

Plate-forme de composants logiciels pour la coordination des adaptations multiples en environnement dynamique

Djalel Chefrour

▶ **To cite this version:**

Djalel Chefrour. Plate-forme de composants logiciels pour la coordination des adaptations multiples en environnement dynamique. Réseaux et télécommunications [cs.NI]. Université Rennes 1, 2005. Français. NNT : . tel-00011148

HAL Id: tel-00011148
https://theses.hal.science/tel-00011148

Submitted on 5 Dec 2005

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: 3301

THÈSE

Présentée devant

devant l'université de Rennes 1

pour obtenir

le grade de : Docteur de l'université de Rennes 1
Mention Informatique

par

Djalel Chefrour

Équipe d'accueil : PARIS
École doctorale : MATISSE
Composante universitaire : IFSIC/IRISA

Titre de la thèse :

Plate-forme de composants logiciels pour la coordination des adaptations multiples en environnement dynamique

soutenue le 22 novembre 2005 devant la commission d'examen

M.	Jean-Louis	PAZAT	Président
Mme	Isabelle	DEMEURE	Rapporteur
M.	Jean-François	MEHAUT	Rapporteur
M.	Jean-Marie	GILLIOT	Examinateur
Mme	Françoise	ANDRE	Directeur de thèse

Table des matières

Introduction **11**

I Notions préliminaires **15**

1 Concepts clés de l'adaptabilité **17**
1.1 Définition de l'adaptabilité . . . 17
1.2 Raisons de l'adaptation (Pourquoi ?) . . . 17
1.3 Localisation de l'adaptation (Où ?) . . . 19
1.4 Sujets de l'adaptation (Quoi ?) . . . 20
1.5 Moments de l'adaptation (Quand ?) . . . 21
1.6 Etapes et mécanismes de l'adaptation (Comment ?) . . . 22
1.7 Critères d'analyse des techniques d'adaptation . . . 23

2 Principaux concepts de programmation utilisés pour l'adaptabilité **25**
2.1 Technique de la réflexion . . . 25
2.2 Programmation orientée aspects (AOP) . . . 26
2.3 Programmation basée sur les langages de description d'architectures logicielles . . . 26
2.4 Programmation orientée composants . . . 26
2.4.1 Composant logiciel . . . 27
2.4.2 Les patrons de conception . . . 27
2.4.3 Les canevas de composants . . . 29
2.5 Intergiciel . . . 29
2.5.1 Intergiciel de composants . . . 30
2.6 Conclusion . . . 30

II Etat de l'art sur l'adaptabilité du logiciel **31**

3 Adaptabilité au niveau des infrastructures logicielles **33**
3.1 Les supports système pour l'adaptabilité . . . 33
3.1.1 Systèmes d'exploitation flexibles et extensibles . . . 33
3.1.2 Services distribués adaptables . . . 34

3.2 Les intergiciels adaptables . . . 35
3.2.1 Approche basée sur les contrats de qualité de service . . . 35
3.2.2 Approche basée sur les rôles des composants . . . 37
3.2.3 Approche gérant les conflits d'adaptation . . . 40
3.2.4 Approche basée sur le calcul d'événements . . . 42
3.2.5 Bilan sur les intergiciels adaptables . . . 44
3.3 Les proxies . . . 44
3.3.1 Proxies sur les nœuds intermédiaires du réseau . . . 45
3.3.2 Proxies aux extrémités du réseau . . . 47
3.3.3 Bilan sur l'approche proxy . . . 50
3.4 Conclusion . . . 51

4 Adaptabilité au niveau des applications 53
4.1 Adaptabilité suivant l'approche contrôle . . . 53
4.2 Adaptabilité suivant la programmation orientée aspect . . . 54
4.3 Adaptabilité suivant la programmation basée sur les ADLs . . . 56
4.4 Adaptabilité suivant l'approche composants logiciels . . . 59
4.4.1 Modèle réifiant les dépendances entre les composants . . . 59
4.4.2 Modèle réflexif utilisant les méta-espaces . . . 64
4.4.3 Modèle réflexif et hiérarchique utilisant un MOP . . . 67
4.4.4 Modèle utilisant un protocole d'adaptation graduel . . . 70
4.4.5 Modèle basé sur un langage d'interaction . . . 73
4.4.6 Modèle réactif utilisant le patron *strategy* . . . 74
4.5 Synthèse . . . 76
4.5.1 Revue des mises en œuvre des mécanismes de l'adaptation . . . 77
4.5.2 Guide pour le développement d'un support de l'adaptation . . . 79

