adaptatif du composant.

Processus d'adaptation. L'article [CHS01] décrit le processus d'adaptation d'un composant adaptable distribué (*« Distributed Adaptive Component »* ou DAC). Un DAC est une collection de composants adaptables répartis sur plusieurs machines et qui coopèrent ensemble pour mettre en œuvre un service distribué. Le type de fonctionnalité fournie par un DAC peut être par exemple la mise en œuvre de la fiabilité ou de l'ordonnancement de messages pour des services de communication.

Prise de décision : Dans Cactus, la prise de décision se fait en deux phases : la phase de détection de changement et la phase d'accord.

La première phase consiste à détecter un changement dans l'environnement d'exécution et déterminer s'il est bénéfique de s'adapter en réponse à ce changement. L'adaptateur cherche l'implémentation du composant la plus appropriée en se basant sur des fonctions d'utilité.

Si un nouvel algorithme alternatif doit remplacer l'algorithme actuel suite au changement du contexte d'exécution, l'adaptateur initialise la phase d'accord. Cette phase est

nécessaire pour permettre à tous les composants adaptables d'un DAC de parvenir à un consensus sur la stratégie d'adaptation nommée dans Cactus « action d'adaptation ». Un certain nombre d'approches sont possibles pour atteindre un tel consensus et le modèle ne dicte pas une approche spécifique pour le réaliser. Une des possibilités est de parvenir à un consensus sur l'état global du système et de déterminer l'action d'adaptation correspondante [CHS01]. Dans cette approche, le processus d'accord consiste à diffuser l'état local de chaque composant impliqué aux autres adaptateurs et ensuite chaque adaptateur applique une même fonction déterministe pour calculer l'état global du système. Une fois que l'état global est calculé, il est utilisé comme entrée pour la fonction qui donne l'alternative optimale. Les fonctions sont déterministes, ce qui signifie que tous les adaptateurs prennent la même décision.

Reconfiguration : Suite à la phase d'accord, une phase finale met en œuvre le processus d'exécution qui cordonne l'échange d'un algorithme par un autre de telle sorte que le DAC continue à fonctionner correctement. Cactus propose un protocole multi-étapes, appelé *« graceful adaptation protocol »*, pour appliquer les modifications aux composants. Ce protocole est basé sur l'identification des flux de messages et il est exécuté en trois étapes principales pour adopter une nouvelle alternative : (i) la préparation, (ii) la commutation des sorties et (iii) la commutation des entrées. En effet, les composants d'une application sont organisés hiérarchiquement en couches. Alors, un composant adaptable traite deux flux de messages : les messages sortants en provenance du niveau supérieur qui sont traités et envoyés au niveau inférieur, et les messages entrants en provenance du réseau qui sont traités et délivrés au niveau supérieur.

Pendant la première étape du protocole, chaque composant adaptable se prépare à recevoir les messages entrants par le nouveau AAM ou des messages liés à l'adaptation par l'ancien AAM. La deuxième étape permet à chaque composant de commuter le traitement des messages sortants de l'ancien AAM au nouvel AAM. Au cours de la dernière étape, chaque composant commute la livraison de messages entrants de l'ancien AAM au nouvel AAM. Les deux modules, l'ancien AAM et le nouvel AAM, sont actifs pendant les trois étapes, mais l'ancien AAM arrête le traitement des messages sortants dans la deuxième étape, et arrête le traitement et la diffusion de messages entrants dans la troisième. Selon la sémantique du service, le transfert d'état entre l'ancien AAM et le nouvel AAM se produit si nécessaire dans les étapes (ii) et (iii). Chaque AAM doit fournir une API appropriée pour l'exécution des trois étapes.

La coordination dans ce protocole consiste à permettre d'accomplir l'étape de préparation pour tous les composants avant de passer à la deuxième étape. Cette exigence est mise en œuvre par un protocole de barrière de synchronisation exécutée à la fin de la première étape. Grâce à ce protocole, chaque adaptateur qui arrive sur cette barrière attend jusqu'à ce que le reste des adaptateurs y soit arrivé.

Discussion. Un premier avantage du canevas Cactus est que la prise de décision d'adaptation est un processus distribué qui tient compte de l'état global du système pour choisir un algorithme approprié pour un service distribué. Par ailleurs, la solution qu'il apporte au problème du remplacement de l'algorithme adopté par plusieurs composants participant à un même service distribué permet de ne pas interrompe les communications. La barrière de synchronisation assure que tous les composants impliqués sont prêts à être modifiés.

Cependant, Cactus supporte un seul type de modification qui est le remplacement de l'implémentation des composants. Tous les composants adaptables d'un DAC participent à la phase d'accord et ils appliquent tous la même action d'adaptation qui consiste à permuter

vers un nouvel algorithme. L'approche ne traite pas le cas d'adaptation coordonnée de plusieurs DACs. La plateforme n'offre pas de facilités pour implémenter les mécanismes nécessaires pour la phase d'accord ; c'est au développeur de le faire. La coordination des actions d'adaptation est spécifique au protocole de reconfiguration défini et elle se limite à synchroniser la fin de la réalisation de la première étape par tous les CAMs avec le démarrage des deuxièmes étapes.

Ensemble

Description. Ensemble est un système de communication de groupe [BCH+99]. Il fournit une boîte à outils pour la mise en œuvre d'un protocole composé d'un ensemble de micro-protocoles organisés en couches qui constituent une pile. Chaque micro-protocole est un module qui met en œuvre une partie des opérations de la communication comme la fragmentation, la fiabilité ou l'ordonnancement. Une pile est instanciée pour chaque participant du groupe. Le développeur définit l'organisation de la pile utilisée par un groupe de façon à fournir les propriétés désirées pour le protocole. Chaque micro-protocole traite certains aspects reliés à ces propriétés. Ensemble fournit un support pour l'adaptation dynamique [VRBH+97]. L'adaptation permet de construire une nouvelle pile (c.-à-d. un nouveau protocole) dans laquelle des micro-protocoles initiaux sont remplacés par des nouveaux.

Processus d'adaptation.

Prise de décision : Dans cette approche, une (ou plusieurs) spécification(s) d'hypothèse-garantie est (sont) associée(s) avec chaque micro-protocole. Chaque spécification détermine une garantie sur les événements générés en sortie selon un ensemble d'hypothèses sur les événements en entrée. Selon les propriétés exigées par l'application et les propriétés du réseau, il est possible de déterminer une pile appropriée en utilisant des heuristiques. Un détecteur surveille l'environnement, détecte les violations des hypothèses et fournit des informations utiles pour déterminer une nouvelle configuration. Ce détecteur est implémenté sous forme de micro-protocoles. Suite à la détection d'une violation, les différentes couches peuvent coopérer ensemble pour la prise de décision d'adaptation. Les couches de bas niveau essayent d'abord de s'adapter localement. Si elles ne peuvent pas répondre correctement à un changement particulier de l'environnement, elles propagent les violations de QoS aux couches plus hautes. Éventuellement, l'application peut être notifiée quand le changement de l'infrastructure de communication n'est pas possible. Dans ce cas, l'application décide comment se reconfigurer. Cette approche ne définit pas de mécanismes pour faire coordonner des prises de décisions par des modules répartis sur des machines distinctes.

Reconfiguration : Une reconfiguration est assurée par un protocole appelé *« Protocol Switch Protocol »* ou PSP. PSP est implémenté sous forme d'un ensemble de micro-protocoles. Ce protocole installe de nouvelles piles sur les participants du groupe pour établir un nouveau schéma de communication. Pour cela, PSP arrête les micro-protocoles de l'ancienne pile, instancie la nouvelle pile et la démarre. PSP coordonne les actions d'adaptation du groupe afin d'assurer la cohérence du système. Il permet de synchroniser les participants, les assister pour arrêter leurs communications en cours et redémarrer les communications. Pour cela, PSP élit un coordinateur parmi les participants. Le coordinateur diffuse à tous les participants un message *FINALIZE* contenant la description de la

nouvelle pile, les participants, s'ils ont changé, et le nouvel identifiant de la pile. Chaque participant construit la nouvelle pile et envoie le message à la couche la plus haute de l'ancienne pile pour arrêter les communications en cours. Chaque couche termine sa tâche et passe le message à la couche du dessous. Lorsque le message atteint la dernière couche, un message d'acquittement est retourné au coordinateur. À la réception des acquittements, le coordinateur diffuse un autre message, *START*, destiné à la couche la plus basse de la nouvelle pile. Ce message sera passé aux couches supérieures pour démarrer la nouvelle pile. Ensuite, l'ancienne pile est détruite.

Ensemble ne fait pas de différence entre exécuteur et effecteurs. Le coordinateur joue le rôle d'exécuteur qui contrôle la reconfiguration par émission de messages et coordonne les modifications sur les différents nœuds. Les modules de PSP chez les autres participants peuvent être considérés comme des effecteurs qui terminent les communications avec l'ancienne pile et construisent la nouvelle. Le processus de contrôle d'exécution d'un plan est centralisé. Les auteurs donnent l'exemple d'adaptation de technique d'ordonnancement de multicast de messages. Arrêter l'ancienne configuration implique le retardement du changement jusqu'à ce que tous les messages envoyés par le protocole d'ordonnancement initial soient délivrés.

Discussion. La prise de décision d'adaptation dans Ensemble est possible dans les différentes couches de l'application (ici le protocole de communication). La hiérarchie simplifie la prise de décision et permet de choisir, en partant de la couche la plus basse, la couche la plus appropriée pour faire l'adaptation. En ce qui concerne l'exécution, Ensemble permet de synchroniser les changements sur différentes couches et entre les nœuds distribués. Ensemble est dédié à un seul type d'adaptation : le remplacement de la pile dans un protocole communication. Les mécanismes sont codés entièrement par le développeur. La coopération pour la prise de décision se limite à la propagation des violations d'une couche à une autre. La coordination de la reconfiguration distribuée (plusieurs effecteurs sur plusieurs nœuds) est centralisée. Le déroulement de la reconfiguration est identique pour tous les processus d'exécution en ce qui concerne les types d'actions et leur ordre. La coordination consiste à s'assurer, par la transmission de messages, de la fin des communications en cours avant de démarrer la nouvelle pile.

K-Component

Description. K-Component [DC01] est un modèle de composant qui permet de créer des applications distribuées auto-adaptables. Selon ce modèle, chaque composant K-Component inclut, outre des composants applicatifs, un modèle de leur architecture et des contrats d'adaptation (voir figure 2.2). L'entité qui gère ce modèle et ces contrats est le « gestionnaire de configuration ». Le modèle d'architecture réifie la composition des composants applicatifs sur un nœud et il peut être utilisé pour effectuer l'adaptation locale en utilisant la réflexion architecturale. Les contrats d'adaptation sont des programmes réflexifs reliés aux composants. Il s'agit de règles qui déterminent le comportement d'adaptation à appliquer lorsque certains événements ou changements de contexte se produisent. Les adaptations sont directement effectuées sur le modèle de l'architecture logicielle des composants applicatifs et reproduites sur les composants.

La communication entre le gestionnaire de configuration et les composants applicatifs se fait via des événements d'adaptation. Ces événements sont émis par l'application afin de remonter des informations et par le gestionnaire de configuration afin de commander les reconfigurations. La reconfiguration de l'architecture se fait en utilisant les opérations de

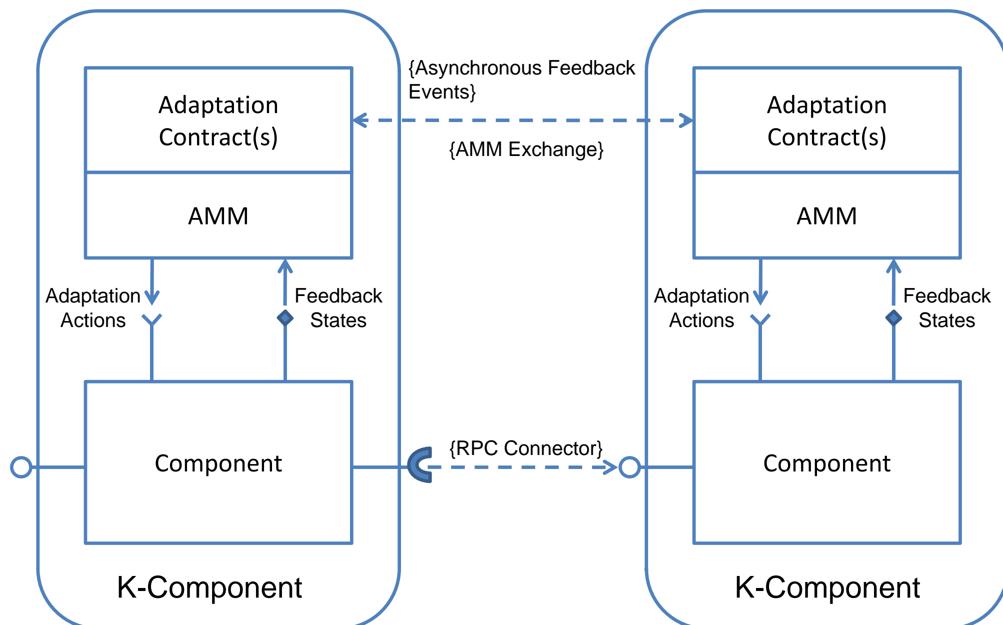

FIGURE 2.2 – Deux composants composites connectés dans deux instances K-Component différentes [DC04]

reconfiguration fournies par les composants applicatifs et elle se base sur un protocole de reconfiguration qui garantit la cohérence et l'intégrité des composants sur un nœud.

Processus d'adaptation.

Prise de décision : Le gestionnaire de configuration détient un ensemble de contrats d'adaptation. Un contrat implémente une politique d'adaptation sous la forme de règles de type ECA décrites dans un langage de programmation déclaratif. Un contrat décrit les conditions de déclenchement des actions d'adaptation à partir des états des composites et des connecteurs du nœud auquel il appartient et à partir des événements envoyés par les autres nœuds.

Une technique appelée apprentissage par renforcement collaboratif (*« collaborative reinforcement learning »* ou CRL) a été utilisée pour coordonner le comportement adaptatif d'un groupe de K-Components dans le but d'établir des propriétés d'autonomie dans des environnements fluctuants et incertains [DC04].

CRL est une version distribuée de l'apprentissage par renforcement (*« reinforcement learning »* ou RL). Le RL consiste à apprendre comment associer des actions à des situations, afin de maximiser quantitativement une récompense. L'apprenant découvre quelles actions donnent une meilleure récompense en les essayant. Ceci est intéressant quand des actions peuvent affecter non seulement les récompenses immédiates mais aussi les récompenses à plus long terme. Les deux propriétés (i) recherche par essai-erreur et (ii) récompense à long terme sont les deux caractéristiques les plus importantes de l'apprentissage par renforcement.

Dans CRL, le problème d'optimisation du système est décomposé en un ensemble de problèmes d'optimisation discrète (*« discrete optimization problems »* ou DOPs). Les

connaissances sont réparties entre les agents qui interagissent seulement avec leurs voisins en leur envoyant leurs résultats. Chaque agent constitue ainsi une table de solutions de DOP, table composée des résultats de résolution de ses voisins.

Pour utiliser la technique CRL, les contrats d'adaptation représentent les agents dans le canevas K-Component. Un K-Component qui a détecté un changement résout un DOP. Il peut ainsi publier sa décision d'adaptation à ses voisins et transférer le contrôle de l'adaptation à certains d'entre eux s'il estime qu'une meilleure décision est possible. Ceux-ci vont lancer de nouveaux DOPs. Ainsi, le comportement autonome du système émerge de prises de décision locales par les composants auto-adaptables après échanges entre voisins.

Dans [DC04], CRL est utilisé pour résoudre un problème d'optimisation consistant à faire de l'équilibrage de charge entre les nœuds d'un réseau de manière décentralisée. Une charge représente tout type d'utilisation de ressources (stockage, unité de calcul...). Lorsqu'un composant reçoit une charge, le contrat d'adaptation du composant prend une décision d'équilibrage de charge. Une fonction d'équilibrage de charge dans chaque contrat d'adaptation calcule le coût de charge local et la capacité des voisins à gérer cette charge. Chaque voisin a un coût de charge annoncé, mais les contrats d'adaptation tiennent compte également du coût de connexion dans la transmission de la charge au voisin. Ainsi, le coût de charge estimé est la somme du coût de la charge annoncée et du coût de la connexion. Lorsque la charge est transmise par un contrat d'adaptation en exécutant des actions d'adaptation, un nouveau DOP est déclenché par le contrat d'adaptation du voisin. De cette manière, le comportement d'équilibrage de charge est assuré de proche en proche par des DOPs.

Reconfiguration : Une fois que la nouvelle configuration a été établie, l'exécution de la reconfiguration est effectuée par chacun des nœuds impactés de manière séparée. Le système K-Component ne propose donc pas de mécanismes pour coordonner les opérations de reconfiguration sur des nœuds différents. Par ailleurs, on peut remarquer que la réussite du transfert de l'état du composant exige que les développeurs de composants implémentent une interface réalisant la copie et le transfert de cet état.

Discussion. Les principales contributions de ce travail sont le modèle K-Component et CRL. On identifie un composant auto-adaptable comme l'élément de construction de systèmes autonomes et l'apprentissage collaboratif par renforcement en tant que technique de coordination pour gérer les comportements d'adaptation collectifs dans le système. K-Component présente un modèle pour créer des applications distribuées auto-adaptables qui sépare le comportement d'adaptation des systèmes de leur comportement fonctionnel.

La distribution du processus de décision et l'utilisation d'une technique d'apprentissage distingue cette approche. Cependant, étant donné que la technique d'apprentissage pour la prise de décision se base sur une recherche par essai-erreur et qu'il existe un tâtonnement dans la sélection des actions d'adaptation à exécuter, on n'est pas toujours garanti de faire le bon choix. Cette technique n'est donc pas appropriée pour les systèmes critiques où la meilleure solution doit être trouvée. De plus, la prise de décision passant de façon successive sur les nœuds voisins fait que le temps de prise de décision peut être long.

Enfin K-Component n'aborde pas le problème d'objectifs d'optimisation multiples et conflictuels. Or, les systèmes décentralisés peuvent souvent avoir besoin d'optimiser plusieurs propriétés.

Système	Adaptation		Prise de décision				Reconfiguration			
	Cible	*Type*	*Choix de stratégie*		*Coordination*		*Protocole*		*Coordination*	
			Technique	*Politique externe*	*Technique*	*Participants*	*Nom*	*Spécialisable*	*Nature du contrôle*	*Technique de coordination*
Cactus	Composants distribués assurant un service	Remplacer les implémentations des composants	Fonctions d'utilité	Non	Diffusion d'état local / fonctions déterministes	Tous les composants assurant le service	Graceful adaptation protocol	Non	Distribuée avec coordination	Barrière de synchronisation
Ensemble	Micro-protocoles d'une pile	Remplacer les composants	Non précisée	Non	Délégation de prise de décision à une autre couche	Différentes couches	Protocol switch protocol	Non	Centralisée	Connaissance de l'état global
K-Component	Composants distribués	Remplacer les composants	Apprentissage par renforcement	Oui	Délégation de prise de décision à un autre nœud	Différents voisins	-	Non	Distribuée sans coordination	-

Figure 2.3 – Caractéristiques des travaux étudiés

2.3.6 Synthèse

Le nombre de travaux qui se sont intéressés à coordonner les activités de multiples gestionnaires d'adaptation est limité. Ces travaux permettent de faire coopérer plusieurs modules logiciels pour la prise de décision et/ou le contrôle d'exécution de plusieurs plans. Cependant, les capacités de coopération sont assez limitées et manquent de généricité.

La figure 2.3 synthétise les trois travaux que nous avons détaillés dans ce chapitre. Ils ont abordé le problème d'adaptation de composants applicatifs distribués. Mais, nous notons que chacun s'est intéressé à un type spécifique d'adaptation.

En ce qui concerne la prise de décision d'adaptation, Ensemble et K-Component se limitent à déléguer la prise de décision respectivement d'une couche à une autre et d'un nœud à un autre nœud voisin. Pour le canevas Cactus, plusieurs gestionnaires d'adaptation peuvent prendre des décisions d'adaptation simultanément. Ces décisions étant basées sur une vue globale de l'application et l'environnement ainsi que des fonctions déterministes, tous les gestionnaires prennent, pour un même événement, la même décision d'adaptation. Aucun mécanisme de coordination explicite n'est donc disponible dans Cactus.

À notre connaissance, aucun travail ne propose des mécanismes permettant des prises de décision distribuées et parallèles, chacune basée sur une vue locale du système, dans un même processus d'adaptation. Le problème de la résolution de conflits entre les décisions de plusieurs gestionnaires d'adaptation n'est donc pas abordé.

Concernant les facilités pour la mise en place de mécanismes de coordination pour la prise de décision, seul K-Component propose une solution réutilisable : la spécification

des contrats d'adaptation dans un langage de programmation déclaratif. Dans Ensemble, plusieurs couches participent à la détermination de la stratégie d'adaptation mais les mécanismes de prise de décision ne sont pas séparés de l'application. Cactus ne propose pas de mécanismes génériques qu'on peut spécialiser selon l'application à adapter.

Pour faciliter la tâche d'un concepteur de systèmes d'adaptation, il nous semble donc intéressant de proposer des mécanismes réutilisables et facilement spécialisables permettant à des décideurs de prendre des décisions localement, tout en se coordonnant quand c'est nécessaire afin d'aboutir à une meilleure solution, d'éviter les incohérences et de détecter et résoudre les conflits éventuels entre eux.

Concernant les mécanismes de reconfiguration, seul Cactus propose un protocole qui n'exige pas de centralisation de la coordination au niveau d'un exécuteur unique. Cependant, la coordination se limite à l'usage de messages et à une synchronisation totale puisqu'il faut attendre que tous les messages de fin d'une phase spécifique soient délivrés avant d'appliquer la phase suivante.

En ce qui concerne les possibilités de spécialisation des mécanismes de reconfiguration, les protocoles existants manquent de flexibilité puisque le type de modification et les étapes de sa réalisation ainsi que le comportement de coordination et les participants sont prédéfinis. Un modèle générique et des mécanismes réutilisables pour le contrôle distribué et coordonné de l'exécution de plans d'adaptation devrait permettre de mieux prendre en compte l'hétérogénéité des besoins.

2.4 Spécialisation des infrastructures d'adaptation

La construction de systèmes auto-adaptables pose plusieurs défis de développement. Les canevas logiciels réutilisables aident à relever ces défis en utilisant des modèles appropriés et en fournissant des services essentiels pour les systèmes auto-adaptables. Plusieurs canevas comme Dynaco [BAP07], Madam [FHS+06] et Rainbow [GCH+04] ont été conçus avec cet objectif. Il s'agit de cadres génériques basés sur des normes et des principes bien établis. Ces canevas définissent généralement l'architecture capable d'assurer les phases d'adaptation. Dans cette section, nous nous intéressons aux possibilités de spécialisation de ces canevas et nous étudions les facilités proposées pour construire un système d'adaptation concret.

2.4.1 Spécialisation de comportement

Les canevas d'adaptation dynamique spécifient un ensemble de types de composants et les règles d'interaction entre eux.

Dans certaines approches, comme dans le cas du canevas Cactus [CHS01], le développeur est responsable d'écrire le code à implémenter pour chaque composant du système d'adaptation. Dans d'autres solutions, les implémentations de canevas de systèmes d'adaptation prennent en charge les tâches communes dans le développement de tels systèmes. En ce qui concerne la prise de décision, elle peut être spécialisée via des politiques d'adaptation externes. La politique définit le comportement attendu d'un décideur. La forme de politique la plus largement utilisée est la politique d'actions qui se présente sous forme de règles événement-condition-action [KC03, DL03]. La plate-forme proposée dans Rainbow fournit un langage de ce type, appelé Stitch, mais spécialement conçu pour représenter les politiques d'adaptation [GCH+04]. D'autres types de politique comme la politique de but (la spécification d'un état désirable) [LLMY05, BAP07] et la politique de fonction d'utilité (l'expression de la valeur de chaque état possible) [CHS01, FHS+06] sont aussi utilisées dans les logiciels auto-adaptables. Rainbow [GCH+04] se base aussi sur la théorie

d'utilité pour calculer la meilleure solution d'adaptation. K-Component propose le langage ACDL (*Adaptation Contract Description Language*) [DC04]. La politique d'adaptation est décrite de manière déclarative à l'aide de ce langage sous la forme de contrats. Ces contrats stipulent les conditions associées aux événements et les opérations d'adaptation architecturales qui doivent être déclenchées par ces mêmes événements.

Dans certains cas, les politiques d'adaptation peuvent être changées à cause de nouvelles exigences envers le système d'adaptation ou de nouvelles conditions d'exécution de l'application. Par exemple, Carisma [CEM03] a abordé cette question et a proposé une technique pour le changement dynamique de politiques. Une interface permet à l'application de demander la désinstallation de la politique courante et l'installation d'une nouvelle.

Quant au contrôle de l'exécution, les protocoles de reconfiguration sont prédéfinis et rigides ou sont à mettre en œuvre par le développeur. Les approches existantes ne proposent pas des mécanismes génériques dont le comportement peut être personnalisé selon l'application visée. Ils fixent d'avance les composants impliqués, les types d'adaptation et l'ordre des actions.

2.4.2 Spécialisation de la répartition des mécanismes d'adaptation

Lorsque plusieurs composants de l'application sont adaptables, un choix est nécessaire pour associer ces composants avec les gestionnaires d'adaptation qui contrôlent leurs processus d'adaptation.

Certaines approches associent à chaque composant adaptable un gestionnaire d'adaptation. Dans le canevas Accord [Liu04], chaque composant autonome intègre un agent de règles qui est chargé de gérer son exécution. Il surveille l'état du composant et son contexte et contrôle les règles de comportement et d'interaction déclenchées. Dans [LLMY05] les auteurs envisagent une hiérarchie d'éléments autonomes. Un gestionnaire autonome est associé à chaque élément. Grâce à cette hiérarchie d'éléments autonomes, il est possible de propager les préoccupations de haut niveau de l'élément racine jusqu'aux éléments au niveau feuilles en sorte que le système atteigne ses objectifs et réponde aux exigences de qualité. Pour Cactus [CHS01], un gestionnaire d'adaptation contrôle chaque composant adaptable. En particulier, il initialise l'adaptation quand il s'aperçoit qu'un autre algorithme alternatif satisfait mieux les exigences selon l'état de l'environnement.

Cette approche totalement distribuée devient complexe lorsque le nombre de composants est élevé. En effet, comme elle gère de manière isolée chaque composant, il est difficile de gérer les dépendances entre ceux-ci. Il faut que chacun prévoit les interactions avec les autres gestionnaires lorsque l'adaptation l'exige.

D'autres travaux permettent d'instancier un gestionnaire d'adaptation sur chaque machine. Dans [MG05], une instance du canevas CASA (*« CASA Runtime System »* ou CRS) est créée sur chaque nœud qui héberge des applications auto-adaptables. Le CRS est chargé de surveiller les changements dans l'environnement d'exécution et d'adapter ces applications. La politique d'adaptation de chaque application est définie dans un contrat d'application. Il n'est pas évoqué de mécanisme de coopération entre les différents nœuds. De même dans [DC04], une instance de K-Component sur chaque machine permet d'adapter un sous ensemble de composants de l'application sur ce nœud et de prendre en compte leurs dépendances internes et externes en utilisant des connecteurs. Le canevas Cholla [BHS09] définit un contrôleur d'adaptation sur chaque nœud. Ce contrôleur gère l'adaptation d'un ensemble de composants organisés en couches sur le nœud qui l'héberge et communique l'état de ses composants aux autres nœuds (et réciproquement). Enfin, dans [CRD09], des services de supervision et de reconfiguration sont distribués en fonction des ressources (mé-

moire, énergie, cadence processeur) disponibles sur chaque dispositif. Pour un nœud non contraint, la totalité des services est déployée. Dans le cas d'un nœud contraint, un service peut être bridé ou même absent.

Ce choix permet de gérer les aspects du contexte sur chaque machine de manière indépendante et de modifier efficacement les composants applicatifs sur le nœud puisqu'ils ne sont pas généralement nombreux. En contrepartie dans ces approches il n'est pas possible que les composants situés sur une même machine puissent être gérés par des gestionnaires distincts, par exemple pour utiliser des algorithmes de prise de décision différents. À l'inverse, les composants sur plusieurs machines pourraient être mieux gérés par un gestionnaire unique. Par exemple, il est plus simple de spécifier une seule politique qui gère plusieurs composants quand les règles d'adaptation sont les mêmes et qu'il existe des dépendances fortes entre eux. Finalement, certains nœuds peuvent ne pas disposer de ressources suffisantes pour exécuter des processus d'adaptation complexes.

Pour conclure, la façon de répartir les mécanismes de la gestion d'adaptation dans le contexte de systèmes distribués mérite une étude plus approfondie. Les infrastructures existantes manquent de flexibilité dans le placement des gestionnaires d'adaptation et dans les critères adoptés pour définir le champ d'action de chacun d'eux. Ces aspects peuvent limiter à la fois la qualité de l'adaptation et la performance du système d'adaptation soit en temps d'exécution, soit en ressources consommées.

2.4.3 Synthèse

Dans la pratique, les canevas et les langages de politiques d'adaptation sont encore rudimentaires notamment pour gérer l'adaptation dynamique d'une manière distribuée et coordonnée. Par ailleurs, les approches existantes pour répartir le système d'adaptation ne sont pas flexibles. Les travaux existants se sont surtout intéressés à la spécialisation du comportement du système d'adaptation mais très peu à sa structure et à sa distribution. On ne trouve pas d'outils pour personnaliser la structure du système d'adaptation et sa distribution. Nos travaux cherchent à améliorer cette situation en facilitant la tâche du concepteur d'adaptation pour atteindre une meilleure qualité de l'adaptation.

2.5 Conclusion

Ce chapitre a présenté le concept d'adaptation dynamique d'abord d'une façon générale puis en mettant l'accent sur les enjeux de la gestion distribuée de l'adaptation. Nous avons examiné des travaux existants d'une part du point de vue conceptuel, d'autre part en ce qui concerne la façon dont ils étaient mis en œuvre. Nous avons sur ce plan particulièrement cherché à identifier les mécanismes pour coordonner plusieurs processus d'adaptation distribués. Enfin nous avons étudié les facilités proposées pour utiliser les infrastructures d'adaptation.

Le nombre de travaux qui ont abordé le problème de coordination d'activités de gestionnaires d'adaptation dans le contexte des applications distribuées est limité. La plupart des travaux dans le domaine de l'adaptation utilisent des algorithmes de prise de décision centralisés. Peu de recherches ont visé à fournir un modèle qui permet de coordonner les activités de plusieurs décideurs distribués pour choisir la stratégie d'adaptation la plus appropriée. Les systèmes qui ont adressé le problème de prises de décision distribuées utilisant plusieurs acteurs qui collaborent ensemble n'ont pas proposé de facilités pour détecter et résoudre les conflits éventuels entre eux car ils s'en tiennent à des schémas établis de coopération qui garantissent statiquement la cohérence. Cette solution trouve ses limites

dans des environnements large échelle, très hétérogènes et très variables.

Quant aux mécanismes de contrôle des exécutions d'actions d'adaptation distribuées, l'accroissement de la complexité des systèmes a mis en évidence les limites des modèles de coordination centralisée [MMB03]. Néanmoins les mécanismes de coordination distribués qui sont proposés sont fortement spécifiques aux applications qu'ils adaptent et au type d'adaptation en question, comme on l'a vu avec le système Cactus. Nous pensons donc que les mécanismes de coordination distribués n'ont pas abordé toutes les fonctions qu'un concepteur d'adaptation pourrait souhaiter.

Par ailleurs, les approches existantes ont défini des moyens pour faciliter la spécialisation du comportement d'un système d'adaptation sans s'intéresser à la personnalisation de la répartition des mécanismes d'adaptation. Des facilités sont nécessaires pour permettre de personnaliser à la fois le comportement, la structure et la distribution d'un système d'adaptation en fonction des besoins.

De notre étude il ressort que de nombreuses améliorations dans le domaine des architectures logicielles pour la coordination des activités de plusieurs gestionnaires d'adaptation distribués sont souhaitables et identifiables. L'effort pour spécifier les mécanismes de coordination demeure actuellement trop important pour satisfaire les besoins grandissant en systèmes d'adaptation pour environnements très dynamiques et hétérogènes. Notre contribution vise à simplifier le développement de systèmes distribués auto-adaptables, en offrant une plate-forme générique de coopération distribuée responsable de coordonner plusieurs modules de contrôle d'adaptation, ainsi que les moyens de la mettre en œuvre et de l'utiliser.

Chapitre 3

Adaptabilité des systèmes de réplication

3.1 Introduction

La réplication de données est une technique souvent utilisée par des applications distribuées pour satisfaire des exigences de disponibilité des données, de temps d'accès à celles-ci [Smi98] ou de performances ou encore de tolérance aux fautes [MMVB00]. Ainsi, par exemple, pour satisfaire des exigences de disponibilité, certaines applications utilisent des solutions qui répliquent les données sur des nœuds du réseau proches des utilisateurs [DPS+94]. Néanmoins, une solution de réplication n'est pas adéquate pour toutes les applications manipulant des données. En effet, chaque application peut avoir des exigences spécifiques comme la réduction du temps d'accès aux données ou la garantie de l'équilibrage de charge de traitements sur les serveurs. De plus, les systèmes de réplication peuvent s'exécuter dans des environnements ayant des caractéristiques différentes. Par ailleurs, l'environnement d'exécution du système de réplication est généralement fluctuant et les exigences envers lui peuvent évoluer au cours du temps. En conséquence, le contexte d'exécution doit être pris en compte dans la gestion de données répliquées afin de choisir la configuration appropriée et de mieux satisfaire les exigences de l'application.

Ce chapitre analyse et compare les travaux qui proposent des solutions de réplication au regard de la prise en compte de l'environnement d'exécution. Il ne s'agit pas de dresser un état de l'art complet des systèmes de réplication mais d'analyser la manière dont l'aspect adaptation et les mécanismes qui y sont associés ont été conçus et mis en place. Les trois paragraphes qui suivent présentent cette analyse et sont structurés suivant trois grandes fonctions d'un système de réplication :

- le placement de répliques, qui décide du nombre et l'emplacement de celles-ci (paragraphe 3.2)
- la sélection de la (ou les) réplique(s) concernée(s) par une demande d'accès (lecture ou écriture) aux données (paragraphe 3.3)
- la cohérence de répliques, qui gère leur divergence (paragraphe 3.4).

Enfin, le paragraphe 3.5 conclut ce chapitre.

3.2 La gestion du placement de répliques

Plusieurs travaux de recherche proposent des stratégies de placement de répliques de manière à satisfaire les exigences de l'application qui les manipule. Dans ce paragraphe,

nous étudions ces approches et nous discutons leurs capacités à prendre en compte l'environnement d'exécution.

3.2.1 Terminologie

Un système de réplication offre un service qui s'occupe de créer les répliques manipulées par une application sur différents sites de l'environnement d'exécution. Dans ce cadre, une *réplique* est une copie d'un objet stockée sur un site. Nous considérons un objet comme pouvant encapsuler des données et/ou des traitements. Ainsi, une photo, un document Web, une table relationnelle, un tuple, un document XML peuvent être des objets mais aussi une instance d'une classe dans un langage orienté objet. De plus, nous considérons qu'un objet peut être composite. Par exemple, un document Web est un objet constitué d'une collection de pages, images, code Javascript...

Le *schéma de réplication* définit le nombre et la localisation des répliques de chaque objet. Une *stratégie de placement* détermine le schéma de réplication pour ces objets. Le problème de placement de répliques (*« Replica Placement Problem »* ou RPP) consiste à décider le schéma de réplication à utiliser [FXL08]. Enfin, le terme *degré de réplication* est souvent utilisé dans la littérature pour désigner le nombre de répliques d'un objet géré par le système de réplication.

3.2.2 Le problème de placement de répliques

Le RPP a été abordé dans de nombreux domaines de recherche comme les bases de données distribuées [Ape88], les serveurs de vidéo [BP96], les serveurs Web [LSBS06, TX04], les réseaux de distribution de contenus [RGE02] et les grilles [FKNT99]. Dans tous ces domaines, le schéma de réplication est généralement calculé comme une fonction de coût. Celle-ci est paramétrée par des métriques qui sont associées (directement ou indirectement) aux exigences de l'application ou du concepteur du système de réplication et qui peuvent être globales et/ou moyennes [FXL08] ou calculées pour chaque utilisateur, voire pour chaque demande d'accès [TX04, JGN06, WLW06, FXL07].

Les deux paragraphes suivants montrent la diversité de ces métriques et exigences dans les travaux existants afin de justifier le besoin d'adaptation.

Métriques et exigences. Deux types d'exigences doivent être satisfaites par la fonction de coût représentant le RPP : les exigences de l'application et les exigences du concepteur du système de réplication. Les premières concernent essentiellement le temps d'accès aux objets répliqués [KKM02] et le taux de demandes d'accès non servies en cas de pannes [NYGS06] ou de partition du réseau [PMND07]. Les deuxièmes ont trait au coût de la réplication en terme de volume de stockage et d'utilisation des ressources réseau. En général, ces dernières s'opposent aux exigences des applications. Ainsi, par exemple, pour réduire le temps d'accès aux objets répliqués il est intéressant d'avoir un degré de réplication important. Néanmoins, plus le degré de réplication est important, plus le volume de stockage utilisé pour la réplication augmente [LA04]. La fonction de coût doit donc correspondre à un compromis entre les deux types d'exigences.

Heuristiques. Le nombre et la complexité des métriques pouvant être utilisées dans la fonction de coût, font du RPP un problème NP-Complet [KKM02]. Ainsi, par exemple, concernant les exigences des applications, le temps d'accès aux objets répliqués peut être estimé en fonction de la distance en nombre de sauts entre le demandeur d'accès et le nœud

qui héberge une réplique [JJK+01], du temps d'accès en lecture [BR01, CKS01, KRR02, RGE02] ou du temps d'accès en lecture et mise à jour [CPP02, KDW01, BFR95].

De même, la littérature est riche en travaux proposant des métriques associées aux exigences du concepteur d'un système de réplication. Par exemple, le volume de stockage utilisé pour la réplication est estimé par [BR01, KRR02] en fonction du volume de stockage dans chaque nœud et dans [JJK+01, RGE02, CPP02, KDW01] en fonction du nombre de répliques de chaque objet.

Des heuristiques ont donc été proposées pour simplifier le calcul du schéma de réplication. Ces heuristiques limitent le nombre de métriques utilisées [KDW01] et/ou leur applicabilité à des conditions d'exécution particulières [KDW01, UC04]. Dans le paragraphe suivant, nous nous intéressons à ces heuristiques et en particulier à la manière dont elles tiennent (ou peuvent tenir) compte de l'évolution de l'environnement d'exécution.

3.2.3 Adaptation et heuristiques

La modification du schéma de réplication peut être nécessaire suite à un changement dans l'environnement d'exécution du système de réplication ou des exigences des applications (ou des utilisateurs) qui en tirent partie. Les travaux dans la littérature tiennent compte de ces changements soit en réévaluant les métriques de l'heuristique, soit en changeant l'heuristique elle-même. Nous analysons ces approches dans les paragraphes qui suivent.

Évaluation des métriques du RPP. La fréquence d'évaluation/estimation des métriques utilisées dans la fonction de coût joue un rôle important pour déterminer la nature adaptative d'une solution au RPP. En effet, cette estimation doit être réalisée à la création de l'objet à répliquer (approches statiques) mais peut également être réalisée régulièrement en fonction de l'évolution des métriques utilisées dans l'heuristique (approches dynamiques) [CWC+07]. Dans les approches statiques, le schéma de réplication est calculé une fois lors du placement des répliques sur les sites. Les répliques persistent jusqu'à ce qu'elles soient supprimées par leur propriétaire ou que leur durée de vie ait expirée. La majorité de travaux ont adopté cette approche de part sa simplicité [JJK+01, KRR02, TX04]. Néanmoins, lorsque les conditions d'exécution changent, le schéma de réplication calculé peut s'avérer inadéquat.

Dans les approches dynamiques, l'évolution des métriques de l'heuristique est surveillée et le schéma de réplication est modifié afin de répondre aux exigences de l'application et ses utilisateurs. Ainsi, par exemple, dans [PKKY03] les auteurs proposent l'algorithme BHR (*« Bandwidth Hierarchy-based Replication »*) qui cherche à minimiser le temps d'accès aux objets manipulés. Pour ce faire, les nœuds sont regroupés en régions. Pour chaque nœud d'une région, la fréquence d'accès à chaque objet est mesurée. Si cette fréquence est considérée importante, une réplique de l'objet est créée dans un nœud de la région s'il n'en existait pas déjà une. Ainsi, les objets le plus souvent accédés par les nœuds d'une région seront toujours présents dans celle-ci.

Changement d'heuristique. Bien que les approches dynamiques permettent de satisfaire certaines exigences des applications malgré des changements des métriques, elles appliquent la même heuristique pour recalculer le schéma de réplication. Or, un changement des exigences ou des conditions d'exécution peut rendre l'heuristique inadaptée. Connaître les caractéristiques des heuristiques pour décider celle qui répondra au mieux aux exigences est alors fondamental.

Dans [FXL08], les auteurs comparent six heuristiques dont l'objectif est de satisfaire les mêmes exigences de temps d'accès pour chaque utilisateur. Les critères de comparaison utilisés incluent la complexité des heuristiques en terme de temps de calcul du schéma de réplication, leur passage à l'échelle, et les métriques qu'elles considèrent. Les auteurs concluent que pour les mêmes exigences de temps d'accès, les algorithmes *L-Greedy-Insert* et *L-Greedy-Delete* offrent un coût de stockage (volume de stockage utilisé par nœud) moindre. Toutefois, il s'agit des deux algorithmes les plus complexes, ce que les rend inadaptés lorsque le temps de calcul du schéma de réplication est critique. Au contraire, *Greedy-Cover* est l'algorithme le plus rapide mais les coûts de stockage sont les plus élevés. Enfin, seulement les algorithmes *Core-Selection* et *TTL-Based* sont distribués et passent à l'échelle.

À notre connaissance, malgré des caractéristiques et des métriques différentes utilisées par les heuristiques proposées dans la littérature, il n'existe pas de travaux qui s'intéressent au changement d'heuristique pour le calcul du schéma de réplication.

3.2.4 Synthèse

Les travaux qui portent sur le problème de placement de répliques permettent de calculer un schéma de réplication en évaluant une fonction de coût. Les paramètres de cette fonction correspondent à des métriques choisies de manière à refléter les exigences des applications ou du concepteur du système de réplication lui-même. La prise en compte de l'évolution des conditions d'exécution, voire des exigences des applications, est effectuée dans la littérature en réévaluant les métriques utilisées pendant l'exécution et en modifiant le schéma de réplication en conséquence. Néanmoins, les approches existantes ont des propriétés différentes et basent leur calcul du schéma de réplication sur un ensemble prédéfini de métriques répondant à des exigences bien définies. Malgré ces différences, aucun travail dans la littérature ne propose le changement d'heuristique pour le calcul du schéma de réplication.

3.3 La sélection de répliques

L'existence de plusieurs répliques d'un objet rend nécessaire la sélection de la (des) réplique(s) appropriée(s) pour traiter les demandes d'accès (lecture ou écriture) clientes. Une *stratégie de sélection* détermine la réplique à sélectionner pour chaque demande d'accès de manière à satisfaire les exigences de l'application et du développeur du système de réplication. Dans cette section, nous nous intéressons aux stratégies de sélection existant dans la littérature et nous analysons leur prise en compte de l'environnement d'exécution et son évolution.

3.3.1 Stratégies de sélection de répliques

Le problème de la sélection de répliques a été abordé dans la littérature en proposant des algorithmes (ou stratégies) qui, en fonction d'un ensemble de métriques, déterminent la réplique à sélectionner pour satisfaire les demandes d'accès aux objets répliqués. À l'image des travaux sur le RPP, les stratégies proposées diffèrent selon les exigences qu'elles cherchent à satisfaire mais aussi selon les métriques qu'elles utilisent pour décider de la réplique à accéder.

En terme d'exigences des applications, la grande majorité des travaux existants considère le temps de réponse aux demandes d'accès aux objets répliqués [FBZA98, SBSV98, VBSS99, VBSS99, ZAFB00, TM05]. Les métriques utilisées pour estimer ce temps sont

très variées et peuvent correspondre à la bande passante du réseau [SSK97], à la distance topologique en nombre de sauts entre le client et les répliques [GS95a] ou au temps de réponse d'un nœud [FBZA98, TM05].

Par ailleurs, les exigences associées au système de réplication lui même sont plus variées et incluent notamment la minimisation de l'utilisation des ressources réseau [FJJ+01, GS95a] et le partage de charge entre les répliques existantes [FBZA98, TM05]. La distance topologique en nombre de sauts entre le client et les répliques est utilisée dans [FJJ+01, GS95a] pour décider de la satisfaction de la première exigence. Pour la deuxième, [FBZA98] utilise le temps moyen d'attente des requêtes sur les nœuds et [TM05] le nombre de requêtes en attente et le taux de traitement sur chaque nœud.

À l'image du RPP, le problème de la sélection de répliques peut être présenté comme l'évaluation d'une fonction de coût dont les métriques utilisées dépendent des exigences à satisfaire. Néanmoins, à la différence du RPP, les solutions proposées sont très simples et se contentent de satisfaire une exigence en utilisant une ou deux métriques. Dans le paragraphe suivant, nous présentons ces solutions et les analysons selon leur capacité à tenir compte de l'évolution de l'environnement d'exécution et des exigences des applications.

3.3.2 Adaptation et stratégies de sélection de répliques

Les préférences pour le choix de la réplique à sélectionner pour répondre à une demande d'accès, peuvent varier d'un utilisateur à un autre. Ainsi, par exemple, un utilisateur peut préférer l'accès à la réplique la plus à jour tandis qu'un autre préfère accéder le plus rapidement possible. Pour cette raison, certains travaux permettent de spécialiser la stratégie de sélection en spécifiant les métriques à utiliser. Au delà de cette spécialisation, des changements dans l'environnement d'exécution et/ou des exigences des applications peuvent rendre nécessaire la réévaluation de la réplique à sélectionner pour satisfaire les demandes d'accès à un objet. Cette opération est réalisée dans la littérature soit en réévaluant les métriques utilisées par la stratégie de sélection, soit en changeant la stratégie utilisée en fonction des nouvelles conditions d'exécution. Nous analysons ces approches dans les paragraphes qui suivent.

Spécialisation de la stratégie. Dans les travaux existants, cette spécialisation consiste à permettre la définition de toutes ou une partie des métriques utilisées par la stratégie pour la sélection de répliques. Un travail important dans ce sens a été proposé par [FM03]. Cet article présente une stratégie de sélection visant à minimiser le temps de réponse à une demande d'accès. Il propose un ensemble de *métriques de base* relatives à la charge du réseau et des nœuds (bande passante disponible, disponibilité d'un nœud) et permet aux utilisateurs d'en définir d'autres en fonction de celles-ci. Ainsi, par exemple, un utilisateur peut définir le temps de latence comme la moyenne des temps de latence mesurés les trois derniers jours. De plus, des politiques de sélection de répliques utilisant les nouvelles métriques peuvent être définies. Des seuils optimums, minimaux et/ou maximaux peuvent ainsi être définis permettant à la stratégie de sélection de décider de la réplique la plus appropriée.

Évaluation des métriques utilisées pour la sélection de répliques. Comme pour les travaux sur le RPP, les stratégies de sélection de répliques peuvent être *statiques*, lorsque les métriques utilisées sont évaluées une seule fois, avant la réception de la première demande d'accès, ou *dynamiques* si elles sont évaluées régulièrement pour tenir compte de leur évolution. Parmi les approches statiques, certaines utilisent des métriques qui ne dé-

pendent pas de l'environnement. Ainsi, par exemple, [KBM94] utilise le principe du *round robin* pour attribuer les demandes d'accès et [Web97] choisit la réplique aléatoirement. D'autres, utilisent des métriques dépendantes de l'environnement comme la distance entre le nœud demandeur d'accès et les répliques en terme de proximité géographique [GS95b] ou de distance topologique en nombre de sauts [GS95a]. Ces stratégies sont simples à mettre en œuvre mais elles peuvent donner des résultats qui ne satisfont pas les exigences des applications dans des environnements fluctuants.

Par ailleurs, les approches dynamiques réévaluent régulièrement les métriques considérées, la réplique sélectionnée pouvant alors changer selon la mesure obtenue. Ainsi, [SSK97] mesure la bande passante, [SBSV98] la latence d'accès au nœud [SBSV98] et [FBZA98, TM05] le temps de réponse d'un nœud. La réévaluation de ces mesures est effectuée soit de manière périodique [SSK97, SBSV98, TM05] soit lorsqu'un nœud détecte le changement de valeur d'une mesure [TM05].

Changement de stratégie. Bien qu'intéressante, la réévaluation des métriques utilise la même politique pour le choix de la réplique à sélectionner. Certains travaux dans la littérature ont cherché à caractériser différentes stratégies de sélection. Ainsi, [TM05] considère trois paramètres pour caractériser une stratégie : le temps de réponse à une demande d'accès, la probabilité d'erreur dans la sélection et la probabilité de surcharge d'un nœud. Les auteurs ont évalué ces trois paramètres pour six stratégies de sélection différentes, l'évaluation étant réalisée par simulation en variant le nombre de clients dans un réseau local. Les résultats de l'évaluation montrent que l'algorithme *balanced selection* offre les meilleurs résultats pour les trois critères comparés. Cependant, rien n'est indiqué concernant sa pertinence pour un environnement autre qu'un réseau local.

3.3.3 Synthèse

Les stratégies de sélection existantes mesurent/estiment un ensemble de métriques qui reflètent les exigences des applications et tiennent compte de l'évolution des conditions d'exécution en réévaluant les métriques utilisées. À la différence du placement, une grande majorité des stratégies de sélection ne prend pas en compte le caractère multicritère du choix de réplique. De plus, certains travaux permettent aux utilisateurs de définir leurs propres métriques et des politiques de sélection en fonction de celles-ci. Bien qu'intéressante, cette spécialisation ne tient pas compte de l'évolution possible des métriques. Or, différentes stratégies de sélection ont des propriétés différentes et s'avèrent adéquates dans des conditions d'exécution particulières. Néanmoins, à notre connaissance des travaux permettant de tenir compte de ces différences en changeant de stratégie en fonction des conditions d'exécution sont absents de la littérature.

3.4 La gestion de la cohérence

La dernière grande fonctionnalité d'un système de réplication concerne la gestion de la cohérence des répliques d'un objet. La *cohérence* est une relation qui définit le degré de similitude entre les répliques d'un objet [Mar03]. Des mécanismes permettant de gérer la cohérence, aussi appelés protocoles de cohérence [Mar03], doivent être présents dans tout système de réplication dont les objets gérés peuvent être modifiés. Ces mécanismes doivent mettre en place des solutions concrètes pour l'ordonnancement des demandes d'accès sur des répliques différentes d'un même objet et pour la propagation des modifications réalisées sur une réplique aux autres.

Plusieurs types de cohérence (aussi appelés *modèles de cohérence*) existent dans la littérature qui mènent à des degrés différents de divergence des répliques [Dra03]. Tandis que la *cohérence forte* garantit que toutes les répliques d'un objet sont identiques du point de vue de l'utilisateur, la *cohérence faible* permet une certaine divergence. Lorsque le système de réplication garantit une cohérence forte, le résultat d'une demande d'accès à une réplique quelconque reflète le résultat de toutes les modifications antérieures à la demande. Lorsque la cohérence garantie est faible, une demande d'accès ne reflète pas forcément toutes les modifications antérieures, mais celles-ci seront répercutées au bout d'un temps fini [Dra03].

Des protocoles de cohérence existent pour assurer les différents modèles de cohérence. Mais, pour un même modèle de cohérence, on trouve aussi dans la littérature plusieurs mécanismes qui le mettent en place. Dans cette section, nous présentons un aperçu des protocoles de cohérence existants et nous montrons leur adéquation à des contextes spécifiques ce qui justifie le besoin d'adaptabilité. Ensuite, nous étudions des approches proposant des protocoles adaptables selon l'environnement d'exécution.

3.4.1 Protocoles de gestion de la cohérence

La littérature distingue principalement deux approches pour gérer la cohérence d'un objet répliqué [BS06]. La première, appelée *réplication pessimiste* ou *impatiente*, évite la divergence entre les répliques d'un objet et garantit donc une cohérence forte. La deuxième, appelée *réplication optimiste* ou *paresseuse*, permet une certaine divergence entre les répliques et garantit donc une cohérence faible. Les deux approches sont exploitées dans les systèmes de fichiers, les bases de données et dans le cadre du travail collaboratif [KBH+88, CP01].

L'objectif de ce paragraphe est de présenter ces approches et d'analyser leurs caractéristiques afin d'illustrer le besoin d'adaptation dans ce domaine.

Réplication pessimiste

Principes. Différents protocoles existent pour éviter la divergence des répliques d'un objet. Ils reposent généralement sur le verrouillage des répliques qui ne sont pas à jour et la libération de verrous lorsqu'elles le sont. La cohérence forte est ainsi garantie et les utilisateurs n'observent pas d'incohérences dans les objets répliqués [BHG87].

Quelques travaux existants. Plusieurs protocoles de cohérence ont été proposés qui mettent en place la réplication pessimiste. Ainsi, le protocole à copie primaire [BHG87, Die94, Ora96] élit une réplique comme étant la copie primaire, les autres étant secondaires. Les demandes d'accès en lecture peuvent être réalisées sur n'importe quelle réplique tandis que les mises à jour doivent être effectuées sur la copie primaire. Lors d'une écriture, toutes les répliques sont verrouillées et la mise à jour est propagée de la copie primaire aux secondaires. Si la copie primaire devient défaillante, les répliques restantes assurent l'élection d'une nouvelle.

Simple à mettre en œuvre et à administrer, le protocole à copie primaire garantit néanmoins une faible disponibilité en écriture (limitée à une seule réplique). Ainsi, d'autres protocoles ont été proposés qui autorisent les mises à jour sur plusieurs répliques tout en permettant les lectures sur n'importe quelle réplique. Par exemple, le protocole ROWA (*« Read One Write All »*) [BHG87] autorise les écritures sur n'importe quelle réplique, la propagation de celles-ci aux autres répliques étant atomique. Cette atomicité des mises à jour rend le protocole inadapté lorsque des répliques sont inaccessibles. L'extension proposée par le protocole ROWAA (*« Read One Write All Available »*) cherche à résoudre ce problème en ignorant les répliques qui ne répondent pas lors de la propagation d'une mise

à jour. Il est alors nécessaire de mettre à jour ces répliques avec l'état courant lorsqu'elles redeviennent disponibles. Ce protocole tolère *n-1* pannes, où n est le nombre de répliques, lorsque celles-ci ne partitionnent pas le réseau. En cas de partition, aucun mécanisme de récupération n'est prévu.

Par ailleurs, des protocoles basés sur des quorums ont été proposés [Gif79, Her86, Kel99]. Un quorum est un ensemble minimal de nœuds qui doivent donner leur accord pour qu'une demande d'accès à un objet soit servie. La taille des quorums est en général différente pour les demandes en lecture et en écriture et peut aussi changer d'un objet à un autre. Les quorums sont calculés de manière à assurer qu'une demande d'accès ait toujours comme résultat la réplique la plus à jour. Comparés au ROWAA, ils diminuent le nombre de messages à échanger lors d'une écriture (moins de n messages, où n est le nombre de répliques) et, certains tolèrent les partitions du réseau en reconfigurant les quorums disponibles en cas de déconnexions [Her86].

Enfin, il existe des protocoles basés sur des *jetons*, essentiellement utilisés pour prévenir les conflits dans des systèmes où des nœuds peuvent être déconnectés [GM95, DD08]. Pour accéder à un objet, un nœud doit posséder un jeton (il existe un jeton unique par objet répliqué). Par exemple, [DD08] propose un algorithme qui se base sur un arbre fixe constitué des nœuds hébergeant une réplique locale d'une donnée. Les messages servant à maintenir la cohérence sont routés selon cet arbre. Une seule réplique peut être modifiée à un instant donné. Quand une modification d'une réplique est envisagée, le nœud envoie une requête au possesseur du jeton et, dès sa réception, pose un verrou sur la donnée. Lorsqu'il a fini la modification, il demande le relâchement du verrou et propage ses modifications. Bien que ces protocoles réduisent le nombre de messages associés aux mises à jour, les communications associées à l'échange des jetons restent importantes.

Analyse. En prévenant l'apparition de divergences entre les répliques d'un objet, l'approche pessimiste privilégie l'unicité de vue sur toutes les répliques d'un objet, la disponibilité de celles-ci étant secondaire. L'unicité de vue peut être indispensable pour certains types d'applications ou dans des contextes d'exécution particuliers mais induit un nombre élevé d'échanges de messages pour prévenir les éventuelles divergences. De plus, l'approche pessimiste limite considérablement la disponibilité des objets répliqués lorsque la fréquence de modifications est élevée. En effet, des répliques restent inaccessibles pendant leur mise à jour. De plus, l'accès peut être bloqué suite à la déconnexion d'un site. Ainsi, cette approche est mal adaptée à des environnements où des déconnexions de nœuds sont possibles et lorsqu'on exige un temps d'accès rapide.

Par ailleurs, chaque protocole de cohérence utilisé pour assurer le modèle de cohérence forte a des caractéristiques particulières. Le choix du protocole adéquat dépend essentiellement de l'échelle de l'environnement et de la fiabilité des sites et des liaisons entre eux. Il dépend aussi des caractéristiques de l'application, en particulier, l'origine, la fréquence et la nature des accès aux objets répliqués. L'adaptabilité des mécanismes de gestion de la cohérence s'avère donc une propriété intéressante.

Réplication optimiste

Principes. L'approche optimiste accepte consciemment des accès concurrents en lecture et en écriture sur des répliques différentes d'un même objet. Lors d'une demande d'accès, il est possible de lire ou d'écrire une réplique qui n'est pas à jour et des mécanismes sont généralement mis en place afin de *réconcilier* des répliques divergentes et de *résoudre des conflits* détectés lors de la réconciliation [RHR+94, KS95].

Quelques travaux existants. La réplication optimiste a été proposée dans de nombreux domaines comme les systèmes de fichiers répartis ([KS95, RHR+94, Rat98]), les bases de données ([PST+97, BK97]) ou encore dans le cadre du travail collaboratif (*« Computer-Supported Cooperative Work »*, CSCW) ([KBH+88, SE98, AFK+95]).

Dans le domaine des systèmes de fichiers, Coda [KS92] a proposé l'utilisation de la réplication optimiste. Il utilise des vecteurs de version pour détecter des conflits entre des répliques d'un fichier. Afin de permettre la résolution automatique de conflits, les utilisateurs peuvent associer aux fichiers un logiciel de résolution de conflits (*« Application-Specific Resolvers »*, ou ASRs). Dès la détection d'un conflit, Coda localise l'ASR associé au fichier et l'exécute.

Bayou [PST+97] est l'un des travaux les plus marquants dans le domaine des bases de données. Il a été conçu pour des bases de données avec des utilisateurs mobiles. Le processus de réconciliation se base sur le transfert des opérations réalisées sur les répliques. Chaque nœud exécute les demandes d'accès dans un ordre arbitraire ; les transactions restent considérées comme provisoires. L'ordre de sérialisation global est l'ordre d'exécution sur un site désigné comme primaire. Les autres annulent leurs états provisoires et ré-exécutent les transactions validées dans l'ordre de leur validation sur le site primaire.

Enfin, dans le domaine du travail collaboratif, un des travaux marquants est le système Lotus Notes [KBH+88] qui permet le travail coopératif entre des utilisateurs mobiles dans une entreprise. Le processus de réconciliation est basé sur la propagation de l'état des répliques et non pas des opérations réalisées sur celles-ci. Lors d'un conflit, il utilise la politique « le dernier écrivain gagne » (*« Last-Writer Wins »*) pour sa résolution.

Par ailleurs, l'analyse de l'état de l'art sur la réplication optimiste montre de nombreuses différences entre les protocoles existants [SS05].

Tout d'abord, un processus de réconciliation dépend du type d'objet répliqué [KRSD01]. Les différences incluent aussi la topologie de communication pour la propagation des mises à jour. Certains protocoles sont mono-maître [MIC98, AL92] ce qui signifie que toutes les mises à jour sont réalisées sur un site maître puis propagées aux autres répliques. D'autres protocoles sont multi-maître [RHR+94, KS95, Rat98, KRSD01]. Pour cette approche, les mises à jour peuvent être effectuées sur n'importe quelle réplique puis échangées en arrière-plan.

Par ailleurs, la propagation des mises à jour peut se faire sous forme d'état [RHR+94, Rat98] ou sous la forme d'opérations [PST+97, KRSD01]. Dans le premier cas, on propage les mises à jour sous la forme du contenu de la donnée répliquée dans sa version actuelle. Dans le deuxième cas, on propage les modifications sous la forme d'une séquence d'opérations qui ont fait évoluer l'état de la réplique vers l'état courant.

Enfin, nous trouvons dans la littérature deux approches de détection de conflits : la détection syntaxique [KS92] et sémantique [KRSD01]. L'approche syntaxique utilise l'ordre d'exécution des opérations et l'approche sémantique utilise les connaissances sur la sémantique des opérations pour détecter les conflits. En ce qui concerne la résolution des conflits, elle peut être manuelle [CP01] ou automatique [KS95]. Le premier type présente deux versions possibles de la réplique. À partir des deux versions de l'objet, un utilisateur peut alors en créer une nouvelle ou fusionner les versions ou encore annuler les effets et soumettre à nouveau l'opération. La résolution automatique des conflits est effectuée par une procédure spécifique à l'application qui prend les deux versions d'un objet et en propose une nouvelle. Généralement, elle est basée sur des contraintes ou des pré-conditions [KRSD01] associées aux opérations.

Analyse. La réplication optimiste autorise des divergences entre les répliques d'un objet et donc garantit un modèle de cohérence faible. Elle ne peut donc être utilisée que lorsque l'application tolère ces divergences. Si tel est le cas, la réplication optimiste est bien adaptée aux environnements où des déconnexions sont possibles : un nœud peut accéder à un objet en étant déconnecté puis, lors de sa reconnexion, le processus de réconciliation effectue les changements sur les autres répliques. De plus, les demandes d'accès sont exécutées sans les délais supplémentaires de la propagation atomique des mises à jour.

Les protocoles mono-maître sont appropriés lorsque la majorité des accès sont en lecture ou s'il y a un écrivain unique. Ils sont simples à mettre en œuvre et évitent certains conflits. Cependant, dans les environnements où la connectivité est limitée et changeante, le nœud maître peut devenir difficile à atteindre.

Pour les protocoles multi-maître avec propagation d'état [RHR+94, Rat98], la surcharge du réseau liée aux échanges dépend de la taille des répliques et la résolution de conflits s'avère difficile du fait de la perte de la sémantique des opérations réalisées [DPS+94, KRSD01]. Ils sont ainsi mieux adaptés lorsque les objets répliqués sont de petite taille, le taux de conflits est faible, et que ceux-ci peuvent être résolus par une règle syntaxique simple comme « le dernier écrivain gagne ».

Concernant les protocoles multi-maître qui propagent les opérations effectuées, la surcharge dûe aux échanges est indépendante de la taille de l'objet mais la complexité algorithmique et l'occupation de l'espace de stockage par les journaux des opérations sont importants.

De manière semblable à la réplication pessimiste, de nombreuses solutions de réplication optimiste existent dans la littérature, chacune étant adéquate à un environnement d'exécution particulier. Proposer des mécanismes permettant de changer la solution adoptée en cours d'exécution peut donc s'avérer intéressant.

Synthèse

Chacune des approches étudiées est appropriée pour satisfaire un ensemble d'exigences comme le degré de disponibilité et de divergence des répliques. La réplication optimiste tolère l'évolutivité des réseaux et peut être intéressante pour certaines applications comme celles qui permettent le travail en mode déconnecté sur les terminaux mobiles. Toutefois, la complexité algorithmique pour maintenir la cohérence des répliques est généralement élevée. De plus, les conflits exigent habituellement un processus de résolution spécifique à l'application. Quant à la réplication pessimiste, elle est parfaitement adaptée aux environnements à petite échelle, avec la disponibilité permanente des nœuds et une connectivité forte.

La gestion de la cohérence d'un système de réplication doit donc prendre en compte les spécificités de l'application et de l'environnement sous-jacent. En général, les caractéristiques du réseau, du type d'objets répliqués et des opérations d'accès ainsi que les besoins des clients en terme de délai d'accès et de modèle de cohérence définissent quelle solution doit être utilisée pour l'application visée. Les mécanismes de gestion de la cohérence doivent donc être génériques et spécialisables pour couvrir un large spectre d'applications. De plus, certains aspects du contexte pouvant varier pendant l'exécution de l'application, ces mécanismes doivent être conçus pour s'adapter à ces évolutions. La section suivante présente des recherches qui ont traité ces aspects.

3.4.2 Adaptation et protocoles de gestion de la cohérence

Dans ce paragraphe, nous étudions la prise en compte de l'évolution de l'environnement d'exécution par les mécanismes de gestion de cohérence existants et montrons leurs limites dans cette prise en compte. Tout d'abord, nous présentons des approches qui permettent la spécialisation des mécanismes. Ensuite, nous abordons des approches qui vont au delà de la spécialisation et qui permettent l'adaptation dynamique des mécanismes proposés.

Systèmes spécialisables

Certains travaux se sont intéressés à la spécialisation des mécanismes de gestion de la cohérence. Ils proposent des canevas qui guident et donc facilitent le développement de tels mécanismes. En général, ces canevas supportent plusieurs approches de gestion de la cohérence et permettent aux développeurs de les configurer selon les besoins applicatifs et l'environnement d'exécution. Cette configuration peut concerner trois types d'actions : la paramétrisation, le choix d'implémentation et l'assemblage de composants, dans le cas d'un canevas à base de ce type d'entités. Dans la suite, nous présentons des travaux représentatifs de chaque type de spécialisation.

Travaux existants.

Paramétrisation : En ce qui concerne la paramétrisation, le canevas TACT (*« Tunable Availability/Consistency Tradeoff »*) [YV02] définit un modèle générique de gestion de cohérence basé sur des unités de cohérence, appelées *conits*. Un conit contrôle la divergence d'une réplique par des seuils, les contraintes de divergence devant être définies par les utilisateurs du système pour chaque réplique. Ainsi, TACT permet la mise en place de différents degrés d'optimisme en fonction des besoins applicatifs.

Choix d'implémentation : Deux travaux sont représentatifs de ce type de spécialisation. Le premier est Globe [KKST98] qui fournit une plate-forme à base d'objets pour le développement d'applications distribuées large échelle et qui permet la réplication de ceux-ci. Il propose pour ceci un modèle d'objet qui inclut une interface générique définissant les opérations de gestion de cohérence. Cette interface doit être implantée en fonction des exigences en terme de cohérence pour chaque objet répliqué.

Le deuxième système représentatif de ce type de spécialisation est IceCube [KRSD01, PSM03]. Il s'agit d'un système de réconciliation générique à base du transfert des opérations qui permet d'intégrer des contraintes spécifiques aux applications. En effet, il traite la réconciliation comme un problème d'optimisation : elle consiste à exécuter une combinaison optimale de mises à jour concurrentes. Pour calculer cette combinaison, il utilise des contraintes entre les opérations effectuées sur les objets répliqués, contraintes qui ont été définies par les développeurs des applications utilisant le système. À partir des journaux présents sur les différents sites, il calcule automatiquement un journal optimal contenant le nombre maximum de mises à jour non conflictuelles qui répondent à ces contraintes.

Choix de l'assemblage de composants : Certaines approches permettent de spécifier un assemblage de composants spécifique structurant les mécanismes de gestion de cohérence. Notamment, [Dra03] réalise un travail important de décomposition des fonctions d'un système de gestion de cohérence afin de proposer le canevas RS2.7. Ce canevas inclut les fonctions identifiées comme obligatoires dans toute solution de gestion de cohérence et il peut être configuré pour en inclure d'autres, optionnelles. Cette spécialisation

s'effectue en définissant l'assemblage des composants correspondant aux fonctions souhaitées, l'implémentation de chaque composant pouvant aussi être spécialisée.

Analyse. Dans tous ces travaux, la spécialisation permet la construction d'un service de gestion de la cohérence sur mesure pour les objets à répliquer. Néanmoins, les capacités d'adaptation des approches existantes sont réduites du fait du nombre limité de types d'adaptation supportées. En effet, la paramétrisation ou le choix d'implémentation limite les stratégies possibles de gestion de la cohérence. Le paradigme à composants adopté par le canevas RS2.7 élargit les possibilités d'adaptation en facilitant, en plus, la composition de composants qui encapsulent le code métier. Nous pensons que ce travail est sur la bonne voie et nous nous en inspirons pour notre proposition.

Cependant, la « simple » configuration d'un canevas de gestion de cohérence n'est pas suffisant pour tenir compte des évolutions des conditions d'exécution et des exigences des applications. La proposition d'un canevas incluant des mécanismes d'adaptation dynamique s'avère nécessaire.

Systèmes auto-adaptables

L'adaptation dynamique dans les solutions existantes permet de prendre en compte l'évolution des exigences des applications et de l'état de l'environnement d'exécution. À notre connaissance, très peu de travaux proposent des systèmes de gestion de cohérence auto-adaptables. Nous avons relevé trois travaux existants. Nous les caractérisons selon les types d'adaptation qu'ils permettent et s'ils séparent la gestion de l'adaptation et du code métier ou non. Dans la suite nous décrivons et analysons ces travaux.

FRACS. FRACS [ZZ03] (*« Flexible Replication Architecture for a Consistency Spectrum »*) est un système de gestion de cohérence qui permet uniquement la modification de paramètres et qui ne sépare pas les mécanismes d'adaptation de ceux de gestion de cohérence, ce qui le rend plus complexe à faire évoluer. Une *fenêtre de mises à jour* est attribuée à chaque réplique et sert de tampon de tentatives de mises à jour locales non validées globalement. Lorsqu'une réplique accumule un nombre prédéfini de tentatives, elle doit les propager aux autres répliques pour validation. Les nouvelles demandes de modification sont bloquées jusqu'à ce que les tentatives de mises à jour aient été validées et retirées de la fenêtre de mises à jour. En fonction de la taille de cette fenêtre, le degré de divergence d'une réplique peut être différent.

Dans FRACS, l'adaptation dynamique consiste à modifier la taille de la fenêtre de mises à jour des répliques en fonction de l'évolution de la charge engendrée par la propagation des mises à jour. Ainsi, un paramètre, appelé *poids*, est mesuré pour chaque réplique. Ce poids peut correspondre à différentes mesures comme le taux de mises à jour. Toutefois, il s'agit d'une connaissance locale de chaque réplique. Pour construire la vue globale du système sur chaque site, les poids sont inclus dans les messages de mise à jour échangés entre les répliques. Ainsi, chaque réplique maintient une vision des poids de toutes les autres répliques et l'utilise pour modifier la taille de la fenêtre.

L'apport principal de cette approche est de permettre de changer dynamiquement le type de cohérence de chaque réplique (cohérence forte lorsque la taille de la fenêtre de mises à jour est égale à 0 et cohérence faible dans le cas contraire). Cependant, il s'agit d'une adaptation paramétrique limitée à la modification d'un paramètre, la taille de la fenêtre de mises à jour. Par ailleurs, les aspects contextuels observés qui guident l'adaptation sont limités essentiellement au taux de mises à jour et à la latence de communication. De plus,

les mécanismes d'adaptation n'étant pas séparés du code métier, il est difficile de prendre en compte d'autres aspects du contexte. Enfin, chaque réplique peut décider de modifier la taille de la fenêtre en fonction des poids de toutes les répliques, aucune coordination de ces décisions n'est prévue dans l'approche.

DARX. DARX (*« Dynamic Agent Replication eXtension »*) [MBS03] est un canevas pour la conception d'applications distribuées multi-agents qui utilise des mécanismes de réplication adaptatifs pour assurer la tolérance aux pannes. Dans ce système, chaque agent peut être répliqué. Pour chaque agent répliqué, le modèle de cohérence utilisé peut être différent et plusieurs modèles peuvent coexister. Les modèles choisis dépendent des exigences de traitements des agents et des caractéristiques de l'environnement.

À l'image de FRACS, DARX permet la modification dynamique du modèle de cohérence utilisé pour un agent répliqué. Il utilise également une fenêtre de mises à jour locales non validées globalement. Cependant, dans DARX la décision de changement de la taille de la fenêtre est prise par un maître élu parmi les répliques d'un agent. Chaque réplique évalue son degré de cohérence et l'envoie au maître élu. En fonction de l'ensemble des degrés de cohérence, il décide du nouveau modèle à adopter et le diffuse aux répliques.

Bien que DARX considère la décision d'adaptation en proposant l'utilisation d'un maître par agent répliqué, la présence dans le système de plusieurs maîtres pose le problème de la coordination des décisions d'adaptation de chacun. Cependant, cette approche ne propose pas de solution pour résoudre ce type de problème. De plus, les mécanismes d'adaptation ne sont pas modulaires ce qui complexifie l'implantation de nouveaux mécanismes ou la modification des existants.

L'approche de J. C. Leonardo. Comme pour FRACS et DARX, [LYO03] propose de changer dynamiquement le protocole de cohérence utilisé pour la gestion d'un objet répliqué. Dans ce travail, le modèle objet Juice [OTY01] a été utilisé pour inclure les mécanismes d'adaptation. Ce modèle permet aux objets de changer leur implémentation à chaud en fonction des changements dans l'environnement d'exécution. En effet, l'objet à adapter est un objet composite qui encapsule trois autres objets. Le premier gère l'état et l'implémentation courante de l'objet adaptable. Le deuxième objet fournit des stratégies d'adaptation à appliquer lorsque des changements se produisent. Enfin, le troisième objet applique l'action d'adaptation.

De par l'utilisation du modèle objet Juice, cette approche sépare la gestion de la cohérence de l'adaptation. Chaque réplique est implantée selon le modèle Juice et encapsule donc ses stratégies d'adaptation et les moyens de réaliser effectivement les changements. Cependant, cette approche ne fournit pas de mécanismes pour permettre aux répliques d'un objet de prendre une décision de façon collective ou de coopérer avec les répliques d'autres objets.

3.4.3 Synthèse

La spécialisation du service de gestion de la cohérence permet de réutiliser des mécanismes génériques puis de les spécialiser en fonction des exigences des applications et de l'environnement d'exécution. Cependant, les systèmes actuels ne sont pas suffisamment adaptatifs. D'une part, ils se limitent à tenir compte de quelques aspects de l'environnement d'exécution, notamment la fréquence de mises à jour des répliques et les caractéristiques réseau. D'autre part, les types d'adaptation supportés sont la configuration de paramètres

génériques et le changement d'implémentation. Ceci limite le spectre des stratégies de gestion de cohérence qui peuvent être utilisées.

Par ailleurs, les mécanismes d'adaptation sont, en général, imbriqués dans la gestion de la cohérence. Pour utiliser un tel système dans un environnement d'exécution différent, il est souvent nécessaire de le reconstruire à partir de zéro ou de modifier son code source. De plus, la gestion de l'adaptation distribuée et coordonnée n'a pas été abordée dans la littérature.

3.5 Conclusion

Ce chapitre a présenté différents systèmes de réplication existant dans la littérature et les a analysé au regard de leur prise en compte de l'évolution des exigences des applications et de l'environnement. Guidée par les grandes fonctionnalités d'un système de réplication (placement, sélection et cohérence de répliques), cette analyse a mis en évidence la richesse de la littérature en solutions pour ces fonctions. Cette richesse s'explique, en autres, par la nature des propositions : souvent centrées sur une unique fonctionnalité, répondant à des exigences spécifiques et/ou applicables dans des conditions d'exécution particulières.

Par ailleurs, nous avons montré que certains efforts ont été réalisés pour modulariser les solutions, permettre leur spécialisation et les rendre adaptables à l'évolution des exigences et de l'environnement d'exécution. Néanmoins, ces efforts restent insuffisants. D'une part, la spécialisation et l'adaptation aux évolutions se limitent à la modification d'un ensemble de paramètres et, pour la gestion de la cohérence, au changement de l'implémentation ou de l'assemblage des composants qui mettent en place la solution. D'autre part, les approches qui proposent des solutions modulaires restent limitées à une fonctionnalité d'un système de réplication et ne considèrent pas l'adaptation coordonnée de leur système.

Construire un système de réplication générique et spécialisable et qui soit muni des mécanismes nécessaires à la prise en compte de l'évolution des exigences et de l'environnement est une tâche complexe. Néanmoins, un tel système présente l'avantage d'être utilisable par un large spectre d'applications indépendamment de l'environnement dans lequel elles s'exécutent. Dans cette thèse, nous proposons d'aider à la construction d'un tel système en proposant un modèle d'un système de réplication modulaire et flexible. De plus, nous proposons un modèle générique permettant l'adaptation dynamique distribuée et coordonnée et nous l'appliquons à notre modèle de réplication pour rendre un système de réplication auto-adaptable.

Chapitre 4

Modèle d'architecture pour l'adaptation distribuée et coordonnée

4.1 Introduction

Les chapitres précédents ont mis en évidence les avantages et les limitations des approches actuelles pour adapter des applications distribuées et, en particulier, des systèmes de réplication de données. Nous avons identifié des besoins et des défis à surmonter pour améliorer les mécanismes d'adaptation, en particulier, les techniques de distribution et de coordination de la gestion de l'adaptation.

Dans cette thèse, nous proposons une approche générique répondant à cet ensemble de besoins et défis pour le développement d'applications distribuées auto-adaptables. Notre premier objectif est de faciliter la construction du système d'adaptation pour ce type d'applications en proposant des outils couvrant son cycle de vie, de la conception à l'exécution. Le deuxième objectif est de fournir une solution de gestion de l'adaptation dynamique distribuée et coordonnée.

Dans ce chapitre, nous nous intéressons essentiellement à l'aspect architectural d'un système d'adaptation distribué en le décomposant en fonctions élémentaires qui identifient et séparent les différentes préoccupations. De plus, nous définissons des mécanismes pour distribuer et coordonner les activités de gestion de l'adaptation dynamique. Nous les illustrons essentiellement par des exemples d'adaptation de systèmes de réplication.

Dans la suite, nous présentons les notations et la terminologie qu'on adopte. Ensuite, nous précisons les principes de notre approche pour la construction d'applications distribuées auto-adaptables. Puis, nous présentons notre architecture logicielle pour la gestion distribuée et coordonnée d'adaptation dynamique.

4.2 Notations et terminologie

Dans cette section, nous définissons le concept de composant ainsi que la terminologie et les notations que nous utilisons dans ce document.

4.2.1 Composant logiciel et modèle architectural

Nous considérons une *application* comme une collection de composants logiciels qui collaborent ensemble et qui peuvent être déployés sur plusieurs machines. Nous définissons un *composant logiciel* comme étant une entité logicielle réutilisable qui fournit et

requiert des interfaces fonctionnelles (liées à la logique applicative) et qui fournit un ensemble d'interfaces de contrôle [1]. Les interfaces fournies sont appelées *interfaces serveur* ; les interfaces requises sont des *interfaces client*. Les interfaces de contrôle permettent de gérer des aspects non fonctionnels comme suspendre/reprendre l'exécution d'un composant et connecter/déconnecter les composants. Ces interfaces sont toujours de type serveur.

La description de l'architecture logicielle d'une application s'intéresse aux composants, à leurs connexions et aux informations de configuration et de déploiement des composants qui la constituent. Il s'agit d'une vision statique à un instant *t*. Au centre de la figure 4.1 nous présentons un exemple de description de l'architecture logicielle d'une application. Elle est constituée de quatre composants (« `comp1` » - « `comp4` ») de trois types différents (« `Type1` » - « `Type3` ») et de leurs connexions. Pour chaque composant, des informations nécessaires à son déploiement y sont associées.

Une architecture logicielle est instanciée et déployée (voir partie basse de la figure 4.1) en utilisant les informations associées aux composants qui le constituent. Elle doit pouvoir évoluer suite à des changements dans le contexte d'exécution. Afin de permettre cette évolution, nous utilisons la notion de *modèle architectural*. Un modèle architectural définit les types de composants existant dans une famille de logiciels et les contraintes à respecter par cette famille, notamment en terme de nombre d'instances de composants, de connexions autorisées entre les types et de paramètres de configuration des types de composants. Il contient ainsi les éléments communs à tous les systèmes qu'on peut instancier et identifie les points de variation possibles. La partie haute de la figure 4.1 présente un exemple de modèle architectural pour une famille d'applications. Il correspond au modèle architectural respecté par la description d'architecture au centre de la même figure. Une application est ainsi développée en spécialisant un modèle architectural défini pendant une phase initiale de conception.

4.2.2 Modélisation d'une architecture logicielle

Nous utilisons UML 2.3 [Gro10] pour la modélisation des composants d'une architecture et leur assemblage. En effet, ce standard propose la notion de *« component »*, pour la modélisation de logiciels à base de composants. Associés à cette notion, plusieurs autres concepts sont définis, notamment celui d'interface offerte et requise, de port, de structure composite et de connecteur. Nous les introduisons rapidement dans ce paragraphe.

UML 2.3 définit un *composant* comme étant une partie d'un système qui encapsule son contenu et qui définit son comportement en terme d'interfaces fournies et requises. Cette définition est compatible avec notre définition de composant. Les *interfaces* contiennent un ensemble d'opérations et de contraintes OCL [Gro10]. Un *port* est un regroupement d'interfaces qui définit un point pour la conduite des interactions entre le composant et son environnement. Nous utilisons dans nos modèles des ports simples (*simple ports*). Il s'agit de ports qui ont une interface unique. Un port est appelé *port fourni* lorsqu'il s'agit d'une interface fournie et *port requis* si l'interface est requise.

Un composant peut être atomique ou composite. Un *composant atomique* est un élément exécutable du système. Un *composant composite* est défini comme un ensemble cohérent de parties appelées *parts*. Dans nos modèles, chaque partie correspond à un composant logiciel.

La connexion entre les interfaces requises et fournies se fait au moyen de *connecteurs*.

1. Nous utilisons le terme *composant* pour faire référence aussi bien à une instance exécutable qu'au composant lui même. Lorsqu'une ambiguïté est possible, le terme *instance de composant* sera utilisé pour désigner une instance de composant en cours d'exécution.

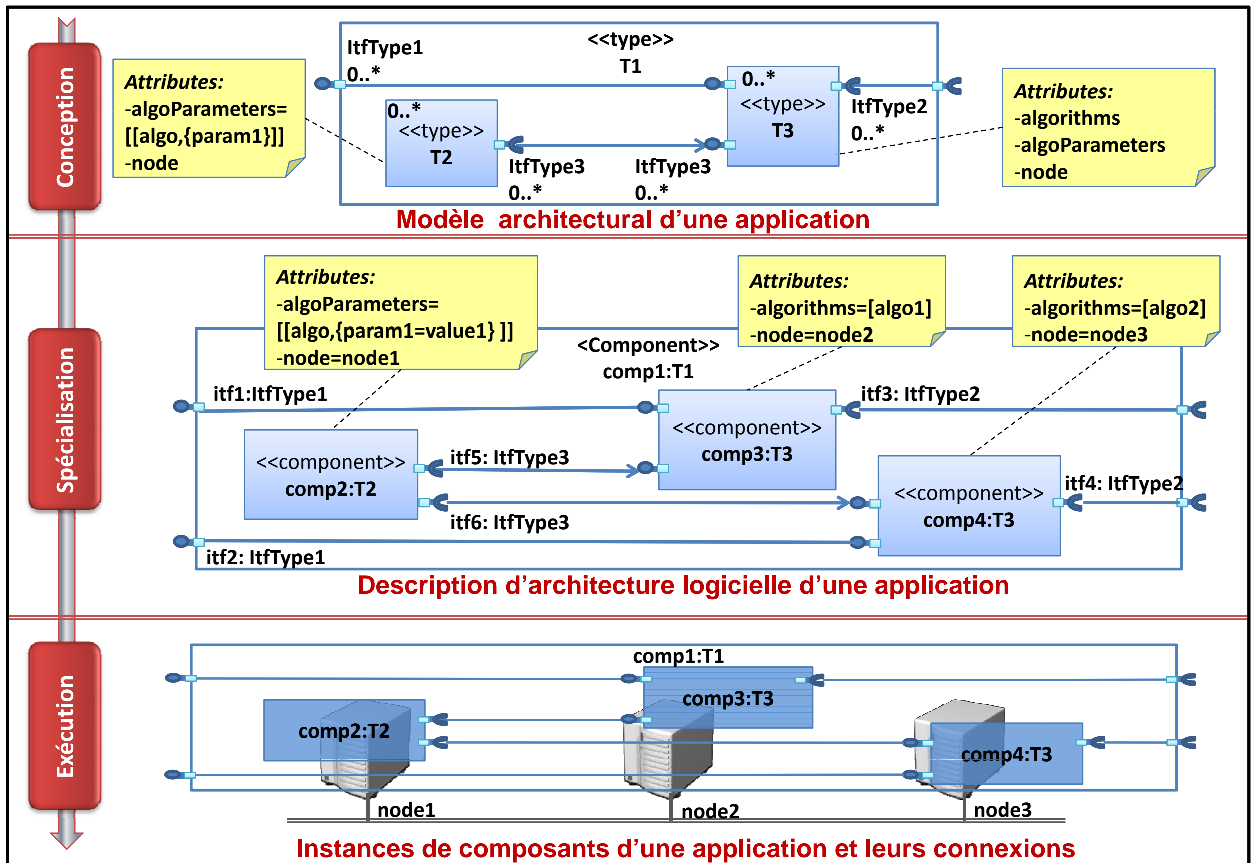

FIGURE 4.1 – Description d'un modèle architectural pour une famille d'applications

On distingue deux types de connecteurs : les connecteurs de délégation et les connecteurs d'assemblage (voir figure 4.2). Le *connecteur de délégation* relie une interface externe via un port à une partie (*part*) interne réalisant l'interface. Cela illustre le transfert de l'appel entre le composant et la partie qui réalise l'opération. Le *connecteur d'assemblage* est un connecteur qui exprime la liaison entre deux ports, un composant fournit donc des opérations que l'autre utilise.

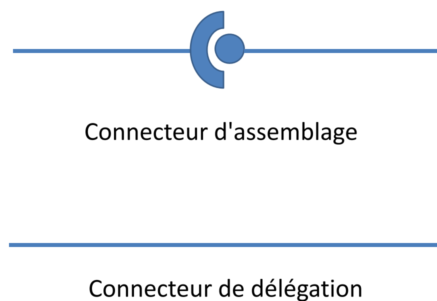

FIGURE 4.2 – Notation graphique des connecteurs UML 2.3

En ce qui concerne la représentation graphique d'un composant, nous utilisons le stéréo-

type « `component` », standard dans UML, pour indiquer qu'il s'agit d'un composant (voir figure 4.3). Les instances d'un composant et ses interfaces peuvent être anonymes ou nommées. Dans le premier cas, nous représentons le nom d'une instance (resp. interface) sous la forme « `:type du composant` » (resp. « `:type d'interface` ») et dans le deuxième, sous la forme « `nom du composant :type du composant` » (resp. « `nom d'interface :type d'interface` »). De plus, dans le but de faciliter la distinction entre les différents types d'interface dans notre représentation graphique des composants, nous suivons les conventions suivantes :

- les interfaces fonctionnelles serveur sont présentées sur le côté gauche du composant,
- les interfaces fonctionnelles client sont présentées sur le côté droit du composant et
- les interfaces de contrôle sont présentées sur la partie haute du composant.

Enfin, nous avons choisi d'annoter un composant par une liste d'attributs et leurs valeurs. Ces attributs représentent les points de variation permettant de spécialiser un composant différemment ou l'adapter dynamiquement en faisant des choix spécifiques sur les valeurs des attributs définis. Les attributs possibles et leurs significations seront décrits dans le paragraphe suivant.

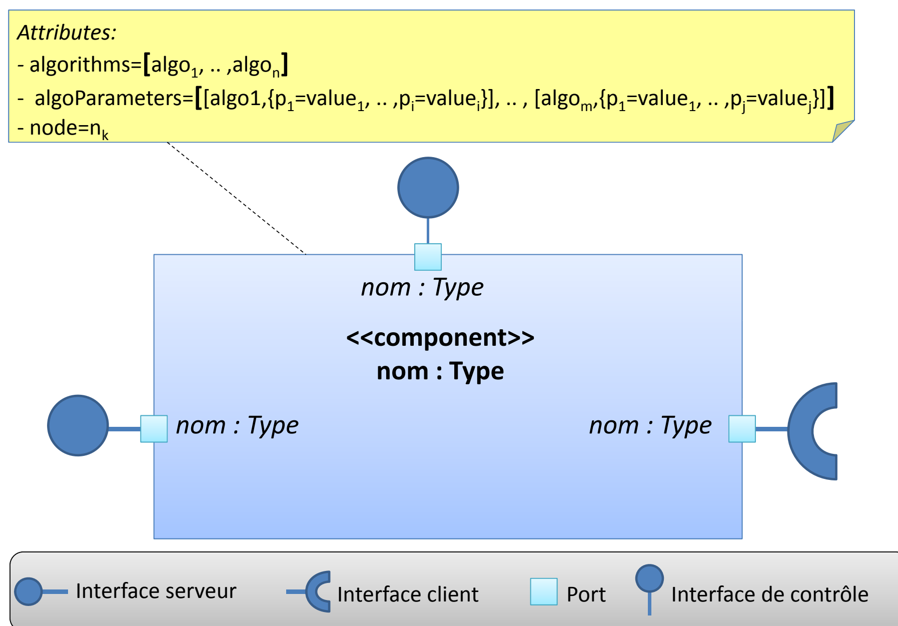

FIGURE 4.3 – Notation graphique d'un composant logiciel

4.2.3 Description d'un modèle architectural

UML ne définit pas de diagramme spécifique pour décrire un modèle architectural. Ainsi, dans ce paragraphe nous présentons notre propre description et sa sémantique et nous donnons sa représentation graphique.

Nous présentons d'abord les éléments pouvant être présents dans un modèle architectural puis nous donnons la représentation graphique de ces éléments.

- *Type d'interface.* Un type d'interface est décrit par une liste d'opérations (leur signature), un rôle (client ou serveur), et une multiplicité.
- *Type de port.* Sa description comprend uniquement le type de l'interface associée.
- *Type de composant primitif.* Il est décrit par le type de ses ports et une multiplicité.

- *Type de composant composite.* Sa description inclut le type des composants qu'il peut contenir, les connexions possibles entre eux et le type de ses ports.
- *Multiplicité d'un type de composant.* Il s'agit d'un intervalle `[min..max]` d'entiers positifs qui indique le nombre possible d'instances de composants de ce type. Nous appelons ce nombre *cardinalité du composant.* La figure 4.4 définit les multiplicités possibles et leur signification.

0..1	Aucune ou une instance
1	Une instance exactement
0..* ou *	Aucune ou plusieurs instances
1..*	Une instance ou plusieurs (au moins une)

FIGURE 4.4 – Multiplicités possibles d'un type de composant

- *Multiplicité d'un type d'interface client.* Il s'agit d'un intervalle `[min..max]` d'entiers positifs qui spécifie le nombre possible d'interfaces de ce type pour un composant qui le déclare. Nous appelons ce nombre *cardinalité de l'interface.* Un nombre `max` égal à *1* signifie que le composant doit avoir au maximum une interface de ce type. Si le nombre `max` est ***, le composant peut avoir un nombre arbitraire d'interfaces de ce type.
  Par ailleurs, lorsqu'un type de composant déclare un nombre `min` de *0*, un composant peut s'exécuter sans avoir une interface de ce type connectée à une interface serveur. Lorsque ce nombre est égal à *1*, la ou les interfaces instanciées doivent être connectées à des interfaces serveur pour que le composant puisse s'exécuter.
- *Multiplicité d'un type d'interface serveur.* Il s'agit aussi d'un intervalle `[min..max]` d'entiers positifs qui spécifie le nombre de composants que peuvent utiliser ce type d'interface. Comme dans le cas des interfaces client, nous appelons ce nombre *cardinalité de l'interface.* Si le nombre `min` est *0*, l'interface serveur peut ne pas être fournie. Si `min` est *1*, l'interface est utilisée par au moins une interface client.
  Par ailleurs, lorsque le nombre `max` est *1*, l'interface peut être utilisée par une seule interface client. Si ce nombre est ***, elle peut être utilisée par un nombre arbitraire d'interfaces client.
- *Attributs.* Les attributs d'un type de composant identifient les propriétés qu'on peut configurer. Ces propriétés concernent l'emplacement physique du composant (attribut *node*), ses comportements possibles, exprimés par les algorithmes qu'il peut encapsuler (attribut *algorithms*), et une liste possible de paramètres de configuration de chaque algorithme (*algoParameters*). Lors d'une spécialisation, un composant peut avoir un nombre arbitraire d'algorithmes, chacun ayant (ou pas) des paramètres de configuration. Ainsi, par exemple, dans la figure 4.1, l'algorithme du composant `comp2` ne peut pas être modifié mais il est possible d'agir sur ses paramètres. L'algorithme des composants `comp3` et `comp4` est modifiable mais n'a pas de paramètres.

La figure 4.5 représente la notation graphique que nous adoptons pour la représentation d'un type de composant. Le stéréotype « `type` », standard dans UML 2.3, est utilisé pour indiquer qu'il s'agit d'un type de composant. Nous précisons en dessous du stéréotype le nom du type. Nous représentons à gauche le type des interfaces serveur, à droite le type des

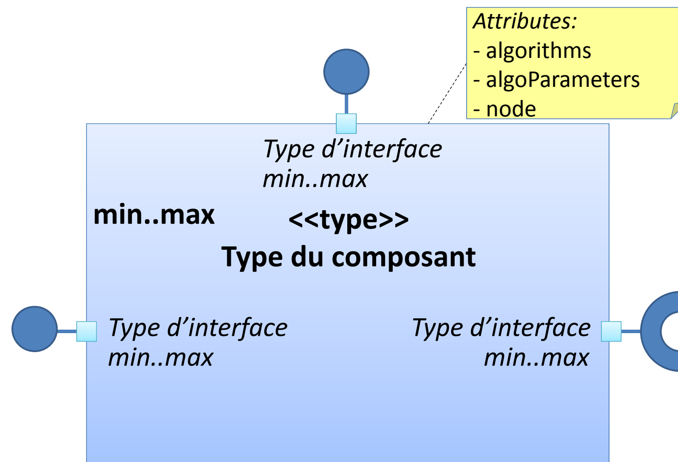

FIGURE 4.5 – Notation graphique d'un type de composant

interfaces client et en haut le type des interfaces de contrôle. Une multiplicité est indiquée pour chaque type d'interface ainsi que pour le type de composant lui-même. Les notations graphiques de type de port et de type d'interface sont les mêmes que celles utilisés respectivement pour le port et l'interface afin de montrer facilement dans les schémas la correspondance entre un modèle architectural et une description d'architecture logicielle. Enfin, la liste des attributs d'un composant est représentée par une annotation. Cette annotation n'est pas nécessaire dans le cas d'un composant composite puisqu'on peut annoter les composants qu'il inclut.

4.3 Principes de construction d'applications auto-adaptables

Nous suivons le principe de séparation entre l'aspect adaptation et l'aspect métier. Ce principe assure l'externalisation des mécanismes de contrôle de l'adaptation de l'application et il a été suivi dans plusieurs approches existantes [GCH+04, FHS+06, OMT08].

Si l'application à adapter est elle même conçue de façon modulaire, son adaptation sera plus fine. L'adaptation d'applications conçues de manière non modulaire peut être effectuée mais elle sera moins précise.

Pour qu'une application soit adaptable, un ou plusieurs de ses composants doivent l'être. Un composant adaptable inclut des moyens pour le surveiller et modifier son comportement. Il peut s'agir des mécanismes de bas niveau fournis par le modèle de composant utilisé (par exemple, création/suppression de composants) mais il peut aussi s'agir d'opérations plus évoluées qui se basent sur ces mécanismes. Dans les deux cas, les composants adaptables exposent des interfaces de contrôle qui incluent ces opérations, interfaces qui pourront être utilisées par un système d'adaptation concret. C'est en utilisant ces interfaces qu'un système d'adaptation agit et surveille une application. Nous appelons l'ensemble système d'adaptation - application, une *application auto-adaptable*.