III Plate-forme de composants adaptables *Aceel* 83

5 Problématique : la coordination des adaptations multiples 85
5.1 Problèmes des adaptations multiples des composants logiciels . . . 85
5.2 Solutions existantes pour la coordination des adaptations multiples . . . 87
5.3 Conclusion . . . 88

6 Conception de la plate-forme *Aceel* 89
6.1 Introduction : objectifs de développement d'*Aceel* . . . 89
6.2 Modèle de composant réflexif adaptable *Aceel* . . . 90
6.2.1 Aspects structurels . . . 90
6.2.2 Aspects dynamiques . . . 96
6.3 Service de surveillance de l'environnement . . . 99
6.4 Intergiciel de coordination des adaptations . . . 100
6.4.1 Architecture de l'intergiciel de coordination . . . 101
6.4.2 Politique de coordination des composants interagissants . . . 102

6.4.3 Interprétation des politiques d'adaptation des composants 103
6.4.4 Prise des décisions d'adaptations 104
6.4.5 Exécution des adaptations 110
6.5 Conclusion 110

7 Mise en œuvre et expérimentation de la plate-forme *Aceel* 113
7.1 Mise en œuvre du modèle de composants *Aceel* 113
7.1.1 Choix des langages de programmation 113
7.1.2 Canevas *Aceel* 114
7.2 Mise en œuvre du service de surveillance 116
7.2.1 Moniteurs de ressources 117
7.2.2 Protocole ADMP 118
7.3 Mise en œuvre de l'intergiciel de coordination 118
7.3.1 Mise en œuvre de *l'optimiseur* 119
7.3.2 Mise en œuvre du *coordinateur* 120
7.3.3 Mise en œuvre du *synchroniseur* 121
7.3.4 Traitement des cas particuliers 122
7.4 Expérimentation d'*Aceel* en environnement mobile 124
7.4.1 Navigateur Web adaptable 124
7.4.2 Application de vidéo à la demande adaptable 125
7.4.3 Coordination des composants adaptables interagissants 127
7.4.4 Coordination des applications adaptables concurrentes 127
7.5 Conclusion 128

8 Conclusion 129
8.1 Rappel des objectifs 129
8.2 Démarche suivie 129
8.3 Bilan scientifique 130
8.4 Limites et perspectives 131

Références 134

A DTDs *Aceel* 147
A.1 DTD XML pour les politiques d'adaptation 147
A.2 DTD XML pour les politiques de coordination 148

B Politiques de l'intergiciel de coordination 149
B.1 Politique d'adaptation de l'*optimisateur* 149
B.2 Politique d'adaptation du *coordinateur* 150

C Politiques des exemples *Aceel* 151
C.1 Politique d'adaptation du navigateur Web 151
C.2 Politique d'adaptation du récepteur vidéo 152
C.3 Politique d'adaptation de l'émetteur vidéo 153
C.4 Politique de coordination de l'application vidéo 155

Table des figures

2.1 Structure du patron de conception *observer*. 28
2.2 Structure du patron de conception *strategy*. 28
2.3 Structure du patron de conception *memento*. 29
2.4 La couche intergicielle. 30

3.1 Méta-espaces du système *Apertos*. 34
3.2 Gestion du cache dans le système de fichiers distribués *Coda*. 35
3.3 *QuO* s'exécutant au-dessus de CORBA. 36
3.4 Intergiciel *m3*. 38
3.5 Exemple de script *MEADL*. 38
3.6 Interaction entre les composants *m3*. 39
3.7 Exemple de profil d'application s'exécutant sur *Carisma*. 41
3.8 Plate-forme d'Efstratiou et coll. pour l'adaptation coordonnée. 42
3.9 Exemple de politique d'adaptation basée sur les *fluents*. 43
3.10 Architecture de *Conductor*. 45
3.11 Nœud *Conductor*. 46
3.12 Planification du redéploiement des adaptateurs dans *Conductor*. 46
3.13 Architecture de *Puppeteer*. 48
3.14 Adaptation multi-applications avec *HATS*. 50

4.1 Architecture générique d'un contrôleur auto-adaptable. 54
4.2 Combinaison sélective des aspects dans *Lasagne*. 55
4.3 Le canevas *Rainbow*. 57
4.4 Exemple d'une stratégie d'adaptation *Rainbow*. 58
4.5 La classe abstraite `ComponentConfigurator`. 60
4.6 Réification des dépendances d'un composant. 60
4.7 Spécialisation de la méthode `eventOnHookedComponent()`. 62
4.8 Reconfiguration des stratégies de *dynamicTAO*. 63
4.9 Architecture d'*OpenCOM*. 65
4.10 Architecture d'*OpenORB*. 66
4.11 Extension de type MOP pour *Fractal*. 68
4.12 Architecture de *Cactus*. 70
4.13 Structure d'un composant *Cactus*. 71
4.14 Protocole d'adaptation graduelle de *Cactus*. 72