Quant au système d'adaptation, nous le concevons de façon modulaire afin de faciliter sa spécialisation selon l'application cible à adapter (voir figure 4.6). Pour faciliter sa construction, nous proposons deux outils. Le premier est un modèle architectural pour la gestion de l'adaptation. Il spécifie les contraintes à respecter par tous les systèmes d'adaptation. Un expert en adaptation fournit la description de l'architecture du système d'adaptation. Dans cette description, les composants que doivent constituer un système d'adaptation, les

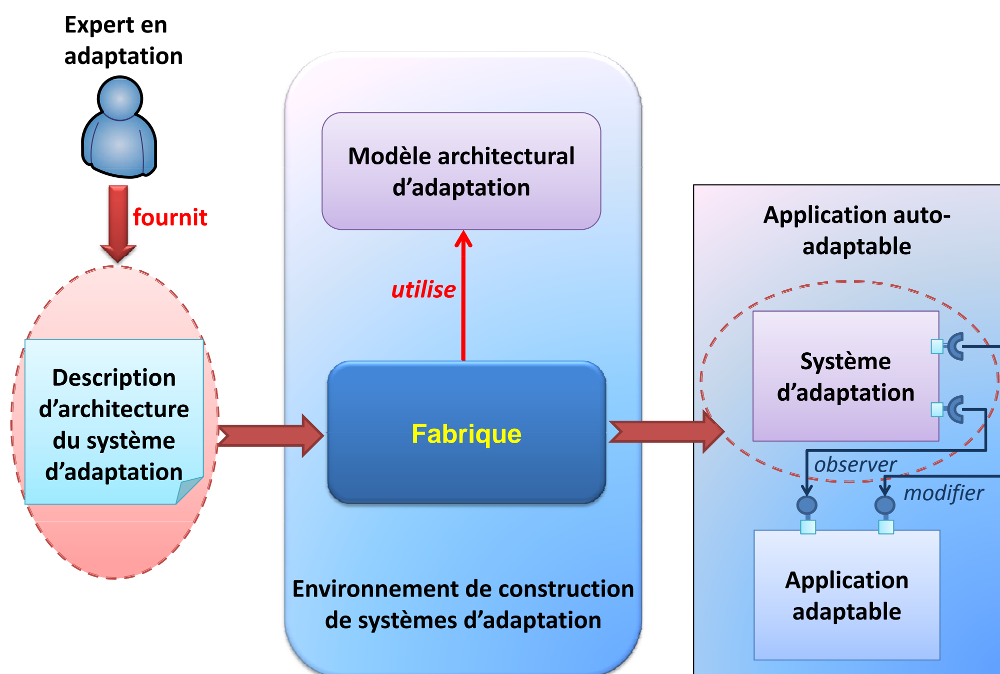

FIGURE 4.6 – Principes de construction d'un système d'adaptation

connexions entre eux et les connexions avec les composants de l'application sont spécifiés.

Le deuxième outil est une fabrique qui, à partir de la description d'architecture du système d'adaptation à déployer, (1) vérifie que les contraintes du modèle architectural sont bien respectées par l'architecture décrite, (2) instancie le système d'adaptation, (3) recherche les références vers les interfaces de contrôle d'adaptation fournies par l'application cible et (4) connecte le système d'adaptation avec l'application selon les connexions spécifiées.

4.4 Fonctions pour l'adaptation dynamique distribuée

Nous considérons un système d'adaptation dynamique comme un assemblage de composants qui implantent les fonctions nécessaires à rendre une application auto-adaptable. Dans un système d'adaptation distribué, les composants qui implantent chaque fonction peuvent être exécutés sur des sites différents. Un tel système peut être utilisé pour adapter une application distribuée. Dans ce paragraphe, nous présentons ces fonctions et leurs dépendances (voir figure 4.7), leur modélisation par assemblage de composants et leur distribution seront adressées dans la section 4.5.

Les deux fonctions principales sont la gestion du contexte et la gestion d'adaptation. Cette séparation permet de distinguer deux étapes importantes de l'adaptation : la surveillance et la détection de changements du contexte d'exécution d'une part, et la réaction au changement d'autre part. La distribution logique et physique des entités logicielles assurant chaque fonction peut être ainsi étudiée séparément. Associées à ces deux fonctions principales, l'observation et la reconfiguration réalisent effectivement les opérations de surveillance et de modification de l'application [FLG06]. Enfin, à la différence des travaux existants, nous considérons la coordination comme une fonction à part entière qui peut

être assurée par des types de composants dédiés comme le montrera le paragraphe 4.5.6.

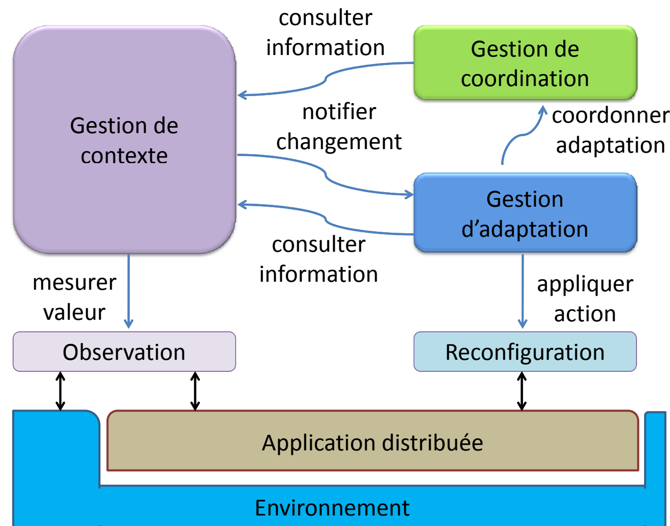

FIGURE 4.7 – Fonctions pour l'adaptation dynamique distribuée

4.4.1 Observation et reconfiguration

Ces deux fonctions permettent respectivement d'ajouter des sondes et d'agir sur l'application. La fonction d'observation assure la collecte des informations sur l'application et son environnement d'exécution. Cette collecte se fait par un ensemble de capteurs physiques comme une caméra pour localiser un patient à domicile. Elle est assurée aussi par des capteurs logiques qui permettent d'ajouter des sondes aux composants de l'application. Par exemple, les requêtes d'accès aux données peuvent être interceptées pour calculer le nombre de lectures et le nombre d'écritures des données pendant un intervalle de temps.

Par ailleurs, la fonction de reconfiguration met en œuvre les actions d'adaptation par un ensemble d'effecteurs. Un effecteur est une entité logicielle spécialisée qui applique des actions d'adaptation primitives comme changer la valeur d'un paramètre ou supprimer une connexion entre deux composants. Ces actions doivent être définies selon les modifications de l'application souhaitées et les facilités associées au modèle de composants choisi pour l'implémentation.

Ainsi, ces deux fonctions ajoutent à l'application les mécanismes de base nécessaires pour être surveillée et adaptée. Nous utiliserons le terme « *application adaptable* » pour désigner l'ensemble des fonctionnalités d'une application et des effecteurs et/ou capteurs logiques.

4.4.2 Gestion du contexte et d'adaptation

Ces deux fonctions assurent les activités nécessaires pour contrôler l'adaptation de l'application distribuée. La première collecte, interprète et agrège les données brutes fournies par les capteurs. Elle rend l'application sensible au contexte et détecte les changements qui nécessitent l'adaptation. Elle notifie ces changements aux entités appropriées qui participent à la réalisation de la fonction de gestion d'adaptation.

La fonction de gestion d'adaptation détermine quelle partie ou quel service de l'application doit être adapté et les moyens d'y parvenir. Pour cela, cette fonction doit décider

des modifications à apporter à l'application, définir les moyens de l'atteindre et contrôler l'exécution des actions d'adaptation réalisées par les effecteurs.

Par la suite, nous utiliserons le terme *« application auto-adaptable »* pour désigner une application adaptable enrichie avec la fonction de gestion d'adaptation et de gestion de contexte.

4.4.3 Coordination

Cette fonction permet de coordonner les activités de la gestion d'adaptation qui peuvent être réparties entre plusieurs composants logiciels. Ainsi, elle permet à ces composants de prendre des décisions d'adaptation de façon collective et de coordonner le contrôle de l'exécution des actions d'adaptation réparties quand nécessaire.

4.5 Modèle architectural pour la gestion distribuée de l'adaptation dynamique

L'adaptation dynamique distribuée doit être basée sur un modèle, une méthodologie et des outils permettant d'en maîtriser la conception et l'implémentation. Pour ceci, nous avons défini un modèle architectural pour développer des systèmes d'adaptation distribués. Ce modèle spécifie la structure et la sémantique d'une architecture d'adaptation distribuée et impose des contraintes explicites pour construire un système d'adaptation respectant l'architecture. Cette structuration est un moyen de briser la complexité inhérente d'un système d'adaptation distribué en le divisant en éléments coopérants et réutilisables. De plus, le modèle définit des points de variation entre une famille de systèmes d'adaptation. Il sera l'élément clé du processus de développement d'un système d'adaptation.

L'apport principal de notre modèle est qu'il permet de gérer l'adaptation de manière distribuée et coordonnée et d'exprimer des points de variation entre les systèmes d'adaptation.

Dans ce paragraphe, nous présentons notre modèle architectural pour distribuer la gestion d'adaptation. Ensuite, nous étendons le modèle pour supporter la coordination des activités des entités logicielles qui gèrent l'adaptation de façon distribuée et nous précisons les techniques de coordination des prises de décision et des contrôles de modifications d'une application. Finalement, nous décrivons en détail le protocole de négociation que nous avons défini dans le but d'assurer la coordination des prises de décisions d'adaptation.

Pour la définition de notre modèle, nous considérons un ensemble de contraintes qui sont : (i) l'indépendance vis-à-vis des standards et des modèles de composants existants, (ii) la flexibilité et l'extensibilité du modèle, (iii) la modularité des mécanismes de la gestion d'adaptation et (iv) la prise en compte de la nature distribuée des logiciels à adapter et de l'hétérogénéité éventuelle de leurs environnements d'exécution.

4.5.1 Types de composants pour la gestion de l'adaptation

Les deux fonctions principales pour rendre une application distribuée auto-adaptable sont la gestion de contexte et la gestion d'adaptation. Elles sont assurées par deux types de composants composites : « `ContextManager` » (gestionnaire de contexte) et « `Adaptation-Manager` » (gestionnaire d'adaptation).

Un composant peut être observable et/ou adaptable. L'observation et l'adaptation sont assurées par les capteurs et les effecteurs qui exposent respectivement une interface d'observation de type « `ObserveItf` » et une interface de reconfiguration de type « `ModifyItf` »

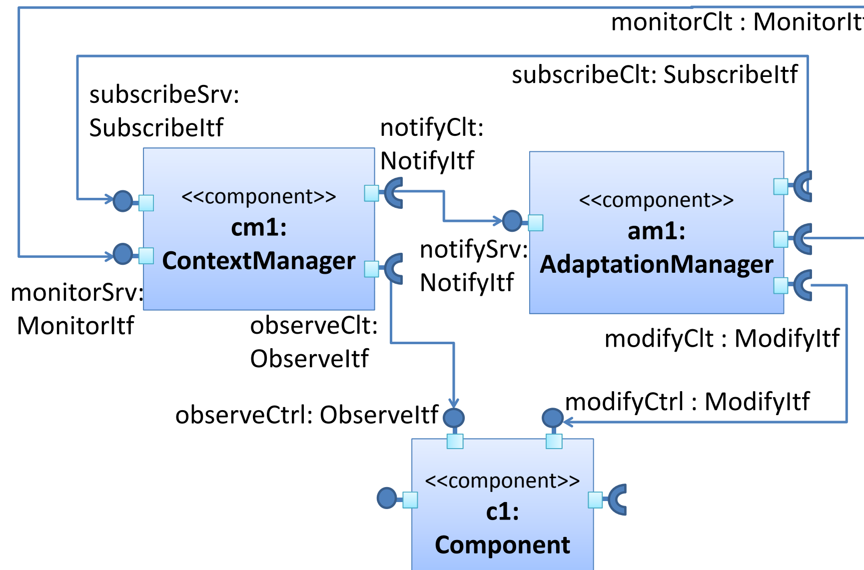

FIGURE 4.8 – Exemple d'un composant auto-adaptable

comme le montre la figure 4.8. Nous appelons *composant adaptable* un composant de l'application qui offre une interface de type « `ModifyItf` » pour permettre sa reconfiguration. Un composant qui inclut des mécanismes pour le surveiller est appelé *composant observable*. Pour qu'un composant soit auto-adaptable, il doit être interconnecté avec un gestionnaire d'adaptation et éventuellement avec un gestionnaire de contexte.

Un composant de type « `ContextManager` » interagit avec des capteurs associés à l'environnement d'exécution et à l'application afin de collecter les informations nécessaires pour caractériser le contexte d'exécution.

Un composant de type « `AdaptationManager` » interagit avec les effecteurs pour contrôler l'exécution des actions à l'échelle d'un ensemble de composants afin d'appliquer des adaptations de comportement, de structure et/ou de distribution de l'application.

Une interaction asynchrone par un mécanisme de « publication-souscription » (*publish-subscribe*) permet de notifier via une interface de type « `NotifyItf` » le gestionnaire d'adaptation des changements du contexte pouvant nécessiter une adaptation. Un gestionnaire d'adaptation s'inscrit à des événements auprès du gestionnaire de contexte en utilisant une interface de type « `SubscribeItf` ». La détection d'un changement approprié du contexte déclenche la notification de l'abonné. Le gestionnaire d'adaptation peut ausssi interroger le gestionnaire de contexte en utilisant l'interface de type « `MonitorItf` » pour récupérer des informations utiles à l'adaptation en mode requête/réponse.

4.5.2 Distribution de la gestion de l'adaptation dynamique

Pour distribuer la gestion de l'adaptation, notre approche permet de définir une architecture d'un système d'adaptation composé d'un nombre arbitraire de composants de type « `ContextManager` » (multiplicité `1..*`) et d'un nombre arbitraire de composants de type « `AdaptationManager` » (multiplicité `1..*`). Le contrôle de l'adaptabilité d'une application distribuée résulte des activités des différents gestionnaires qu'inclut le système d'adaptation (voir figure 4.9).

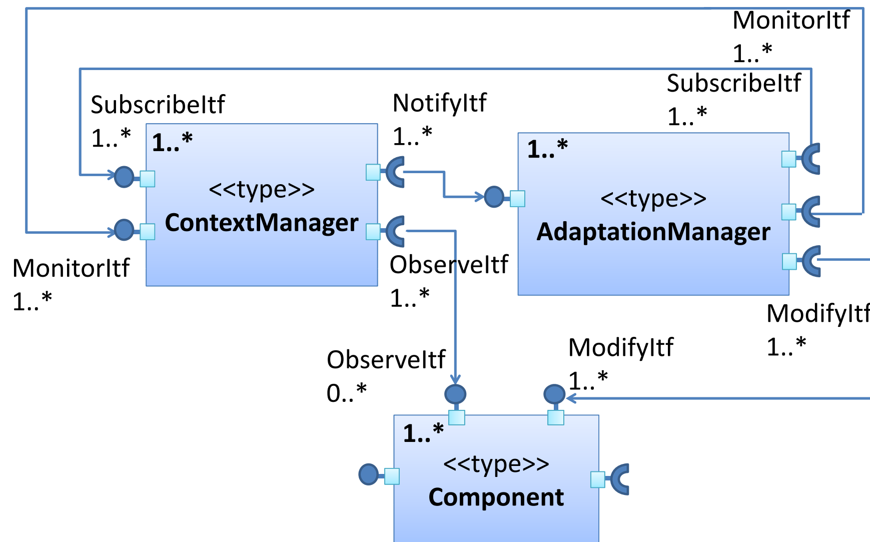

FIGURE 4.9 – Modèle architectural pour une gestion distribuée d'adaptation

Gestionnaires de contexte et d'adaptation. La portée des activités de chaque gestionnaire du contexte est limitée à un environnement spécifique (WLAN, un groupe de nœuds voisins dans un réseau ad hoc...) ou à un ensemble particulier d'aspects contextuels (par exemple un gestionnaire de contexte pour chaque couche : application, réseau et système d'exploitation). En effet, plusieurs gestionnaires de contexte peuvent être utilisés surtout lorsque l'environnement d'exécution est fortement hétérogène et qu'un grand volume d'informations contextuelles doit être géré. La gestion de contexte devient plus efficace puisque chacun d'eux est dédié à des aspects contextuels particuliers. En outre, l'échange d'informations de contexte (collecte des mesures et notification des changements) peut être limité à des sous-domaines réseau pour réduire le trafic réseau et la latence.

Enfin, un gestionnaire de contexte pouvant utiliser un nombre arbitraire de capteurs physiques ou logiques, une interface client de type « `ObserveItf` » a une multiplicité `1..*`.

Notre approche limite aussi la portée des activités de chaque gestionnaire d'adaptation à un sous-ensemble de composants de l'application distribuée. Un sous-ensemble est constitué de composants qui collaborent pour assurer un ou plusieurs services spécifiques et/ou sont placés proches géographiquement. La politique d'adaptation d'une application distribuée est décomposée en sous-politiques, chacune tient compte d'une partie limitée d'informations de contexte et elle est spécialisée pour adapter un sous-ensemble des composants de l'application. Par conséquent, chaque gestionnaire a un champ d'action limité et le comportement d'adaptation résulte des différentes actions locales des différents gestionnaires d'adaptation.

Chaque gestionnaire d'adaptation s'abonne à des événements auprès d'un ou plusieurs gestionnaires de contexte selon les composants qu'il adapte et les champs d'action de ces gestionnaires. Chaque gestionnaire de contexte notifie les gestionnaires d'adaptation abonnés des événements qui les concernent et répondent aux requêtes de consultation de contexte provenant des différents gestionnaires d'adaptation. Pour cela, les interfaces client et serveur de type « `MonitorItf` », « `SubscribeItf` » et « `NotifyItf` » ont une multiplicité `1..*`.

Enfin, un gestionnaire d'adaptation reconfigurant un nombre arbitraire de composants de l'application, l'interface client de type « `ModifyItf` » a une multiplicité `1..*`.

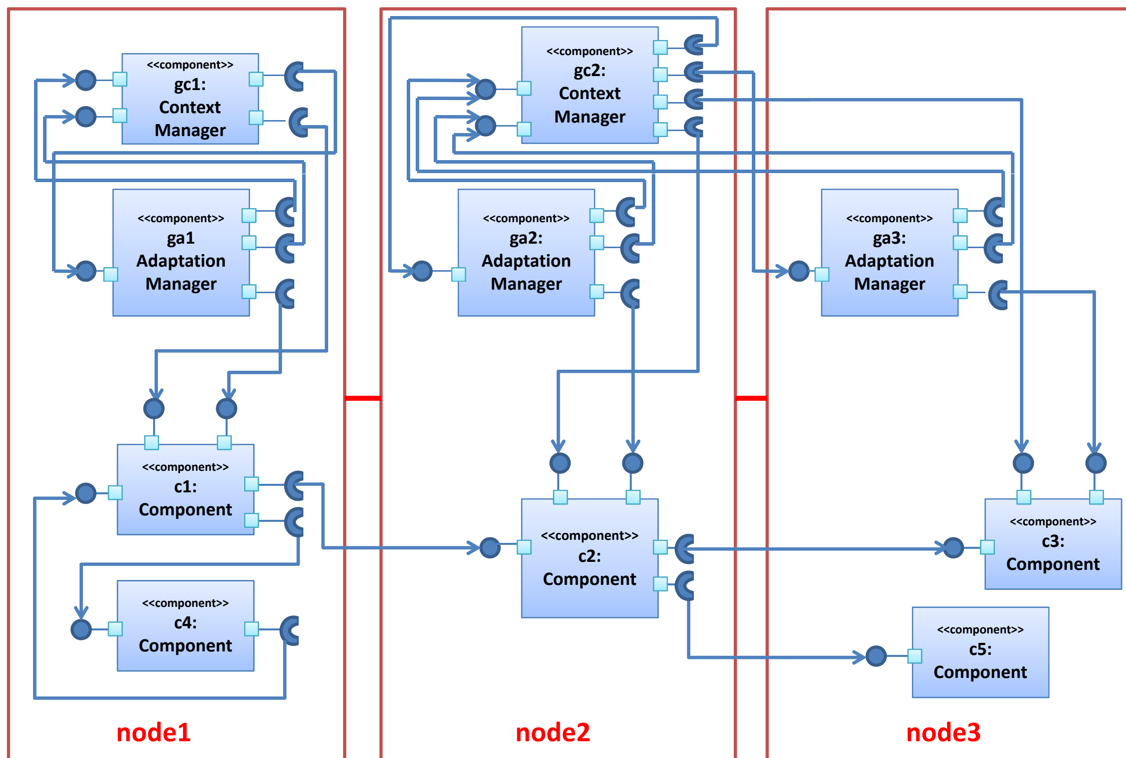

Figure 4.10 – Exemple de distribution de la gestion d'adaptation dynamique

La figure 4.10 montre un exemple d'une application et un ensemble d'instances de gestionnaires de contexte et d'adaptation qui lui sont associées afin de la rendre auto-adaptable. L'application est composée de 5 composants répartis sur trois nœuds, trois parmi eux sont observables et adaptables. Le système d'adaptation se constitue de 2 composants de type « `ContextManager` » et de 3 composants de type « `AdaptationManager` ». Le gestionnaire de contexte *gc1* et le gestionnaire d'adaptation *ga1* sont associés seulement au composant adaptable *c1* car il est placé loin géographiquement des autres composants de l'application. Les composants adaptables *c2* et *c3* sont placés sur deux machines distinctes et proches. Les aspects contextuels auxquels on s'intéresse pour ces deux composants sont similaires. Ainsi, on instancie un seul gestionnaire de contexte *gc2* pour les gérer. On applique des politiques d'adaptation différentes aux deux composants. Donc chacun est géré par un gestionnaire d'adaptation distinct (*ga2* et *ga3*) hébergé sur la même machine que le composant qu'il adapte.

Coordination. Le modèle présenté permet de distribuer la gestion de l'adaptation dynamique. Cependant certains composants de l'application peuvent être dépendants. Ainsi, certains scénarios d'adaptation nécessitent la coordination des activités des gestionnaires d'adaptation pour prendre des décisions de façon collective et pour reconfigurer des composants applicatifs de manière coordonnée. La coordination s'apparente aux interactions

entre un groupe de gestionnaires d'adaptation pour coordonner leurs activités de prises de décisions et de contrôle des reconfigurations des différents composants. Pour cela, les gestionnaires d'adaptation doivent intégrer des composants appropriés pour la coordination. Notre proposition pour gérer l'aspect coordination sera présentée dans le paragraphe 4.5.6.

4.5.3 Modèle architectural du gestionnaire de contexte

Nous avons défini un modèle architectural de base de gestionnaire de contexte (« `ContextManager` ») en nous inspirant de travaux existants sur les systèmes sensibles au contexte [DAS01, CFJ03, KMK+03, GPZ04, RCS08].

Dans notre modèle, un composant de type « `ContextManager` » est un composite qui inclut 6 types de composants primitifs présentés dans la figure 4.11. Un gestionnaire de contexte doit contenir une seule instance de chacun (multiplicité `1`). Comme il s'agit d'un modèle de base, des extensions sont possibles pour ajouter de nouvelles fonctions ou pour raffiner les existantes.

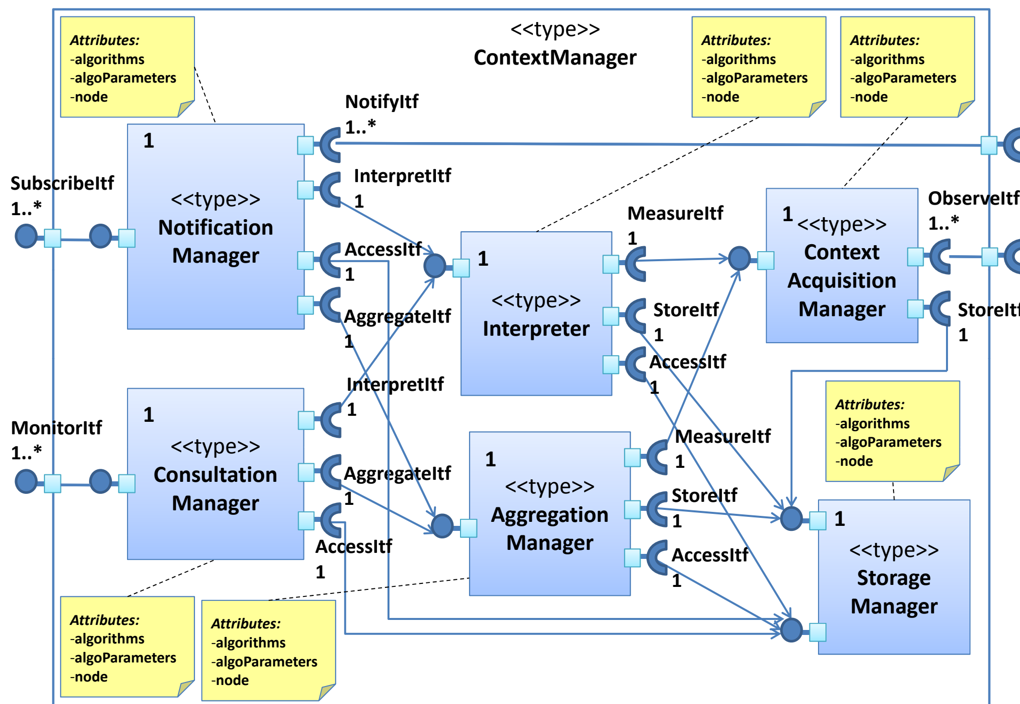

FIGURE 4.11 – Modèle architectural d'un gestionnaire de contexte

Le composant de type « `ContextAcquisitionManager` » (gestionnaire d'acquisition de contexte) permet de collecter des mesures, les traiter et générer des données contextuelles exploitables par les services d'adaptation. Il prend en entrée des mesures des capteurs physiques ou logiques en utilisant des interfaces client de type « `ObserveItf` » et expose des données contextuelles de haut niveau. Par exemple, il peut s'agir de calculer la localisation de l'utilisateur par triangularisation. Ces données sont stockées pour être utilisées par les autres services. Le gestionnaire peut observer plusieurs entités donc l'interface client de type « `ObserveItf` » a une multiplicité `1..*`.

Le composant de type « `Interpreter` » (interpréteur) interprète des données fournies par le gestionnaire d'acquisition. Les données reçues sont interprétées séparément pour chaque type de mesure afin de fournir une donnée contextuelle significative. Par exemple,

un événement de diminution de bande passante peut être à lui seul non représentatif. Par contre s'il se répète dans le temps, il peut indiquer que l'utilisateur s'éloigne d'un point d'accès et donc être significatif. L'interpréteur stocke donc, si besoin, les valeurs mesurées par type d'événement.

La fonction du composant de type « `AggregationManager` » (gestionnaire d'agrégation) consiste à regrouper un ensemble de données contextuelles de type différent afin d'en créer de nouvelles. Tout d'abord, il collecte les données contextuelles nécessaires qui peuvent être stockées pour les utiliser ultérieurement. Une fois toutes les informations sont obtenues, elles sont traitées pour créer une nouvelle information contextuelle. Par exemple, avec les mesures du temps de transmission et du débit du réseau on peut déterminer si le réseau est chargé.

Le composant de type « `NotificationManager` » (gestionnaire de notifications) permet de transmettre les événements du changement du contexte pertinents pour un ou plusieurs gestionnaires d'adaptation via l'interface client de type « `NotifyItf` ». Pour cela cette interface a une multiplicité `1..*`. Par exemple, suite au dépassement d'un seuil sur une jauge de consommation de la mémoire, l'attribut mémoire disponible passe de l'état *suffisante* à l'état *insuffisante*. Les gestionnaires d'adaptation abonnés vont, sur réception de cet évènement, démarrer un processus d'adaptation.

Le composant de type « `ConsultationManager` » (gestionnaire de consultation) permet à un nombre arbitraire de gestionnaires d'adaptation de consulter des données contextuelles via l'interface de type « `MonitorItf` ». Par exemple, on demande des informations sur la disponibilité des ressources sur un groupe de serveurs. Ces informations permettent de choisir sur quel serveur il faut migrer un composant de l'application.

Enfin, le composant de type « `StorageManager` » (gestionnaire de stockage) permet aux autres composants de stocker des données contextuelles et d'y accéder via les interfaces de types « `StoreItf` » et « `AccessItf` ». Par exemple, le gestionnaire d'acquisition de contexte demande à ce type de composant de stocker des données que le gestionnaire d'agrégation utilise.

4.5.4 Modèle architectural du gestionnaire d'adaptation

La figure 4.12 montre le modèle architectural de base d'un composant de type « `AdaptationManager` ». Ce modèle ne permet pas l'interaction entre des composants du même type (« `AdaptationManager` ») et il sera étendu pour permettre une gestion distribuée et coordonnée de l'adaptation.

Ce modèle spécifie trois types de sous-composants essentiels pour la gestion de l'adaptation : « `DecisionMaker` » (décideur), « `Planner` » (planificateur) et « `Executor` » (exécuteur). Un gestionnaire d'adaptation est un composite qui contient une seule instance de chaque type (multiplicité `1`). Les trois types de composant ont un attribut *node*. Ainsi, on peut spécialiser le nœud sur lequel est deployé un composant de chaque type. De plus, il est possible de spécialiser le comportement des composants de type « `DecisionMaker` » et « `Planner` » en donnant une valeur à l'attribut *algoParameters*. Cet attribut sert à configurer l'algorithme qu'implémente le composant. La spécialisation de comportement d'un gestionnaire d'adaptation sera détaillée dans le paragraphe suivant.

Le composant de type « `DecisionMaker` » se charge des prises de décisions d'adaptation et il fournit en sortie une stratégie d'adaptation à appliquer. Dans ce but, il s'abonne à des événements auprès d'un ou plusieurs gestionnaires de contexte en utilisant son interface client de type « `SubscribeItf` » ayant une multiplicité `1..*`. Il est donc notifié des changements du contexte par les gestionnaires de contexte via l'interface serveur de

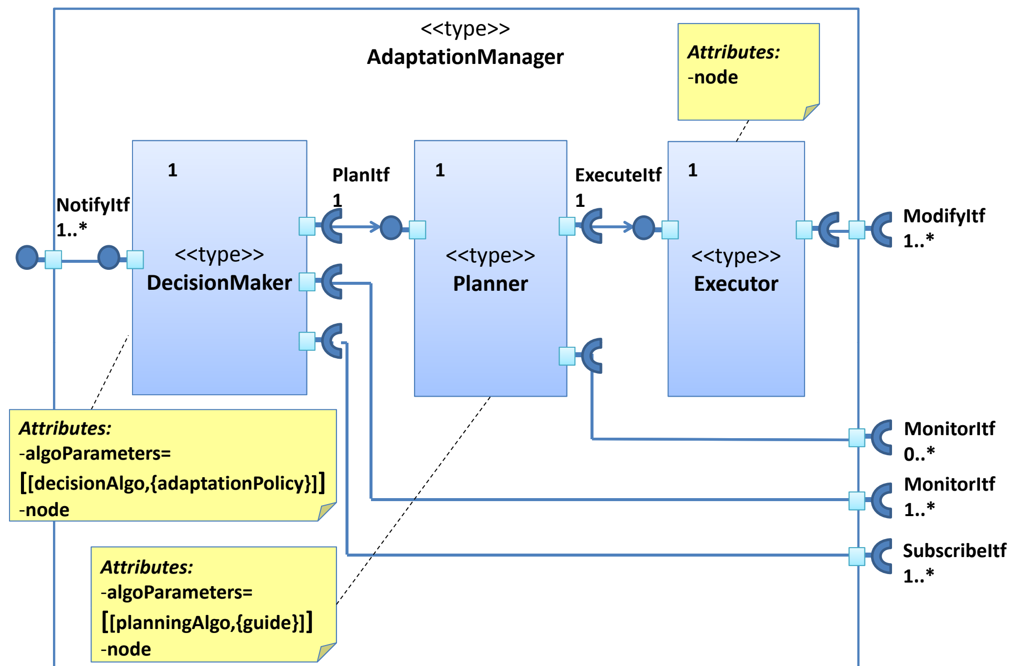

FIGURE 4.12 – Modèle architectural d'un gestionnaire d'adaptation

type « `NotifyItf` ». Il peut être nécessaire aussi d'effectuer des mesures particulières du contexte en complément des informations contenues dans l'événement. C'est pourquoi le décideur peut être lié à un ou plusieurs gestionnaires de contexte par l'intermédiaire de l'interface client de type « `MonitorItf` ». Le décideur choisit la stratégie appropriée qui précise les changements à apporter à la configuration actuelle (par exemple, modifier l'algorithme du placement des répliques). Il en informe le composant de type « `Planner` » en utilisant l'interface client de type « `PlanItf` ».

Le composant de type « `Planner` » identifie une séquence d'actions d'adaptation sous forme d'un plan pour mettre en œuvre la stratégie choisie par le décideur. Par exemple, il peut s'agir de désactiver un composant, le remplacer par un autre et démarrer le nouveau composant. Un composant de type « `Planner` » peut requérir des informations contextuelles particulières au sujet de l'application ou l'environnement. Pour cela, il a une interface client de type « `MonitorItf` » ayant une multiplicité `1..*` pour interagir avec un nombre arbitraire de gestionnaire de contexte. Séparer la prise de décision et la planification permet de distinguer l'objectif à atteindre (la stratégie) de la manière de l'atteindre (le plan). Ainsi, une même stratégie peut être adoptée par plusieurs composants, mais exécutée de manières différentes. Le planificateur envoie le plan d'adaptation à l'exécuteur en utilisant l'interface de type « `ExecuteItf` ».

Le composant de type « `Executor` » contrôle l'application du plan d'adaptation qu'il reçoit. Pour cela, il interagit avec les effecteurs associés aux composants via l'interface client de type « `ModifyItf` » qui a une multiplicité `1..*`. Il prend en compte l'exécution de l'application adaptable qui est en cours et cordonne l'application des actions lorsque plusieurs composants de l'application sont impliqués.

4.5.5 Spécialisation de la gestion d'adaptation

Nous adoptons une approche orientée politique décrite dans le paragraphe 2.4.1 où la logique d'adaptation est décrite sous forme de politiques externes, dissociées de l'implémentation des services. Cette approche présente de nombreux avantages. En premier lieu, elle facilite le développement du système d'adaptation. En exprimant l'adaptation dans un langage de haut niveau, le développeur n'est pas obligé de comprendre tous les détails de la mise en œuvre du service d'adaptation lors de la définition ou la lecture d'une politique donnée. En deuxième lieu, il est plus facile d'analyser les dépendances entre les différents aspects de l'adaptation et de détecter les conflits potentiels. En troisième lieu, cette approche ouvre la porte à la réutilisation des mêmes services d'adaptation dans des contextes différents en les spécialisant différemment selon les besoins d'adaptation de l'application. Enfin, avec le niveau de découplage obtenu, il est plus facile de permettre le changement dans la logique d'adaptation en cours d'exécution, sans qu'il soit nécessaire de recompiler et de redéployer le système d'adaptation.

Dans notre approche, une politique est un ensemble de règles sous la forme événement-condition-action (ECA). Chaque règle se compose d'un *« trigger »* qui se déclenche lorsqu'un événement est reçu, d'une condition qui doit être vérifiée, et d'une action à exécuter. En effet, l'une des exigences clés des mécanismes d'adaptation proposées est la facilité de leur spécialisation par un intervenant humain. Les règles ECA est un paradigme bien connu dans le domaine de la représentation des connaissances [BL04, Sow00] et elles offrent une bonne lisibilité de la représentation de celles-ci. De plus, l'utilisation de ces règles aborde l'un des objectifs prioritaires lors de la conception d'un système, à savoir la similitude à la pensée humaine et l'amélioration de la facilité d'utilisation [BKI06]. Deux types de composants du modèle d'un gestionnaire d'adaptation utilisent la notion de politique : le décideur et le planificateur. Dans les deux cas, la politique spécialise le comportement du composant. Ainsi, l'attribut *algoParameters* pour ces types inclut un seul paramètre correspondant à la politique à utiliser.

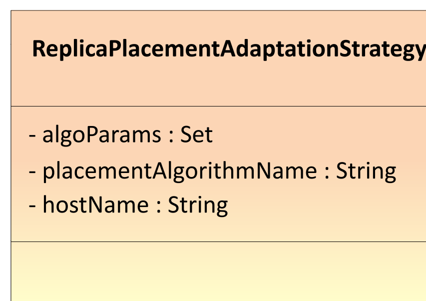

FIGURE 4.13 – Classe UML représentant un type de stratégies d'adaptation d'un composant de placement de répliques

Le décideur interprète une politique que nous appelons *politique d'adaptation*. Cette politique spécifie le raisonnement qui permet de décider quelle modification est exigée. Le décideur utilise la politique pour choisir un type de stratégie d'adaptation et le paramétrer. Ainsi, par exemple, la figure 4.13 montre le type disponible pour définir des stratégies d'adaptation pour un composant de gestion du placement de répliques. Dans ce type, le nom de l'algorithme de placement ainsi que ses paramètres de configuration et le nœud où doit être déployé le composant peuvent être définis. Une stratégie d'adaptation consistant à adopter un algorithme de placement aléatoire avec un degré de réplication égal à 10, consiste

donc à initialiser l'attribut *placementAlgorithmName* avec « *RandomPlacementStrategy* » et l'attribut *algoParams* à la valeur *10*. Une autre stratégie d'adaptation possible est de migrer le composant sur une autre machine. Dans ce cas, le nom du nouveau nœud qui va héberger le composant doit être spécifié dans la variable *hostName*. Le nombre d'attributs dans ce type de stratégie est suffisant pour appliquer deux types d'adaptation : la modification du comportement (algorithme et paramètres) et la modification de la distribution.

Un planificateur interprète un autre type de politique que nous appelons *guide* comme dans [BAP07]. Le guide permet au planificateur de choisir un type de plan d'adaptation approprié à une stratégie d'adaptation et de configurer ses variables en se basant sur les informations indiquées dans la stratégie. Par exemple, il peut extraire, à partir de la stratégie d'adaptation, le nom d'algorithme à utiliser et les valeurs de ses paramètres de configuration.

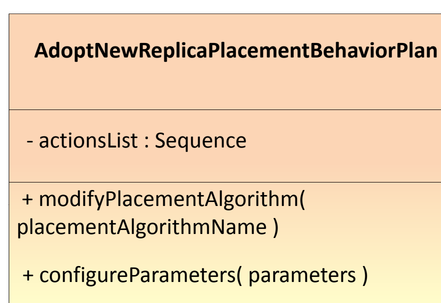

Figure 4.14 – Classe UML représentant un plan d'adaptation de stratégie de placement de répliques

La figure 4.14 présente un plan qui modifie l'algorithme de placement qu'implémente un composant d'un système de réplication. Il contient une première action d'adaptation *modifyPlacementAlgorithm* pour modifier l'algorithme qu'implémente le composant et une deuxième action *configureParameters* qui configure ses paramètres (par exemple, l'attribut qui précise le nombre de répliques souhaitées).

Le paramétrage des stratégies et des plans d'adaptation génériques permet de réduire l'effort pour définir les diverses stratégies et plans possibles pour une application particulière. Par exemple, ajouter un nouvel algorithme de placement des repliques ne nécessite pas forcément de créer un nouveau type de plan.

En ce qui concerne l'exécuteur, son comportement ne peut pas être spécialisé par une politique. En effet, celui-ci ne fait qu'exécuter les actions du plan qu'il reçoit.

4.5.6 Coordination des activités de gestionnaires d'adaptation

La distribution de la gestion d'adaptation se traduit par une autonomie de chaque gestionnaire d'adaptation. Ces activités consistent à prendre la décision d'adaptation, déterminer le plan d'adaptation et contrôler l'exécution du plan. Cependant, elles ne peuvent pas se réaliser de manière isolée dans tous les scénarios d'adaptation. Certains scénarios nécessitent la coordination des activités d'un groupe de gestionnaires d'adaptation lorsqu'il existe des dépendances entre leurs buts et entre les actions d'adaptation qu'ils appliquent.

En effet, chaque gestionnaire d'adaptation a une vue réduite de l'application et de son environnement d'exécution. Il manque de connaissances concernant d'autres composants, les ressources qu'ils requièrent, les préférences des utilisateurs et de leurs services. Par conséquent, les différents gestionnaires d'adaptation peuvent prendre des décisions conflic-

tuelles si chacun effectue ses prises de décision de façon individuelle. Par exemple, un gestionnaire d'adaptation ne doit pas appliquer des actions d'adaptation qui augmentent le taux d'utilisation de ressources partagées par les entités qu'il adapte en ignorant les besoins en ressources d'autres entités associées avec d'autres gestionnaires d'adaptation. De plus, un gestionnaire d'adaptation peut avoir besoin d'exploiter les possibilités d'adaptation d'autres composants sous le contrôle d'autres gestionnaires. Par exemple, le changement du comportement d'un composant peut améliorer considérablement la qualité de service d'une application si un autre composant change lui aussi de comportement.

Par ailleurs, des processus de contrôle d'exécution de plusieurs plans qui modifient les composants de l'application en parallèle peuvent laisser le système dans un état non cohérent. Ceci provient du manque du contrôle de l'état global de l'application pendant et après l'adaptation et aussi du manque d'informations sur l'avancement de l'exécution des différents plans. Ainsi, il est indispensable de coordonner l'exécution de plusieurs plans lorsqu'il existe des dépendances entre les composants qu'ils modifient.

Pour cela, notre approche définit des mécanismes pour :

- assurer la validité et l'efficacité des stratégies d'adaptation choisies en parallèle,
- fournir des plans qui sont applicables parallèlement et qui incluent les traitements nécessaires pour coordonner leurs exécutions,
- coordonner les exécutions simultanées de plusieurs plans d'adaptation.

Ces mécanismes sont décrits dans les deux paragraphes suivants.

Coordination des prises de décision d'adaptation

Principe. La coordination de prises de décision vise à emmener un groupe de gestionnaires d'adaptation dans une prise de décision commune pour assurer des décisions non-conflictuelles et complémentaires. La prise de décision de manière collective résulte de la détermination d'une stratégie globale. Nous définissons une *stratégie globale* comme l'ensemble des stratégies que les gestionnaires d'adaptation choisissent pendant un processus de prise de décision. Par exemple, une stratégie globale peut se composer de deux stratégies complémentaires choisies par un gestionnaire d'adaptation associé au service de placement et un autre associé au service de la gestion de la cohérence des répliques. Le premier choisit de passer à un nouvel algorithme qui permet la création de répliques de certaines données sur les terminaux mobiles. Le deuxième décide de passer à un protocole de cohérence optimiste pour ces données.

Les prises de décision doivent être pertinentes par rapport à la situation globale courante et doivent prévoir l'effet des stratégies d'adaptation pour éviter des situations incohérentes. Un gestionnaire d'adaptation peut ne pas posséder suffisamment d'informations locales lui permettant de choisir une stratégie d'adaptation adéquate. Par contre, il doit choisir les stratégies qui permettent de satisfaire un maximum de ses objectifs en respectant des contraintes envers les composants contrôlés par d'autres gestionnaires. La principale préoccupation de la coordination de prises de décision doit donc être d'assurer l'adéquation d'une stratégie d'adaptation avec les contraintes suivantes : (i) maintenir une composition cohérente des services de l'application, (ii) garantir les intérêts des autres entités applicatives en terme de partage de ressources et de services et (iii) assurer de bonnes performances du système global. Pour cela, notre approche se base sur le principe de faire participer plusieurs gestionnaires dans un même processus de prise de décision et de prendre en compte les effets non locaux de stratégies d'adaptation résultant de décisions locales. Ainsi, la construction d'une stratégie globale permet de choisir les stratégies d'adaptation locales de telle sorte que de mauvais choix soient évités.

La prise de décisions coordonnées est réalisée en trois étapes. Les deux premières sont assurées au niveau d'un décideur et consistent à (1) récupérer les informations de contexte nécessaires et (2) choisir une stratégie d'adaptation. À la fin de l'étape 2, le processus de coordination (étape 3) est initié. Nous supposons ici qu'un seul gestionnaire d'adaptation déclenche ce processus. Il est appelé l'*initiateur de coordination*. L'étape 3 implique les décideurs des autres gestionnaires appelés eux les *participants*. Au cours de cette étape, les décideurs peuvent avoir besoin de récupérer des informations de contexte et de faire un raisonnement pour déterminer la stratégie globale définitive. Le cas où plusieurs décideurs initieraient simultanément une coordination pourrait être traité par un abandon du rôle d'intiateur sauf pour l'un d'entre eux. Les paragraphes qui suivent détaillent le déroulement des trois étapes de prise de décision.

Types de composants pour la coordination de prises de décisions. Notre modèle architectural définit des types d'interfaces additionnels au composant de type « `DecisionMaker` » et un nouveau type de composant « `Negotiator` » (négociateur) de sorte à permettre à un décideur de prendre part à un processus de coordination. La figure 4.15 présente le modèle spécifiant les deux types de composants « `DecisionMaker` » et « `Negotiator` » et les liaisons possibles entre eux pour rendre un décideur coopératif.

L'interface optionnelle de type « `CoordinateItf` » avec une multiplicité `0..*` permet à un décideur de communiquer des messages pour un nombre arbitraire de décideurs. La politique d'adaptation doit définir le moment où l'interaction se produit, les décideurs impliqués, et le contenu des messages. Un composant optionnel de type « `Negotiator` » peut être instancié et interconnecté avec le décideur afin de permettre la négociation d'une stratégie globale. Dans ce cas, un décideur peut entamer des négociations, en utilisant l'interface optionnelle de type « `NegotiateItf` » et un décideur d'un gestionnaire participant peut être notifié sur le résultat de négociation réussie, grâce à l'interface de type « `NotifyItf` ».

Par ailleurs, les composants de type « `Negotiator` » inclus dans différents gestionnaires d'adaptation sont reliés entre eux via des interfaces de type « `ProposeItf` ». Chaque négociateur expose une interface serveur de ce type et utilise une interface client du même type et de multiplicité `0..*`. Il a une interface client de type « `MonitorItf` » avec une multiplicité `1..*` pour récupérer des informations contextuelles. Ainsi, chaque composant de type « `Negotiator` » a la possibilité de communiquer avec un nombre arbitraire de négociateurs et de gestionnaires de contexte.

La politique d'adaptation précise les situations nécessitant le déclenchement de la négociation par un composant de type « `DecisionMaker` ». Une politique de négociation spécialise le comportement du composant de type « `Negotiator` » en spécifiant les participants et les règles de contrôle de l'état d'avancement de chaque processus de négociation. Ces politiques sont spécifiées dans l'attribut *algoParameters* des composants « `DecisionMaker` » et « `Negotiator` » respectivement. Un autre attribut des mêmes composants, l'attribut *node* permet de choisir leur placement.

Modèles de coordination de prises de décisions. Nous avons défini trois modèles de coordination de prises de décision entre services de décision : le choix de stratégie par un maître, la publication de stratégie et la négociation de stratégie.

1. *Choix de stratégie par un maître* :
   Ce modèle permet à l'initiateur de la coordination de se comporter comme un maître et considère les participants comme des esclaves qui suivent ses ordres. Le maître

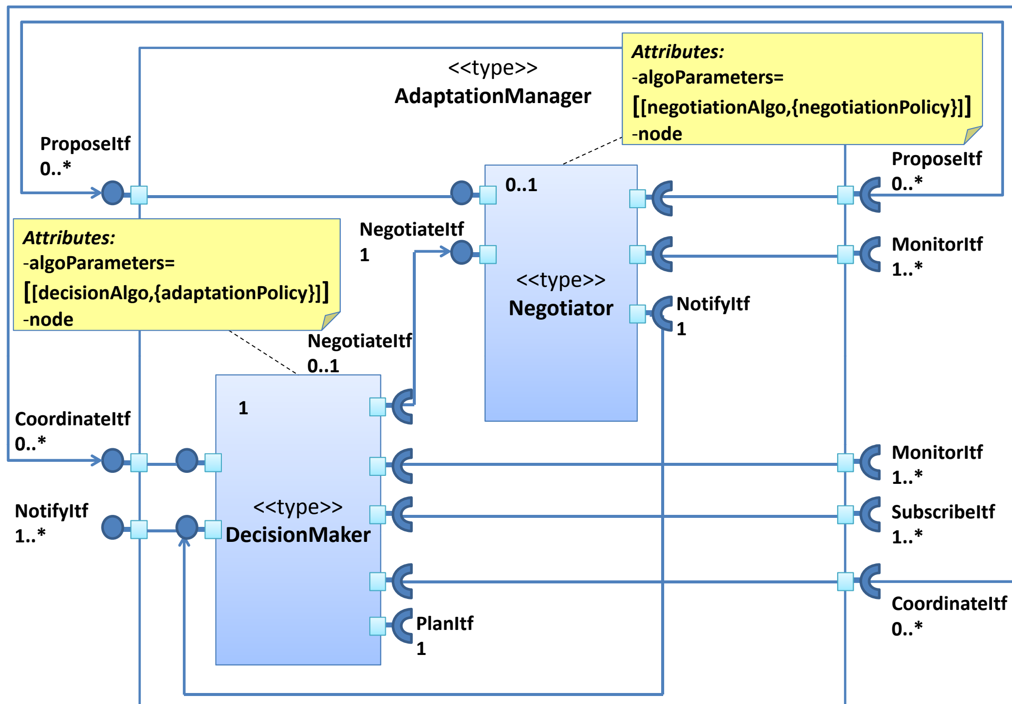

Figure 4.15 – Modèle architectural pour des décideurs coopératifs

choisit de façon individuelle une stratégie globale. Ensuite, il assigne la tâche d'appliquer une stratégie spécifique à chaque participant, en utilisant les interfaces de type « `CoordinateItf` ». Le message envoyé à chaque participant indique la stratégie et qu'il doit appliquer. Chaque participant démarre un processus d'adaptation pour appliquer la stratégie assignée en appelant son planificateur.

Ce modèle est simple à spécifier et limite les coûts de communication entre les décideurs.

2. *Publication de stratégie* :

   Ce deuxième modèle permet à l'initiateur de coordination de notifier les participants de la stratégie qu'il a choisie pour lui-même. Une fois un participant notifié, celui-ci analyse son contexte et choisit une stratégie appropriée en tenant compte de la décision prise par l'initiateur. La stratégie choisie doit être cohérente et complémentaire à celle de l'initiateur. Comme dans le premier modèle, l'interaction entre l'initiateur et les participants via les interfaces de type « `CoordinateItf` » est unidirectionnelle et un message envoyé à chaque participant indique la stratégie adoptée par l'initiateur. Par exemple, un décideur associé à un gestionnaire de placement de répliques d'un groupe de données peut choisir une stratégie consistant à augmenter le nombre des répliques. Après, il annonce cette stratégie au décideur associé à un autre gestionnaire de placement de répliques d'un deuxième groupe de données. Ce deuxième décideur déclenche un processus d'adaptation en choisissant pour ses données une stratégie consistant à réduire le nombre de leurs répliques car il trouve que la capacité de stockage globale est faible.

   Ce modèle peut être utilisé lorsque chaque participant dispose de possibilités pour assurer une stratégie globale cohérente et lorsqu'il est facile de gérer les dépendances entre les services fournis par les composants impliqués.

3. *Négociation de stratégie* :

   Ce modèle permet l'établissement d'un accord sur la stratégie globale à appliquer par un processus de négociation entre un groupe de gestionnaires d'adaptation. Par exemple, la négociation peut avoir lieu entre les gestionnaires d'adaptation associés aux composants assurant le service de propagation de mises à jour s'exécutant dans des environnements de caractéristiques différentes. Ces gestionnaires négocient le protocole de cohérence des répliques à adopter pour améliorer la performance du système de réplication en tenant compte des caractéristiques des différents environnements comme la disponibilité des ressources.

   Dans ce modèle, les négociateurs échangent des propositions et des contre-propositions afin de parvenir à un accord. L'initiateur informe les autres de la stratégie qu'il veut adopter et/ou demande d'adopter une stratégie spécifique. L'interaction entre les négociateurs permet d'accepter ou de refuser les propositions et de modifier la stratégie globale progressivement jusqu'à trouver un accord ou considérer que la négociation a échoué. Dans le processus de négociation, les négociateurs peuvent avoir des intérêts différents envers chaque configuration de l'application et des intentions différentes. Le contexte local est utilisé pour faire une proposition ou une contre-proposition afin de déterminer la décision finale.

   L'apport majeur de ce modèle est de rendre possible la modification de la stratégie initiale choisie par l'initiateur et de construire la stratégie globale progressivement en prenant en compte les différents contextes locaux des composants impliqués. Ceci garantit une meilleure efficacité de la stratégie finale quand la négociation réussit. Cependant, celle-ci augmente la complexité du processus de décision en particulier lorsque le nombre de participants et d'alternatives d'adaptation sont importants. En effet, les gestionnaires risquent de tarder à converger vers une décision finale et consomment des ressources pour le calcul et l'échange des propositions et des contre-propositions. Nous reviendrons plus en détail sur ce modèle dans le paragraphe 4.5.7.

Chaque modèle de coordination de prises de décision est adéquat pour une famille de processus d'adaptation particulière. Le choix du modèle dépend essentiellement des dépendances entre les composants de l'application, du nombre de décideurs et de la complexité du processus de coordination. L'utilisation de politiques rend notre approche flexible. En effet chaque décideur peut supporter un ou plusieurs modèles. Par exemple, un décideur peut être un esclave dans un scénario d'adaptation et un initiateur de la négociation dans un autre scénario. De plus, chaque décideur peut être spécialisé de sorte qu'il choisit un modèle à utiliser dans un processus d'adaptation en fonction du contexte. Par exemple, il se comporte comme un maître lorsque les ressources réseau disponibles sont faibles et il joue le rôle d'un initiateur de négociation lorsque la connectivité est élevée. Les capacités de prise de décision de chaque gestionnaire d'adaptation sont spécifiées en incorporant ou non un négociateur et au travers de la spécification de sa politique d'adaptation.

Coordination des exécutions parallèles de plans d'adaptation

La coordination des exécutions de plans adresse le problème de l'application de plusieurs plans d'adaptation en parallèle par un groupe de gestionnaires d'adaptation lorsque les plans contiennent des actions d'adaptation de composants dépendants. Dans de tels cas, l'application de ces plans doit être coordonnée dans le sens où un enchaînement spécifique de l'exécution de leurs actions doit être réalisé afin d'aboutir à une configuration cohérente de l'application. Il peut s'agir par exemple de suivre un ordre particulier pour appliquer d'abord une action *a1* d'un plan *P1* puis une action *a2* d'un plan *P2*.

Dans notre approche, l'aspect coordination des exécutions de plans d'adaptation est considéré dès la phase de planification. Au cours de cette phase, on insère dans les plans d'adaptation des traitements nécessaires pour coordonner leur exécution lors de la phase d'exécution. Ces traitements sont spécifiés dans un plan sous la forme d'actions de coordination en plus des actions d'adaptation. Nous appelons *plan coordonné* un plan qui inclut de telles actions de coordination permettant de prendre en compte les dépendances envers d'autres plans qui s'exécutent en parallèle. Ensuite, pendant la phase d'exécution, l'interaction de gestionnaires d'adaptation permet de coordonner les exécutions des plans d'adaptation.

La détermination d'un plan coordonné est assurée par le type de composant « `Planner` ». Pour exécuter des plans de façon coordonnée, le gestionnaire d'adaptation doit inclure un sous-composant additionnel de type « `Coordinator` » et le composant de type « `Executor` » est étendu par de nouveaux types d'interfaces pour communiquer avec son coordinateur.

Dans la suite, nous décrivons comment se fait la planification de plans d'adaptation coordonnés. Ensuite, nous précisons comment un groupe de plans coordonnés sont appliqués par des exécuteurs coopérants. Un exemple d'adaptation de protocole de cohérence des répliques illustrera notre proposition.

Planification de plans coordonnés. Une planification de plans coordonnés consiste à déterminer des plans d'adaptation à exécuter simultanément par différents exécuteurs. Pour cela, le planificateur associé à chacun de ces exécuteurs détermine un plan en considérant les dépendances entre des actions d'adaptation spécifiées dans ces plans. Chaque planificateur interprète son propre guide d'adaptation qui lui permet de choisir et paramétrer le plan adéquat incluant les actions de coordination nécessaires.

Les actions de coordination incluent deux types de commandes. Le premier type permet à un exécuteur de diffuser des informations locales nécessaires pour le contrôle d'exécution d'autres plans. Ces informations concernent l'avancement de l'exécution de son plan ou les états des composants applicatifs qu'il contrôle. Le deuxième type de commandes permet à l'exécuteur de déléguer au coordinateur la prise de décision concernant la suite du déroulement de l'exécution du plan lorsqu'elle dépend d'autres plans. Le coordinateur analyse dans ce cas des informations diffusées par d'autres exécuteurs et indique à cet exécuteur comment poursuivre l'exécution du plan. Par exemple, il l'ordonne de démarrer l'action d'adaptation suivante ou d'ignorer un ensemble d'actions.

Les informations liées à l'avancement de l'exécution sont utiles pour assurer un ordre d'exécution spécifique de certaines actions d'adaptation incluses dans des plans distincts. En effet, les actions d'adaptation ne sont pas identiques dans tous les plans d'adaptation et les planificateurs ne disposent pas des informations sur la durée de l'exécution de chaque action. Par conséquent, la vitesse de l'exécution de chaque plan ne peut pas être prévue d'avance et elle varie d'un plan à un autre. Cependant, certaines actions doivent être appliquées avant d'autres. Alors, le contrôle de l'exécution doit faire que cet ordre soit respecté en se basant sur les informations partagées.

Par ailleurs, les actions d'adaptation ne sont pas définies de manière définitive au cours de la planification. Pendant l'exécution des plans, le service de coordination prend des décisions concernant l'exécution de certaines actions d'adaptation. Par exemple, certaines actions peuvent se trouver inutiles ou un choix entre des actions alternatives doit se faire. En effet, un planificateur ne dispose pas toujours des informations sur les états des entités qui sont contrôlées par les autres gestionnaires d'adaptation. De plus, les actions d'adaptation n'exigent pas toutes le blocage de certains services de l'application. Ainsi, le blocage peut avoir lieu pendant des périodes spécifiques de l'application d'un plan et pas forcément

dès le début de son exécution. C'est au cours de l'exécution des plans que le service de coordination disposera des informations nécessaires sur l'avancement de cette exécution et sur l'état courant de l'application pour pouvoir décider la suite du déroulement de l'exécution des actions d'adaptation.

Nous considérons l'exemple de 4 gestionnaires d'adaptation (*am1..am4*) qui contrôlent ensemble l'adaptation du protocole de cohérence. Comme nous le verrons dans le chapitre suivant, ce protocole est géré par deux types de composants, les gestionnaires de propagation de mises à jour et les gestionnaires d'accès, un gestionnaire de propagation de mises à jour étant connecté à un ensemble de gestionnaires d'accès. Nous allons décrire un scénario d'adaptation pour passer d'un protocole optimiste à un protocole pessimiste à copie primaire [Rat98]. La stratégie reçue par chaque planificateur des 4 gestionnaires spécifie le nom du protocole (*primary copy protocol*), le groupe de données concerné (*group1*) et la liste des gestionnaires d'adaptation qui participent à l'adaptation (*am1..am4*).

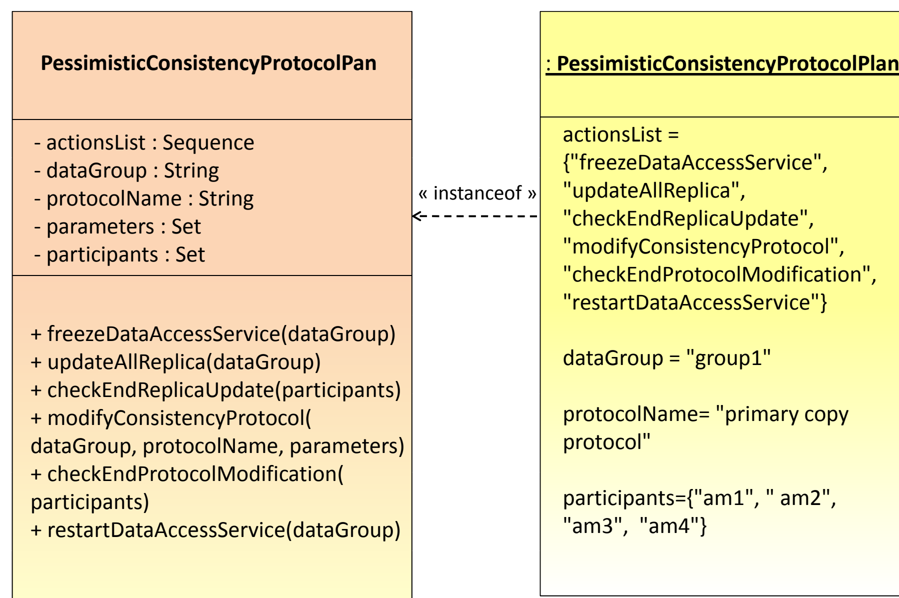

FIGURE 4.16 – Classe UML et l'objet associé définissant une collection d'actions ordonnées d'un plan d'adaptation coordonné

Chaque planificateur choisit et paramètre le plan approprié qui inclut des actions de coordination. La figure 4.16 montre le plan d'adaptation fourni par chaque planificateur pour notre scénario. Nous représentons le plan sous forme d'un objet associé à la classe UML dont il est instance. Le premier attribut *actionsList* est une collection ordonnée qui précise les noms des actions à appliquer et leur ordre. Le deuxième attribut *dataGroup* est une chaîne de caractères qui représente le nom du groupe de données concerné par l'adaptation. L'attribut *protocolName* fixe le nom du protocole à appliquer. Le dernier attribut *participants* est une collection qui précise les noms des gestionnaires d'adaptation impliqués. Il s'agit ici d'un même plan à appliquer par les 4 exécuteurs. Chacun d'eux est responsable de l'adaptation d'un gestionnaire de propagation de mises à jour et des gestionnaires d'accès qui y sont connectés. Ce plan contient quatre actions d'adaptation et sera appliqué simultanément par les exécuteurs impliqués. Tout d'abord, l'action *freezeDataAccessService* active l'interception des requêtes d'accès au groupe de données *group1* et bloque leur traitement. Puis, l'action *updateAllReplica* force la mise à jour de toutes les

répliques de ces données puisque les mises à jour manquantes et les conflits potentiels entre elles ne peuvent pas être gérés par le nouveau protocole après l'adaptation. L'action *modifyConsistencyProtocol* change le comportement du gestionnaire de propagation de mises à jour et des gestionnaires d'accès correspondants afin de changer de protocole adopté pour le groupe de données *group1*. Enfin, *restartDataAccessService* réactive le service d'accès au groupe de données et lance le traitement des requêtes d'accès aux données en attente.

Dans cet exemple, deux actions de coordination sont nécessaires. L'action *checkEndReplicaUpdate* contrôle la fin de la mise à jour de toutes les répliques, car ceci nécessite la collaboration des gestionnaires de propagation de mise à jour avant de changer leur comportement. L'action *checkEndProtocolModification* permet à chaque exécuteur de vérifier que le comportement de tous les gestionnaires de mise à jour et des gestionnaires d'accès ont été adaptés.

Exécution de plans coordonnés. La coordination des activités d'un groupe d'exécuteurs consiste à exercer un contrôle global de l'exécution des différents plans. Ce contrôle est assuré par un service de coordination en plus du contrôle local exercé par chaque exécuteur en appliquant les actions d'adaptation définies dans son propre plan.

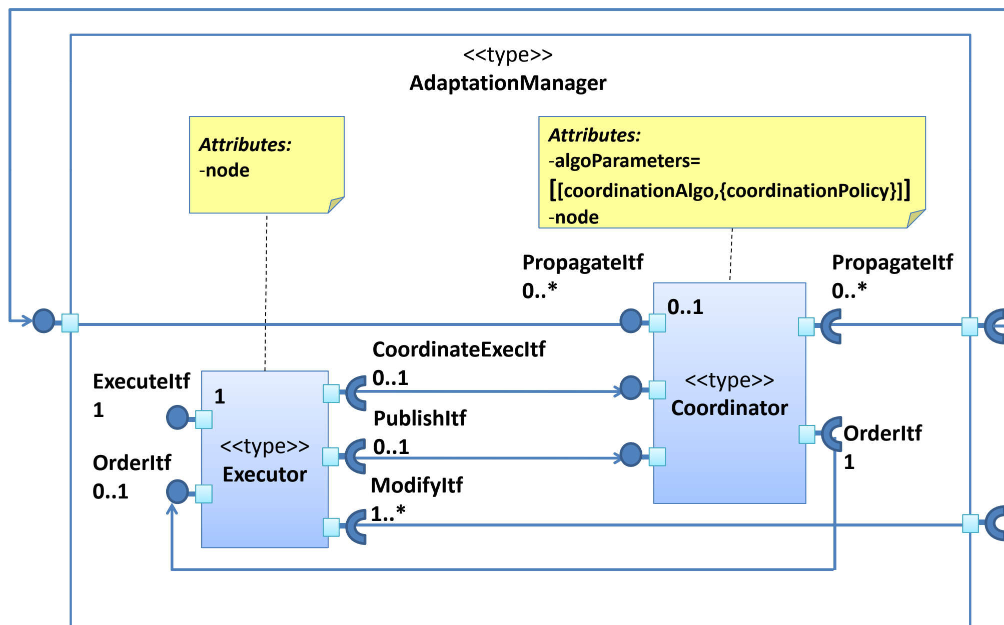

FIGURE 4.17 – Modèle architectural pour des exécuteurs coopératifs

Notre modèle architectural définit un composant optionnel de type « `Coordinator` » (coordinateur) qui est associé à un composant de type « `Executor` » pour lui donner la capacité de coordonner ses activités avec d'autres gestionnaires d'adaptation (voir figure 4.17). Un composant de type « `Coordinator` » est spécialisé par deux attributs : *algoParameters* et *node*. Le premier doit inclure un paramètre correspondant à une politique de coordination. Le deuxième permet de choisir le placement du composant. Il peut s'agir du même nœud qui héberge l'exécuteur ou d'un autre nœud.

En fonction des actions de coordination présentes dans un plan, un exécuteur communique avec le coordinateur. Celui-ci peut échanger des informations avec d'autres coordina-

teurs et produire des ordres vers l'exécuteur en interprétant une politique de coordination. Cette politique est un ensemble de règles sous la forme événement-condition-action. La séparation des préoccupations de la coordination des exécuteurs et du contrôle de l'exécution d'un plan ainsi que l'utilisation de politiques externes permet une mise en œuvre identique de l'exécuteur avec ou sans coordination et la réutilisation des mécanismes de coordination du contrôle d'exécution. Le contrôle global est quant à lui, réalisé de façon distribuée et coordonnée puisqu'il est réparti sur les différents gestionnaires d'adaptation incluant chacun un coordinateur.

Pendant la phase d'exécution, les exécuteurs échangent des informations sur l'avancement de l'exécution des plans d'adaptation en utilisant les coordinateurs. Par exemple, un exécuteur associé à un gestionnaire d'accès doit attendre, avant d'autoriser le traitement des requêtes d'accès aux données, que les autres exécuteurs associés aux gestionnaires de propagation de mises à jour terminent la modification de leur comportement. En outre, un exécuteur peut partager avec d'autres des informations sur l'état des composants qu'il gére. Ce type d'information peut être essentiel pour faire progresser l'application des actions d'adaptation d'une manière cohérente. Par exemple, un exécuteur a besoin de savoir si les autres gestionnaires de propagation de mises à jour qui ne sont pas sur son contrôle stockent certaines mises à jour des répliques à propager. Cette connaissance est utile pour décider de changer le comportement du gestionnaire de propagation de mises à jour qu'il adapte ou attendre jusqu'à ce que toutes les mises à jour soient propagées.

Dans notre modèle architectural, un composant de type « `Executor` » a une interface client de type « `PublishItf` » pour communiquer les informations qui sont utiles pour le progrès de l'exécution des autres plans. Le composant de type « `Coordinator` » détermine le groupe de coordinateurs concernés par l'information et utilise l'interface client de type « `PropagateItf` » pour communiquer avec ce groupe. Ce type d'interface a une multiplicité `0..*` afin de pouvoir interagir avec un nombre arbitraire de composants de type « `Coordinator` ».

Le composant de type « `Executor` » peut également utiliser une interface client de type « `CoordinateExecItf` » pour déléguer au coordinateur la prise de décision à propos de la suite du déroulement de l'exécution du plan. Le coordinateur prend la décision en se basant sur les informations reçues des autres coordinateurs. Il en informe l'exécuteur en envoyant un ordre via l'interface de type « `OrderItf` », ordre tel que *« ignorer l'action d'adaptation suivante »* ou *« appliquer l'action suivante »* ou encore *« appliquer un action X spécifiée dans le plan »*.

En reprenant l'exemple d'adaptation du protocole de cohérence cité dans le paragraphe précédent, lorsque un exécuteur atteint l'action de coordination *checkEndProtocolModification* (voir figure 4.16), il utilise l'interface de type « `PublishItf` » pour indiquer que la modification locale du protocole est terminée (méthode `publish(subject = "endProtocolModificationConfirmation", content = "true")`), puis l'interface de type « `CoordinateExecItf` » pour indiquer qu'il doit attendre que tous les exécuteurs aient fini leurs modifications locales pour continuer (méthode `command(subject = "endProtocolModificationCoordination", participants = ["am1","am2","am3","am4"])`). Cet appel est traité par le coordinateur comme un événement et interprète sa politique de coordination. Pour celui-ci, la politique de coordination précise que la condition à vérifier est que tous les autres participants aient publié un message confirmant la fin de la modification locale de protocole avant l'expiration d'un délai d'attente comme le montre la première règle de la figure 4.18.

Le coordinateur lance alors un processus qui attend la réception des publications attendues concernant ce sujet jusqu'à l'expiration d'un délai d'attente. Certaines publications

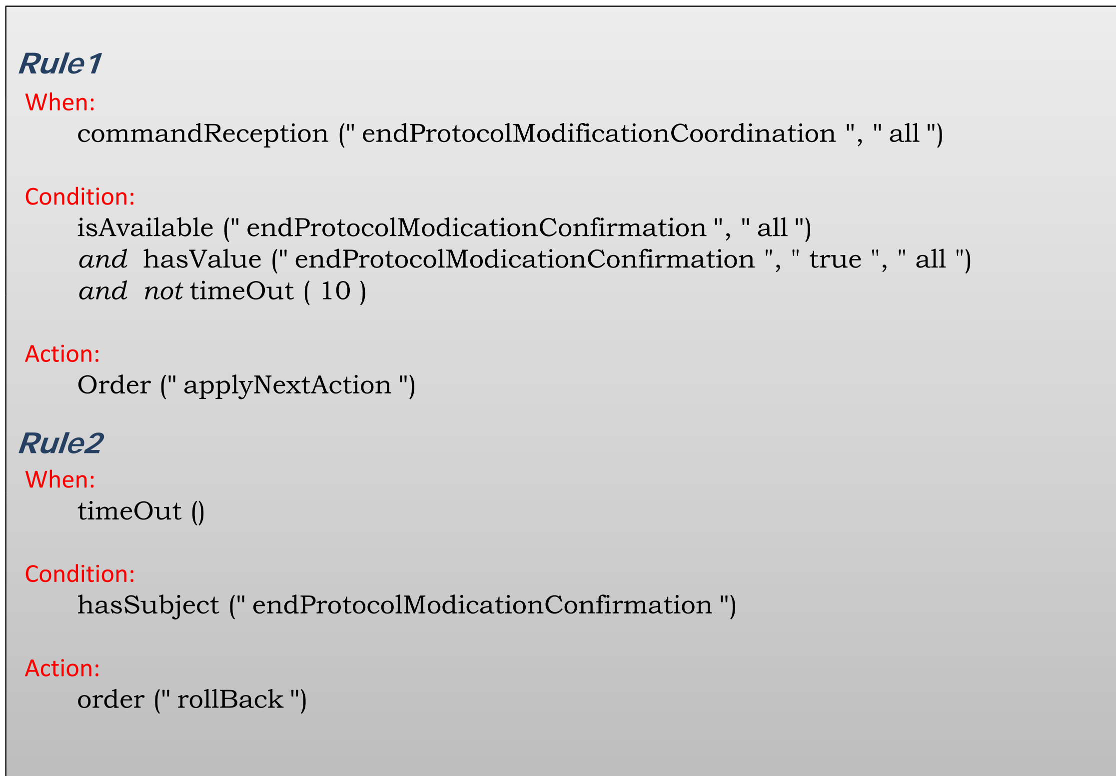

FIGURE 4.18 – Exemple de règles de coordination d'exécution de plans

peuvent être déjà reçues au moment du lacement du processus d'attente selon le coordinateur en question et la vitesse de l'exécution des plans. Supposons que tous les exécuteurs terminent leurs modifications locales et en fassent la publication. Chaque coordinateur obtient les connaissances nécessaires avant que le délai ne s'écoule et elles sont positives. Chacun envoie alors à l'exécuteur auquel il est associé l'ordre *applyNextAction* pour exécuter l'action suivante dans le plan d'adaptation. En cas d'expiration du délai d'attente avant de recevoir toutes les publications, la deuxième règle de la figure 4.18 est déclenchée et l'adaptation est annulée.

4.5.7 Protocole de négociation de stratégies d'adaptation

Principe

La négociation est un processus coopératif par lequel un groupe de gestionnaires d'adaptation parvient à un accord sur une stratégie d'adaptation globale. La négociation doit garantir l'indépendance dans la prise de décision de chacun des gestionnaires d'adaptation et assurer la validité globale d'une décision locale. Pour cela, les négociateurs impliqués évaluent la situation selon les informations dont chacun dispose pour trouver une solution acceptable à base de compromis.

Par exemple, un système de réplication de données médicales peut utiliser initialement un algorithme de placement de répliques basé sur la prédiction du placement des différents professionnels de santé et un protocole de cohérence optimiste. Suite à une situation d'urgence, une adaptation collective est nécessaire. Un premier gestionnaire d'adaptation peut souhaiter changer le comportement du système pour utiliser un algorithme de placement

épidémique. Un autre gestionnaire d'adaptation souhaite un changement du protocole de cohérence pour assurer une cohérence forte. Cependant le coût de propagation de mises à jour risque de devenir élevé dans ce cas. Un processus de négociation assure la coordination des prises de décisions. Le résultat en est par exemple de changer l'algorithme de placement et de garder le même protocole de cohérence optimiste.

Processus de négociation

Lorsqu'un décideur demande la négociation d'une stratégie, le négociateur crée un contrat d'adaptation et initie sa négociation avec un ensemble de participants. Le contrat est un objet qui spécifie l'initiateur, les participants et la stratégie globale à négocier. Celle-ci est spécifiée sous forme de paramètres où chaque paramètre précise une stratégie locale, le gestionnaire chargé de l'appliquer et les négociateurs impliqués dans sa négociation (voir figure 4.19).

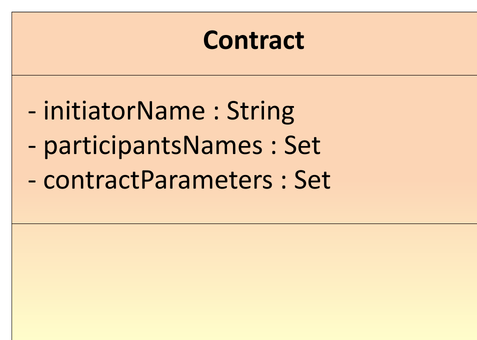

FIGURE 4.19 – Classe UML représentant les attributs d'un contrat négocié

Les participants à chaque processus de négociation peuvent être indiqués explicitement dans la stratégie d'adaptation. Une autre alternative est de faire que la négociation se déroule avec tous les négociateurs auxquels l'initiateur est connecté. Enfin, notre approche permet aussi de découvrir de manière automatique la liste des participants à la négociation. Pour cela, un modèle de dépendances entre les différents composants adaptables peut être fourni aux négociateurs. Le modèle spécifie pour chaque composant les types de ses relations avec les autres composants :

- « `uses` » et « `usedBy` » pour désigner que le composant utilise ou est utilisé par un autre composant,
- « `affects` » et « `affectedBy` » pour désigner que le comportement du composant influence la performance d'un autre composant ou sa performance est influencé par le comportement d'un autre composant.

Dans ce cas, la stratégie d'adaptation spécifie les types de relations à prendre en compte pour que le négociateur découvre automatiquement les participants.

Le processus de négociation est décomposé en trois phases :

1. *La phase de proposition* : Cette phase détermine le contrat à proposer par l'initiateur aux participants.
2. *La phase de conversation* : Phase durant laquelle des propositions ou des demandes de modifications sont échangées. L'initiateur collecte les réponses des participants. Chaque participant peut soit accepter, soit refuser la proposition. Il peut aussi demander ou proposer des modifications du contrat. Suite à une proposition ou une

demande de modification d'au moins un participant, on se retrouve dans une nouvelle phase de proposition.

3. *La phase de décision finale* : Cette phase de décision finale aboutit soit à la confirmation d'un contrat, soit à l'annulation du contrat. Cette décision est prise par l'initiateur selon les réponses des participants aux propositions qu'il leur a faites.

Le comportement de chaque négociateur est spécialisé par une politique de négociation externe, indiquant entre autre comment résoudre les conflits. La première possibilité pour résoudre un conflit consiste à choisir aléatoirement une stratégie parmi celles proposées par les différents participants. La deuxième possibilité se base sur l'association de priorités aux stratégies d'adaptation proposées par les participants.

Le diagramme présenté dans la figure 4.20 décrit la séquence des messages pour la négociation d'un contrat entre un initiateur et des participants. Pour des raisons de clarté, un seul participant y est représenté.

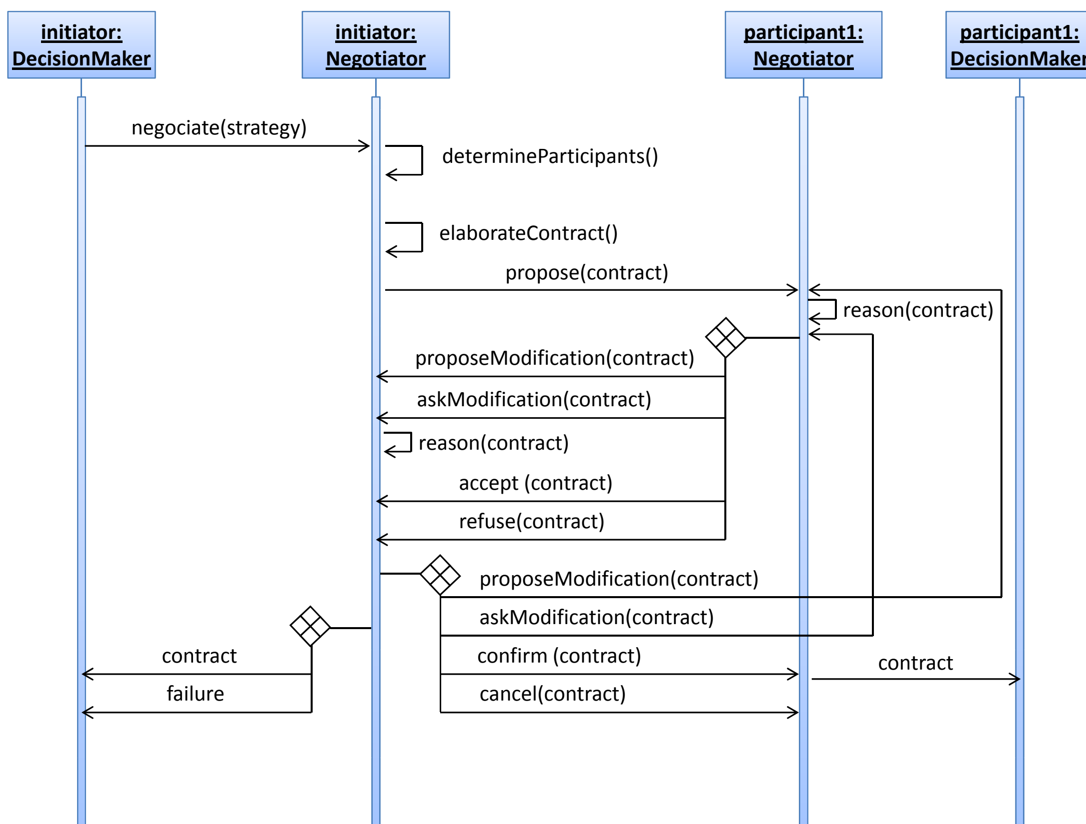

FIGURE 4.20 – Diagramme de séquence de négociation entre deux gestionnaires d'adaptation

Le décideur de l'initiateur choisit une stratégie d'adaptation. Ensuite, il utilise son interface de type « `NegotiateItf` » et demande au négociateur de négocier la stratégie qu'il a choisi. Ce négociateur construit le contrat. L'initiateur utilise les interfaces appropriées de type « `ProposeItf` » pour proposer en parallèle à chaque participant le contrat qui le concerne.

Le négociateur de chaque participant reçoit le contrat et interprète sa politique pour raisonner sur son applicabilité. Il peut alors accepter, refuser ou proposer/demander une

modification du contrat puis, il répond à l'initiateur. Lorsque l'initiateur reçoit toutes les réponses, il raisonne sur les acceptations et/ou l'applicabilité des modifications demandées ou proposées. Lorsque tous les participants acceptent le contrat, la négociation réussit. Dans le cas contraire, il détecte et résout les conflits et il peut alors à son tour proposer/demander une modification du contrat. Le processus de négociation est arrêté si un négociateur refuse un contrat ou si une condition d'arrêt est vérifiée. Cette condition peut être en relation avec le temps de négociation maximal autorisé ou avec le nombre maximal de cycles de négociations.

Si la négociation réussit, le négociateur de l'initiateur retourne au décideur de l'initiateur la stratégie résultante de la négociation et envoie au négociateur de chaque participant le contrat final. À la réception de ce contrat, le négociateur du participant utilise l'interface de type « `NotifyItf` » pour demander au décideur d'adopter la stratégie résultant de la négociation. Dans le cas contraire, le décideur de l'initiateur et les participants sont informés de l'échec de la négociation et l'adaptation est annulée.

4.6 Conclusion

Dans ce chapitre, nous avons présenté les concepts fondamentaux liés à notre approche pour l'adaptation d'applications distribuées à base de composants. Notre contribution consiste à définir une approche pour construire des systèmes d'adaptation distribués. Nous nous sommes intéressés particulièrement à la définition d'un modèle architectural flexible et de mécanismes génériques pour assurer la coordination des activités de plusieurs gestionnaires d'adaptation.

La flexibilité de notre modèle permet de spécialiser le comportement, la structure et la distribution d'un système d'adaptation. Notre approche offre plusieurs manières de coordination des décisions des gestionnaires d'adaptation. En particulier, la négociation de stratégies d'adaptation permet des prises de décision distribuées et parallèles, la résolution automatique des conflits entre eux et la garantie de leur autonomie. Concernant l'exécution de l'adaptation, notre approche permet de paramétrer et d'exécuter divers plans pour différents types de modifications et d'inclure une variété d'actions selon la stratégie à adopter. De plus, elle permet de coordonner l'exécution parallèle et distribuée de plusieurs plans d'adaptation.

La définition d'une fabrique et l'approche orientée politique facilitent la construction de systèmes d'adaptation. Néanmoins, le rôle de l'expert en adaptation est fondamental surtout pour spécialiser le comportement des gestionnaires d'adaptation par des politiques assurant un fonctionnement harmonieux du système global.

La généricité de notre approche la rend applicable à plusieurs familles de logiciels distribués. Le chapitre suivant détaillera comment on peut utiliser notre modèle architectural pour adapter des systèmes de réplication de données.

Chapitre 5

Modèle d'architecture pour des systèmes de réplication de données adaptables

5.1 Introduction

Le chapitre 3 a montré que de nombreuses techniques de réplication de données avaient été proposées pour différents domaines d'application. Cependant, très peu de travaux se sont intéressés aux services de gestion de données répliquées de façon globale et modulaire. De plus, parmi ces travaux, les approches existantes manquent de flexibilité et de méthodologie pour spécialiser un système de réplication en fonction de l'application qui l'utilise et de l'environnement d'exécution sous-jacent.

Par ailleurs, la grande variation du contexte d'exécution dans les environnements distribués n'a pas été suffisamment prise en compte dans les travaux sur la réplication de données. De plus, les systèmes actuels ne disposent pas de mécanismes pour proposer des types d'adaptation différents et pour gérer l'adaptation dynamique de façon distribuée et coordonnée.

Notre objectif est d'apporter une solution à ces problèmes en fournissant un modèle architectural pour la gestion de données répliquées et des outils et méthodes pour faciliter sa spécialisation en fonction des besoins. À l'image du modèle proposé pour l'adaptation dynamique, le modèle architectural pour la réplication de données spécifie le type des composants, leurs paramètres de configuration et leurs connexions possibles pour tout système de réplication. Il exprime ainsi les éléments communs à une famille de systèmes mais aussi les points de variation qui peuvent être utilisés pour modifier dynamiquement une spécialisation particulière du modèle. Cette spécialisation doit être réalisée par un expert qui décide, en fonction des besoins de l'application cible, des composants à instancier et de la valeur initiale des paramètres de configuration.

Ce chapitre présente d'abord notre cas d'étude qui porte sur la réplication de données pour des services de télémédecine. Cette application sera utilisée pour illustrer notre démarche. Ensuite, il présente les principes de développement d'un système de réplication suivant notre approche. Nous précisons dans la section 5.4 les fonctions d'un système de réplication que doit supporter notre modèle architectural. La section 5.5 précise notre modèle architectural pour la réplication de données. Enfin, nous présentons dans la section 5.6 un exemple de spécialisation du modèle pour notre cas d'étude.

5.2 Cas d'étude : gestion de données répliquées pour des services de télémédecine

Le maintien à domicile consiste à prendre en charge dans leur lieu de vie des personnes dépendantes, qu'elles soient âgées, en situation de handicap, malades chroniques ou ayant des difficultés temporaires. Ce maintien à domicile ne peut pas être uniquement constitué de services à destination de la personne dépendante mais aussi des services permettant la collaboration au sein du réseau médical. Un réseau de soins est la collaboration de professionnels de santé de disciplines différentes comme des médecins généralistes, des spécialistes, des pharmaciens, des infirmiers, et des kinésithérapeutes. Dans la suite, nous nous intéressons à des services de télémédecine qui permettent d'effectuer des actes médicaux à distance de manière collaborative.

5.2.1 Scénario d'utilisation : services de télémédecine

Les services de télémédecine que nous envisageons mettent en rapport, à distance, un patient et un ou plusieurs professionnels de santé. Ils visent aussi à faciliter le travail en réseau des établissements de santé et à améliorer la prise en charge des urgences en assurant l'échange de données médicales entre les différents participants impliqués dans les soins d'une personne.

Par ailleurs, des mesures sur l'état de santé du patient dans les conditions de vie de tous les jours peuvent être prises et accédées, à distance, par des professionnels de santé qui peuvent ainsi suivre l'évolution de l'état du patient.

D'une manière générale, ces services proposent une approche globale pour :

- suivre les patients grâce à des dispositifs et des applications installés à domicile et connectés aux établissements de soins,
- récolter et stocker les données médicales pertinentes du patient. Ces données peuvent être du texte comme les comptes rendus du médecin ou la délivrance de médicaments mais aussi des images (radiographies par exemple) ou même des vidéos (comportement du patient pour analyse),
- partager et échanger ces données médicales entre plusieurs professionnels de santé du même ou de différents centres médicaux.

Pour établir notre scénario d'utilisation, nous considérons que le dossier médical d'un patient est stocké par des hébergeurs de données de santé : chaque professionnel ou établissement de santé peut faire appel à un hébergeur différent. Ensuite, chaque professionnel de santé peut avoir une « vue » du contenu de ce dossier qui soit adaptée à ses préoccupations.

Le patient à domicile porte des capteurs dits « physiologiques » qui permettent un suivi en temps réel de ses paramètres biologiques (tension, fréquence cardiaque, rythme respiratoire...). D'autres capteurs peuvent aussi être utilisés pour surveiller l'activité courante du patient (par exemple, des caméras réparties dans chaque pièce) ou détecter les chutes (par exemple, un capteur cinétique). Certaines données mesurées sont stockées, soit sous forme brute, soit après synthèse, sur un serveur dans le domicile de la personne. La présence de ces informations au sein du domicile du patient permet à un soignant mobile, par exemple, d'y avoir un accès rapide en cas de déplacement chez la personne. De plus, les données acquises sont automatiquement transférées par le service de surveillance du patient sur le serveur de(s) l'établissement(s) de santé concerné(s) par les mesures lorsque cette action est pertinente. Puis, elles sont transférées vers l'hébergeur de données de santé correspondant.

5.2.2 Système de réplication de données auto-adaptable pour les services de télémédecine

Comme il a été mentionné dans le paragraphe précédent, une caractéristique importante des services de télémédecine est qu'ils font participer des professionnels de santé de plusieurs services et même de plusieurs établissements pour soigner un patient de manière collaborative (voir figure 5.1). Ce type d'application doit prévoir des fonctionnalités donnant aux différents acteurs un accès efficace aux données médicales qui les concernent et qui sont souvent partagées entre plusieurs utilisateurs. Pour atteindre cet objectif, une solution intéressante est d'utiliser un système de réplication des données médicales.

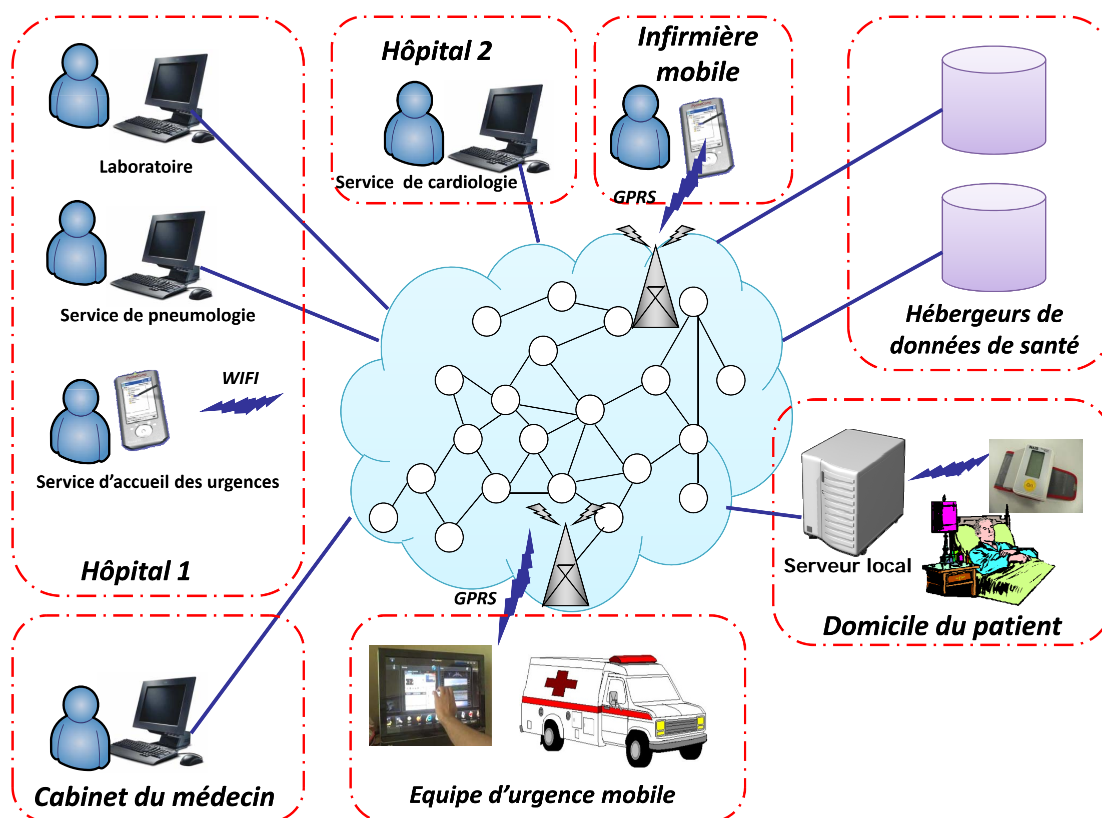

FIGURE 5.1 – Environnement d'exécution des services de télémédecine considérés

Le système de réplication de données doit tenir compte du contexte d'exécution et s'adapter en réponse à des changements. Ce contexte inclut notamment les caractéristiques des données, les capacités des terminaux mobiles, la localisation des acteurs, et leurs profils et préférences. Dans notre cas d'étude, le contexte d'exécution est dynamique : les utilisateurs peuvent se déplacer avec des terminaux mobiles (infirmières à domicile par exemple), l'importance des données accédées peut varier en fonction de l'état de santé de la personne, le nombre de sites disponibles pour héberger les répliques peut changer... Ainsi, une configuration choisie pour le système de réplication à un instant donné, peut ne pas être adéquate suite à un changement du contexte.

Par ailleurs, les exigences envers le système de réplication peuvent évoluer au cours du temps. Par exemple, en cas d'urgence, les contraintes imposées sur le temps d'accès aux données deviennent plus strictes. De plus, les contraintes sur la cohérence des répliques

peuvent changer. Par exemple, dans une situation normale de suivi du patient, l'accès à une réplique d'une donnée médicale divergente (par exemple, la dernière température prise) peut être toléré mais, une cohérence forte entre les répliques est nécessaire en situation d'urgence. Ainsi, nous visons à rendre les services de gestion de données répliquées auto-adaptables selon le contexte d'exécution et extensibles pour supporter des nouvelles stratégies de réplication.

5.3 Principes de développement de systèmes de réplication auto-adaptables

Pour faciliter la construction de systèmes de réplication à base de composants, nous définissons un modèle architectural de manière similaire à notre modèle pour l'adaptation dynamique. L'utilisation de notre modèle de réplication permet de définir une architecture logicielle d'un système de réplication. Le modèle exprime la variabilité entre plusieurs systèmes de réplication en définissant un ensemble d'éléments communs entre eux et aussi des caractéristiques qui les différencient.

La fabrique est capable de créer un système concret en utilisant le modèle de réplication et la description de l'architecture logicielle fournie par l'expert en réplication. À l'image du système d'adaptation, cette description définit les composants à instancier, les connexions entre eux et les valeurs des attributs nécessaires pour configurer leurs comportements et leurs placements (voir figure 5.2). La spécialisation du comportement d'un système de réplication se fait en fournissant les algorithmes à utiliser par les différents composants et leurs paramètres d'initialisation. La fabrique utilise le modèle architectural de réplication pour vérifier si la description fournie respecte l'ensemble des contraintes spécifiés par ce modèle.