4.15 Architecture de la plate-forme *MolèNE*. 74
4.16 Politique d'adaptation d'un composant *MolèNE*. 75
4.17 Architecture d'un composant *MolèNE*. 76

6.1 Plate-forme de composants adaptables *Aceel*. 90
6.2 Modèle de composant adaptable *Aceel*. 93
6.3 Grammaire des politiques d'adaptation. 95
6.4 Service de surveillance de l'environnement. 100
6.5 Architecture de l'intergiciel de coordination des adaptations. 102
6.6 Grammaire des politiques de coordination. 103

7.1 Classe générique pour les moniteurs de ressources. 117
7.2 Architecture de *l'optimiseur*. 119
7.3 Architecture du *coordinateur*. 120
7.4 Coordination des adaptations sur des sites pairs. 123
7.5 Application de vidéo à la demande basée sur *Aceel*. 126
7.6 Applications *Aceel* en environnement mobile. 127

Liste des tableaux

1.1 Exemples d'adaptations dans différents niveaux logiciels [Friday99]. . . . 21

4.1 Comparaison des différents travaux sur l'adaptabilité du logiciel. 78

7.1 Moniteurs de ressources 117

Introduction

Cadre du travail et motivations

Les progrès réalisés dans les réseaux de communication ont entraîné l'émergence des *systèmes distribués*. De nos jours, ces systèmes connaissent une grande prolifération due au grand essor qu'a connu l'Internet lors de la dernière décennie, ainsi qu'au récent et fort développement des environnements mobiles. Un système distribué est distinct d'un réseau d'ordinateurs car il apporte aux développeurs une abstraction des détails de l'infrastructure et des supports logiciels pour gérer les problèmes introduits par la distribution [Tanenbau96]. Parmi ces problèmes nous trouvons l'hétérogénéité des plates-formes d'exécution, la complexité des infrastructures, la concurrence, le passage à l'échelle, la tolérance aux fautes et la sécurité [Coulouri01, Tanenbau02, Warfield01]. Ces problèmes ont reçu un intérêt plus ou moins important de la part des chercheurs qui leurs ont proposé plusieurs solutions. En particulier, ils ont développé les *intergiciels* qui gèrent la complexité inhérente à la distribution et masquent l'hétérogénéité des plates-formes [Campbell99, Schantz02].

L'émergence des réseaux sans-fil et des équipements mobiles [Varshney00, Salkintz99] a projeté au devant de la scène un problème qui n'avait pas beaucoup d'importance dans les systèmes distribués traditionnels (c.-à-d. des stations de travail reliées par des réseaux filaires). Il s'agit des variations dynamiques dans l'environnement d'exécution des applications. Ces variations sont principalement dues aux caractéristiques technologiques d'une plate-forme particulière ou à l'hétérogénéité entre plusieurs plates-formes. Par exemple, dans un réseau sans-fil WiFi (IEEE 802.11x) la bande passante varie de façon importante en fonction de la mobilité de l'équipement portable.

Pour faire face à ce problème, les développeurs doivent concevoir des applications dites *adaptables*. C'est à dire, des applications dont les comportements peuvent être modifiés en fonction de l'état de leur environnement d'exécution. Notre intérêt pour *l'adaptabilité* vient du fait que c'est une notion clé pour les applications déployées dans des environnements dynamiques tels que les environnements mobiles. En effet, sans adaptation, ces applications ne peuvent continuer leur exécution ou fournir à l'utilisateur une qualité de service acceptable. A titre d'exemple, la lecture d'un flux vidéo transmis à travers un réseau sans-fil peut être interrompue ou devenir saccadée si la bande passante chute considérablement. Ce type de variations ne constitue plus un cas exceptionnel mais une caractéristique intrinsèque des environnements dynamiques.

Actuellement le développement des applications adaptables est réalisé de façon *ad*

hoc. C'est un travail complexe qui nécessite des compétences supplémentaires en plus de celles liées au métier de chaque application, en particulier dans le cas des adaptations dynamiques. En effet, les développeurs doivent gérer plusieurs types de codes qui réalisent des fonctions nécessaires à l'adaptation comme la détection et notification des variations des ressources, le choix des modifications à réaliser en fonction de ces variations, l'exécution de ces modifications de façon cohérente qui préserve la stabilité et l'intégrité du système, la coordination des adaptations multiples entre des entités interagissantes et/ou concurrentes, etc.