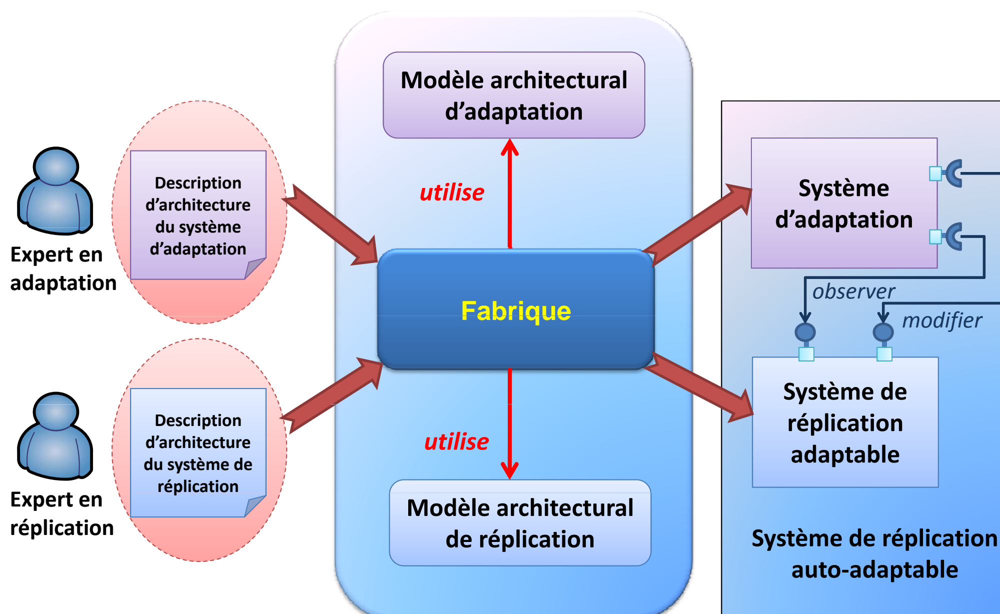

FIGURE 5.2 – Approche pour construire un système de réplication auto-adaptable

Par ailleurs, un système de réplication doit pouvoir être observé et modifié par un système d'adaptation construit à partir du modèle architectural présenté dans le chapitre précédent. Pour ceci, la description de l'architecture logicielle du système doit inclure les définitions d'interfaces nécessaires pour l'observation et/ou la modification des différentes composants.

5.4 Fonctions d'un système de réplication de données

Nous considérons un système de réplication comme un assemblage de composants logiciels qui implémentent l'ensemble des fonctions nécessaires pour gérer des données répliquées. Nous séparons les traitements de gestion de réplication des données répliquées elles-mêmes. Ainsi, un objet répliqué n'incorpore pas des traitements spécifiques pour cette gestion. Nous employons dans la suite le terme « application » pour désigner le logiciel qui utilise les services du système de réplication. Un ou plusieurs « utilisateurs » d'une application peuvent demander de répliquer ou lire/écrire des données répliquées. Un utilisateur peut être un module logiciel de l'application ou un utilisateur humain de celle-ci.

Après analyse d'un ensemble de travaux de recherche comme [Mar03, Dra03, TM05, FXL08], nous avons identifié des fonctions que nous considérons fondamentales dans un système de réplication. Dans la suite, nous décrivons ces fonctions. Elles seront présentes dans notre modèle architectural pour la réplication de données.

Nous considérons qu'un système de réplication possède 6 fonctions principales comme le montre la figure 5.3. Il s'agit d'une fonction de gestion du placement des répliques, une fonction de gestion d'interrogation des données, une fonction de manipulation des répliques, une fonction de contrôle d'accès aux répliques, une fonction de propagation des mises à jour entre les répliques et une fonction de contrôle de concurrence d'accès aux répliques. La figure précise aussi les dépendances entre ces fonctions et entre une application et un système de réplication dans l'objectif de maîtriser l'impact de l'adaptation statique ou dynamique d'une fonction sur les autres.

5.4.1 La gestion du placement des répliques

La fonction de gestion du placement des répliques répond aux demandes de création et de suppression de données provenant de l'application et décide le degré de réplication et les sites sur lesquels seront placées les répliques (le schéma de réplication présenté dans la section 3.2.1). Elle utilise la fonction de manipulation des répliques qui s'occupe de créer et de supprimer effectivement les répliques pour mettre en œuvre les décisions prises par la gestion du placement. La création et la suppression des répliques se font en réponse aux demandes provenant de l'application. De plus, elles peuvent se réaliser d'une manière automatique si la fonction dispose des capacités à déterminer le moment de réplication et la durée de vie des répliques ou à modifier dynamiquement le schéma de réplication.

5.4.2 L'interrogation des données

Cette fonction répond aux demandes d'accès en lecture/écriture aux données provenant de l'application. Elle choisit la réplique appropriée pour exécuter les requêtes d'accès et utilise la fonction de contrôle d'accès pour effectuer les opérations de lecture/écriture sur celle-ci. Une requête provenant de l'application peut être complexe lorsqu'elle contient plusieurs opérations et/ou elle porte sur plusieurs données répliquées. Dans ce cas, la fonction d'interrogation assure la répartition spatiale et temporelle de la requête.

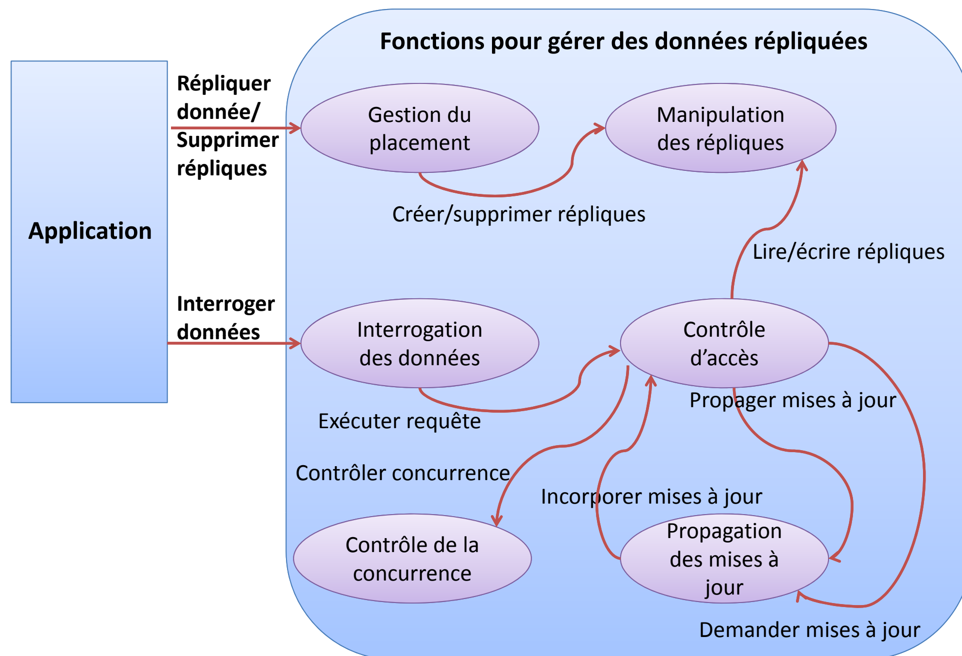

Figure 5.3 – Fonctions d'un système de réplication

5.4.3 Le contrôle d'accès aux répliques

Cette fonction contrôle les accès aux répliques sur les différents sites. L'accès à une réplique peut être une opération de lecture/écriture afin d'exécuter les requêtes provenant de l'application ou une mise à jour qui provient d'une autre réplique. Le contrôle consiste à s'assurer que la réplique à accéder soit disponible et à empêcher, quand c'est nécessaire, l'accès à d'autres répliques de la même donnée. Cette dernière opération requiert la participation de la fonction du contrôle de la concurrence. Le contrôle d'accès inclut aussi la notification des modifications des répliques à la fonction de propagation de mises à jour et, éventuellement, la demande des mises jour manquantes avant d'effectuer une opération de lecture ou d'écriture sur une réplique.

5.4.4 La manipulation des répliques

Cette fonction met en œuvre toutes les opérations de manipulation de répliques. Les demandes de création/suppression de répliques proviennent de la fonction de gestion du placement, les demandes de lecture/écriture de la fonction de contrôle d'accès.

5.4.5 La propagation des mises à jour

Cette fonction propage les mises à jour entre les répliques d'une donnée. Pour cela elle détermine les répliques concernées par une mise à jour et véhicule celle-ci vers les répliques identifiées. De plus, elle est responsable de la détection et résolution de confits lorsque ceci est nécessaire (voir section 3.4).

5.4.6 Le contrôle de la concurrence

Cette fonction contrôle la concurrence d'accès aux différentes répliques de chaque donnée. Elle gère l'ensemble de contraintes que les répliques doivent respecter sur l'ordre d'application des opérations. En effet, il peut être nécessaire de bloquer l'accès jusqu'à ce que l'état d'une réplique reflète un ensemble suffisant de mises à jour ou jusqu'à ce que les mises à jour soient propagées sur un ensemble ou sur toutes les répliques (voir section 3.4).

5.5 Modèle architectural d'un système de réplication

Comme pour l'adaptation dynamique, nous définissons un modèle architectural d'un système de réplication. Il est représenté dans la figure 5.4. Ce modèle définit les types de composants que peut contenir un système de réplication de données, leurs connexions, leurs paramètres de configuration et les interfaces permettant d'observer l'application et de modifier sa configuration. Il identifie alors les éléments communs et les points de variation possibles d'un système de réplication qui portent sur le comportement, la structure et la distribution des composants.

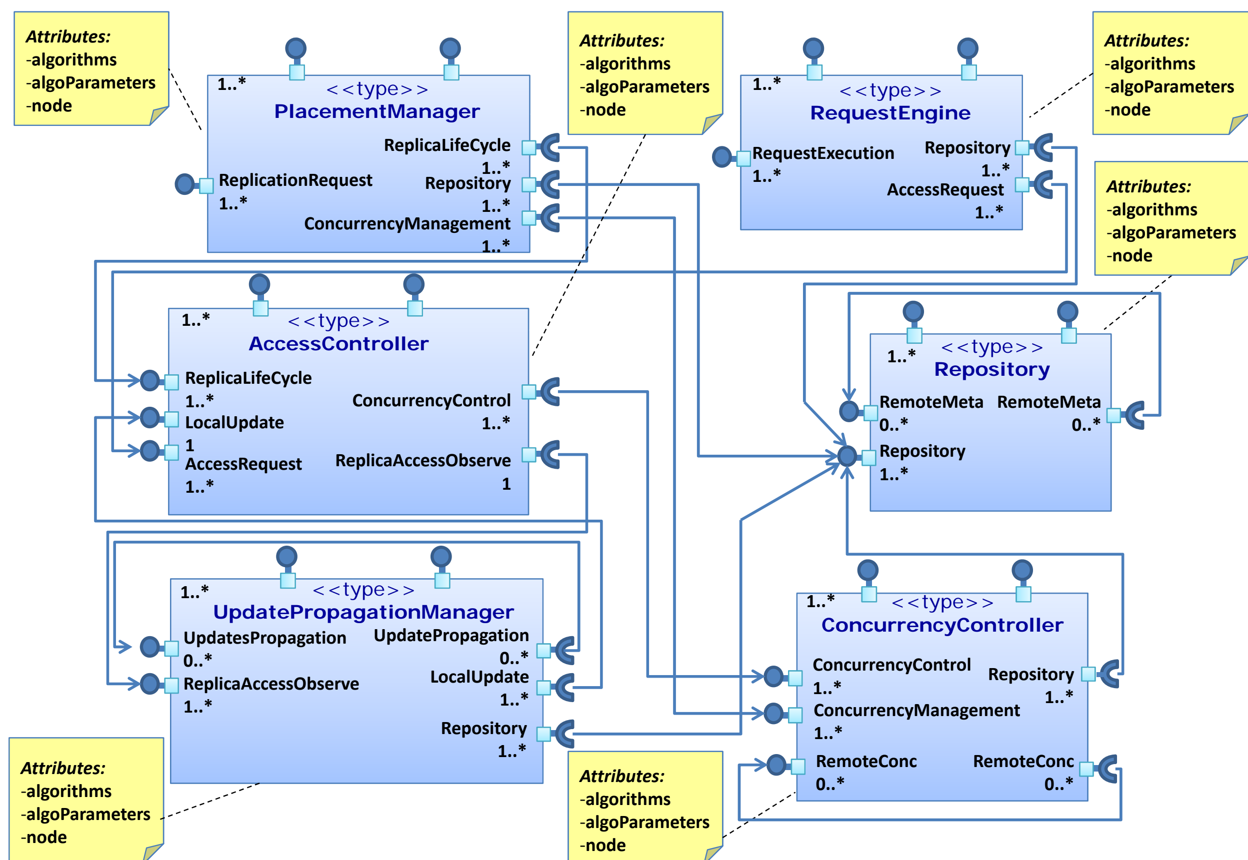

FIGURE 5.4 – Modèle architectural d'un système de réplication de données

5.5.1 Types de composants du modèle

Notre modèle architectural définit six types de composants pour construire un système de réplication (voir figure 5.4). La multiplicité de ces types est de `1..*` pour qu'un système

de réplication puisse inclure une ou plusieurs instances de chaque type. De plus, chaque type de composant fournit deux types d'interface de contrôle dédiées à l'adaptation : une interface d'observation de type « ObserveItf » et une interface de reconfiguration de type « ModifyItf ». Chacun de ces types a une multiplicité 1..*.

Annuaire (« Repository »)

Ce type de composant gère les informations sur les répliques des données (des métadonnées) et les met à disposition d'autres composants d'un système de réplication. Ainsi, par exemple, un gestionnaire de placement demande de mémoriser la localisation des répliques de chaque donnée. Un gestionnaire de propagation de mises à jour consulte cette information pour transmettre les mises à jour vers les sites appropriés. Dans notre modèle, nous imposons que l'emplacement des répliques fasse partie de ces informations mais bien d'autres peuvent y être présentes en fonction de l'implantation concrète.

Il fournit une interface de type « Repository » pour stocker, consulter, modifier et supprimer les métadonnées. La multiplicité de ce type d'interface indique qu'au moins une interface client devra y être connectée, celle d'un composant de type « PlacementManager », « UpdatePropagationManager », « ConcurrencyManager » et/ou « RequestEngine » (voir figure 5.4).

Par ailleurs, la multiplicité du type « Repository » indique qu'une architecture concrète peut inclure une ou plusieurs instances de ce type. Lorsqu'une seule est présente, l'annuaire est implanté de manière centralisée. Lorsque différentes instances sont décrites, il est décentralisé et, dans ce cas, une interface de type « RemoteMeta » peut être fournie (multiplicité 0..*) et requise (multiplicité 0..*). Ce type d'interface sert aux échanges entre composants de ce type pour rendre les métadonnées accessibles de tout annuaire.

Contrôleur d'accès (« AccessController »)

Les composants de ce type implémentent la fonction de contrôle d'accès et de manipulation des répliques. Un composant de ce type reçoit alors des demandes de création/suppression de répliques (interface « ReplicaLifeCycle »), de lecture/écriture de répliques (interface « AccessRequest ») et d'application de mises à jour (interface « LocalUpdate ») d'autres composants du système.

Comme mentionné précédemment, le contrôle d'accès utilise la fonction de contrôle de la concurrence pour s'assurer que les opérations de lecture/écriture peuvent être effectuées. Dans notre modèle, cette interaction se fait par l'interface client de type « ConcurrencyControl » de multiplicité 1..* pour pouvoir utiliser les services d'un nombre arbitraire de composants de gestion de concurrence.

Enfin, l'interface de type « ReplicaAccessObserve » et de multiplicité 1 est utilisée pour informer des accès des répliques à un composant de type « UpdatePropagationManager ».

Gestionnaire de propagation de MAJ (« UpdatePropagationManager »)

Les composants de ce type assurent la fonction de propagation de mises à jour entre les répliques. Ils sont notifiés des tentatives d'accès aux données via l'interface serveur de type « ReplicaAccessObserve ». En effet, pour certains protocoles, cette notification peut déclencher la mise à jour de la réplique sélectionnée avant d'effectuer l'opération de lecture ou d'écriture de la réplique. Pour d'autres, seulement la notification de modification de la réplique est nécessaire afin de mettre à jour les autres répliques de la même donnée. Dans

les deux cas, le composant peut mémoriser les modifications et décide à quel moment les envoyer aux autres répliques de la donnée. Il s'occupe aussi, si nécessaire, de détecter et résoudre les conflits afin de faire converger les répliques vers le même état.

Par ailleurs, l'interface de type « `LocalUpdate` » (multiplicité `1..*`) est utilisée pour transmettre les mises à jour aux composants de type « `AccessController` » qui appliquent les opérations sur les répliques.

Enfin, les interfaces client et serveur de type « `UpdatePropagation` » (multiplicité `0..*`) sont utilisées pour communiquer les mises à jour à un nombre arbitraire de composants du même type.

Moteur des requêtes (« `RequestEngine` »)

Les composants de ce type implémentent la fonction d'interrogation de données. Pour cela, ce type fournit une interface « `RequestExecution` » pour recevoir les requêtes d'accès aux données de la part de l'application. Ensuite, il choisit les répliques à utiliser pour satisfaire ces requêtes. Une interface de type « `AccessRequest` » de multiplicité `1..*` est utilisée pour transmettre les opérations aux composants de type « `AccessController` » qui les réaliseront sur les répliques sélectionnées. Enfin, ce type de composants dispose d'une interface « `Repository` », de multiplicité `1..*`, pour interagir avec un nombre arbitraire de composants de type « `Repository` » afin de manipuler les informations utiles pour la sélection de répliques.

Gestionnaire de placement (« `PlacementManager` »)

Les composants de ce type implémentent les fonctions du placement présentées précédemment. Ainsi, une interface serveur de type « `ReplicationRequest` » doit être fournie pour recevoir des requêtes de création/suppression de données de la part de l'application. Pour chacune de ces requêtes, il détermine le schéma de réplication (nombre de répliques et sites sur lesquels les répliques seront/sont placées) et génère les demandes nécessaires pour la création/suppression des répliques concernées. Il utilise pour ceci une interface client de type « `ReplicaLifeCycle` » de multiplicité `1..*` afin d'interagir avec un nombre arbitraire de composants de type « `AccessController` ».

Par ailleurs, il requiert une interface client de type « `Repository` » de multiplicité `1..*` pour être connecté à un nombre arbitraire de composants de type « `Repository` » ce que lui permet de manipuler les métadonnées sur les répliques (notamment leur placement).

Enfin, ce type de composants inclut aussi une interface client de type « `Concurrency-Controller` » de multiplicité `1..*`. Elle est utilisée pour enregistrer une nouvelle donnée à un « `ConcurrencyController` » pour qu'il contrôle la concurrence d'accès à ses répliques.

Contrôleur de concurrence (« `ConcurrencyController` »)

Un composant de ce type applique la politique de gestion de concurrence d'accès aux répliques. Il implémente des mécanismes de verrouillage des répliques lorsque le protocole de cohérence est pessimiste. Il peut permettre certains accès concurrents sur plusieurs répliques lorsque le protocole est optimiste. Il offre une interface de type « `ConcurrencyControl` » pour être notifié des opérations d'accès et les autoriser ou non. La deuxième interface fournie est de type « `ConcurrencyManagement` » et permet de recevoir les informations nécessaires pour la gestion de la concurrence des répliques lorsqu'une donnée est répliquée. Par exemple, un composant de type « `PlacementManager` » peut fournir au contrôleur de concurrence le nom de l'annuaire qui gère la donnée qu'il a répliqué.

Par ailleurs, l'interface client « `Repository` » de multiplicité `1..*` permet l'interaction avec un nombre arbitraire de composants de type « `Repository` » et l'accès aux métadonnées, notamment la localisation des répliques.

Enfin, lorsque le contrôle de concurrence est décentralisé, les interfaces client et serveur de type « `RemoteConc` » (multiplicité `0..*`) servent aux échanges entre composants de ce type.

5.5.2 Points de variation du modèle architectural

Le modèle architectural d'un système de réplication inclut les éléments communs à tout système de réplication mais aussi des points de variation. Des valeurs sont données à ces points lors de la spécialisation du modèle par l'expert en réplication. Elles peuvent, de plus, être modifiées par le système d'adaptation en fonction des politiques définies par l'expert en adaptation. Ces points de variation concernent, dans notre modèle, le comportement des composants (les algorithmes que les composants implémentent et les valeurs de leurs paramètres de configuration), la structure du système de réplication (le nombre de composants de chaque type et les connexions entre eux) et la distribution des composants (les placements possibles des composants). Dans ce paragraphe, nous présentons comment ces points de variation sont décrits dans le modèle ainsi que les principes et contraintes qui leur sont propres.

Variabilité de comportement

Points de variation. Le choix du comportement des différents composants d'un système de réplication se fait pendant la spécialisation ou l'adaptation dynamique de celui-ci. Les points de variation qui permettent de fixer ce choix sont spécifiés dans le modèle architectural en annotant les types de composants par deux attributs (voir figure 5.4) : l'algorithme qu'un composant applique et ses paramètres de configuration.

Bien qu'un composant puisse appliquer par défaut un seul algorithme qui peut être configuré par un ou plusieurs paramètres, il peut être intéressant de lui permettre d'appliquer plusieurs algorithmes alternatifs. Ainsi, un composant de type « `PlacementManager` » peut être utilisé par plusieurs utilisateurs différents[1]. Chaque utilisateur peut avoir des exigences différentes selon les caractéristiques des données à répliquer. Il est donc intéressant de permettre aux composants de ce type de choisir l'algorithme à utiliser en fonction de la source des requêtes. La même démarche peut être menée pour les composants de type « `RequestEngine` », le choix de la réplique sélectionnée pouvant dépendre de l'utilisateur qui demande l'accès. Enfin, les composants de type « `AccessController` », « `UpdatePropagationManager` » et « `ConcurrencyController` » définissent, ensemble, le protocole de cohérence à utiliser pour les différentes données répliquées. Ce protocole pouvant être différent selon les caractéristiques de la donnée et les exigences de son utilisation, il est intéressant de permettre à ce type de composants de choisir l'algorithme en fonction de la donnée concernée.

Classification des données et des utilisateurs. Nous appelons *« classification »* la technique qui permet de répartir des données ou des utilisateurs en groupes afin d'en appliquer des algorithmes différents.

Comme mentionné précédemment, la classification des données peut guider le choix du

1. La définition du terme *utilisateur* a été donnée dans la section 5.4.

protocole de cohérence [2]. Chaque donnée peut appartenir à un groupe auquel un algorithme spécifique est appliqué pour gérer la cohérence de ses répliques. Par exemple, la figure 5.5 montre deux groupes de données et un algorithme pour chaque groupe. L'algorithme *AlgoCohérence1* est appliqué pour le groupe contenant les deux données *D1* et *D2*. Le deuxième algorithme *AlgoCohérence2* est appliqué pour le deuxième groupe contenant la donnée *D3*.

Par ailleurs, les utilisateurs peuvent être classés afin que les composants de type « `PlacementManager` » et « `RequestEngine` » puissent effectuer des choix d'algorithme. Bien évidemment, les critères de classification sont généralement différents pour les deux composants. Par exemple, la figure 5.5 montre 4 utilisateurs : *Utilisateur1* peut uniquement accéder à des répliques tandis que *Utilisateur2* peut aussi en demander la création. Enfin, *Utilisateur3* et *Utilisateur4* ne peuvent que demander la création de répliques. Concernant le moteur de requêtes, les deux utilisateurs appartiennent à deux groupes distincts. Le moteur de requêtes applique l'algorithme *AlgoSélection1* lorsqu'il répond aux requêtes d'accès de *Utilisateur1* et *AlgoSélection2* pour celles de *Utilisateur2*.

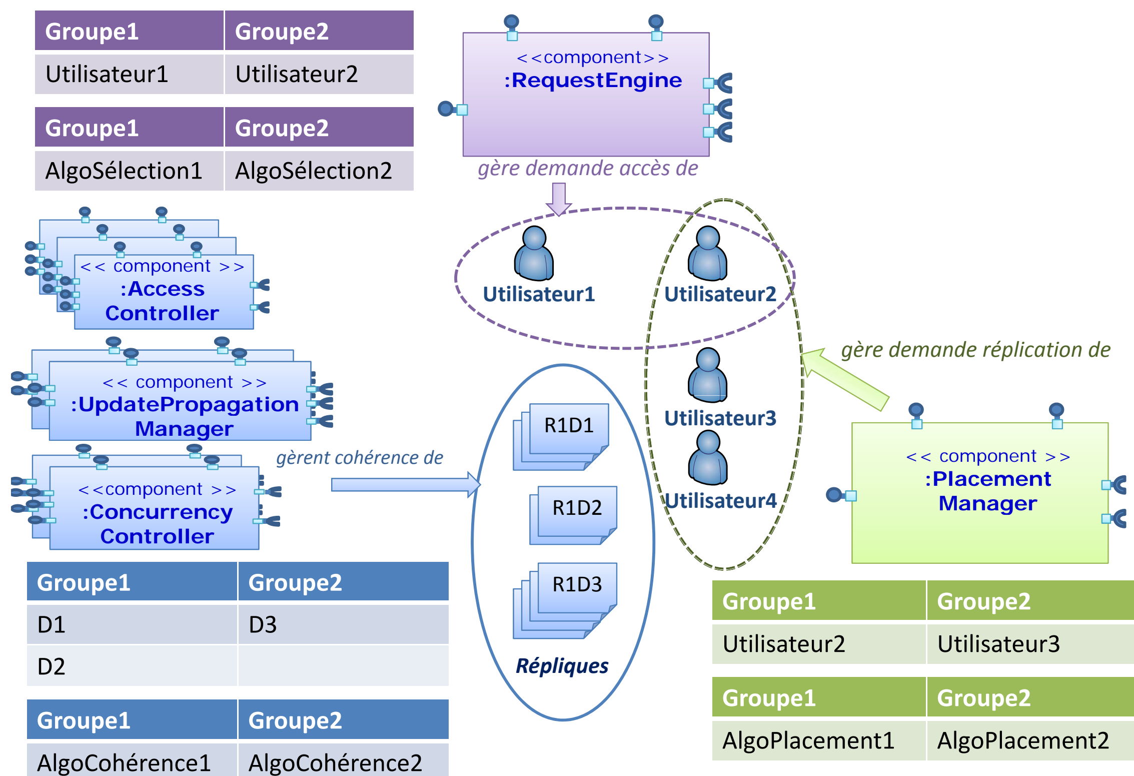

FIGURE 5.5 – Exemple de classification de données et utilisateurs

Par ailleurs, un gestionnaire de placement applique *AlgoPlacement1* quand *Utilisateur2* demande de répliquer une donnée et applique *AlgoPlacement2* pour l'*Utilisateur3*. Par exemple, *Utilisateur2* peut être un médecin mobile qui soumet des requêtes de réplication de données essentiellement pour réduire la latence de ses futurs accès à celles-ci. Par contre, *Utilisateur3* peut être un module du service de surveillance du patient qui demande de répliquer des mesures faites pour tous les utilisateurs potentiels.

La classification des données et utilisateurs est optionnelle. Un algorithme par défaut est

2. Ce choix correspond à celui de l'algorithme utilisé dans chacun des trois types de composant qui assurent le protocole.

défini pour chaque composant lors de la spécialisation du modèle architectural en donnant des valeurs adéquates pour les attributs *algorithms* et *algoParameters*. Lorsqu'un composant est appelé, il vérifie si une classification existe pour la donnée ou l'utilisateur concerné. Si elle n'existe pas, l'algorithme par défaut est utilisé. C'est le cas de l'*Utilisateur4* pour le placement dans la figure 5.5. Si elle existe, le composant choisira l'algorithme correspondant au groupe auquel appartient la donnée ou l'utilisateur selon le composant.

Par ailleurs, la mise en place de la classification des données et des utilisateurs tire partie du système d'adaptation. En effet, la définition des groupes est mise en place par des politiques d'adaptation exécutées par le système d'adaptation. Ainsi, par exemple, celui-ci peut décider de créer un nouvel groupe d'utilisateurs avec un algorithme particulier pour un composant de type « `RequestEngine` » lorsqu'un nouvel utilisateur se connecte. Le nouveau groupe contient initialement cet utilisateur et pourra en contenir d'autres en fonction de l'arrivée d'autres utilisateurs.

Dans le cadre de la classification, l'adaptation dynamique permet essentiellement de changer l'algorithme appliqué par défaut. Une fois des groupes sont définis, elle permet aussi de changer le groupe auquel une donnée ou un utilisateur appartient. Dans ce cas, il peut s'agir de la (ou le) déplacer dans un autre groupe existant ou dans un nouveau groupe avec un nouvel algorithme à appliquer.

Structure interne d'un composant. La structure interne d'un composant supportant la variabilité de comportement est définie selon le patron de conception *« Stratégie »* [GHJV95] qui permet de définir des familles d'algorithmes encapsulés, interchangeables et associés à des situations spécifiques. Ce patron propose la délégation de l'exécution de l'algorithme à des objets interchangeables qui respectent une interface commune. Ainsi, la structure interne d'un composant primitif adaptable est constituée de deux types d'objets, *Selector* et *Strategy* (voir figure 5.6). Le premier sélectionne l'algorithme à utiliser pour chaque requête au composant. Il s'agit de l'algorithme par défaut (*defaultAlgorithm* dans la figure) si aucun groupe n'est identifié pour la donnée ou l'utilisateur concernés par la requête, ou l'algorithme correspondant si la classification existe. Cette classification est réifiée par les deux attributs *groupMembersMap* et *groupStrategyMap* de la figure. Le premier classe les données ou les utilisateurs par groupes. Le deuxième identifie l'algorithme à utiliser pour chaque groupe.

Variabilité de structure et de distribution

Points de variation. Notre modèle architectural inclut deux types de points de variation qui affectent la structure d'un système de réplication : la multiplicité des types de composant et celle des types d'interface. La première définit les contraintes à respecter sur la cardinalité de composants d'un type lors de la spécialisation et de l'adaptation dynamique d'un système de réplication concret. Modifier les cardinalités des types de composants requiert la prise en compte des connexions de ces composants et, donc, la modification possible de la cardinalité des interfaces concernées. La structure du système de réplication est ainsi modifiée.

La multiplicité des types d'interface définit les contraintes à respecter sur le nombre des composants connectés à une interface serveur et celui des interfaces client d'un type de composant. Elles doivent être respectées lors de la spécialisation et de l'adaptation dynamique d'un système de réplication concret. Cette multiplicité décide alors du nombre de connexions entre les composants. Changer la cardinalité d'interface permet donc de modifier la structure du système de réplication. Ainsi, par exemple, lors d'une adaptation,

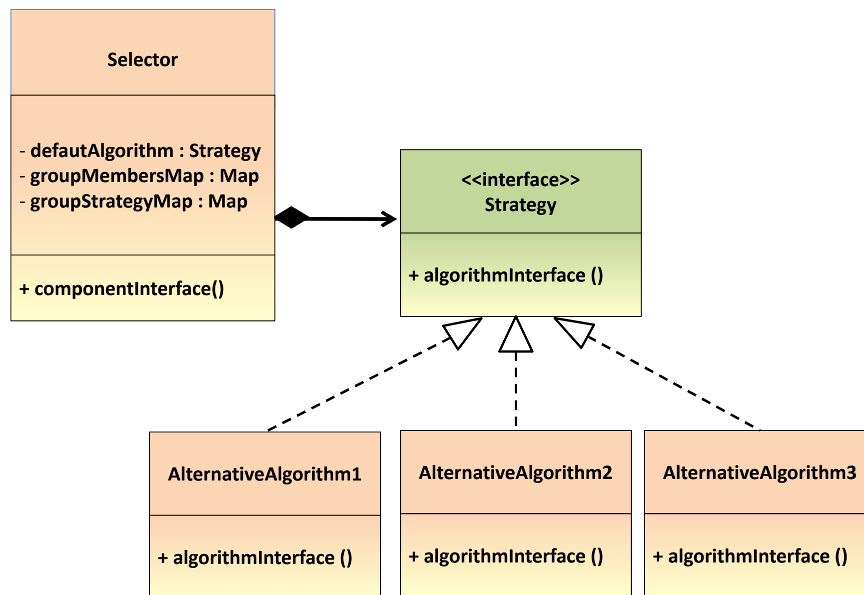

Figure 5.6 – Structure interne d'un composant primitif à comportement variable

la connexion d'un contrôleur d'accès avec un gestionnaire de propagation de MAJ peut être remplacée par une nouvelle connexion avec un autre gestionnaire de propagation afin de changer la répartition de la charge.

Concernant la distribution des composants d'un système de réplication, nous incluons dans le modèle architectural un point de variation dans tous les types de composant. Il s'agit de l'attribut *node* qui définit le nœud sur lequel le composant sera déployé. La valeur de cet attribut pour un composant est fixé lors de la spécialisation du modèle architectural et peut être modifié lors d'une adaptation (en cours d'exécution) pour mettre en place la migration de composants.

Contraintes d'instanciation. Le choix des cardinalités des composants et des interfaces ainsi que des nœuds sur lesquels les composants sont déployés est guidé par des contraintes que nous appelons *contraintes d'instanciation* qui doivent être respectées par l'expert de réplication et lors de toute adaptation modifiant la structure ou la distribution du système de réplication. La figure 5.7 présente ces contraintes sous forme d'un diagramme de classes UML. À gauche de la figure, nous avons représenté les concepts qui guident l'instanciation et la distribution et à droite, les composants de notre modèle.

- Concernant les données, les répliques et les nœuds, la figure indique qu'une donnée a un ensemble de répliques chacune d'entre elles étant déployée sur un seul nœud.
- Le type « `AccessController` ». Dans notre modèle, un composant de ce type doit être déployé sur chaque nœud pouvant héberger des répliques. Il assure les fonctions de contrôle d'accès et de manipulation de celles-ci.
- Les types « `PlacementManager` » et « `RequestEngine` ». Dans notre modèle, chaque utilisateur est associé à un seul composant de ces types qui gèrent ses requêtes de lecture/écriture et de création/suppression de répliques. Les experts en réplication et en adaptation sont libres de choisir quels utilisateurs sont gérés par quel composant et l'emplacement de ceux-ci.
- Le type « `UpdatePropagationManager` ». Le choix du nombre de ce type de com-

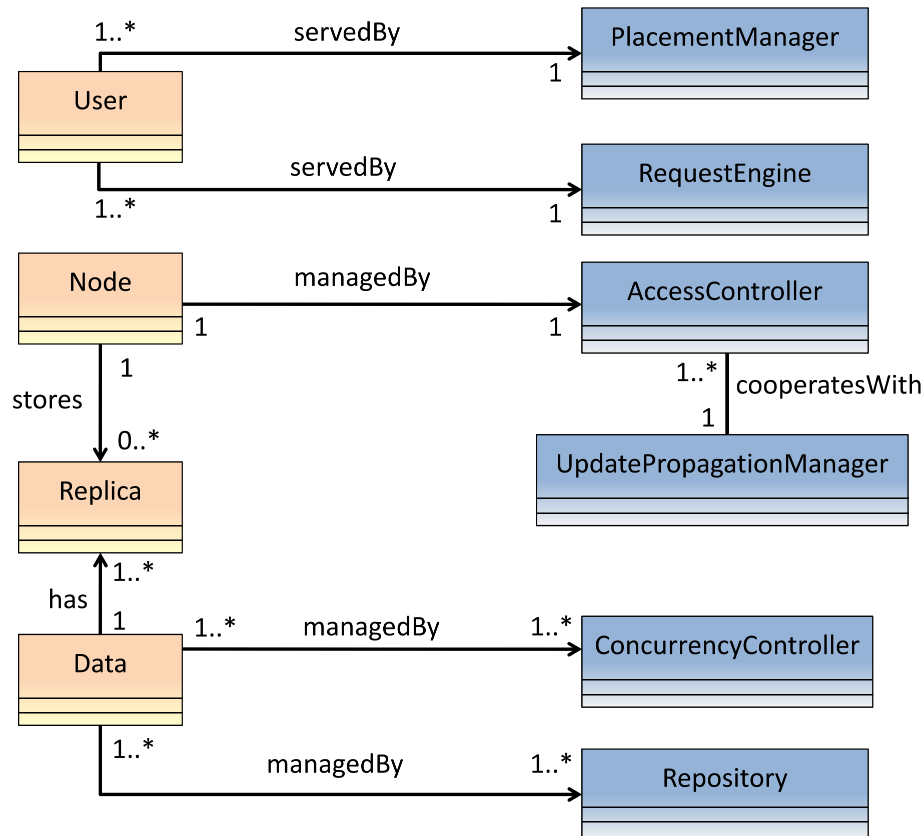

FIGURE 5.7 – Contraintes sur l'instanciation de composants d'un système de réplication

posants se base sur les contraintes présentées dans la figure 5.7. Ces contraintes indiquent que chaque composant de type « `UpdatePropagationManager` » est connecté à un ensemble de composants de type « `AccessController` ». À l'inverse, un composant de type « `AccessController` » est connecté à un seul composant de type « `UpdatePropagationManager` ». Ces contraintes signifient qu'un seul composant « `UpdatePropagationManager` » est responsable d'initier la propagation des mises à jour soumises par l'ensemble des composants « `AccessController` » auxquels il est connecté. On dit alors que le composant assure la mise à jour des répliques sur le l'ensemble des nœuds qui hébergent ces contrôleurs d'accès.
De plus, un composant de type « `UpdatePropagationManager` » se charge de répercuter les mises à jour provenant d'autres composants du même type sur les répliques hébergées par cet ensemble de nœuds. Ainsi, lorsque des répliques d'une même donnée se trouvent sur des nœuds gérées par différents « `UpdatePropagationManager` », ceux-ci coopèrent pour mettre à jour toutes les répliques.

- Le type « `ConcurrencyController` ». Une donnée peut être gérée par un ou plusieurs composants de ce type. Dans le premier cas, le composant « `AccessController` » concerné par une demande d'accès interagit avec le composant « `ConcurrencyController` » correspondant à la donnée accédée. Dans le deuxième cas, le composant « `AccessController` » concerné par la demande d'accès coopère avec un des composants « `ConcurrencyController` » qui interagit avec d'autres composants du même type pour assurer le contrôle de la concurrence.

- Le type « `Repository` ». Les métadonnées d'une donnée peuvent être gérées par un ou plusieurs composants de ce type. Lorsqu'elles sont gérées par un seul, les composants clients de celui-ci doivent connaître le composant concerné pour chaque donnée. Dans le cas contraire, les métadonnées d'une même donnée sont réparties sur plusieurs composants. Dans ce cas, ceux-ci doivent coopérer afin que les métadonnées soient disponibles par tout composant de type « `Repository` ».

5.6 Cas d'étude : exemple de spécialisation du modèle architectural

Pour illustrer les contraintes présentées dans le paragraphe précédent, nous décrivons ci-après un exemple de répartition d'un système de réplication. Puis, nous allons décrire des scénarios de réplication d'une donnée et d'accès à celle-ci afin d'illustrer un comportement possible du système. Ensuite, nous présentons un exemple de spécialisation d'un système d'adaptation au système de réplication.

5.6.1 Répartition de composants d'un système de réplication

La figure 5.8 présente un exemple de système de réplication pour le cas d'étude de la section 5.2. Les composants du système de réplication sont distribués sur 9 nœuds interconnectés. Sur chacun des nœuds pouvant héberger des répliques, un composant de type « `AccessController` » est instancié qui manipulera donc les répliques locales.

Le personnel de santé de l'hôpital *H1* peut demander d'accéder et de répliquer des données. Il manipule les données médicales du patient de manière fréquente. Un composant *req1* de type « `RequestEngine` » et un autre *pla1* de type « `PlacementManager` » sont créés sur le serveur de l'hôpital. Ils gèrent les demandes d'accès et de création/suppression de données du personnel de santé de celui-ci.

Concernant le personnel de l'hôpital *H2*, il peut accéder à des répliques existantes mais ne peut pas en créer. En effet, il manipule les données rarement et il intervient essentiellement dans les situations d'urgence pour collaborer avec le personnel du premier hôpital. Un composant *req2* de type « `RequestEngine` » est donc instancié sur le serveur de l'hôpital pour gérer les demandes d'accès du personnel de l'hôpital *H2*.

Sur le serveur au domicile du patient, le composant *pla2* de type « `PlacementManager` » s'occupe de la réplication des mesures faites pour le patient. Ce serveur est suffisamment puissant pour héberger et exécuter un ensemble de composants du système de réplication. Des répliques de taille réduite peuvent être placées sur ce serveur car sa capacité de stockage est limitée.

Un composant de type « `UpdatePropagationManager` » assure la mise à jour des répliques hébergées par un ensemble de nœuds. Dans cet exemple, quatre composants de ce type (*upd1..upd4*) ont été instanciés. Chacun s'occupe d'une partie des nœuds de sorte à assurer l'équilibrage de la charge de mise à jour des répliques. Ces composants sont connectés entre eux sous forme d'anneau pour l'échange des mises à jour lorsque ceci est nécessaire. Ils sont placés sur des nœuds ayant de bonnes capacités de calcul et ils sont proches des contrôleurs d'accès avec lesquels ils interagissent.

Enfin, le composant *repo1* de type « `Repository` » et le composant *conc1* de type « `ConcurrencyController` » sont créés sur le serveur de l'hôpital H1 et gèrent les données créées par le composant *pla1* sur ce même serveur. En effet, ces deux composants interagissent le plus avec les autres composants sur ce serveur. De même, les composants *repo2* et *conc2* de type « `Repository` » et « `ConcurrencyController` » sur un serveur au domicile

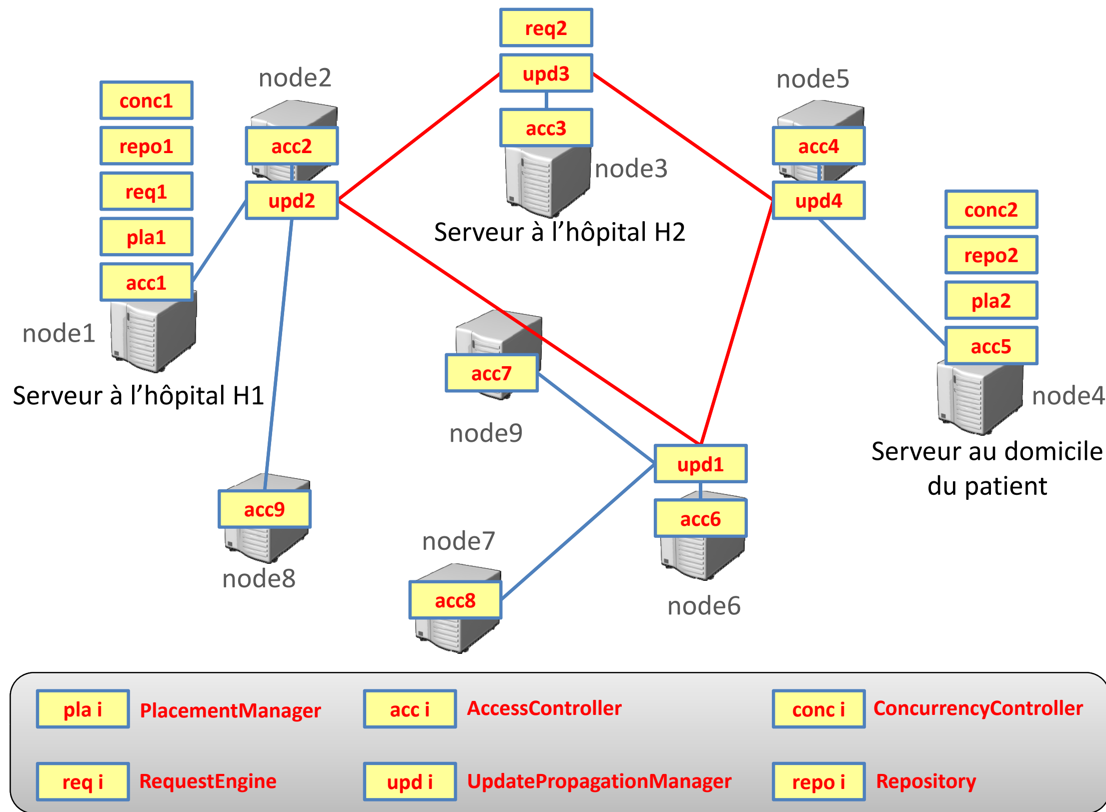

FIGURE 5.8 – Exemple de répartition des composants d'un système de réplication

du patient gèrent les données créées par le composant *pla2* de type « `PlacementManager` » sur ce même serveur.

Chaque composant a des connexions vers ceux avec lesquels il interagit. Sur la figure, nous ne montrons que celles qui sont importantes pour la compréhension des scénarios présentés dans le paragraphe ci-après.

5.6.2 Comportement du système

Les différents composants d'un système de réplication fournissent des services et collaborent ensemble pour assurer les différentes fonctions du système. Dans ce paragraphe, nous présentons le comportement de ces composants et leurs interactions sur des scénarios de réplication de données et d'accès aux répliques. Les scénarios sont décrits sous forme de diagrammes de séquence UML.

Dans nos scénarios, nous considérons un système de réplication qui assure un placement de données aléatoire, une sélection des répliques qui réduit la latence réseau et un protocole de cohérence pessimiste ROWA [BHG87] (voir section 3.4).

Les métadonnées sur chaque donnée sont gérés par un seul composant de type « `Repository` » et un seul composant « `ConcurrencyController` » intervient pour gérer les répliques de chaque donnée.

Réplication d'une donnée. La figure 5.9 décrit un scénario de réplication d'une donnée *D2* initiée par un utilisateur de l'hôpital *H1*.

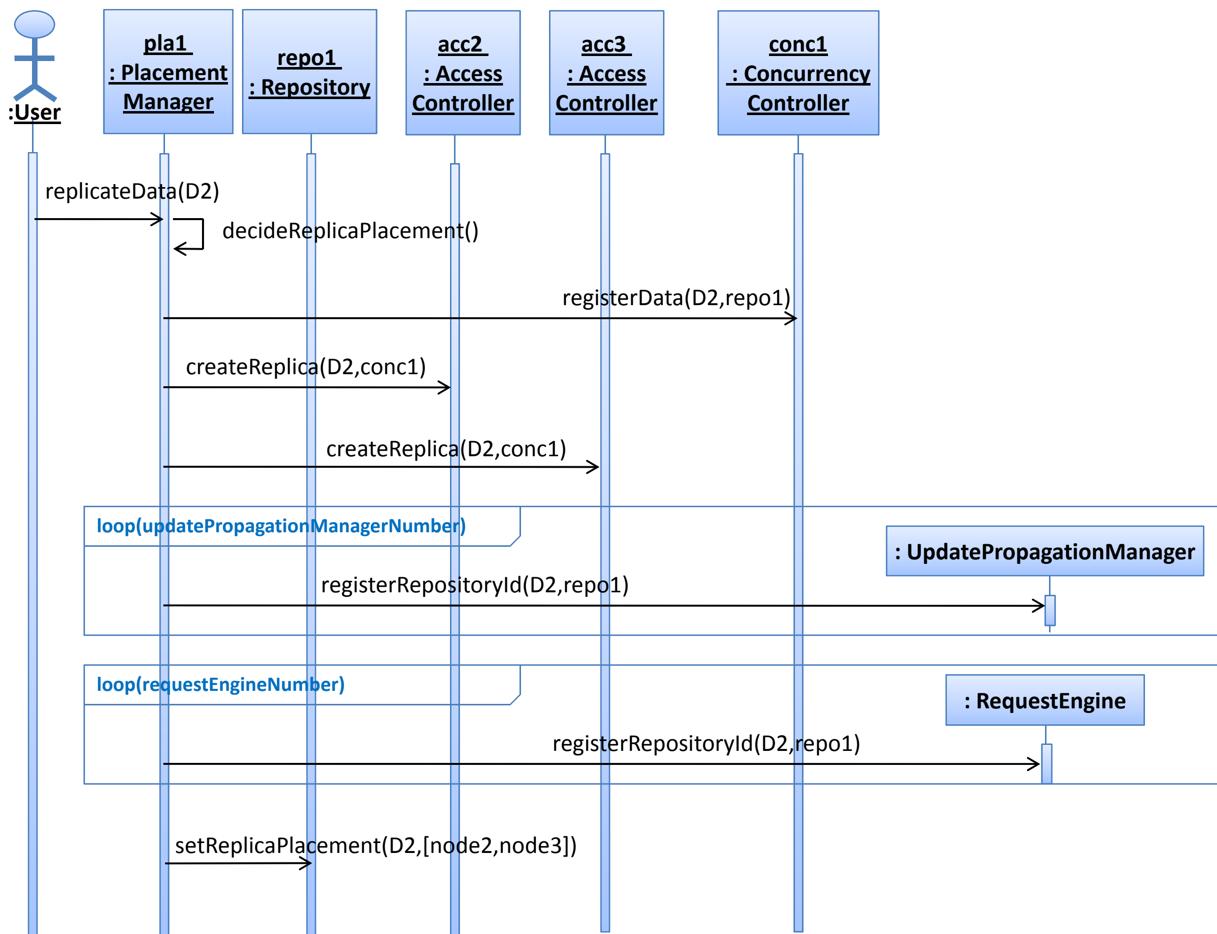

FIGURE 5.9 – Diagramme de séquence de réplication d'une donnée

L'utilisateur envoie sa requête vers le composant *pla1* qui gère les créations/suppressions provenant des utilisateurs de l'hôpital *H1*. Ce composant décide du schéma de réplication : il choisit de créer deux répliques sur les nœuds *node2* et *node3*. Puis, il attribue au composant *conc1* le contrôle de concurrence des accès aux répliques de la donnée en lui précisant le nom de l'annuaire *repo1* qui gère les métadonnées de la nouvelle donnée.

Il interagit ensuite avec les contrôleurs d'accès *acc2* et *acc3* responsables des nœuds choisis en leur demandant de créer chacun une réplique locale et il précise l'identifiant du composant *conc1* responsable du contrôle de la concurrence de la donnée.

Puis, le composant *pla1* informe les composants de type « `UpdatePropagationManager` » et « `RequestEngine` » de l'identifiant du composant *repo1* qui gère les métadonnées de la nouvelle donnée.

Enfin, *pla1* demande au composant *repo1* de stocker le placement des répliques. La donnée est considérée en cours de réplication tant que cette dernière opération n'est pas réalisée. Les demandes d'accès sont mises en attente jusqu'à ce que l'annuaire soit capable de répondre à la demande d'un moteur de requêtes qui essaye de consulter le placement des répliques de la donnée.

Accès à une donnée. Dans la figure 5.10, nous représentons le scénario d'une lecture d'une donnée *D2* par un utilisateur de l'hôpital *H1*. Pour des raisons de clarté, nous ne

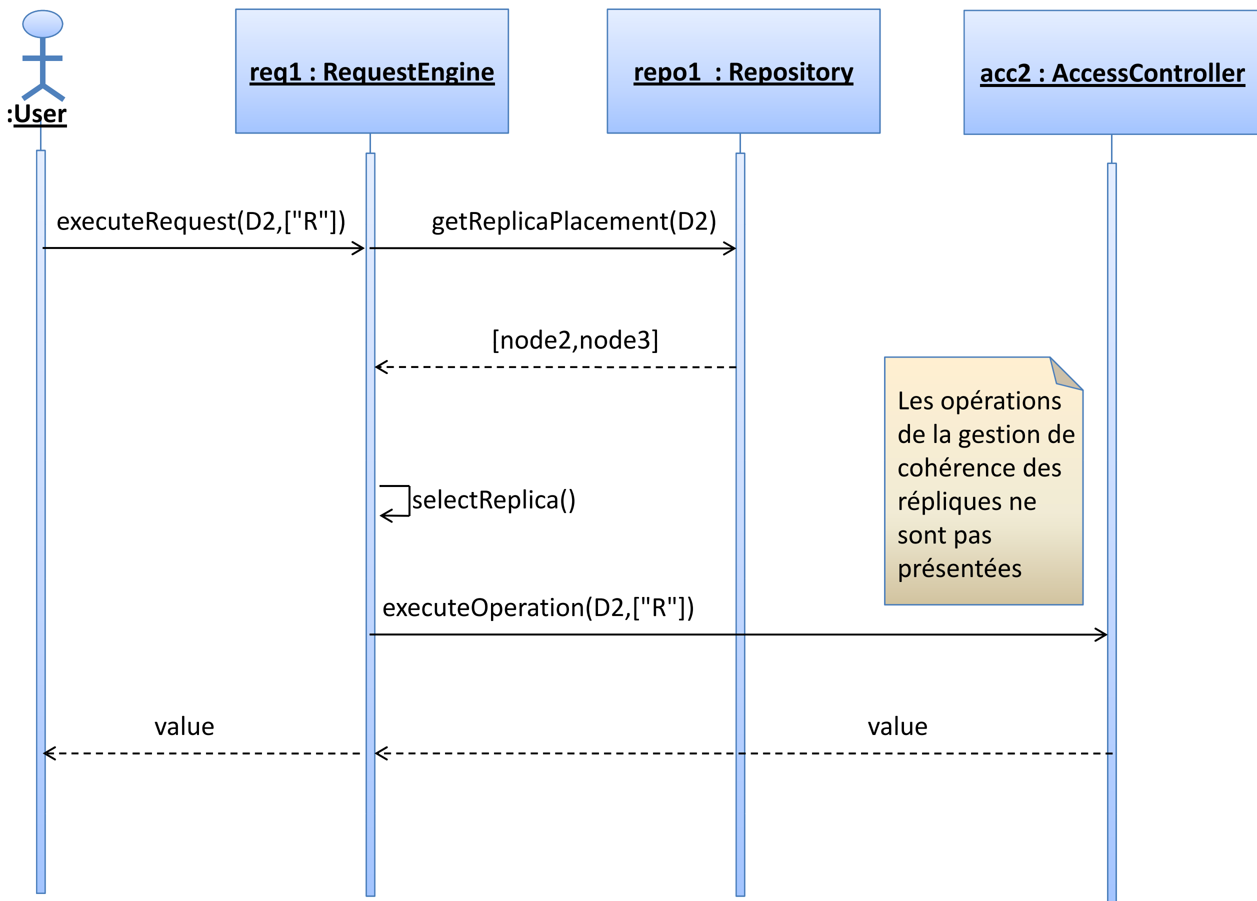

FIGURE 5.10 – Diagramme de séquence d'accès à une donnée

décrivons pas les traitements nécessaires pour la gestion de la cohérence.

L'utilisateur envoie sa requête de lecture au composant *req1* qui gère les requêtes des utilisateurs de l'hôpital *H1*. Celui-ci consulte le composant *repo1* qui connaît le placement des répliques puis, il sélectionne la réplique hébergée par le nœud *node2* à utiliser.

Ensuite, il envoie la requête de lecture au composant *acc2* sur le nœud hébergeant la réplique choisie. Ce dernier retourne à l'utilisateur le résultat qui est la valeur de la réplique lue.

Dans le cas d'écriture de la donnée, le scénario est similaire et il se termine par le renvoi de l'acquittement de la réalisation de l'écriture.

Gestion de la cohérence. Le diagramme représenté dans la figure 5.11 décrit les traitements de gestion de cohérence en cas de lecture puis, en cas d'écriture de la donnée *D2* par un utilisateur de l'hôpital *H1*.

Lorsque le composant *acc2* reçoit l'opération de lecture du composant *req1*, il interagit avec le composant *conc1* responsable de la donnée pour lui notifier la tentative de lecture. Celui-ci verrouille l'accès en lecture et en écriture à cette réplique et l'accès en écriture à toutes les autres répliques de la donnée *D2*[3]. Puis, il informe *acc2* que la réplique est dis-

3. Le protocole ROWA (« *Read One Write All* ») autorise l'écriture sur n'importe quelle réplique, la propagation de celle-ci aux autres répliques étant atomique. Il autorise aussi des lectures simultanées sur

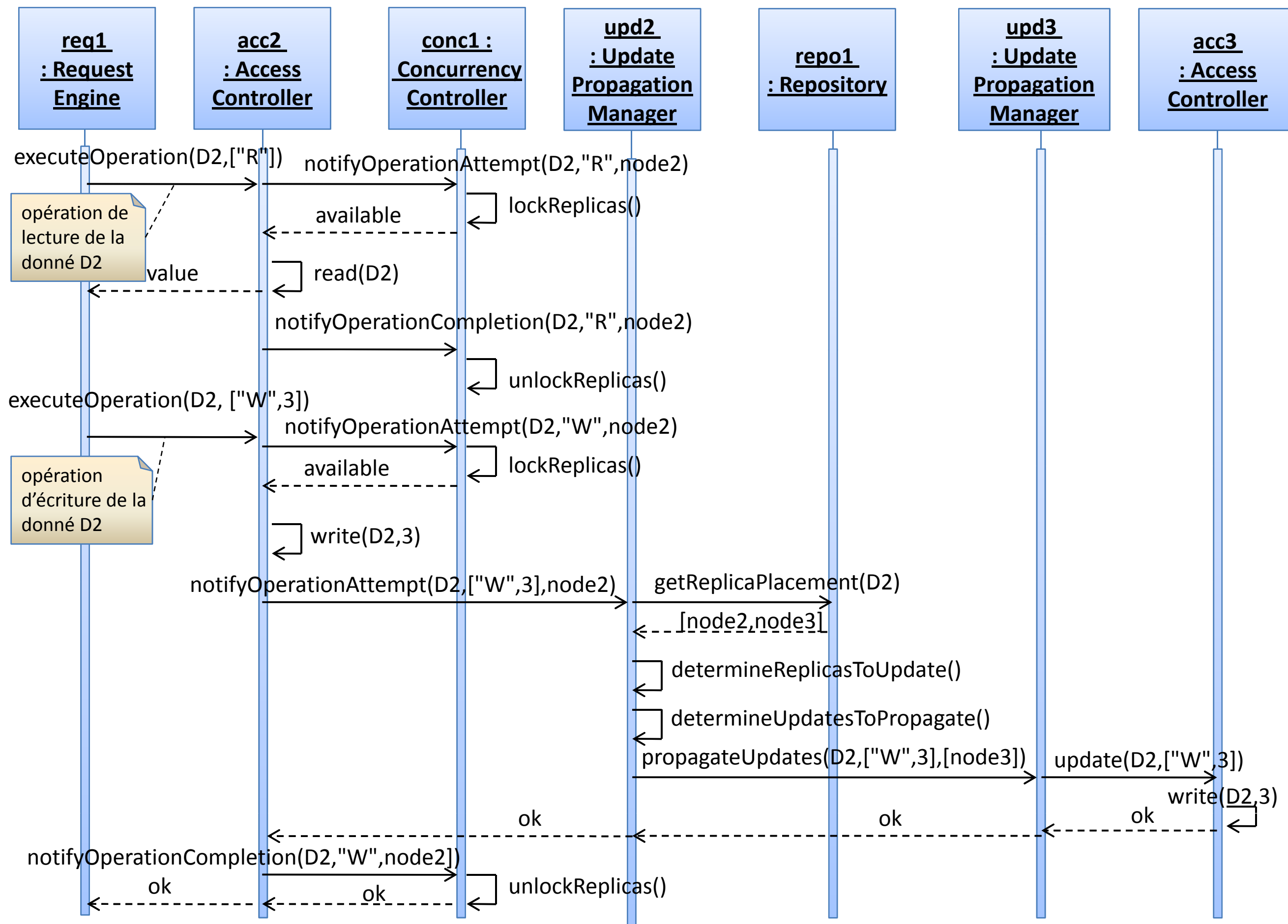

FIGURE 5.11 – Diagramme de séquence de gestion de cohérence suite à une lecture/écriture d'une donnée

ponible pour la lecture. Ce dernier réalise l'opération, retourne la valeur lue au composant *req1* et notifie la fin de l'opération à *conc1* qui libère tous les verrous.

Lorsque l'opération est une écriture, *conc1* verrouille toutes les répliques de la donnée en lecture et en écriture dès qu'il est notifié de la tentative d'écriture. Ensuite, *acc2* modifie la réplique choisie et notifie le composant *upd2* de cette opération.

Le composant *upd2* consulte alors l'annuaire *repo1* pour récupérer le placement des répliques de la donnée *D2*. Il détermine les répliques à mettre à jour et les mises à jour à propager [4], puis, il initie la diffusion de l'opération aux autres composants de propagation de MAJ. Cette diffusion étant un parcours d'anneau dans notre scénario, *upd3* reçoit l'opération et conclut qu'il ne doit pas propager la mise à jour à d'autres composants de propagation. Il interagit avec le contrôleur d'accès *acc3* responsable du nœud *node3* qui héberge la réplique à mettre à jour et lui soumet l'opération.

Enfin, le contrôleur d'accès *acc2* initial est informé de la réussite de la mise à jour et notifie le contrôleur de concurrence *conc1* de la fin de l'opération pour libérer tous les verrous sur les répliques.

plusieurs répliques d'une donnée.

4. Dans notre scénario, il s'agit de transmettre l'opération d'écriture qui vient d'être faite aux autres répliques de la donnée : la réplique hébergée par le nœud *node3*.

5.6.3 Spécialisation d'un système d'adaptation

Afin d'illustrer l'articulation entre un système de réplication et un système d'adaptation, ce paragraphe présente un exemple de démarche que l'expert en adaptation pourra suivre. Dans cet exemple, il doit analyser les caractéristiques du système de réplication à adapter et de son environnement d'exécution pour spécialiser un système d'adaptation approprié.

Nous considérons un ensemble d'hypothèses selon lesquelles la spécialisation est faite. D'abord, l'expert constate que les composants du système de réplication s'exécutent dans des environnements hétérogènes et que l'environnement global est large. Alors, il décide de distribuer la gestion de contexte ainsi que la gestion d'adaptation. De plus, l'objectif de l'expert est de sélectionner des composants applicatifs dans le champ d'action de chaque gestionnaire de contexte et d'adaptation de sorte qu'ils soient proches géographiquement. Enfin, les fonctions de placement de répliques, d'interrogation de données et de gestion de cohérence des répliques sont sensibles à des aspects de contexte différents et le système d'adaptation leur applique des stratégies d'adaptation différentes. L'expert décide ainsi de définir des gestionnaires d'adaptation séparés pour chaque fonction.

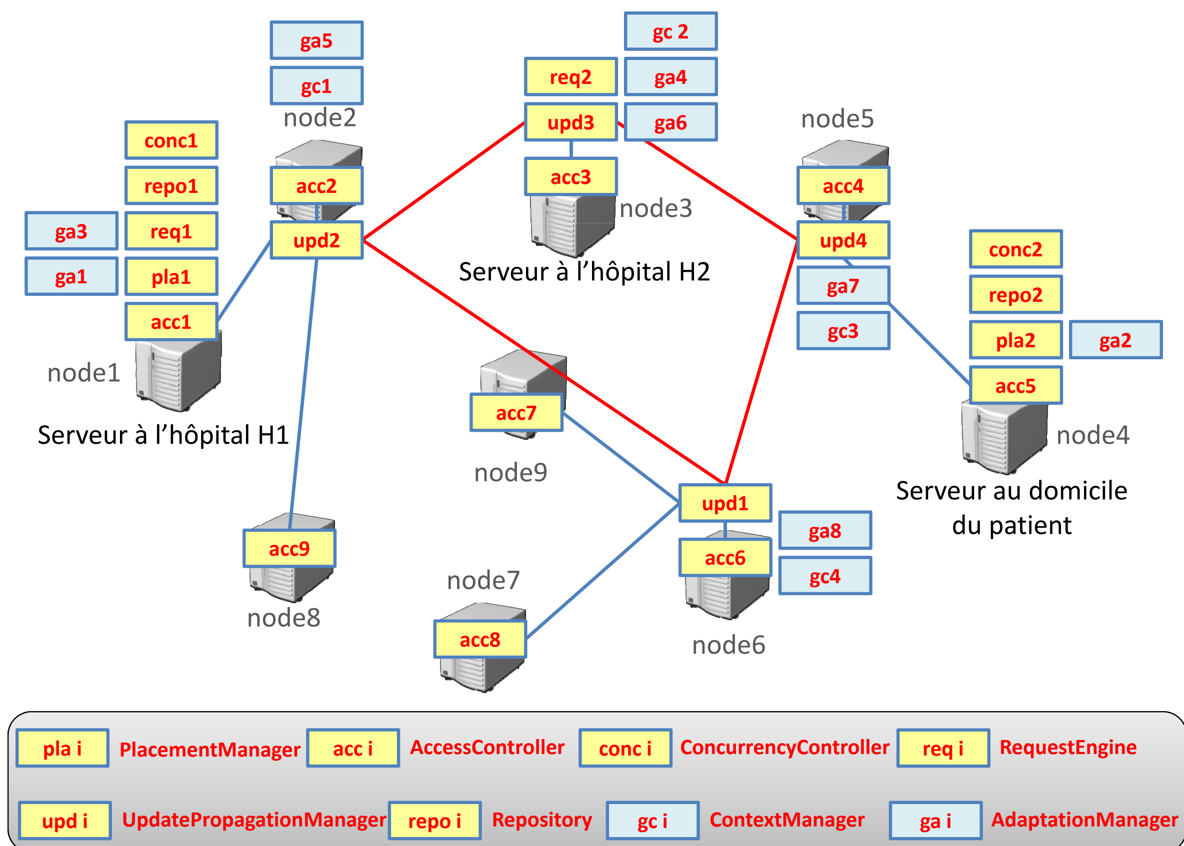

FIGURE 5.12 – Exemple de répartition des composants d'un système d'adaptation

Sous ces hypothèses, l'expert choisit quatre gestionnaires de contexte (*gc1..gc4*) (voir figure 5.12). Les gestionnaires *gc1* et *gc2* surveillent le contexte respectivement au niveau de l'hôpital H1 et l'hôpital H2. Le gestionnaire *gc3* s'intéresse aux informations contextelles liées au nœud qui l'héberge et au patient. Finalement, le gestionnaire *gc4* gère le contexte au niveau des nœuds 6, 7 et 9. Les types d'informations sur le contexte gérées dans chaque

environnement diffèrent. Par exemple, *gc1* tient compte des aspects liés à la mobilité des utilisateurs comme la connectivité et leurs emplacements alors que *gc3* se distingue par la prise en compte de la situation du patient.

De plus, l'expert définit plusieurs gestionnaires d'adaptation comme le montre la figure 5.12. D'abord, les deux gestionnaires de placement répliquent des données différentes et sont éloignés géographiquement. Ainsi, un gestionnaire d'adaptation est associé à chacun d'eux (*ga1* et *ga2*). Ensuite, les moteurs de requêtes sont utilisés par des utilisateurs distincts dans des environnements distants et hétérogènes. Ainsi, il associe à chaque moteur un gestionnaire d'adaptation spécifique (*ga3* et *ga4*). Puis, les gestionnaires de propagation de mises à jour sont déployés dans des environnements hétérogènes et sont distants. Donc, l'expert en adaptation associe à chacun un gestionnaire d'adaptation (*ga5..ga8*). Enfin, le contrôleur de concurrence *conc1* est dans le champ d'action de *ga5* et *conc2* dans celle de *ga7*. Les quatre gestionnaires d'adaptation responsables de la gestion de cohérence (*ga5..ga8*) coopèrent. Chacun inclut donc un négociateur et un coordinateur d'exécution et est connecté aux trois autres. Par ailleurs, les services de placement et de gestion de cohérence sont dépendants. Ainsi, l'expert choisit un gestionnaire d'adaptation coopératif associé à un gestionnaire de placement coopératif et le connecte aux gestionnaires d'adaptation des gestionnaires de propagation. Les gestionnaires d'adaptation associés aux moteurs de requêtes ne sont pas coopératifs. En effet, le choix du comportement d'un moteur de requêtes ne dépend pas des comportements des autres composants applicatifs. Selon les composants applicatifs qu'il adapte, chaque gestionnaire d'adaptation est connecté au gestionnaire de contexte fournissant les informations nécessaires. L'expert spécialise aussi le comportement des différents composants du système d'adaptation.

5.7 Conclusion

Dans ce chapitre, nous avons présenté notre modèle architectural pour la réplication de données et nos principes de développement de systèmes de réplication auto-adaptables. Nous nous sommes intéressés aux différentes fonctions d'un système de réplication de façon globale et nous les avons conçues de manière modulaire à base de composants logiciels. Notre modèle supporte la variabilité de la structure, du comportement et de la distribution du système de réplication. Ainsi, il offre une grande flexibilité pour spécialiser un système de réplication en fonction d'une application cible et pour l'adapter dynamiquement. En effet, notre approche supporte divers types de modification du système de réplication et l'adaptation peut concerner seulement un sous ensemble de données ou d'utilisateurs pour certaines de ses fonctions. Pour assurer l'adaptation dynamique, le système de réplication offre des interfaces spécifiques pour le connecter à un système d'adaptation dédié.

Les types d'interfaces spécifiées dans notre modèle architectural prennent en compte l'existence de plusieurs approches de réplication en définissant plusieurs types d'opérations possibles et plusieurs possibilités d'interaction entre les composants. L'implémentation de chaque interface peut varier selon le comportement souhaité. Cependant, il est envisageable d'ajouter de nouveaux types interfaces, voire de nouveaux types de composants. Ceci peut être nécessaire pour enrichir notre modèle et élargir le spectre de systèmes de réplication pouvant être définis.

Notre approche qui se base sur la définition de deux modèles architecturaux, le premier pour l'adaptation et le second pour la réplication, peut être suivie pour rendre d'autres applications auto-adaptables. La généricité de nos modèles permet de mettre en œuvre notre approche avec plusieurs modèles de composants logiciels existants. Le chapitre suivant décrira notre implémentation des modèles et de la fabrique ainsi que l'évaluation de notre

approche.

Chapitre 6

Implémentation et expérimentation

6.1 Introduction

Dans ce chapitre, nous présentons la mise en œuvre de notre proposition et sa validation par des expérimentations. Nous décrivons les implémentations de nos deux modèles architecturaux de systèmes d'adaptation et de systèmes de réplication ainsi que notre fabrique qui ont été présentés dans les deux chapitres précédents. Nous avons choisi le modèle de composants Fractal pour réaliser notre prototype. Nous avons utilisé Julia, l'implantation de référence en Java de ce modèle. Notre prototype permet ainsi de construire un système de réplication auto-adaptable comme un assemblage de composants Fractal distribués sur plusieurs nœuds. Dans la suite de ce chapitre, le premier paragraphe introduit le modèle de composants Fractal et un ensemble d'outils qui y sont associés et que nous avons utilisés pour le développement de notre prototype. Le paragraphe 6.3 présente celui-ci. Enfin, le dernier paragraphe décrit des expérimentations que nous avons menées pour évaluer les performances des implémentations réalisées et discute les résultats obtenus.

6.2 Le modèle de composants Fractal et les outils associés

Pour implémenter notre proposition, nous avons choisi le modèle de composants Fractal. Ce choix a été motivé par (i) la nature hiérarchique du modèle qui facilite l'implémentation de nos modèles architecturaux, (ii) les facilités qu'il offre pour mettre en place l'adaptation grâce à la réflexivité (un composant peut s'inspecter et se modifier), (iii) l'extensibilité du modèle qui permet de personnaliser les capacités de contrôle de chaque composant afin de définir celles dédiées à l'adaptation et (iv) l'activité importante autour du modèle et ses implantations pour le développement d'applications à base de composants (noyaux de systèmes d'exploitation, bibliothèques de communication, intergiciels à messages, systèmes de gestion de persistance...).

6.2.1 Le modèle de composants Fractal

Fractal [BCL+06] est un modèle de composants proposé par le consortium OW2 [1]. Un composant Fractal est une brique de base pour la construction d'applications qui est manipulé à l'aide de ses interfaces. Un composant Fractal fournit et requiert des interfaces fonctionnelles et fournit un ensemble d'interfaces de contrôle. La liste de ces dernières inclut, entre autres, une interface dédiée à la gestion du cycle de vie du composant et au contrôle du contenu du composant lorsqu'il s'agit d'un composite.

1. http ://fractal.ow2.org/

On distingue deux parties dans la structure d'un composant Fractal :

- *Une membrane.* Elle contient toutes les interfaces du composant ainsi que l'implémentation des interfaces de contrôle.
- *Un contenu.* Il comprend un ensemble de sous-composants ou d'objets qui implémentent les interfaces fonctionnelles du composant.

Dans Fractal, il existe deux types de composant selon le contenu de celui-ci :

- *Composant primitif.* Le contenu d'un composant primitif est un objet.
- *Composant composite.* Le contenu d'un composant composite est un ensemble de composants (composite ou primitif).

L'exemple de la figure 6.1 décrit un composant composite *HelloWorld* contenant deux composants *Client* et *Server*. Ces composants représentent un client et un serveur connectés via une interface *s*. Une interface *r* fournie par le composant client est exportée en dehors de la composition.

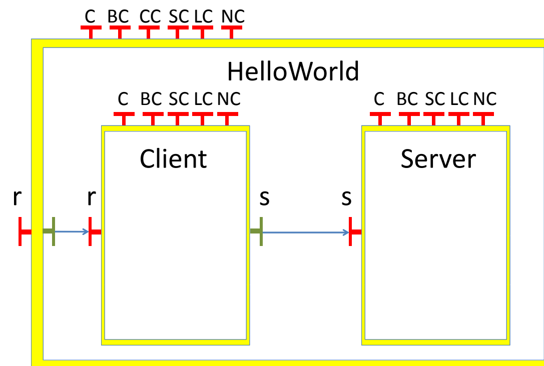

FIGURE 6.1 – Représentation graphique d'un composite *HelloWorld*

Par convention, on représente à gauche et en rouge les interfaces serveur, et à droite et en vert les interfaces client. Les interfaces de contrôle sont représentées sur la partie haute du composant.

Les composants sont liés les uns aux autres via leurs interfaces fonctionnelles. Les connexions entre interfaces sont nommées « *binding* ». Un composant Fractal a un cycle de vie. Il peut être dans l'état *stopped* (arrêté) ou *started* (en marche).

Le modèle Fractal est un modèle de composants typé. Il définit un système de types pour les composants et les interfaces. Le type d'un composant Fractal est l'ensemble des types de ses interfaces fonctionnelles client et serveur. Un type d'interface est constitué du nom de l'interface, de sa signature, de son rôle (client ou serveur), de sa contingence (obligatoire ou optionnelle) et de sa cardinalité (unique ou multiple). Le typage en Fractal rend possible la vérification de l'assemblage de composants.

Pour un composant, la spécification Fractal définit différents contrôleurs :

- *AttributeController ou AC.* Il permet d'accéder et de modifier les attributs du composant.
- *BindingController ou BC.* Il permet de gérer les liaisons du composant avec les autres composants.
- *ContentController ou CC.* Il permet de gérer le contenu du composant. Ce contrôleur existe seulement pour les composants composites. Il est utilisé pour connaître l'ensemble des sous-composants et des interfaces internes du composant. Il permet aussi d'ajouter ou supprimer des sous-composants.

- *LifeCycleController ou LC.* Il permet de connaître l'état d'exécution du composant, de l'arrêter et de le démarrer.
- *NameController ou NC.* Il permet de connaître ou modifier le nom du composant.
- *SuperController ou SC.* Il permet de connaître le composant père (contenant le composant) du composant.
- *Component.* Il permet d'accéder aux interfaces externes (clients ou serveurs) du composant.
- *Interface.* Il permet d'obtenir des informations sur les interfaces du composant : nom, propriétaire...

Dans la figure 6.1, chacun des composants primitifs *Client* et *Server* dispose de cinq interfaces de contrôle : *Component*, *BindingController*, *SuperController*, *LifeCycleController* et *NameController*. Le composite *HelloWorld* a une interface de contrôle en plus qui est l'interface *ContentController*.

6.2.2 Julia : une implémentation de Fractal

Julia [2] est l'implémentation de référence du modèle Fractal pour des applications Java. Il s'agit d'un canevas logiciel qui permet d'instancier des composants Fractal dont le contenu est implanté sous forme de classes Java. Ce canevas fournit un ensemble de contrôleurs standards et une fabrique de composants qui permet d'assembler les composants avec l'implémentation des contenus. Le canevas se présente sous la forme d'une librairie qui s'exécute au dessus d'une machine virtuelle Java (JVM) classique.

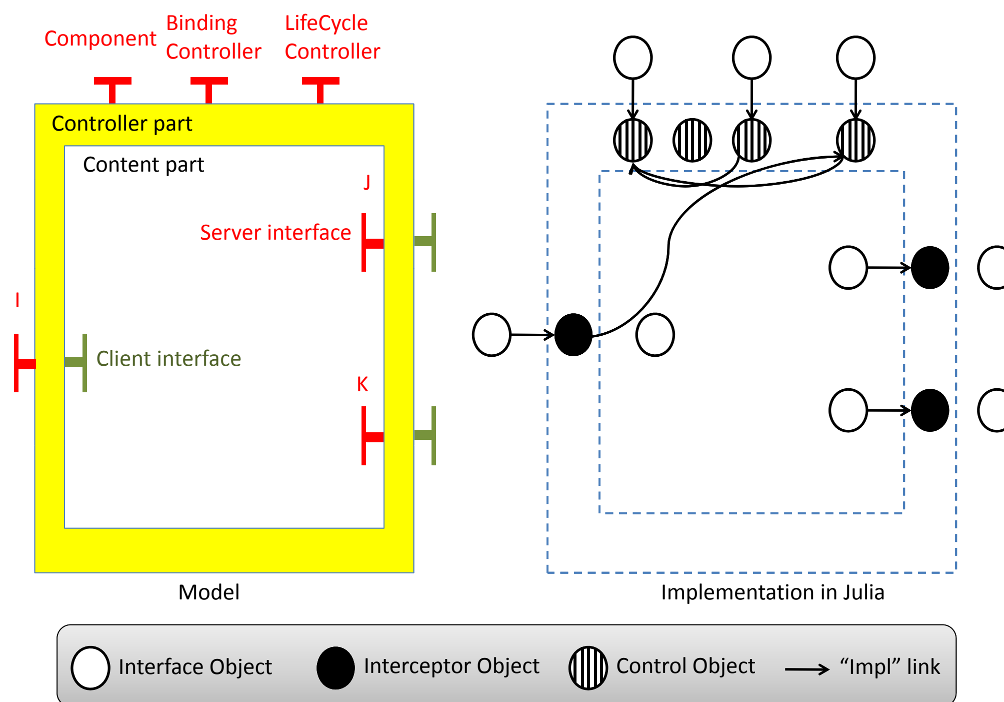

FIGURE 6.2 – Julia : l'implantation de référence en Java du modèle Fractal

Dans Julia, un composant en exécution est représenté par différents objets Java. On peut séparer ces objets en 3 groupes distincts (voir figure 6.2) :

2. http ://fractal.ow2.org/current/doc/javadoc/julia/

- Les objets qui implémentent les interfaces du composant (*Interface Objects*). Il y a un objet par interface du composant. Chaque objet a un lien (*« impl » link*) vers un objet qui implémente l'interface Java et à laquelle tous les appels de méthode sont délégués. Cette référence est nulle pour les interfaces client.
- Les objets qui implémentent les contrôleurs du composant (*Control Objects* et *Interceptor Objects*). Un objet contrôleur peut implémenter zéro, une ou plusieurs interfaces de contrôle.
- Les objets qui implémentent le contenu du composant dans le cas d'un composant primitif (non représentés sur la figure 6.2).

Par ailleurs, on distingue deux types d'objets pour représenter l'implémentation de l'interface de contrôle :

- *Intercepteur.* Un intercepteur (*Interceptor Object*) permet d'intercepter les appels entrants et sortants sur les interfaces fonctionnelles.
- *Contrôleur.* Un contrôleur (*Control Object*) implémente une partie de l'interface de contrôle. Il peut avoir des références vers d'autres contrôleurs et/ou intercepteurs.

Dans Julia, la fabrique peut être utilisée soit directement en Java (interface programmatique), soit en utilisant le langage de description d'architecture (ADL) de Fractal (interface déclarative). Ce langage est présenté dans le paragraphe suivant.

6.2.3 Fractal ADL : le langage de description d'architecture de Fractal

L'architecture d'une application à base de composants Fractal peut être décrite à l'aide du langage de description d'architecture Fractal ADL [3]. Cet ADL permet, de manière déclarative, de définir des configurations d'une application à base de composants Fractal avec une syntaxe basée sur XML. Les différents éléments d'une architecture (composants, interfaces...) sont définis dans des modules XML séparés conformes à une DTD. Un atout de cet ADL est son extensibilité. Par exemple, pour ajouter des contrôleurs dédiés à l'adaptation, l'ADL peut être étendu facilement pour prendre en compte ces nouveaux contrôleurs. De plus, de multiples usages sont possibles d'une définition ADL comme le déploiement, la vérification et l'analyse de l'architecture.

Fractal ADL est constitué de deux parties : un langage basé sur XML et une usine qui permet de traiter les définitions réalisées à l'aide du langage. Nous présentons ces deux éléments dans les deux paragraphes suivants.

Le langage Fractal ADL

L'ADL de Fractal est basé sur XML et est défini par un ensemble de DTDs dont les éléments définis peuvent être utilisés dans toute description Fractal ADL. La figure 6.3 illustre un exemple de description en Fractal ADL du composant que nous avons présenté graphiquement dans la figure 6.1. L'élément *definition* doit être utilisé pour démarrer la description de tout composant. Il a un attribut obligatoire *name* qui spécifie le nom du composant décrit. Ensuite, un élément XML *component* peut être ajouté comme un sous-élément d'un élément *definition* ou d'un autre élément *component* pour spécifier des sous-composants. Dans la figure 6.3, les composants *Client* et *Server* sont contenus dans le composite *HelloWorld.*

Fractal ADL définit d'autres éléments et attributs XML pour spécifier l'architecture d'une application, les interfaces d'un composant, l'implantation des composants primitifs et la partie contrôle des composants. De plus, concernant les informations de déploiement,

3. http ://fractal.ow2.org/current/doc/javadoc/fractal-adl/

le langage permet de préciser sur quels nœuds les composants sont placés. Dans notre exemple, le client sera déployé sur le nœud *node1* et le serveur sur le nœud *node2*.

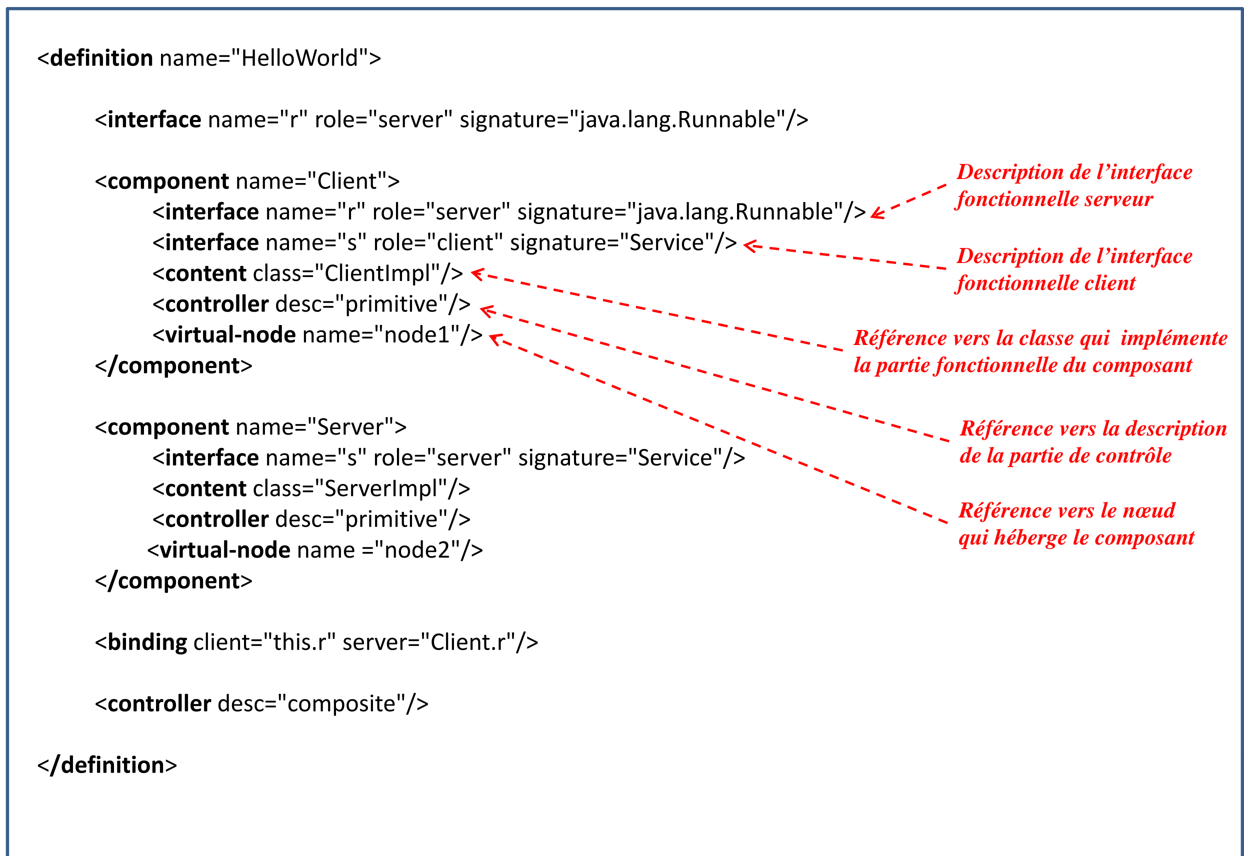

FIGURE 6.3 – Exemple de description de composants en Fractal ADL

Par ailleurs, Fractal ADL permet d'exprimer des relations de référencement et d'héritage entre des descriptions ADL afin de faciliter l'écriture de définitions ADL.

L'usine de Fractal ADL

L'usine de Fractal ADL permet de traiter les descriptions écrites en Fractal ADL. Elle est implantée par cinq composants Fractal (voir figure 6.4) qui traitent les différents éléments de Fractal ADL décrits dans le paragraphe précédent.

- Le composant *Factory* assure le contrôle de l'exécution de l'usine. Après avoir initialisé les composants de l'usine, il appelle successivement les composants *Loader*, *Compiler* et *Scheduler*.
- Le composant *Loader* est en charge de construire une AST (*Abstract Syntactic Tree*) à partir des fichiers ADL qui décrivent l'application. Il contient une chaîne de sous-composants. En particulier, un composant *Parser* lit les fichiers de description et les transforme en AST puis un ensemble de composants analysent l'AST généré pour des vérifications sémantiques particulières.
- Le composant *Compiler* est en charge de parcourir l'AST et de créer des tâches à exécuter afin de déployer l'architecture spécifiée. Ce composant contient un ensemble de sous-composants chacun dédié à la création d'une tâche spécifique comme la création d'un composant, l'établissement d'une liaison...

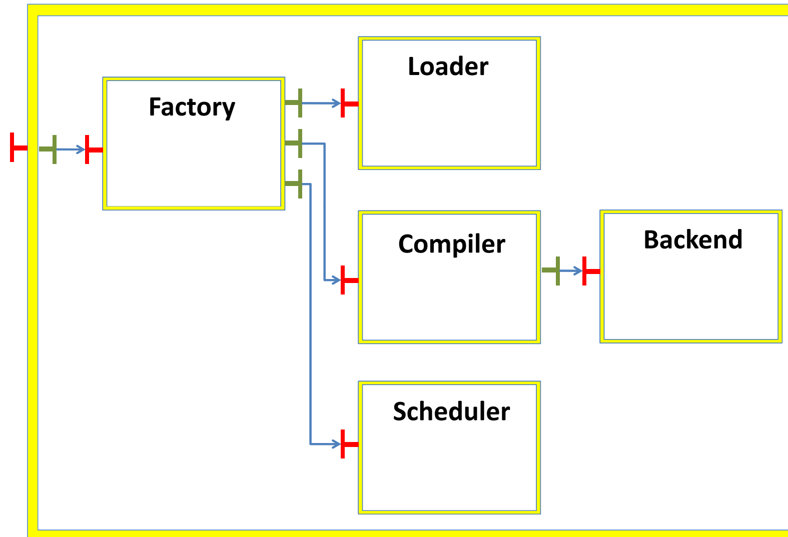

FIGURE 6.4 – Architecture de l'usine de Fractal ADL

- Le composant *Backend* encapsule un ensemble de sous-composants fournissant l'implantation concrète des tâches créées par le *Compiler*. Des composants *Backend* différents peuvent être utilisés pour obtenir des caractéristiques de déploiement différentes. Par exemple, parmi les *Backend* fournis dans le paquetage de l'usine de Fractal ADL, on en trouve un qui déploie directement une configuration ADL sur une machine virtuelle Java, alors qu'un autre génère le code de déploiement.
- Le composant *Scheduler* exécute les tâches créées par le composant *Compiler* dans un ordre correct en résolvant leurs contraintes de dépendance.

6.2.4 Fractal RMI

Fractal RMI [4] est un intergiciel qui permet des communications synchrones distantes entre des composants Fractal. Il est constitué d'une dizaine de composants Fractal regroupés dans un composant composite qui offre l'interface *NamingContext*. Une instance de ce composite est nécessaire sur chaque nœud (ou par machine virtuelle).

L'interface *NamingContext* permet de gérer l'espace de nommage utilisé par Fractal RMI : création d'un nom pour un objet local et création d'une chaîne de liaison vers un objet distant désigné par son nom.

Parmi les composants internes de Fractal RMI, on trouve des composants implémentant des protocoles et notamment le protocole d'invocation de méthode à distance, un composant pour gérer le paquetage et le dépaquetage des messages, un composant pour créer des talons et squelettes, ainsi que des composants de gestion de ressources (cache de tampons mémoire pour lire et écrire les messages, pools de threads...).

6.3 Notre prototype

Afin de proposer des outils concrets pour la construction de systèmes de réplication auto-adaptables et valider notre approche par des expérimentations, nous avons réalisé

4. http ://fractal.ow2.org/current/doc/javadoc/fractal-rmi/

une implémentation de notre proposition. Cette implémentation représente une infrastructure pour le développement et la mise en place de systèmes de réplication auto-adaptables par assemblage de composants Fractal. Elle doit notamment supporter la variabilité des configurations (implémentation, structure, distribution) de systèmes de réplication et de systèmes d'adaptation. Elle implante alors le modèle architectural d'un système d'adaptation distribué et celui d'un système de réplication de données en utilisant des mécanismes offerts par Fractal. De plus, nous avons implanté une fabrique qui instancie un système de réplication et un système d'adaptation et les interconnecte ensemble. Dans la suite, nous décrivons l'implémentation de notre modèle architectural de systèmes d'adaptation puis, celle de notre modèle architectural de réplication. Ensuite, nous présentons la mise en œuvre de la fabrique.

6.3.1 Implémentation du modèle architectural d'adaptation

L'implémentation de notre modèle architectural d'adaptation avec le modèle de composants Fractal nécessite la mise en œuvre des différents types de composants d'un système d'adaptation, la définition des points de variation et l'expression de contraintes spécifiées dans notre modèle. De plus, nous avons développé un ensemble de traitements communs spécialisables qui représentent des implantations réutilisables des composants d'un système d'adaptation.

Projection du modèle architectural vers le modèle de composants Fractal

La projection de notre modèle vers le modèle de composants Fractal a nécessité de prendre en compte quelques différences structurelles entre le modèle Fractal et le modèle de composants UML. Notamment, la notion de port est absente dans Fractal. En effet, les points d'entrée d'un composant dans le modèle Fractal sont définis par les interfaces de contrôle et les interfaces fonctionnelles. Il n'existe pas non plus de notion explicite de connecteur, bien qu'on puisse très bien utiliser un composant standard pour jouer ce rôle.

Par ailleurs, le modèle Fractal introduit le concept de liaison (*binding*) comme support des interactions entre les composants. Une liaison est un lien orienté, correspondant à un canal de communication, entre une interface client et une interface serveur. Nous utilisons Fractal RMI qui fournit une fabrique de liaisons (*binding factory*) pour créer des liaisons entre des composants distribués.

Enfin, la description d'un type de composant dans nos modèles inclut en plus des types des interfaces fonctionnelles, les points de variation de comportement et de distribution des composants et le type des interfaces de contrôle dédiées à l'adaptation pour les composants applicatifs.

Description des types des composants du système d'adaptation

Nos avons mis en œuvre les définitions de types de composants primitifs contenus dans les composants de type « `ContextManager` » et « `AdaptationManager` » à l'aide du langage Fractal ADL.

Nous répartissons la définition ADL de chaque type de composant primitif dans deux fichiers. Le premier décrit l'ensemble des types des interfaces fonctionnelles du composant. Le deuxième fichier étend (via un attribut *extends*) cette description en y rajoutant les autres éléments. Il précise des attributs du composant qui spécialisent son comportement, une référence au fichier d'implémentation du contenu, une référence à un descripteur de la

partie de contrôle du composant et une référence au nœud qui doit héberger une instance du composant.

Points de variation. Pour exprimer les points de variation, les descriptions en Fractal ADL contiennent des arguments dont les valeurs sont attribuées au moment de l'instanciation des composants. Ces arguments représentent les points de variation reliés au comportement et au placement du composant. Un argument peut représenter la classe qu'implante le composant ou un paramètre de configuration de l'algorithme qu'implémente le composant. De plus, un argument représente le nœud qui héberge le composant. Par exemple, les figures 6.5 et 6.6 représentent respectivement la description des types d'interfaces fonctionnelles en Fractal ADL d'un composant de type « `Negotiator` » et la définition ADL complète du composant qui l'étend.

La première description montre les différents types des interfaces du composant. Il a deux interfaces serveur de type « `NegotiateItf` » et « `ProposeItf` » et trois interfaces client de type « `ProposeItf` », « `MonitorItf` » et « `NotifyItf` ». Les multiplicités des interfaces sont exprimés avec les attributs *cardinality* et *contingency*. Par exemple, la multiplicité 0..* de l'interface client de type « `ProposeItf` » est exprimée en définissant une contingence optionnelle (nombre d'interfaces peut être égale à 0) et une cardinalité collection (nombre d'interfaces peut être supérieure à 1).

Dans la définition complète (figure 6.6), nous distinguons deux arguments : *negotiationPolicy* et *remoteNode*. Le premier permet de choisir une classe Java qui représente la politique de négociation utilisée par le composant. Cette classe est un paramètre de configuration de l'algorithme de négociation qu'implante le composant. Le deuxième argument sert à fixer le nœud qui héberge le composant. La classe qu'implante le composant (*fr.irisa.coordination.decision.implementations.NegotiatorImpl*) et la description des interfaces de contrôle (*primitive*) sont communs à tous les composants de ce type.

Des descriptions similaires avec un argument représentant une politique à choisir par l'expert sont définies pour les autres composants de type « `DecisionMaker` », « `Planner` » et « `Coordinator` ». En effet, le comportement d'un composant d'un de ces types est spécialisable par une politique.

Assemblage de composants. Concernant l'assemblage des composants, la structure peut varier d'un système d'adaptation à un autre. Plusieurs choix sont possibles pour le nombre de gestionnaires d'adaptation et de gestionnaires de contexte ainsi que pour la composition d'un gestionnaire d'adaptation. De plus, le nombre d'instances de certains types d'interface varie et la manière d'interconnecter les composants diffèrent d'un système d'adaptation à un autre. En conséquence, nous avons défini une API de programmation pour notre fabrique. L'expert en adaptation utilise cette API pour spécifier l'architecture souhaitée du système d'adaptation. Une description détaillée de notre fabrique est présentée dans le paragraphe 6.3.3.

Interfaces de contrôle. En Julia, la description des interfaces de contrôle des composants est réalisée dans un fichier de configuration particulier, `julia.cfg`. Ce fichier est structuré en différentes parties définissant notamment les interfaces des contrôleurs et leur implémentation. Pour un système d'adaptation, nous utilisons les interfaces et implantations définies par défaut dans ce fichier.

```
<?xml version="1.0" encoding="ISO-8859-1" ?>
<!DOCTYPE definition PUBLIC "-//objectweb.org//DTD Fractal ADL 2.0//EN"
"classpath://org/objectweb/fractal/adl/xml/standard.dtd">

<definition name="fr.irisa.coordination.decision.api.NegotiatorAbstract">

<interface signature="fr.irisa. coordination.decision.api.NegotiateItf" role="server" name="negotiate"/>

<interface signature="fr.irisa. Coordination.decision.api.ProposeItf" role="server" name="propose"
contingency="optional"/>

<interface signature="fr.irisa. coordination.decision.api.ProposeItf" role="client" name="proposer"
cardinality="collection" contingency="optional"/>

<interface signature="fr.irisa.adaptation.contextManagement.api.MonitorItf" role="client"
name="monitor" cardinality="collection" contingency="optional"/>

<interface signature="fr.irisa.adaptation.decision.api.NotifyItf" role="client" name="notifier"/>

</definition>
```

FIGURE 6.5 – Description des types d'interfaces fonctionnelles d'un composant de type négociateur

Implémentations réutilisables des gestionnaires d'adaptation

Pour faciliter la mise en place d'un système d'adaptation, nous avons mis en œuvre des traitements communs qui implémentent les fonctionnalités des gestionnaires d'adaptation.