En regardant de près ces différentes tâches, il est facile de constater que seuls le choix des actions d'adaptation à réaliser et la coordination des entités interagissantes dépendent du code métier des applications. Les autres tâches sont indépendantes de ce code et il est donc inutile de les refaire à chaque développement d'une nouvelle application adaptable. En réalité, il y a un grand besoin de factoriser ces tâches nécessaires à l'adaptation et les rendre réutilisables afin de gagner du temps et faciliter le travail des développeurs.

Contributions

L'objectif de notre travail était de proposer des mécanismes génériques pour permettre l'adaptabilité des logiciels de manière *systématique* (par opposition à *ad hoc*). De nombreux chercheurs issus de communautés différentes se sont intéressés à cette question. Parmi ces chercheurs, beaucoup ont considéré le problème sous l'angle de l'approche basée sur les composants logiciels qui est l'approche dominante dans le développement des applications distribuées. Toutefois, la majorité a proposé des modèles gérant l'adaptation au niveau individuel du composant sans attacher beaucoup d'importance aux deux niveaux d'abstraction supérieurs, qui sont l'application et le système distribué. Dans le premier niveau, il faut considérer les adaptations de plusieurs composants interagissants qui forment une seule application. Dans le deuxième niveau, il faut considérer les adaptations de plusieurs composants qui partagent les ressources de l'environnement et qui appartiennent à plusieurs applications concurrentes. Les rares travaux qui ont abordé le problème complexe des adaptations multiples n'ont considéré que l'un ou l'autre de ces deux niveaux mais pas les deux en même temps. De plus, ils sont restés liés à une variation précise (p. ex. la bande passante) ou à un type d'application particulier (p. ex. les applications bureautiques) et sont donc loin d'être génériques.

Dans notre travail, nous avons identifié les mécanismes génériques qui supportent les adaptations individuelles et les adaptations multiples au sein d'un canevas logiciel réutilisable. Notre canevas est bâti autour d'un modèle de composants logiciels adaptables. Ceux-ci sont déployés au-dessus d'un intergiciel dédié à la coordination des adaptations multiples et d'un service sous-jacent pour la surveillance de l'environnement. En instaurant la séparation des préoccupations, notre approche facilite la tâche du développeur à qui nous offrons des langages déclaratifs pour spécifier les politiques qui contrôlent les adaptations des composants dans des scripts séparés. Notre modèle de composants

Aceel (Adaptive ComponEnt modEL) utilise la réflexivité et les techniques objet pour permettre l'introspection de l'état du composant et l'intercession de son code métier à l'aide d'actions d'adaptation bien définies. Basé sur la théorie de la coordination, notre intergiciel assure la cohérence des composants liés par des dépendances d'interaction et évite les conflits entre les composants liés par des dépendances de partage des ressources. Ces différentes tâches sont modélisées sous la forme d'un problème d'optimisation combinatoire de la qualité de service offerte sous les contraintes de l'environnement.

Plan de la thèse

Ce mémoire est organisé en trois parties. La première partie introduit une définition de la notion d'adaptabilité du logiciel (cf. chapitre 1) et rappelle les techniques et les approches de programmation adoptées pour mettre en œuvre cette notion (cf. chapitre 2). La deuxième partie contient un état de l'art sur l'adaptabilité du logiciel. Dans le chapitre 3 nous présentons l'adaptabilité au niveau des infrastructures logicielles, tandis que dans le chapitre 4 nous exposons les travaux sur l'adaptabilité au niveau des applications.

La troisième partie présente en détails la problématique à laquelle nous nous sommes confronté (cf. chapitre 5) et la solution que nous avons apporté. Le chapitre 6 détaille la conception de notre plate-forme de composants en décrivant le modèle de composants adaptables *Aceel*, le service de surveillance de l'environnement et l'intergiciel de coordination des adaptations multiples. Dans le chapitre 7, nous exposons une implantation prototype de notre plate-forme de composants en commençant par une argumentation du choix des langages de programmation utilisés. Ensuite, nous présentons une expérimentation de cette implantation prototype avec deux applications adaptables en environnement mobile . Il s'agit d'un navigateur Web et d'une application de vidéo à la demande. L'objectif étant d'illustrer l'utilisation d'*Aceel* pour rendre adaptables des codes légataires et, surtout, de montrer comment les mécanismes proposés gèrent concrètement l'adaptation au niveau d'un composant individuel, puis au niveau de composants interagissants et, ensuite, au niveau de plusieurs composants concurrents.