Les classes qu'implantent les composants de type « `DecisionMaker` », « `Planner` », « `Negotiator` » et « `Coordinator` » sont développées pour être communes à tous les gestionnaires d'adaptation et à tous les systèmes d'adaptation. L'implémentation de chacun est spécialisée par une politique externe définie comme un attribut du composant. Au cours de leur exécution, la politique associée à chacun de ces composants Fractal peut être changée en modifiant la valeur de l'attribut correspondant.

Le composant de type « `Executor` » est réutilisable sans qu'il soit nécessaire de spécialiser son comportement. La classe qu'implante ce composant est capable d'appliquer les actions de tout plan qu'il reçoit.

La figure 6.7 montre la correspondance entre la description d'un composant de type « `Negotiator` » avec un attribut représentant sa politique et les traitements nécessaires pour configurer le composant. Ces traitements sont inclus dans la classe Java *NegotiatorImpl* qu'implante le composant et en particulier dans la méthode `setNegotiationPolicy`.

La description en Fractal ADL du composant précise que celui-ci a un attribut *negotiationPolicy*. Cette politique est décrite sous forme d'une classe Java et elle est nommée *NegotiationPolicyOfConsistencyAdaptation1* dans cet exemple. La fabrique utilise le contrôleur d'attributs de Julia pour affecter une valeur à l'attribut *negotiationPolicy* dans la classe Java qu'implante le composant. L'appel de la méthode de l'interface de contrôle d'attributs réalise une instanciation d'un objet de la classe représentant la politique et

```
<?xml version="1.0" encoding="ISO-8859-1" ?>
<!DOCTYPE definition PUBLIC "-//objectweb.org//DTD Fractal ADL 2.0//EN"
"classpath://org/objectweb/fractal/adl/xml/standard.dtd">

<definition name="fr.irisa.coordination.decision.api.Negotiator"
extends="fr.irisa.coordination.decision.api.NegotiatorAbstract" arguments="negotiationPolicy
,remoteNode">

<content class="fr.irisa. coordination.decision.implementations.NegotiatorImpl"/>

<attributes signature="fr.irisa.cooperation.decision.implementations.negotiation.NegotiatorAttributes">
        <attribute name="negotiationPolicy" value="${negotiationPolicy}"/>
</attributes>

<controller desc="primitive"/>

<virtual-node name="${remoteNode}"/>

</definition>
```

FIGURE 6.6 – Définition d'un composant de type négociateur en Fractal ADL

affecte cet objet à *negotiationPolicy*.

Concernant la forme des politiques, nous nous sommes inspirés du paradigme ECA (Événement - Condition - Action). Une politique est donc constituée d'un ensemble de règles, chacune constituée de trois éléments :

- une spécification d'un type d'événement qui active la règle lorsqu'il a lieu ;
- une condition qui est expression booléenne définissant la situation dans laquelle on applique l'action ;
- une action à effectuer si la condition est satisfaite.

La figure 6.8 montre un exemple de règle d'une politique de négociation. Cette politique spécialise le comportement d'un négociateur ayant la capacité de participer à des processus de négociation à propos le changement du protocole de cohérence de répliques.

Dans cette règle, un test est effectué pour vérifier qu'il s'agit de négociation d'adaptation du protocole de cohérence. Ensuite, la règle extrait, à partir du contrat d'adaptation proposé, la stratégie d'adaptation souhaitée. Cette stratégie précise le nom du protocole visé. La règle utilise l'interface client de type « `MonitorItf` » du négociateur reliée au gestionnaire de contexte *contextManager0*. Cette interface permet de mesurer la fréquence des écritures sur les sites dont le gestionnaire de propagation de mises à jour en question est responsable. Si cette fréquence est élevée et la stratégie d'adaptation vise à appliquer un protocole de cohérence optimiste, le négociateur accepte la proposition spécifiée dans le contrat de négociation. Un attribut du contrat d'adaptation est affecté par la valeur *accept*.

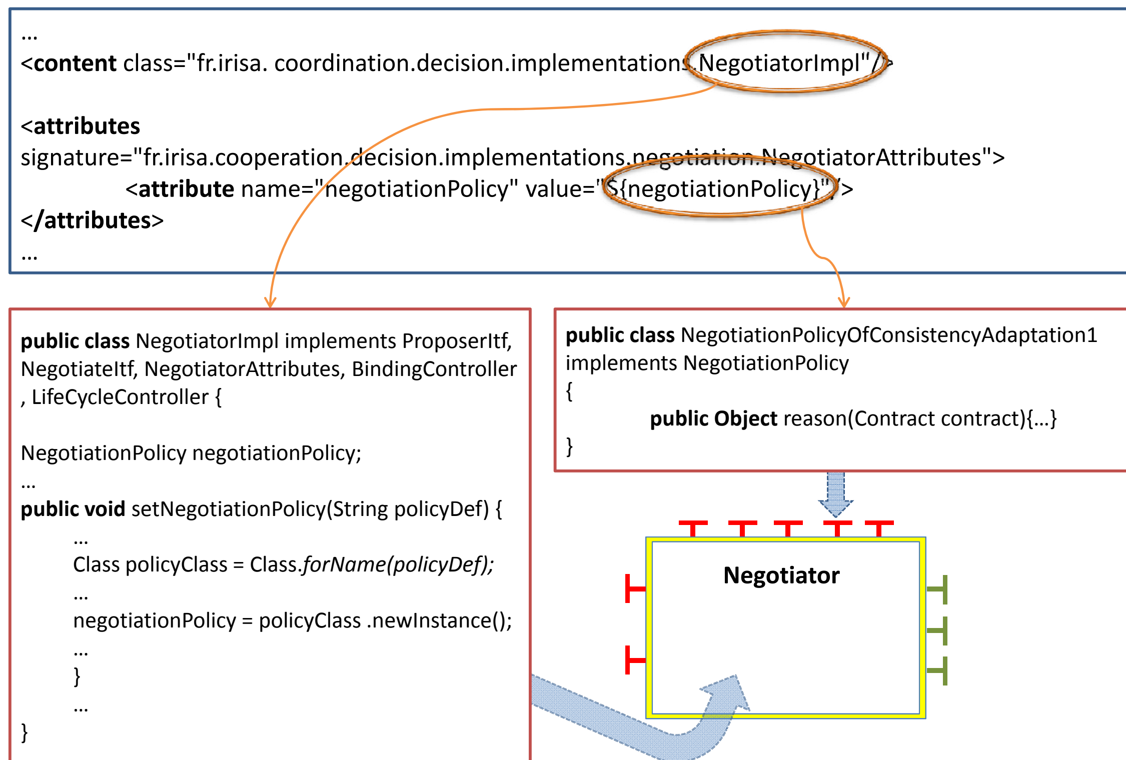

Figure 6.7 – Spécialisation du comportement d'un négociateur

Implémentations réutilisables des composants de gestion du contexte

Pour pouvoir adapter une application aux évolutions de son contexte d'exécution, il faut, bien évidemment, connaître ce contexte. Selon l'application à adapter, un choix concernant les algorithmes qu'implantent les composants de chaque gestionnaire de contexte chargés de cette fonctionnalité est nécessaire. En particulier, les composants de type « `ContextAcquisitionManager` », « `Interpreter` » et « `AggregationManager` » qui se chargent de l'acquisition des données brutes et du raisonnement sur ces données doivent être spécialisés selon les caractéristiques du contexte qui guident les processus d'adaptation. Nous avons réalisé une implémentation de ces composants de sorte à pouvoir gérer plusieurs types d'informations contextuelles nécessaires pour l'adaptation de systèmes de réplication. Ces implémentations peuvent être réutilisées et éventuellement étendues en fonction du système de réplication en question. Pour d'autres types d'application, il est nécessaire de définir d'autres implémentations spécifiques à l'application en question.

6.3.2 Implémentation du modèle architectural de réplication

L'implémentation de notre modèle architectural de système de réplication de données met en œuvre les types de composants du système et les contraintes de paramétrage et d'assemblage des composants, des traitements réutilisables pour la gestion de données répliquées et aussi des mécanismes pour rendre les composants observables et/ou adaptables.

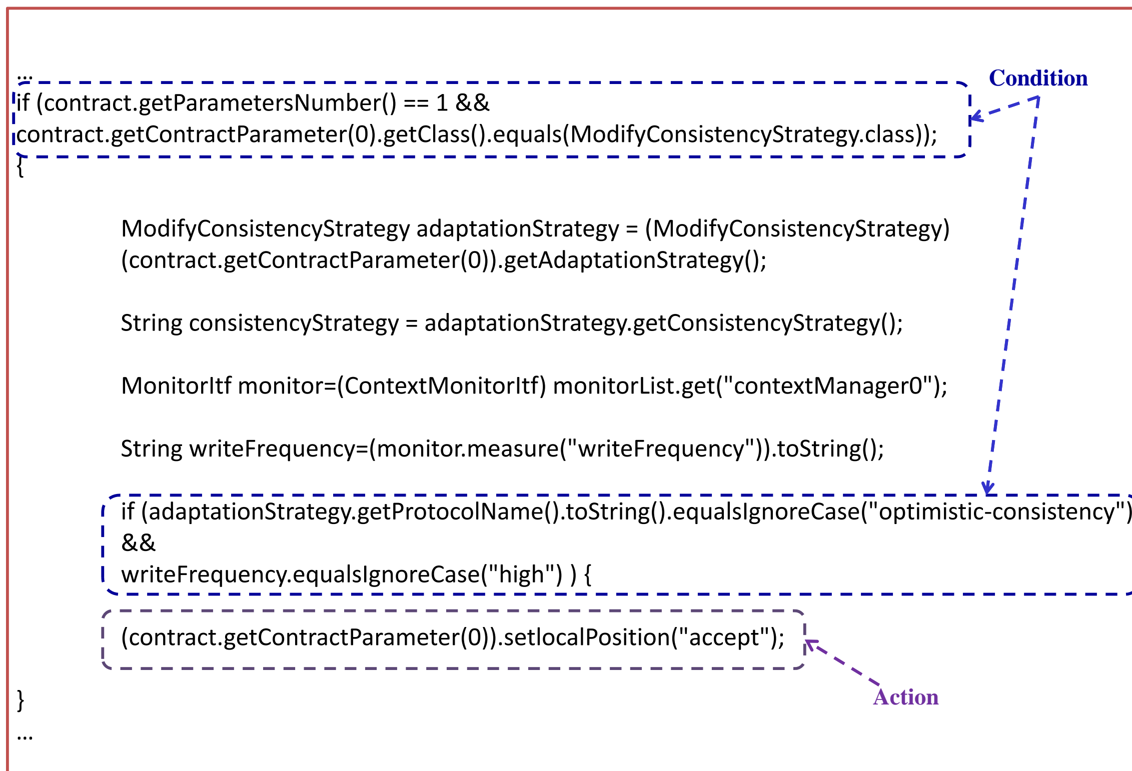

Figure 6.8 – Exemple d'une règle de négociation d'adaptation du protocole de cohérence

Description des types des composants du système de réplication

Nous avons décrit les types des composants de notre modèle architectural de réplication avec le langage Fractal ADL. Comme nous l'avons fait pour le modèle d'adaptation, un premier fichier ADL décrit le type des interfaces fonctionnelles du type de composant. Puis, un deuxième fichier ADL rajoute les autres éléments : un attribut précisant le nom de l'algorithme qui spécialise son comportement, l'implémentation du contenu, une référence à un descripteur des interfaces de contrôle du composant et le nom du nœud qui héberge le composant.

Points de variation. Parmi les éléments ajoutés dans le deuxième ADL, le nom de l'algorithme et le nom du nœud doivent être fixés par l'expert en réplication pour instancier le composant. Ces deux éléments représentent donc des points de variation du système de réplication. Ils servent à choisir le comportement et le placement du composant. Par ailleurs, et à la différence des systèmes d'adaptation, nous ne définissons pas dans l'ADL des éléments représentant les paramètres de chaque algorithme puisque, dans Fractal ADL, il est possible de spécifier seulement des paramètres de type primitif ce qui n'est pas suffisant pour certains algorithmes. De plus, les paramètres diffèrent d'un algorithme à un autre en terme de nombre, nom et type. Les paramètres sont donc définis dans la classe Java qui implémente l'algorithme sans les déclarer dans la description ADL du composant. Nous avons développé un contrôleur spécifique utilisé pour configurer ces paramètres. Il sera décrit dans le paragraphe suivant.

Assemblage des composants. La description de l'assemblage des composants du système de réplication se fait au moment de sa spécialisation. Il n'est pas possible d'écrire toutes les configurations possibles d'un système de réplication en Fractal ADL. Par exemple, le nombre de composants de chaque type que peut inclure un système est arbitraire. Comme pour le système d'adaptation, la description d'architecture est réalisée via l'interface fournie par la fabrique. Cette interface masque l'utilisation de Fractal ADL et de l'API Fractal pour instancier les composants sur les nœuds appropriés, les configurer et les interconnecter.

Interfaces de contrôle. Lors de la définition ADL des types de composants, la partie contrôle doit être spécifiée. Dans le système de réplication, elle doit inclure les nouveaux contrôleurs dédiés à la configuration et à l'adaptation des composants. Cette extension des capacités de contrôle des composants est décrite dans les deux paragraphes suivants.

Spécialisation d'un système de réplication

Pour spécialiser la structure d'un système de réplication, la fabrique utilise l'ensemble des interfaces de contrôle standards offrant des primitives de base pour configurer les composants. Ces primitives permettent au moment de déploiement du système de réplication d'instancier ces composants et de les assembler.

Pour spécialiser le comportement nous définissons un nouveau contrôleur que nous appelons le « contrôleur de configuration » (*ConfigurationController*). Ce contrôleur remplace le contrôleur d'attributs standard de Julia qui permet de manipuler seulement des attributs de type primitif comme les entiers et les chaînes de caractères. Ce nouveau contrôleur permet de définir les valeurs des paramètres de configuration de l'algorithme implémenté quels que soient leurs types (un tableau d'entiers, une instance d'une classe Java spécifique...). Il fournit pour ceci une interface qui sera implémentée de la même manière par tous les composants. Une partie de cette interface est présentée dans la figure 6.9. Elle définit des opérations pour manipuler deux sortes de paramètres : l'algorithme (*algorithm*) et ses paramètres (*filed*). La réflexivité en Java est utilisée pour manipuler les paramètres de l'algorithme indépendamment de leur type.

```
public interface ConfigureItf {

    void setAlgorithm (String algorithm);

    String getAlgorithm ();

    Object getField (String fieldName);

    void setField (String fieldName,
    Object fieldValue);
    …
}
```

FIGURE 6.9 – Interface fournie par un contrôleur de configuration

Observation et modification d'un système de réplication

Dans notre prototype, l'adaptation dynamique est réalisée par des objets séparés des objets Java qui implémentent les fonctionnalités des composants. Elle se base sur deux

nouveaux contrôleurs que peut inclure la membrane de chaque composant du système de réplication.

Le premier est le contrôleur d'observation (*ObservationController*) qui joue le rôle de capteur. Il est utilisé pour surveiller le composant applicatif auquel il appartient. Par exemple, il permet de compter le nombre d'appels d'une méthode implémentée par le composant de type « `AccessController` » demandant l'accès à une donnée.

Le deuxième est le contrôleur de modification (*ModificationController*) qui joue le rôle d'effecteur. Il exécute les opérations d'adaptation du composant applicatif auquel il appartient. Par exemple, il modifie la valeur du paramètre qui indique le nombre de répliques à placer sur le réseau.

La figure 6.10 représente la description de la partie de contrôle d'un composant de type « `PlacementManager` ». Elle inclut les interfaces de contrôle standards *component-itf*, *binding-controller-itf*, *super-controller-itf*, *lifecycle-controller-itf* et *name-controller-itf*. De plus, on y rajoute l'interface commune à tous les composants du système de réplication *configuration-controller-itf* et les deux interfaces de contrôle dédiées à l'adaptation *observation-controller-itf* et *modification-controller-itf*.

```
# control part of a placement manager
(placementManagerControllers
 (
  'interface-class-generator
  (
  ## control interfaces
   'component-itf
   'binding-controller-itf
   'super-controller-itf
   'lifecycle-controller-itf
   'name-controller-itf
   'configuration-controller-itf
   'observation-controller-itf
   'modification-controller-itf
  )
  (
  ## control objets
   'component-impl
   'container-binding-controller-impl
   'super-controller-impl
   'lifecycle-controller-impl
   'name-controller-impl
   'configuration-controller-impl
   'observation-controller-placement-impl
   'modification-controller-placement-impl
  )
...
```

Figure 6.10 – Description de la partie de contrôle d'un gestionnaire de placement

Contrôleurs d'observation des composants. Pour surveiller les composants, un gestionnaire de contexte utilise l'interface fournie par le contrôleur d'observation. Cette interface permet de mesurer les valeurs décrivant la configuration du composant et des caractéristiques de l'exécution de ses fonctionnalités.

L'implémentation de l'interface d'observation inclut un ensemble d'opérations pour surveiller le composant. Chaque opération consiste en un traitement nécessaire pour acquérir une information contextuelle brute. Par exemple, l'opération *getUserNumber* est fournie par le contrôleur d'observation appartenant à un composant de type « `PlacementManager` » ou « `RequestEngine` » pour mesurer le nombre d'utilisateurs connectés au composant.

```
public interface ObserveItf
{

    public Object applyObservation(String name,
    Object... params)

    throws IllegalObservationException,
    InvocationTargetException;

}
```

FIGURE 6.11 – Interface fournie par un contrôleur d'observation

L'interface offre une méthode *applyObservation* qui permet d'invoquer une de ces opérations. La méthode prend en argument le nom de l'opération et un nombre arbitraire d'arguments. Dans notre cas, la méthode accepte un nombre arbitraire de paramètres de type *Object* pour invoquer l'opération.

Le contrôleur d'observation utilise les capacités d'introspection du modèle Fractal permettant de connaître l'état de la configuration d'un système. Par exemple, l'opération *getFcSubComponent* du *ContentController* permet de connaître l'ensemble des sous composants d'un composant donné. Donc, le contrôleur d'observation peut utiliser ces primitives. Ainsi, il n'est pas nécessaire d'utiliser la description de l'architecture réalisée au moment de la spécialisation évitant ainsi de maintenir cette description à jour.

Le contrôleur d'observation peut aussi utiliser les opérations fournies par le contrôleur de configuration appartenant au même composant ce qui lui permet par exemple de mesurer la taille d'un tableau contenant les identifiants des utilisateurs afin de déterminer leur nombre.

Par ailleurs, le contrôleur d'observation peut intercepter les appels entrants et sortants d'un composant pour observer l'exécution des fonctionnalités du composant. Il peut ainsi observer les appels des méthodes (nombre d'appels, paramètres d'appel, résultat d'invocation...) et mesurer la performance du système comme le temps de réponse à une demande d'accès à une donnée ou le temps de placement de répliques.

Les opérations exposées par l'interface d'observation masquent l'utilisation des autres contrôleurs. Un composant de type « `ContextManager` » est lié à une seule interface fournie par le composant applicatif qu'il doit observer. Il est donc plus facile de spécialiser le comportement d'un gestionnaire de contexte pour acquérir les données contextuelles brutes grâce à l'utilisation d'opérations bien définies offertes par le contrôleur d'observation.

Contrôleurs de modification des composants. Un gestionnaire d'adaptation assure la modification d'un système de réplication en utilisant les interfaces fournies par les contrôleurs de modification. Chaque contrôleur de modification expose une interface pour invoquer un ensemble d'opérations pour l'adaptation du composant auquel il appartient. Ces

opérations définies dans le contrôleur représentent des actions d'adaptation et elles peuvent être composées pour constituer des actions d'adaptation plus complexes spécifiées au niveau des plans d'adaptation.

L'interface du contrôleur de modification définit une seule méthode *applyModification* comme le montre la figure 6.12. La méthode a un argument *name* représentant le nom de l'opération d'adaptation et un nombre d'arguments variable de type *Object*.

```
public interface ModifyItf
{
    public Object applyModification(String name,
    Object... params)

    throws IllegalModificationException,
    InvocationTargetException;

}
```

FIGURE 6.12 – Interface fournie par un contrôleur de modification

Pour réaliser ses opérations, le contrôleur de modification utilise des opérations primitives d'intercession des contrôleurs standards et les opérations du contrôleur de configuration. Le tableau suivant présente les opérations primitives décrites dans la spécification Fractal pour chaque contrôleur.

Ces opérations produisent des modifications simples d'une configuration comme l'ajout de liaison ou la modification de la valeur d'un attribut d'un composant. De plus, l'ensemble de ces opérations est restreint. Le contrôleur de modification permet alors de composer et d'étendre ces opérations pour augmenter les capacités de modification du système de réplication. Il expose des opérations de plus haut niveau afin de masquer l'utilisation des primitives des autres contrôleurs.

Pour un système de réplication, trois types de modifications doivent être possibles : comportementale (algorithmes et paramètres), structurelle et de distribution. Nous présentons ces adaptations par la suite.

Modification de comportement : La modification du comportement d'un composant implique le changement de l'algorithme utilisé ou des valeurs des paramètres de celui-ci. Le contrôleur de modification utilise le contrôleur de configuration pour assurer ces changements. Ce dernier offre les méthodes nécessaires pour fixer l'algorithme à utiliser et ses paramètres de configuration. Cependant des traitements supplémentaires peuvent être requis avant ou après la modification afin de ramener le système dans un état spécifique. Par exemple, avant de changer le protocole de cohérence des répliques, il peut s'avérer nécessaire de forcer la mise à jour de toutes les répliques (l'opération *updateAllReplica*) car les mises à jour non encore propagées ne peuvent pas être traitées par le nouveau protocole. C'est le contrôleur de modification qui offre les actions nécessaires pour assurer ces traitements.

Par ailleurs, le contrôleur de modification inclut aussi des opérations pour appliquer la technique de classification que nous avons présentée dans la section 5.5.2. Nous nous basons sur le patron de conception « Stratégie » qui permet de définir pour un composant une

Contrôleur	Opérations dans l'API Java
ContentController	• *void addFcSubComponent(Component subComponent)* : ajoute le sous-composant donné au composant • *void removeFcSubComponent(Component subComponent)* : retire le sous-composant donné du composant
BindingController	• *void bindFc(String clientItfName, Object serverItf)* : lie l'interface cliente dont le nom est donné en paramètre à une interface serveur passée en paramètre • *void unbindFc(String clientItfName)* : rompt le lien de l'interface cliente donnée
LifeCycleController	• *void startFc()* : démarre l'exécution des activités dans le composant • *void stopFc()* : suspend l'exécution des activités dans le composant
NameController	• *void setFcName(String name)* : change le nom du composant
AttributeController	• *void setX(Object value)* : change la valeur de l'attribut du composant
Genericfactory	• *Component newFcInstance(Type type, Object controllerDesc, Object contentDesc)* : crée un nouveau composant
Factory	• *Component newFcInstance()* : crée un nouveau composant

Figure 6.13 – Opérations primitives d'intercession dans Fractal

famille d'algorithmes encapsulés, interchangeables et associés à des contextes spécifiques (voir figure 5.6).

La classe *Selector* délègue l'exécution d'une fonctionnalité du composant à l'un des algorithmes. Elle implémente aussi l'interface du contrôleur de configuration afin de pouvoir modifier les algorithmes utilisés et leurs paramètres de configuration. Cette interface inclut les opérations primitives nécessaires pour appliquer la technique de classification. Ces opérations sont représentées dans la figure 6.14 et peuvent être utilisées par le contrôleur de modification. Premièrement, les méthodes *setGroup* et *getGroup* permettent d'attribuer une donnée ou un utilisateur identifié par l'argument *id* à un groupe d'identifiant *groupId* et de récupérer l'identifiant du groupe respectivement. Les méthodes *setAlgorithm* et *getAlgorithm* permettent de consulter et modifier l'algorithme adopté pour un groupe ayant l'identifiant *groupId*. Enfin, *getField* et *setField* permettent de manipuler les paramètres de configuration d'un algorithme adopté pour un groupe particulier.

Modification de la structure et de la distribution : Pour adapter la structure, les contrôleurs de modification des composants applicatifs offrent les opérations suivantes :

- une opération qui suspend l'activité du composant en utilisant le contrôleur de cycle vie,
- une opération qui déconnecte une liste d'interfaces client du composant en utilisant le contrôleur de liaisons,
- une opération qui connecte une liste d'interfaces client du composant en utilisant le contrôleur de liaisons,
- une opération qui démarre l'activité du composant en utilisant le contrôleur de cycle

```
public interface ConfigureItf {
    ...
    void  setGroup (String id, String groupId);

    String  getGroup (String id);

    void  setAlgorithm (String groupId,  String algorithm);

    String  getAlgorithm (String groupId);

    Object  getField (String groupId,  String fieldName);

    void  setField (String groupId, String fieldName, Object
    fieldValue);
}
```

FIGURE 6.14 – Opérations fournies par un contrôleur de configuration pour la classification

vie,
- une opération qui récupère l'état d'un composant en récupérant les valeurs des attributs de l'objet qu'il implante (nécessaire quand le composant va être remplacé),
- une opération qui injecte un état au composant en attribuant des valeurs aux attributs de l'objet qu'il implante (nécessaire quand le composant a été remplacé).

La migration d'un composant se fait en utilisant aussi les opérations décrites. Elle consiste à suspendre l'activité du composant à migrer et à supprimer ses liaisons. Après l'instanciation du composant sur le nouveau nœud, le système d'adaptation recrée de nouveau les liaisons et démarre l'activité du composant. Les opérations concernant la gestion de l'état sont obligatoires pour certains algorithmes de gestion de données répliquées.

Algorithmes interchangeables et réutilisables pour la gestion de données répliquées

Nous avons implémenté une bibliothèque d'algorithmes réutilisables pour tester notre prototype. Ces algorithmes spécialisent différemment le comportement du système de réplication en ce qui concerne les fonctionnalités de placement des répliques, la sélection de répliques et la gestion de cohérence des répliques. Pour la fonction de placement de répliques, nous avons implémenté trois algorithmes : le premier place les répliques aléatoirement, le deuxième les place sur tous les sites disponibles et le troisième choisit les sites avec une latence réseau minimale pour un ensemble d'utilisateurs.

Pour la sélection, un premier algorithme choisit une réplique aléatoirement et un deuxième choisit la réplique qui minimise la latence entre le client et le site sélectionné. Enfin, concernant la gestion de cohérence, nous avons implémenté un protocole pessimiste et un autre optimiste. Le premier correspond au protocole ROWA (voir paragraphe 3.4.1) et le deuxième est inspiré du système Coda. Il propage les mises à jour sous forme d'opérations et utilise des vecteurs de versions pour réconcilier les répliques. Chacun des composants de type « `UpdatePropagationManager` », « `AccessController` » et « `ConcurrencyController` » implémente un algorithme spécifique selon le protocole.

6.3.3 Fabrique pour construire des systèmes de réplication auto-adaptables

Rôle de la fabrique

Pour assurer une mise en œuvre simple d'un système de réplication et d'adaptation conforme à nos deux modèles architecturaux, nous avons développé une fabrique de systèmes de réplication auto-adaptables.

L'intérêt principal de la fabrique est qu'elle masque l'utilisation de l'API Fractal et du langage Fractal ADL. Ainsi, les experts en réplication et en adaptation utilisent des méthodes pour créer des composants et d'autres pour les connecter. Une fois la configuration du système de réplication auto-adaptable définie, la fabrique se charge de mettre en place le système d'une façon automatique.

La cohérence entre la description de l'architecture Fractal, des interfaces et les informations des contrôleurs est garantie par la fabrique. Par exemple, dans Fractal, le nom des interfaces doit être rigoureusement le même entre la description d'architecture et l'utilisation de ces noms dans le contrôleur de liaisons. De même, les signatures des interfaces Java ou des classes d'implémentation doivent être référencées dans le fichier de description d'architecture. Sans support, cette cohérence est difficile à maintenir et donc propice aux erreurs. L'utilisation de notre fabrique permet de supprimer le travail lié au maintien de la cohérence entre les éléments de spécifications et de se concentrer uniquement sur les choix concernant les points de variation.

Utilisation de la fabrique

La fabrique fournit une interface qui définit des méthodes pour instancier, configurer et interconnecter les différents composants pour construire un système de réplication auto-adaptable. L'expert invoque ces méthodes pour décrire la configuration souhaitée du système. La fabrique se charge de mettre en place le système en fonction de cette configuration.

Construction du système d'adaptation. La figure 6.15 montre un ensemble de méthodes de l'interface offerte par la fabrique. Ces méthodes représentent les opérations de base pour construire un système d'adaptation.

Les quatre premières méthodes permettent d'instancier et d'ajouter des gestionnaires d'adaptation au système d'adaptation. La première permet d'ajouter un gestionnaire d'adaptation qui n'inclut pas de composants assurant la coordination (négociateur et coordinateur). Les trois autres permettent au gestionnaire d'adaptation d'inclure seulement un négociateur ou seulement un coordinateur ou aussi un négociateur et un coordinateur en même temps.

La spécialisation du comportement nécessite de fixer les politiques à utiliser entre un ensemble de politiques alternatives sous forme de classes Java. Un expert peut écrire lui-même des politiques pour spécialiser le système d'adaptation. Il peut aussi réutiliser ou éditer des politiques existantes. L'expert spécifie dans les paramètres des méthodes les politiques à utiliser. Parmi les paramètres, il y a aussi le nom du gestionnaire d'adaptation et les identifiants des nœuds où seront placés les composants primitifs qu'inclut le gestionnaire d'adaptation.

Un autre ensemble constitué de trois méthodes permet de connecter deux gestionnaires d'adaptation ensemble afin de coordonner leurs activités. Les liaisons peuvent être entre deux décideurs, deux négociateurs ou deux coordinateurs inclus dans deux gestionnaires d'adaptation distincts. Deux arguments de chaque méthode spécifient les noms des deux

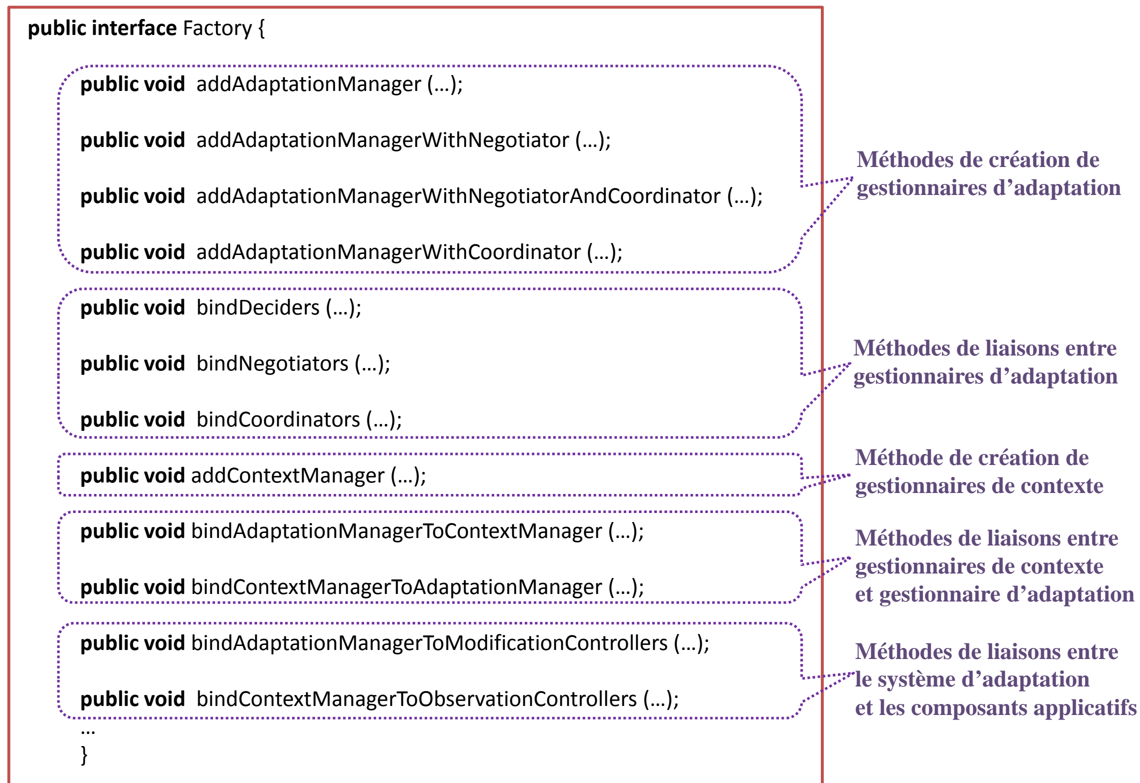

FIGURE 6.15 – Opérations de base pour construire un système d'adaptation

gestionnaires d'adaptation à connecter. Un troisième argument précise si la liaison est unidirectionnelle (une interface client du premier composant est liée à une interface serveur du second) ou bidirectionnelle (chacun des gestionnaires d'adaptation a une interface client liée à une interface serveur de l'autre).

Par ailleurs, une méthode permet d'ajouter un gestionnaire de contexte au système d'adaptation. Les arguments consistent en le nom du gestionnaire, les noms des classes qu'implantent ses sous-composants et le placement de ces derniers. L'expert peut utiliser les méthodes *bindAdaptationManagerToContextManager* et *bindContextManagerToAdaptationManager* pour connecter un gestionnaire de contexte et un gestionnaire d'adaptation en précisant les noms des deux composants.

Enfin, deux méthodes permettent d'interconnecter les composants du système d'adaptation aux interfaces de contrôle dédiées à l'adaptation qui sont fournies par les composants applicatifs. Les composants applicatifs doivent évidement être créés avant de réaliser ces connexions. La première méthode *bindAdaptationManagerToModificationControllers* permet de lier un exécuteur inclus dans un gestionnaire d'adaptation à un ensemble d'interfaces de contrôle de modification fournies par des composants applicatifs. La deuxième méthode *bindContextManagerToObservationControllers* relie un composant de type « `ContextAcquisitionManager` » à des interfaces de contrôle d'observation fournies par les composants applicatifs.

D'autres méthodes plus avancées ont également été définies. Chacune de ces méthodes permet de connecter plusieurs composants à la fois afin de faciliter la description de l'architecture logicielle. Par exemple, la méthode *bindNegotiatorToAllNegotiators* permet de connecter un négociateur particulier à tous les autres négociateurs inclus dans le système d'adaptation.

Le nombre d'instances des interfaces client d'un système d'adaptation liées aux interfaces de contrôle fournies par les composants applicatifs est variable. Il y a les interfaces client de type « `ModifyItf` » d'un composant de type « `Executor` » inclus dans un gestionnaire d'adaptation et les interfaces client de type « `ObserveItf` » d'un composant de type « `ContextAcquisitionManager` » que contient un gestionnaire de contexte. Le nombre des interfaces client d'un gestionnaire d'adaptation pour communiquer avec d'autres gestionnaires d'adaptation ou avec des gestionnaires de contexte varie aussi ainsi que le nombre d'interfaces client d'un gestionnaire de contexte liées aux gestionnaires d'adaptation. Pour toutes ces interfaces, la description ADL des types de composants précise un préfixe pour les noms des interfaces. Après, le nom complet est attribué par la fabrique de manière automatique et transparente à la création de la liaison au moyen d'un entier ajouté pour chaque type d'interface et incrémenté en fonction du nombre d'instances de l'interface. Par exemple, un négociateur (voir figure 6.5) lié à trois autres négociateurs aura trois interfaces client de type « `ProposeItf` » ayant les noms : *proposer0*, *proposer1* et *proposer2*. L'expert précise alors seulement les noms ou les types des composants qu'il souhaite interconnecter.

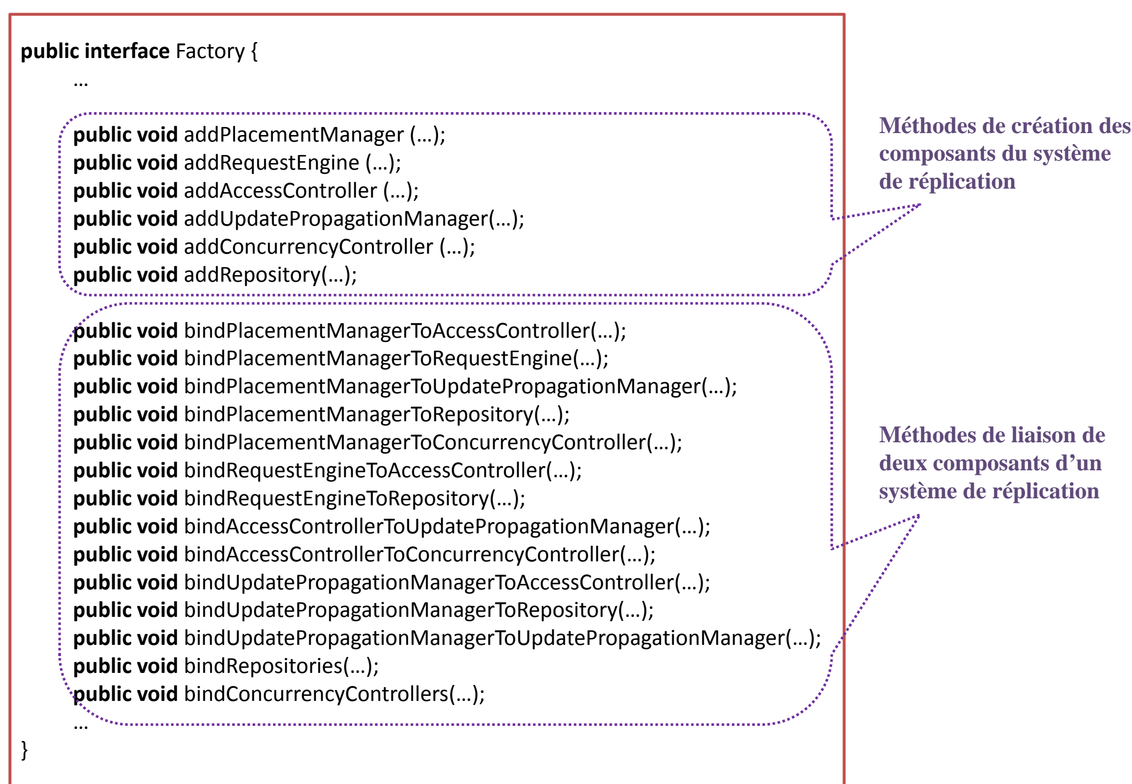

Figure 6.16 – Opérations de base pour construire un système de réplication

Construction du système de réplication. Comme pour le système d'adaptation, la fabrique expose des méthodes permettant de créer les composants d'un système de réplication et de les interconnecter. La figure 6.16 montre les méthodes fournies par l'interface de la fabrique pour cette création. Les six premières méthodes sont utilisées pour instancier des composants de chacun des 6 types de composant que peut inclure un système de réplication. Les paramètres de chaque méthode définissent le nom du composant, l'algorithme qu'il utilise et son placement. Toutes les autres méthodes de la figure représentent

les opérations de base permettant la création de liaisons autorisées entre les composants. Chaque méthode lie l'interface client d'un premier composant à une interface serveur du même type fournie par un autre composant. Les deux composants sont de même type ou de types différents selon les contraintes fixées dans notre modèle architectural.

L'interface offre d'autres méthodes non représentées dans la figure. Chacune crée des liaisons entre plus de deux composants. Par exemple, la méthode *bindPlacementManager-ToAllAccessController* permet de lier un composant de type « `PlacementManager` » à tous les composants de type « `AccessController` ».

À l'image du système d'adaptation, les noms des interfaces de type collection dans un système de réplication sont gérés par la fabrique de façon transparente.

Architecture de la fabrique

La fabrique est composée d'un ensemble de composants Fractal qui mettent en place le système de réplication auto-adaptable et d'un ensemble de fichiers et bibliothèques (voir figure 6.17). Le composant *SARS Factory* (*« Self-Adaptable Replication System »*) inclut un composant *SARS Builder* qui fournit l'interface introduite dans le paragraphe précédent. Ce composant utilise les opérations fournies par les différents composants prédéfinis dans Julia : *Bootstrap Component*, *Fractal ADL Factory* et *Fractal RMI Binder*. Les deux premiers sont utilisés pour instancier de nouveaux composants primitifs. Le dernier sert à mettre en place des liaisons réparties entre des composants Fractal distribués. Ces composants utilisent des fichiers et des bibliothèques que nous avons implémentés afin de mettre en œuvre les composants du système de réplication et du système d'adaptation en Fractal :

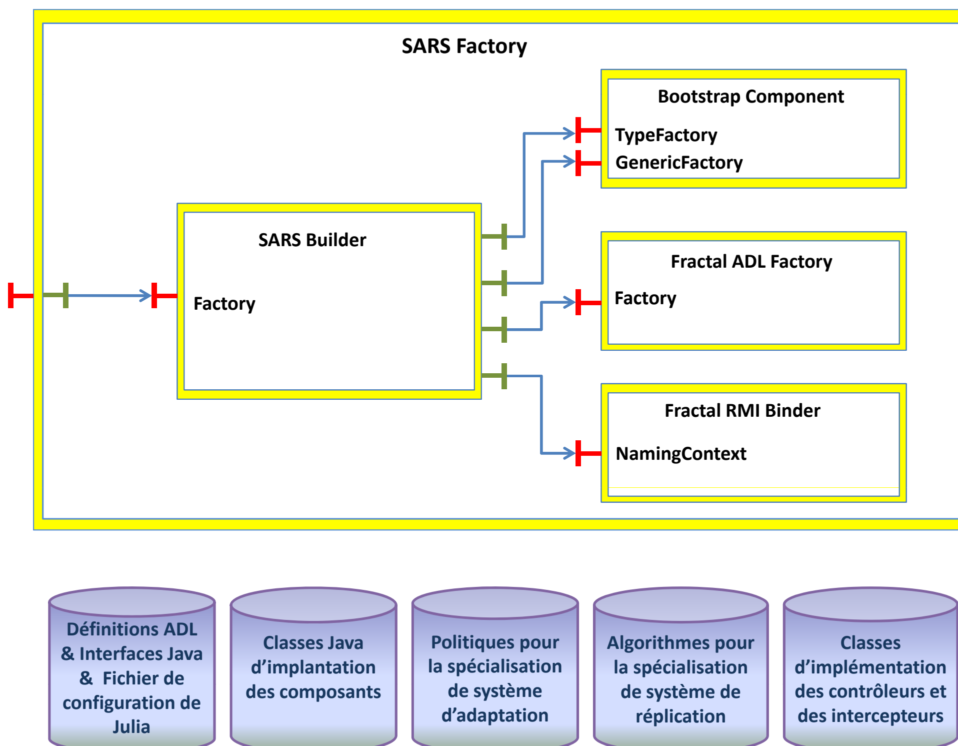

FIGURE 6.17 – Architecture de la fabrique

- Nous avons mis en œuvre les définitions ADL des types de composants et les interfaces Java que fournissent les composants du système de réplication et du système d'adaptation. De plus, le fichier de configuration de Julia inclut les descriptions des contrôleurs des différents types de composants. Ces fichiers définissent, ensemble, un squelette d'un système de réplication et celui d'un système d'adaptation.
- Nous avons développé les classes qu'implantent les composants. En particulier, les classes qu'implantent un gestionnaire d'adaptation et les composants d'un système de réplication sont réutilisables et on peut les spécialiser selon le besoin.
- Afin de spécialiser le système de réplication, nous avons réalisé une bibliothèque d'algorithmes pour la gestion de données répliquées. Nous avons défini aussi un ensemble de politiques pour spécialiser les gestionnaires d'adaptation. Ces politiques peuvent être éditées ou d'autres peuvent être définies pour assurer un comportement différent d'un système d'adaptation
- Des classes Java ont été développées pour implémenter les interfaces de contrôle dédiées à l'adaptation. Les opérations définies sont génériques et permettent plusieurs types d'observation et de modification des composants d'un système de réplication. Ces classes peuvent être étendues par de nouvelles opérations en cas de besoin.

6.4 Évaluation de performances

Cette partie présente des expériences réalisées pour évaluer notre travail. L'évaluation est composée de deux catégories d'expériences. En premier lieu, nous avons cherché à mettre en évidence l'intérêt de la gestion distribuée de l'adaptation dynamique et ses impacts. En deuxième lieu, nous avons évalué l'intérêt de l'adaptation pour améliorer la qualité des services d'un système de réplication.

6.4.1 Plateforme expérimentale

Nos expériences ont été réalisées sur une grappe de serveurs qui comprend 10 machines Dell PowerEdge 1855 équipées de processeurs Bipro Intel Xeon 3.40GHz et disposant de 4 gigaoctets de mémoire. Le réseau d'interconnexion est un réseau GigabitEthernet. Ces serveurs représentent les nœuds qui hébergent les composants d'un système de réplication et ceux d'un système d'adaptation. Une machine Dell Latitude D630 équipée d'un processeur 1.20GHz et ayant 2 gigaoctets de mémoire a été utilisée comme terminal utilisateur. Depuis cette machine, on envoie des requêtes de réplication de données et des requêtes d'accès aux répliques de ces données.

Les serveurs sont relativement performants par rapport à d'autres types de terminaux. Cependant, les résultats des expériences peuvent être étudiés indépendamment de la configuration matérielle. En effet, nos expérimentations se basent sur des comparaisons entre des expériences réalisées en parallèle avec différentes configurations de systèmes d'adaptation et dans les mêmes conditions. D'autres expériences comparent les performances d'un même système de réplication réparti sur un ensemble de serveurs avant et après son adaptation dynamique.

Du point de vue logiciel, chaque serveur exécute un système Linux. Notre prototype utilise l'implémentation de référence Julia 2.1.3 du modèle de composants Fractal. Nous utilisons la version 1.5.0 de la machine virtuelle Java de Sun Microsystems.

6.4.2 Comparaison des approches de gestion d'adaptation : centralisée et distribuée

Nos expériences ont visé à déterminer l'impact de la distribution de la gestion d'adaptation sur les performances du système d'adaptation, et plus précisément son temps de réponse. L'objectif est d'évaluer les coûts de coordination des activités des gestionnaires d'adaptation. À cette fin, nous avons examiné le scénario d'adaptation du protocole de cohérence pour un groupe de données médicales. Nous avons considéré la configuration du système de réplication décrite dans le paragraphe 5.6 du chapitre précédent. L'adaptation concerne donc plusieurs composants répartis : quatre composants de type « `UpdatePropagationManager` », huit composants de type « `AccessController` » et deux composants de type « `ConcurrencyController` ».

Le système d'adaptation est composé de quatre composants de type « `ContextManager` » et quatre composants de type « `AdaptationManager` ». Un gestionnaire de contexte et un gestionnaire d'adaptation, instanciés sur le même nœud, sont associés à chaque gestionnaire de propagation de mises à jour et aux contrôleurs d'accès qui y sont reliés. Il s'agit du même nœud qui héberge le gestionnaire de propagation de mises à jour. Chacun des deux contrôleurs de concurrence est géré par un couple (gestionnaire de contexte, gestionnaire d'adaptation) qui sont les plus proches de lui géographiquement.

Chaque gestionnaire d'adaptation inclut un composant de type « `Negotiator` » et un autre de type « `Coordinator` » pour avoir la capacité de coopérer avec les autres. En effet, dans le scénario d'adaptation considéré, les quatre négociateurs coopèrent ensemble pour choisir une stratégie d'adaptation du protocole de cohérence. Deux cycles de négociation produisent un accord sur la stratégie. Ensuite, chaque planificateur détermine un plan d'adaptation coordonné pour modifier les composants dans le champs d'action du gestionnaire d'adaptation en question. Enfin, les quatre exécuteurs et les coordinateurs qui y sont associés se chargent d'appliquer les quatre plans de manière coordonnée. Il s'agit des mêmes actions d'adaptation décrites dans la figure 4.16 mais deux des plans incluent de plus les actions nécessaires pour modifier le comportement du contrôleur de concurrence.

Le comportement de ce système d'adaptation distribué a été comparé à un système d'adaptation centralisé. Dans le cas centralisé, un seul gestionnaire de contexte et un seul gestionnaire d'adaptation sont instanciés et sont chargés d'adapter tous les composants assurant le protocole de cohérence. Le gestionnaire de contexte surveille tous ces composants. Le gestionnaire d'adaptation choisit la stratégie d'adaptation puis il détermine et applique un plan d'adaptation. L'exécuteur inclu dans ce gestionnaire est lié à tous les composants applicatifs pour pouvoir les modifier.

La première expérience que nous décrivons concerne le temps de prise de décision dans les deux configurations. Nous mesurons le temps entre la réception de l'événement déclenchant l'adaptation et le choix de la stratégie d'adaptation, y compris la consultation des informations contextuelles. Nous faisons varier le nombre de paramètres du contexte analysés afin de modifier la complexité du processus de décision.

La figure 6.18 montre le temps de prise de décision mesuré pour les deux configurations. Elle montre que le processus de prise décision distribuée donne un délai supplémentaire lorsque le nombre de paramètres analysés est petit (inférieur à 6). Cela est dû au temps passé dans les deux cycles de négociation. Toutefois, la distribution donne une moyenne de 35% d'amélioration des performances lorsque le nombre de paramètres est supérieur à 6. Cette amélioration est due au fait que chaque décideur est plus proche géographiquement du gestionnaire du contexte avec lequel il communique pour consulter les informations contextuelles. De plus, chaque décideur analyse un nombre réduit de paramètres du contexte et les analyses se réalisent en parallèle. Ainsi, nous constatons que le modèle de négocia-

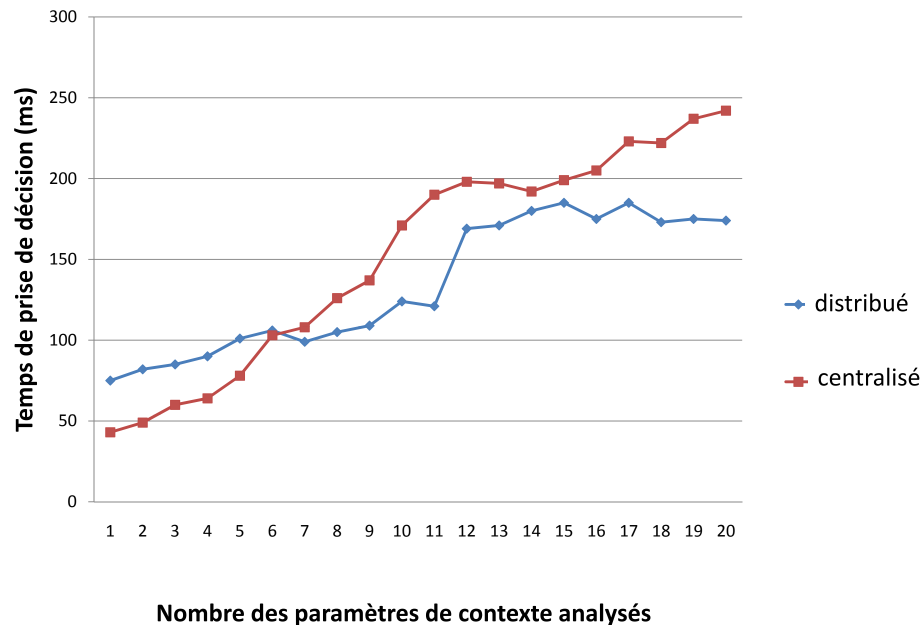

FIGURE 6.18 – Temps de prise de décision avec une configuration centralisée vs. distribuée

tion peut être approprié en particulier lorsque la consultation et l'analyse des informations contextuelles sont coûteuses.

Une deuxième expérience concerne le temps d'exécution pour les configurations distribuée et centralisée. Dans le cas centralisé, on mesure le temps entre le début et la fin de l'exécution du plan d'adaptation. Dans le cas distribué, on mesure le temps entre le début de l'exécution du premier plan et la fin de l'exécution de tous les plans.

Nous augmentons le nombre de répliques à mettre à jour afin d'augmenter le nombre d'appels des opérations d'adaptation primitives à effectuer. La figure 6.19 montre les résultats. Nous notons qu'une action de mise à jour des répliques d'une donnée nécessite un temps d'exécution important. Par ailleurs, le contrôle distribué rend l'exécution plus rapide malgré la surcharge de coordination. Ceci est dû au parallélisme dans l'exécution des plans et la diminution de la latence de communication entre chaque exécuteur et les effecteurs qu'il contrôle. Dans cette expérience, le gain moyen dans le temps avec gestion de l'exécution distribuée est de 358 millisecondes. La distribution de la gestion de l'exécution est appropriée pour des plans d'adaptation coûteux qui contiennent des actions nécessitant un temps d'exécution élevé et des interactions intenses entre l'exécuteur et les effecteurs (les contrôleurs de modification).

Nos expériences montrent des avantages significatifs dans le temps de réponse des processus d'adaptation distribués. Toutefois, les coûts de coordination peuvent contrebalancer les avantages de la gestion distribuée d'adaptation. L'expert d'adaptation doit tenir compte des coûts de coordination (en temps et en utilisation des ressources de calcul et réseau) et essayer de trouver un compromis entre les avantages de la distribution de la gestion d'adaptation et les coûts de coordination.

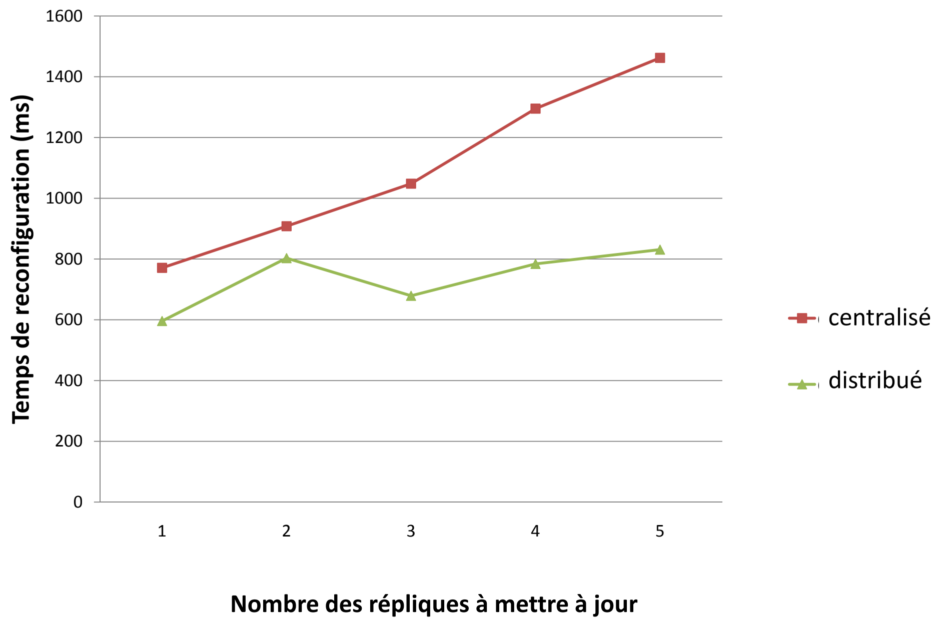

FIGURE 6.19 – Temps d'exécution avec une configuration centralisée vs. distribuée

6.4.3 Gains et coûts de l'adaptation de la gestion de données répliquées

Nous avons mené un ensemble d'expériences pour déterminer si l'adaptation dynamique peut améliorer les performances du système de réplication et satisfaire les exigences des utilisateurs de façon significative dans plusieurs situations.

Une des expériences a consisté à réaliser un processus d'adaptation du protocole de cohérence pour passer d'un protocole pessimiste à optimiste. Nous nous sommes intéressés à une donnée modifiable ayant quatre répliques. Dans notre expérience un client effectue une lecture suivie d'une écriture une vingtaine de fois de manière à avoir des demandes d'accès avant, au cours et après le processus d'adaptation.

L'objectif de cette expérience est de déterminer d'une part le coût d'adaptation en terme de temps de blocage des requêtes d'accès aux données à cause du processus d'adaptation et d'autre part le gain d'adaptation en terme de réduction de temps d'accès aux données.

Nous avons mesuré le temps moyen de réponse à une demande d'écriture de la donnée avant et après l'adaptation ainsi que le coût de cette adaptation.

Le temps moyen d'écriture était de 198 ms avant adaptation et 151 ms après adaptation. Ces mesures mettent en évidence le gain de l'application d'un nouveau protocole de cohérence. Le temps d'arrêt de service est égal à 288 ms. Ce coût d'adaptation peut être compensé s'il y a par la suite une longue série d'accès à la donnée. De manière générale, le gain total est égal à la somme des gains de toutes les requêtes d'accès à la donnée lancées pendant et après le changement de comportement du système de réplication. Ce gain dépend du nombre de requêtes d'accès et du moment de leur lancement :

- *Le nombre de requêtes.* Le gain total peut être important quand le nombre de requêtes est grand. Par exemple, si après la fin d'adaptation, les clients exécutent 10 requêtes d'écriture, le gain total est 10 * 47 ms.
- *Le moment de lancement de requêtes.* La requête peut être bloquée à cause de l'arrêt

de service pendant l'adaptation et/ou le traitement des requêtes précédentes qui ont étés bloquées à cause de l'adaptation. Dans le cas contraire, la requête est exécutée sans attente due à la réalisation de l'adaptation.

D'autres exemples d'adaptation pouvant être bénéfiques pour l'utilisateur concernent le changement de l'algorithme de placement permet de mieux rapprocher les répliques des utilisateurs. L'adaptation peut aussi faire qu'une réplique locale soit créée pour un utilisateur particulier suite à un changement de contexte. Par exemple, on utilise l'algorithme de placement *AnywherePlacement* et on le configure pour placer des répliques sur un sous ensemble spécifique des sites disponibles. L'adaptation du comportement d'un gestionnaire de placement n'est pas coûteuse lorsqu'elle ne nécessite pas de coordination d'adaptation. De plus, une telle adaptation peut avoir lieu suite à l'occurrence d'événements spécifiques qui précèdent la demande d'accès à la donnée en question. Ainsi, on évite le temps d'attente de la fin de l'adaptation. Similairement, l'adaptation de comportement d'un moteur de requêtes n'est pas coûteuse puisqu'elle n'exige pas de coordination.

Enfin, nous avons aussi noté que la modification du comportement par changement d'algorithme est beaucoup moins coûteuse que le remplacement de composant qui est généralement adopté dans les approches orientées composant. On réduit de façon importante le temps de réalisation de la modification. De plus, il est plus simple de définir les plans d'adaptation. C'est un point positif de notre approche lorsque l'application qu'on adapte est distribuée. Nous évitons le remplacement de plusieurs composants dans certains processus d'adaptation pour changer le comportement de l'application.

6.5 Conclusion

Dans ce chapitre, nous avons montré comment implémenter les propositions de modèles architecturaux formulées dans les chapitres 4 et 5 en utilisant le modèle de composants Fractal. Le prototype qui a été décrit est fonctionnel. L'étude expérimentale a montré que ce prototype a été utilisé avec succès pour construire des systèmes de réplication de données auto-adaptables.

L'utilisation de la fabrique montre que le développement de systèmes d'adaptation et de réplication peut être considérablement facilité. En effet, plusieurs tâches de développement sont automatisées au moins partiellement. Cependant, les capacités de la fabrique pourraient être étendues pour automatiser d'avantage le processus de spécialisation d'un système. Par exemple, il serait intéressent d'étudier des travaux sur le déploiement automatique et voir dans quelle mesure ils peuvent être intégrés dans notre fabrique pour rendre le choix de placement des composants automatique. Une autre perspective sera d'intégrer dans la fabrique des mécanismes d'analyse et de vérification des propriétés dynamiques qui concernent les comportements du système comme l'absence d'interblocage. Ainsi, il sera possible de déceler les erreurs dans la spécialisation du comportement. Plusieurs solutions existent dans ce domaine comme les méthodes de model-checking ou les techniques de test et de simulation.

Nos expérimentations montrent que la gestion distribuée de l'adaptation dynamique est profitable dans le contexte des applications distribuées. Cette distribution permet d'améliorer les performances lorsque l'environnement d'exécution est large et très fluctuant. Elle permet l'exécution des activités de chaque phase d'adaptation de manière distribuée et parallèle. Elle peut aussi réduire les coûts des communications comme celles faites pour collecter les données contextuelles brutes et pour contrôler les modifications des composants applicatifs. Par contre, la coordination peut augmenter dans certains cas les coûts d'adaptation en terme de consommation de ressources et de délai comme lorsque le nombre

de cycles de négociation est élevé. L'expert en adaptation doit alors trouver un compromis entre les avantages de la distribution de la gestion d'adaptation et les coûts de coordination. Par ailleurs, nous avons montré que les prises de décisions d'adaptation du système de réplication adéquates permettent d'optimiser le temps d'accès aux données et de garantir leur disponibilité en dépit du fait que le contexte d'exécution change. Les expériences réalisées ont montré que le coût d'adaptation en temps peut être amorti rapidement par le gain obtenu suite à l'adaptation surtout lorsque le nombre d'accès aux données est élevé.

Chapitre 7

Conclusion et perspectives

Pour clore la présentation de notre travail de thèse, nous présentons dans la section 7.1 une synthèse de nos contributions pour la construction d'applications distribuées auto-adaptables puis, dans la section 7.2 les travaux futurs qui compléteraient le travail réalisé.

7.1 Résumé des contributions

Dans cette thèse, nous nous sommes intéressés à étudier l'adaptabilité des applications distribuées. L'objectif principal que nous nous sommes fixé vise à proposer une solution pour permettre la gestion distribuée et coordonnée de l'adaptation dynamique et à faciliter le développement d'applications distribuées auto-adaptables.

Dans cette optique, nous avons proposé une approche générique pour adapter les applications distribuées à base de composants. Puis, nous avons décrit comment notre proposition d'adaptation se décline dans le cas particulier des systèmes de réplication de données.

Notre démarche peut se résumer en trois étapes. La première étape a consisté à définir les fonctionnalités d'un système d'adaptation distribué, son architecture logicielle et la variabilité possible de sa configuration pour le spécialiser selon l'application cible. La deuxième étape a été de définir de façon modulaire l'architecture logicielle d'un système de réplication et la variabilité de sa configuration au moment de sa spécialisation et au cours de son exécution. La troisième et dernière étape a été la proposition d'une fabrique pour faciliter la spécialisation et la mise en place d'un système de réplication et d'un système d'adaptation distribués.

Cette démarche nous a permis de réaliser les contributions suivantes :

- Nous avons proposé un modèle architectural de systèmes d'adaptation distribués. Il propose la structuration d'un tel système à l'aide de types de composants et il fixe des contraintes à respecter dans l'assemblage et la configuration des composants du système. Ce modèle assure la modularité du système d'adaptation et permet de spécialiser son comportement, sa structure ainsi que sa distribution selon l'application à adapter.

  Nous avons également défini un ensemble de mécanismes réutilisables et flexibles pour coordonner les activités de prise de décision d'adaptation d'une part, et de contrôle d'exécutions simultanées de plans d'adaptation d'autre part. Concernant la coordination d'une prise de décision, les mécanismes proposés permettent de distribuer le processus de décision. Ainsi, par exemple, nous avons proposé un modèle de négociation qui permet de réaliser des analyses distribuées et parallèles du contexte d'exécution et de résoudre les conflits éventuels entre les gestionnaires d'adaptation. Par ailleurs, les mécanismes de coordination de l'exécution de plans d'adaptation

permettent de prendre en compte l'avancement des différentes exécutions et les états des composants applicatifs impliqués dans l'adaptation.
- Nous avons défini un modèle architectural de systèmes de réplication pour permettre l'adaptation fine d'un tel système et un large spectre de types de modifications possibles. Ce modèle spécifie des types de composants et les interactions possibles entre eux pour gérer des données répliquées. Il définit des points de variation du comportement, de la structure et de la distribution du système permettant ainsi de définir plusieurs configurations alternatives. Notre modèle permet aussi d'inclure dans un système de réplication des interfaces de contrôle dédiées à l'adaptation. Ces interfaces servent à lui associer un système d'adaptation et elles définissent un ensemble d'opérations primitives pour observer et modifier les composants applicatifs.
- Nous avons proposé une fabrique pour la construction de systèmes de réplication auto-adaptables. Cette fabrique facilite la mise en place du système de réplication et du système d'adaptation associé. Pour cela, elle utilise nos modèles architecturaux et une description de l'architecture du système global souhaitée. La fabrique permet ainsi de masquer l'utilisation du modèle de composants choisi et des outils associés et elle automatise certaines tâches de développement.

Enfin, ces contributions ont été implémentées en utilisant le modèle de composants Fractal et expérimentées dans des scénarios de réplication de données médicales pour des services de prise en charge collaborative de malades chroniques.

7.2 Perspectives

Nos contributions constituent une réponse aux problématiques abordées dans cette thèse. Cependant plusieurs voies d'exploration sont ouvertes, soit pour corriger des limitations, soit pour élargir les perspectives d'utilisation de notre approche.

Validation de la configuration d'un système d'adaptation.
Comme nous l'avons signalé en conclusion du chapitre 6 (section 6.5), notre fabrique effectue des vérifications sur l'assemblage des composants de l'architecture envisagée pour un système d'adaptation. Par contre, aucune vérification n'est proposée au niveau de la spécialisation des comportements des composants. La vérification automatique de la cohérence des comportements choisis pour les différents composants serait intéressante.

Considérons par exemple les politiques de négociation pour la prise de décision. Si une politique indique à un négociateur de choisir une proposition donnée X, on pourrait vérifier que les politiques des participants à la négociation incluent les méthodes pour pouvoir réagir à cette proposition. Ces vérifications doivent être réalisées au moment de la spécialisation et aussi en cours d'exécution du système en cas de changement de politique.

Extension des techniques de décision et de coordination.
Les mécanismes de décision et de coordination proposés dans le chapitre 4 reposent sur le paradigme ECA (Événement - Condition - Action). C'est un paradigme couramment employé dans les systèmes d'adaptation. Cependant, d'autres techniques peuvent être envisageables comme l'apprentissage automatique ou les techniques de prise de décision distribuées inspirées des systèmes de vote ou des systèmes d'enchères. Par ailleurs, il serait intéressant de proposer plusieurs techniques de coordination de plans comme celles citées dans le chapitre 2 et d'en adapter le choix au contexte des systèmes d'adaptation au moment de leur spécialisation.

Fiabilité du système d'adaptation.
Une autre voie de recherche concerne la fiabilité du système d'adaptation. En effet, la probabilité d'une défaillance partielle du système d'adaptation ou du réseau d'interconnexion ne peut pas être négligée dans certains environnements distribués. L'objectif est donc d'assurer la continuité des services d'adaptation et d'éviter des incohérences dans l'application dues à ces pannes. Par exemple, une panne d'un gestionnaire d'adaptation peut bloquer la réponse à une proposition réalisée au cours d'un processus de négociation ou encore interrompre l'exécution d'un plan d'adaptation. La tolérance aux pannes consisterait à détecter les pannes et à les réparer de manière transparente. La gestion de panne peut consister à annuler le processus de prise décision ou d'exécution. Elle peut être aussi la continuation du processus en remplaçant le composant défaillant par un nouveau ou en attribuant la poursuite de la réalisation du processus à un composant homologue existant. Dans une seconde phase de recherche, il serait intéressant de pouvoir prévenir les pannes et empêcher ainsi l'occurrence de certaines d'entre elles avant leur apparition.

Automatisation du processus de spécialisation du système d'adaptation.
Notre fabrique et l'utilisation de politiques facilitent la construction d'un système d'adaptation. Néanmoins, le travail de spécialisation restant à l'expert en adaptation est non négligeable. La semi-automatisation de cette spécialisation s'avère donc une piste de recherche intéressante. La fabrique pourrait notamment être étendue pour choisir automatiquement la structure d'un système d'adaptation et sa distribution. Pour cela, elle pourrait appliquer des règles prenant en compte les caractéristiques de l'application à adapter et de l'environnement d'exécution comme la disponibilité des ressources, la distribution des composants applicatifs et les dépendances entre eux. Les travaux sur le déploiement automatique constituent un point de départ intéressant pour cette perspective.

Adaptabilité du système d'adaptation.
Comme nous l'avons vu dans le chapitre 2, il existe plusieurs techniques pour la prise de décision d'adaptation. Il serait envisageable de définir des mécanismes pour rendre un système d'adaptation capable d'utiliser de multiples techniques et de choisir dynamiquement l'une d'entre elles selon un ensemble de critères. L'algorithme de planification utilisé pourrait aussi être changé pendant l'exécution. Un travail dans cette direction est en cours au sein de notre équipe [FAEDGN+10]. Par ailleurs, la modification de la structure et de la distribution du système d'adaptation pendant l'exécution peut s'avérer nécessaire notamment dans le contexte de l'informatique mobile. Pour ce faire, il faudrait définir des mécanismes pour que le système d'adaptation ait de telles capacités. De plus, comme les composants applicatifs peuvent être déployés sur des terminaux mobiles, il serait intéressant de permettre la modification du champ d'action d'un gestionnaire de contexte et d'un gestionnaire d'adaptation. Cette extension serait possible grâce à des techniques de découverte et de connexion dynamique de composants comme celles proposées dans les travaux de [RRSC10].

Bibliographie

[AC03] Mehmet Aksit and Zièd Choukair. Dynamic, adaptive and reconfigurable systems overview and prospective vision. In *ICDCSW '03 : Proceedings of the 23rd International Conference on Distributed Computing Systems*, page 84, Washington, DC, USA, 2003.

[AFK+95] Larry Allen, Gary Fernandez, Kenneth Kane, David B. Leblang, Debra Minard, and John Posner. Clearcase multisite : Supporting geographically-distributed software development. pages 194–214, 1995.

[AL92] Paul Albitz and Cricket Liu. *DNS and BIND*. O'Reilley & Assoc., California, 1992.

[Ape88] Peter M. G. Apers. Data allocation in distributed database systems. *ACM Transactions on Database Systems*, 13(3) :263–304, September 1988.

[BAP07] Jeremy Buisson, Francoise Andre, and Jean-Louis Pazat. Supporting adaptable applications in grid resource management systems. In *GRID '07 : Proceedings of the 8th IEEE/ACM International Conference on Grid Computing*, pages 58–65, Washington, DC, USA, 2007. IEEE Computer Society.

[BCH+99] Kenneth P. Birman, Robert Constable, Mark Hayden, Jason Hickey, Christoph Kreitz, Robbert Van Renesse, Ohad Rodeh, and Werner Vogels. The horus and ensemble projects : Accomplishments and limitations. Technical report, Ithaca, NY, USA, 1999.

[BCL+06] Eric Bruneton, Thierry Coupaye, Matthieu Leclercq, Vivien Quéma, and Jean-Bernard Stefani. The FRACTAL component model and its support in java. *Softw, Pract. Exper*, 36(11-12) :1257–1284, 2006.

[BFR95] Yair Bartal, Amos Fiat, and Yuval Rabani. Competitive algorithms for distributed data management. *JCSS : Journal of Computer and System Sciences*, 51, 1995.

[BHG87] Philip A. Bernstein, Vassos Hadzilacos, and Nathan Goodman. *Concurrency Control and Recovery in Database Systems*. Addison-Wesley, 1987.

[BHS09] Patrick G. Bridges, Matti A. Hiltunen, and Richard D. Schlichting. Cholla : A framework for composing and coordinating adaptations in networked systems. *IEEE Trans. Comput.*, 58(11) :1456–1469, 2009.

[BK97] Yuri Breitbart and Henry F. Korth. Replication and consistency : Being lazy helps sometimes. In *Proceedings of the Sixteenth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*, pages 173–184, Tucson, Arizona, 12–15 May 1997.

[BKI06] Christoph Beierle and Gabriele Kern-Isberner. Methoden wissensbasierter systeme, 2006.

[BL04] Ronald Brachman and Hector Levesque. *Knowledge Representation and Reasoning*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2004.

[BP96] Chatschik Bisdikian and Baiju V. Patel. Cost-based program allocation for distributed multimedia-on-demand systems. *IEEE MultiMedia*, 3(3) :62–72, 1996.

[BR01] Ivan D. Baev and Rajmohan Rajaraman. Approximation algorithms for data placement in arbitrary networks. In *SODA*, pages 661–670, 2001.

[BS06] Ghalem Belalem and Yahya Slimani. A hybrid approach for consistency management in large scale systems. In *ICNS*, page 71. IEEE Computer Society, 2006.

[CEM03] Licia Capra, Wolfgang Emmerich, and Cecilia Mascolo. Carisma : Context-aware reflective middleware system for mobile applications. *IEEE Transactions on Software Engineering*, 29 :929–945, 2003.

[CFJ03] Harry Chen, Tim Finin, and Anupam Joshi. An intelligent broker for context-aware systems, September 06 2003.

[CHS01] Wen-Ke Chen, Matti A. Hiltunen, and Richard D. Schlichting. Constructing adaptive software in distributed systems. In *ICDCS '01 : Proceedings of the The 21st International Conference on Distributed Computing Systems*, page 635, Washington, DC, USA, 2001. IEEE Computer Society.

[CKS01] Israel Cidon, Shay Kutten, and Ran Soffer. Optimal allocation of electronic content. pages 1773–1780, 2001.

[CP01] Per Cederqvist and Roland Pesch. *Version Management with CVS*. Signum Support AB, 2001.

[CPP02] Stephen A. Cook, Jan K. Pachl, and Irwin S. Pressman. The optimal location of replicas in a network using a READ-ONE-WRITE-ALL policy. *DISTCOMP : Distributed Computing*, 15 :57–66, 2002.

[CRD09] Cyril Cassagnes, Philippe Roose, and Marc Dalmau. KALIMUCHO - software architecture for limited mobile devices. *ACM SIGBED Review*, Volume 6(Number 3) :1–6, 2009.

[CWC+07] Kevin C. Chang, Wei Wang, Lei Chen, Clarence A. Ellis, Ching-Hsien Hsu, Ah C. Tsoi, and Haixun Wang, editors. *Advances in Web and Network Technologies, and Information Management*, volume 4537 of *Lecture Notes in Computer Science*. Springer, 2007.

[DAS01] Anind K. Dey, Gregory D. Abowd, and Daniel Salber. A conceptual framework and a toolkit for supporting the rapid prototyping of context-aware applications. *Human-Computer Interaction*, 16(2/4) :97–166, 2001.

[DC01] Jim Dowling and Vinny Cahill. The K-Component architecture meta-model for self-adaptive software. In *Akinori Yonezawa and Satoshi Matsuoka, editors, Proceedings of 3rd International Conference on Metalevel Architectures and Separation of Crosscutting Concerns (Reflection'2001), LNCS 2192*, pages 81–88. Springer-Verlag, 2001.

[DC04] Jim Dowling and Vinny Cahill. Self-managed decentralised systems using K-Components and collaborative reinforcement learning. In *WOSS '04 : Proceedings of the 1st ACM SIGSOFT workshop on Self-managed systems*, pages 39–43, New York, NY, USA, 2004. ACM.

[DD08] Hoa H. Duong and Isabelle Demeure. Data sharing over mobile ad hoc networks. In *Proceedings of the 8th international conference on New technologies in distributed systems*, NOTERE '08, pages 44 :1–44 :6, New York, NY, USA, 2008. ACM.

[Die94] Daniel J. Dietterich. DEC data distributor : for data replication and data warehousing. In Richard T. Snodgrass and Marianne Winslett, editors, *Proceedings of the 1994 ACM SIGMOD International Conference on Management of Data*, page 468, 24–27 May 1994.

[DL87] Edmund H. Durfee and Victor R. Lesser. Using partial global plans to coordinate distributed problem solvers. Technical report, Amherst, MA, USA, 1987.

[DL03] Pierre-Charles David and Thomas Ledoux. Towards a framework for self-adaptive component-based applications. In *DAIS'03, volume 2893 of LNCS*, pages 1–14. Springer-Verlag, 2003.

[DPS+94] Alan Demers, Karin Petersen, Mike Spreitzer, Douglas Terry, Marvin Theimer, and Brent Welch. The Bayou architecture : Support for data sharing among mobile users. In *Workshop on Mobile Computing Systems and Applications*, Santa Cruz, CA, US, December 1994.

[Dra03] Stéphane Drapeau. *RS2.7 : un Canevas Adaptable de Services de Duplication*. PhD thesis, Institut National Polytechnique de Grenoble, 2003.

[EFDC02] Christos Efstratiou, Adrian Friday, Nigel Davies, and Keith Cheverst. Utilising the event calculus for policy driven adaptation on mobile systems. In *POLICY*, pages 13–24. IEEE Computer Society, 2002.

[FAEDGN+10] Francoise André, Erwan Daubert, Grégory Nain, Brice Morin, and Olivier Barais. F4plan : An approach to build efficient adaptation plans. In *Proceedings of 7th International ICST Conference on Mobile and Ubiquitous Systems (MobiQuitous)*, December 2010.

[FBZA98] Zongming Fei, Samrat Bhattacharjee, Ellen W. Zegura, and Mostafa H. Ammar. A novel server selection technique for improving the response time of a replicated service. In *INFOCOM*, pages 783–791, 1998.

[FHS+06] Jacqueline Floch, Svein Hallsteinsen, Erlend Stav, Frank Eliassen, Ketil Lund, and Eli Gjorven. Using architecture models for runtime adaptability. *IEEE Softw.*, 23(2) :62–70, 2006.

[FJJ+01] Paul Francis, Sugih Jamin, Cheng Jin, Yixin Jin, Danny Raz, Yuval Shavitt, and Lixia Zhang. Idmaps : a global internet host distance estimation service. volume 9, pages 525–540, Piscataway, NJ, USA, October 2001. IEEE Press.

[FKNT99] Ian Foster, Carl Kesselman, Jeffrey M. Nick, and Steven Tuecke, editors. *The Grid : Blueprint for a New Computing Infrastructure*. Morgan-Kaufmann, 1999.

[FLG06] Ramy Farha and Alberto Leon-Garcia. Blueprint for an autonomic service architecture. *Autonomic and Autonomous Systems, International Conference on*, 0 :16, 2006.

[FM03] Corina Ferdean and Mesaac Makpangou. A scalable replica selection strategy based on flexible contracts. In *Proceedings of the The Third IEEE Workshop on Internet Applications*, WIAPP '03, pages 95 – 99, 2003.

[FXL07] Wei Fu, Nong Xiao, and Xicheng Lu. QoS-guaranteed ring replication management with strong consistency. In Kevin Chen-Chuan Chang, Wei Wang, Lei Chen 0002, Clarence A. Ellis, Ching-Hsien Hsu, Ah Chung Tsoi, and Haixun Wang, editors, *APWeb/WAIM Workshops*, volume 4537 of *Lecture Notes in Computer Science*, pages 37–49. Springer, 2007.

[FXL08] Wei Fu, Nong Xiao, and Xicheng Lu. A quantitative survey on QoS-aware replica placement. In *Proceedings of the 2008 Seventh International Conference on Grid and Cooperative Computing*, pages 281–286, Washington, DC, USA, 2008. IEEE Computer Society.

[GCH+04] David Garlan, Shang-Wen Cheng, An-Cheng Huang, Bradley Schmerl, and Peter Steenkiste. Rainbow : Architecture-based self-adaptation with reusable infrastructure. *Computer*, 37(10) :46–54, 2004.

[Geo83] Michael Georgeff. Communication and interaction in multi-agent planning. In *Proceedings of AAAI-83*, pages 125–129, August 1983.

[GHJV95] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns : Elements of Reusable Object-Oriented Software*. Addison Wesley, 1995.

[Gif79] David K. Gifford. Weighted voting for replicated data. *Seventh SOSP*, OSR 13(5) :150–162, December, 1979.

[GM95] V. Guedes and F. Moura. Replica control in MIo-NFS. In *ECOOP'95 Workshop on Mobility and Replication*, Aarhus, Denmark, August 1995.

[GPZ04] Tao Gu, Hung K. Pung, and Da Q. Zhang. A middleware for building context-aware mobile services. In *Vehicular Technology Conference, 2004. VTC 2004-Spring. 2004 IEEE 59th*, volume 5, pages 2656–2660, May 2004.

[Gro10] Object Management Group. *OMG Unified Modeling Language 2.3*. OMG, http ://www.omg.com/uml/, 2010.

[GS95a] James D. Guyton and Michael F. Schwartz. Locating nearby copies of replicated internet servers. In *SIGCOMM*, pages 288–298, 1995.

[GS95b] James Gwertzman and Margo Seltzer. The case for geographical push-caching. In *Proceedings of the HotOS '95 Workshop*, May 1995.

[GS02] David Garlan and Bradley Schmerl. Model-based adaptation for self-healing systems. In *WOSS '02 : Proceedings of the first workshop on Self-healing systems*, pages 27–32, New York, NY, USA, 2002. ACM.

[Hei00] George T. Heineman. A model for designing adaptable software components. *SIGSOFT Softw. Eng. Notes*, 25(1) :55–56, 2000.

[Her86] M. Herlihy. A quorum consensus replication method for abstract data types. *ACM Transactions on Computer Systems*, 4(1) :32–53, 1986.

[JCD02] Imed Jarras and Brahim Chaib-Draa. Aperçu sur les systèmes multi-agents. 2002.

[JGN06] Won Jong Jeon, Indranil Gupta, and Klara Nahrstedt. QoS-aware object replication in overlay networks. In *GLOBECOM*. IEEE, 2006.

[JJK+01] Sugih Jamin, Cheng Jin, Anthony R. Kurc, Danny Raz, and Yuval Shavitt. Constrained mirror placement on the internet. In *IEEE Journal on Selected Areas in Communications*, pages 31–40, 2001.

[KBC02] A. Ketfi, N. Belkhatir, and P. Y. Cunin. Adaptation dynamique, concepts et expérimentations. In *15th International Conference on Software and Systems Engineering and their Applications*, 2002.

[KBH+88] Leonard J. Kawell, Steven Beckhardt, Timothy Halvorsen, Raymond Ozzie, and Irene Greif. Replicated document management in a group communication system. In *Proceedings of ACM CSCW'88 Conference on Computer-Supported Cooperative Work*, page 395. ACM, 1988.

[KBM94] Eric D. Katz, Michelle Butler, and Robert McGrath. A scalable HTTP server : The NCSA prototype. *Computer Networks and ISDN Systems*, 27(2) :155–164, 1994.

[KC03] John Keeney and Vinny Cahill. Chisel : A policy-driven, context-aware, dynamic adaptation framework. In *POLICY '03 : Proceedings of the 4th IEEE International Workshop on Policies for Distributed Systems and Networks*, page 3, Washington, DC, USA, 2003. IEEE Computer Society.

[KDW01] Konstantinos Kalpakis, Koustuv Dasgupta, and Ouri Wolfson. Optimal placement of replicas in trees with read, write, and storage costs. *IEEE Transactions on Parallel and Distributed Systems*, PDS-12(6) :628–637, June 2001.

[Kel99] Peter J. Keleher. Decentralized replicated-object protocols. In *Proceedings of the* 18th *Symposium on Principles of Distributed Computing (PODC)*, pages 143–151, Atlanta, Georgia, USA, 1999.

[KKM02] Magnus Karlsson, Christos Karamanolis, and Mallik Mahalingam. A framework for evaluating replica placement algorithms. Technical Report HPL-2002-219, Hewlett Packard Laboratories, August 16 2002.

[KKST98] Anne-Marie Kermarrec, Ihor Kuz, Maarten V. Steen, and Andrew S. Tanenbaum. A framework for consistent, replicated web objects. In *ICDCS*, pages 276–291, 1998.

[KMK+03] Panu Korpipaa, Jani Mantyjarvi, Juha Kela, Heikki Keranen, and Esko J. Malm. Managing context information in mobile devices. *Pervasive Computing, IEEE*, 2(3) :42–51, 2003.

[KRR02] Jussi Kangasharju, James W. Roberts, and Keith W. Ross. Object replication strategies in content distribution networks. *Computer Communications*, 25(4) :376–383, 2002.

[KRSD01] Anne-Marie Kermarrec, Antony Rowstron, Marc Shapiro, and Peter Druschel. The IceCube approach to the reconciliation of divergent replicas. In *Principles of Distributed Computing (PODC)*, 2001.

[KS92] James J. Kistler and Mahadey Satyanarayanan. Disconnected operation in the coda file system. *ACM Trans. on Computer Sys.*, 10(1) :3, February 1992.

[KS95] Puneet Kumar and Mahadev Satyanarayanan. Flexible and safe resolution of file conflicts. In *USENIX Winter*, pages 95–106, 1995.

[LA04] Thanasis Loukopoulos and Ishfaq Ahmad. Static and adaptive distributed data replication using genetic algorithms. *Journal of Parallel and Distributed Computing*, 64(11) :1270–1285, November 2004.

[Lad00] Robert Laddaga. Active software. In *Proceedings of the first international workshop on Self-adaptive software*, IWSAS' 2000, pages 11–26, Secaucus, NJ, USA, 2000. Springer-Verlag New York, Inc.

[LH05] Oussama Layaida and Daniel Hagimont. Designing self-adaptive multimedia applications through hierarchical reconfiguration. In Lea Kutvonen and Nancy Alonistioti, editors, *DAIS*, volume 3543 of *Lecture Notes in Computer Science*, pages 95–107. Springer, 2005.

[Liu04] Hua Liu. A component-based programming model for autonomic applications. In *ICAC '04 : Proceedings of the First International Conference on Autonomic Computing*, pages 10–17, Washington, DC, USA, 2004. IEEE Computer Society.

[LLMY05] Alexei Lapouchnian, Sotirios Liaskos, John Mylopoulos, and Yijun Yu. Towards requirements-driven autonomic systems design. In *DEAS '05 : Proceedings of the 2005 workshop on Design and evolution of autonomic application software*, pages 1–7, New York, NY, USA, 2005. ACM.

[LR00] Meir M. Lehman and Juan F. Ramil. Towards a theory of software evolution - and its practical impact (working paper). In *Invited Talk, Proceedings Intl. Symposium on Principles of Softw. Evolution, ISPSE 2000, 1-2 Nov*, pages 2–11. Press, 2000.

[LSBS06] Nikolaos Laoutaris, Georgios Smaragdakis, Azer Bestavros, and Ioannis Stavrakakis. Mistreatment in distributed caching groups : Causes and implications. IEEE, 2006.

[LYO03] Juan C. Leonardo, Takaichi Yoshida, and Kentaro Oda. An adaptable replication scheme for reliable distributed object-oriented computing. In *AINA*, pages 602–606. IEEE Computer Society, 2003.

[Mar92] Frank V. Martial. *Coordinating Plans of Autonomous Agents.* Springer-Verlag New York, Inc., Secaucus, NJ, USA, 1992.

[Mar03] Vania Marangozova. *Duplication et cohérence configurables dans les applications réparties à base de composants.* PhD thesis, Université Joseph Fourier - Grenoble 1, 2003.

[MBS03] Olivier Marin, Marin Bertier, and Pierre Sens. DARX - A framework for the fault-tolerant support of agent software. pages 406–418. IEEE Computer Society, 2003.

[MG99] Kaveh Moazami-Goudarzi. *Consistency preserving dynamic reconfiguration of distributed systems.* PhD thesis, Imperial College, London, 1999.

[MG05] Arun Mukhija and Martin Glinz. Runtime adaptation of applications through dynamic recomposition of components. In Michael Beigl and Paul Lukowicz, editors, *ARCS*, volume 3432 of *Lecture Notes in Computer Science*, pages 124–138. Springer, 2005.

[MIC98] SUN MICROSYSTEMS. *Sun directory services 3.1 administration guide*, 1998.

[MMB03] Alberto Montresor, Hein Meling, and Ozalp Babaoglu. Toward self-organizing, self-repairing and resilient distributed systems. In Andre Schiper, Alexander A. Shvartsman, Hakim Weatherspoon, and Ben Y. Zhao, editors, *Future Directions in Distributed Computing, Research and Position Papers*, volume 2584 of *Lecture Notes in Computer Science (LNCS)*, pages 119–126. Springer-Verlag, New York, 2003.

[MMVB00] Carlo Marchetti, Massimo Mecella, Antonino Virgillito, and Roberto Baldoni. An interoperable replication logic for CORBA systems. In *DOA*, pages 7–16, 2000.

[MSKC04] Philip K. McKinley, Seyed Masoud Sadjadi, Eric P. Kasten, and Betty H. C. Cheng. Composing adaptive software. *Computer*, 37(7) :56–64, 2004.

[NYGS06] Suman Nath, Haifeng Yu, Phillip B. Gibbons, and Srinivasan Seshan. Subtleties in tolerating correlated failures in wide-area storage systems. In *NSDI*. USENIX, 2006.

[OGT+99] Peyman Oreizy, Michael M. Gorlick, Richard N. Taylor, Dennis Heimbigner, Gregory Johnson, Nenad Medvidovic, Alex Quilici, David S. Rosenblum, and Alexander L. Wolf. An architecture-based approach to self-adaptive software. *IEEE Intelligent Systems*, 14(3) :54–62, 1999.

[OMT08] Peyman Oreizy, Nenad Medvidovic, and Richard N. Taylor. Runtime software adaptation : framework, approaches, and styles. In Wilhelm Schäfer, Matthew B. Dwyer, and Volker Gruhn, editors, *30th International Conference on Software Engineering (ICSE 2008), Leipzig, Germany, May 10-18, 2008, Companion Volume*, pages 899–910. ACM, 2008.

[Ora96] *Oracle7 Server Distributed Systems Manual*, 1996.

[OTY01] Kentaro Oda, Shin'ichi Tazuneki, and Takaichi Yoshida. The flying object for an open distributed environment. In *ICOIN*, pages 87–92, 2001.

[PKKY03] Sang-Min Park, Jai-Hoon Kim, Young-Bae Ko, and Won-Sik Yoon. Dynamic data grid replication strategy based on internet hierarchy. In Minglu Li, Xian-He Sun, Qianni Deng, and Jun Ni, editors, *GCC (2)*, volume 3033 of *Lecture Notes in Computer Science*, pages 838–846. Springer, 2003.

[PMND07] Guilhem Paroux, Ludovic Martin, Julien Nowalczyk, and Isabelle Demeure. Transhumance : A power-sensitive middleware for data sharing on mobile ad hoc networks. In *Proceedings of the seventh international Workshop on Applications and Services in Wireless Networks (ASWN)*, 2007.

[PSM03] Nuno M. Preguiça, Marc Shapiro, and Caroline Matheson. Semantics-based reconciliation for collaborative and mobile environments. In Robert Meersman, Zahir Tari, and Douglas C. Schmidt, editors, *On The Move to Meaningful Internet Systems 2003 : CoopIS, DOA, and ODBASE - OTM Confederated International Conferences, CoopIS, DOA, and ODBASE 2003, Catania, Sicily, Italy, November 3-7, 2003*, volume 2888 of *Lecture Notes in Computer Science*, pages 38–55. Springer, 2003.

[PST+97] Karin Petersen, Mike Spreitzer, Douglas B. Terry, Marvin Theimer, and Alan J. Demers. Flexible update propagation for weakly consistent replication. In *SOSP*, pages 288–301, 1997.

[PYS98] Daniel Paul, Sudhakar Yalamanchili, and Karsten Schwan. Decision models for adaptive resource management in multiprocessor systems, November 13 1998.

[Rat98] David Howard Ratner. *Roam : a scalable replication system for mobile and distributed computing*. PhD thesis, University of California, Los Angeles, 1998.

[RCS08] Romain Rouvoy, Denis Conan, and Lionel Seinturier. Software architecture patterns for a context-processing middleware framework. *IEEE Distributed Systems Online*, 9(6), 2008.

[RGE02] Pavlin Radoslavov, Ramesh Govindan, and Deborah Estrin. Topology-informed internet replica placement. *Computer Communications*, 25(4) :384–392, 2002.

[RHR+94] Peter Reiher, John Heidemann, David Ratner, Greg Skinner, and Gerald Popek. Resolving file conflicts in the Ficus file system. In *Proceedings of the USENIX Summer 1994 Technical Conference*, pages 183–195, Boston, Massachusetts, USA, June 1994.

[RLR06] Liliana Rosa, Antonia Lopes, and Luis Rodrigues. Policy-driven adaptation of protocol stacks. In *ICAS '06 : Proceedings of the International Conference on Autonomic and Autonomous Systems*, page 5, Washington, DC, USA, 2006. IEEE Computer Society.

[RRSC10] Daniel Romero, Romain Rouvoy, Lionel Seinturier, and Pierre Carton. Service discovery in ubiquitous feedback control loops. In *DAIS*, pages 112–125, 2010.

[SBSV98] Mehmet Sayal, Yuri Breitbart, Peter Scheuermann, and Radek Vingralek. Selection algorithms for replicated web servers. *SIGMETRICS Performance Evaluation Review*, 26(3) :44–50, 1998.

[SC01] Nary Subramanian and Lawrence Chung. Software architecture adaptability : an nfr approach. In *IWPSE '01 : Proceedings of the 4th International Workshop on Principles of Software Evolution*, pages 52–61, New York, NY, USA, 2001. ACM.

[SE98] Chengzheng Sun and Clarence A. Ellis. Operational transformation in real-time group editors : Issues, algorithms, and achievements. pages 59–68, 1998.

[Smi98] Chris Smith. Fault tolerant corba. Technical report, 1998.

[Sow00] John F. Sowa. *Knowledge representation : logical, philosophical and computational foundations.* Brooks/Cole Publishing Co., Pacific Grove, CA, USA, 2000.

[SS05] Yasushi Saito and Marc Shapiro. Optimistic replication. *ACM Comput. Surv.*, 37 :42–81, March 2005.

[SSK97] Srinivasan Seshan, Mark Stemm, and Randy H. Katz. SPAND : Shared passive network performance discovery. In *Proceedings of the USENIX Symposium on Internet Technologies and Systems (ITS-97)*, pages 135–146, Berkeley, December 8–11 1997. USENIX Association.

[TM05] Ceryen Tan and Kevin Mills. Performance characterization of decentralized algorithms for replica selection in distributed object systems. In *WOSP*, pages 257–262. ACM, 2005.

[TS09] Eli Tilevich and Yannis Smaragdakis. J-orchestra : Enhancing java programs with distribution capabilities. *ACM Trans. Softw. Eng. Methodol.*, 19(1) :1–40, 2009.

[TX04] Xueyan Tang and Jianliang Xu. On replica placement for QoS-aware content distribution. In *INFOCOM*, 2004.

[UC04] Oren Unger and Israel Cidon. Optimal content location in multicast based overlay networks with content updates. *World Wide Web*, 7(3) :315–336, 2004.

[VBSS99] Radek Vingralek, Yuri Breitbart, Mehmet Sayal, and Peter Scheuermann. Web++ : A system for fast and reliable web service. In *Proceedings of 1999 USENIX Annual Technical Conference*, pages 171–184, June 1999.

[VRBH+97] Robbert V. Renesse, Ken Birman, Mark Hayden, Alexey Vaysburd, and David Karr. Building adaptive systems using ensemble. Technical report, Ithaca, NY, USA, 1997.

[Web97] WebLogic. *Load Balancing in a Cluster, WebLogic Server 7.0*, 1997.

[Weg03] Maarten Wegdam. *Dynamic reconfiguration and load distribution in component middleware.* PhD thesis, University of Twente, the Netherlands, 2003.

[WLW06] Hsiangkai Wang, Pangfeng Liu, and Jan-Jan Wu. A QoS-aware heuristic algorithm for replica placement. In *GRID*, pages 96–103. IEEE, 2006.

[YRP99] Mark Yarvis, Peter Reiher, and Gerald J. Popek. Conductor : A framework for distributed adaptation. In *Workshop on Hot Topics in Operating Systems*, March 28 1999.

[YV02] Haifeng Yu and Amin Vahdat. Design and evaluation of a conit-based continuous consistency model for replicated services. *ACM Trans. Comput. Syst.*, 20 :239–282, August 2002.

[ZAFB00] Ellen W. Zegura, Mostafa H. Ammar, Zongming Fei, and Samrat Bhattacharjee. Application-layer anycasting : a server selection architecture and use in a replicated web service. *IEEE/ACM Trans. Netw*, 8(4) :455–466, 2000.

[ZYCM04] Ji Zhang, Zhenxiao Yang, Betty H. C. Cheng, and Philip K. McKinley. Adding safeness to dynamic adaptation techniques. In *Proceedings of ICSE 2004 Workshop on Architecting Dependable Systems*, Edinburgh, Scotland, UK, May 2004.

[ZZ03] Chi Zhang and Zheng Zhang. Trading replication consistency for performance and availability : an adaptive approach. In *23th International Conference on Distributed Computing Systems (23th ICDCS'03)*, pages 687–695, Providence, RI, May 2003. IEEE Computer Society.