Ce mémoire est conclu par le chapitre 8 où nous synthétisons les résultats obtenus, nous établissons les limites du travail réalisé et nous proposons les perspectives pour des travaux futurs.

Première partie

Notions préliminaires

Chaptire 1

Concepts clés de l'adaptabilité

Ce chapitre introduit une définition de la notion d'adaptabilité des applications. Ensuite, il approfondit cette définition à travers les réponses aux questions : pourquoi (s')adapter, qui adapte quoi, quand le fait-il et comment ? Puis, il explicite des critères pour examiner différentes techniques d'adaptation.

1.1 Définition de l'adaptabilité

Nous définissons la notion d'adaptabilité comme la faculté de modifier une application pour qu'elle continue à assurer sa fonction en dépit des variations des conditions de son environnement[1]. En plus de l'application elle-même, le terme « environnement » englobe aussi les applications concurrentes, les ressources de la plate-forme d'exécution et les éventuels utilisateurs humains. Toutefois, cette définition de l'adaptabilité étant un peu vaste et pour bien cerner notre sujet d'étude, nous allons la préciser en essayant de répondre aux questions suivantes : *pourquoi* s'adapter ? *Où* implanter l'adaptation ? Qu'est ce *qui* est modifié, *quand* et *comment* ?

1.2 Raisons de l'adaptation (Pourquoi ?)

Il y a trois raisons majeures pour adapter une application. Chacune d'elles définit un type d'adaptations différent. Ces raisons de l'adaptation sont :

Adaptations réactives

Les adaptations réactives sont des adaptations dynamiques à court terme, réalisées en cours d'exécution. Leur objectif est d'accommoder les applications aux changements de leur environnement. Il peut s'agir de changements dans le comportement de l'utilisateur d'une application, des modifications importantes des données manipulées, des variations des niveaux des ressources disponibles (p. ex. mémoire vive,

[1] Cette définition est basée sur celle du verbe « adapter » dans le dictionnaire Hachette.

connexion réseau, charge CPU, etc.) ou de l'apparition/disparition d'une ressource en cours d'exécution (p. ex. un volume de stockage sur une clé USB, une imprimante reliée à un réseau sans-fil, un nœud dans une grille de calculateurs, etc.) Ces changements caractérisent davantage les environnements très dynamiques tels les environnements mobiles [Varshney00, Satyanar96a], les grappes de PC (grilles de calculateurs) [Buisson04, Cheng02] et les périphériques de type *Plug and Play*. Etant donné que la prolifération de ce genre d'environnement remonte à une douzaine d'années seulement [Satyanar96a, Baker99], la recherche sur l'adaptation réactive est relativement récente.

Adaptations évolutives

Comme leur nom l'indique, les adaptations évolutives ont lieu suite à l'évolution des besoins des utilisateurs d'une application. Elles font partie du processus de génie logiciel qui vise à étendre les fonctionnalités d'une application, corriger ses erreurs ou augmenter sa performance. Ce sont donc des adaptations à long terme qui prennent la forme de mises à jour logicielles. Les adaptations évolutives entrent généralement dans la catégorie des adaptations statiques réalisées avant l'exécution. Cependant, dans le cas des systèmes critiques à haute disponibilité (p. ex. les systèmes de télécommunication et les systèmes bancaires), les mises à jour doivent être réalisées à chaud ce qui fait qu'elles se comportent comme des adaptations dynamiques.

Contrairement à la recherche sur l'adaptation réactive, la recherche sur la mise à jour à chaud des applications est plus ancienne. Par exemple, l'un des travaux traitant le remplacement à la volée d'un module dans un système date de 1976 [Fabry76]. Ceci dit, les deux types d'adaptations ont des objectifs différents mais les mêmes techniques peuvent être utilisées pour les mettre en œuvre.

Adaptations pour l'intégration

Le recours à l'adaptation peut être nécessaire dans la programmation par intégration[2] (composition) de modules (composants) logiciels existants. Dans ce cas l'adaptation vise à rendre inter-opérables des modules qui ont des interfaces ou des protocoles d'interaction incompatibles, le plus souvent car ils ont été implantés par des développeurs différents. Ce type d'adaptation peut aussi avoir comme objectif le rajout de nouvelles interactions entre les composants ou l'intégration de services non fonctionnels (p. ex. sécurité, transactions) non anticipés [Charfi04].

L'adaptation pour l'intégration est généralement mise en œuvre par des techniques statiques comme l'héritage dans les langages objets, les Wrappers [Gamma95, Hölzle93, Troya98], la super-imposition [Bosch97], et les interfaces actives [Heineman98]. Le lecteur pourra trouver plus d'informations sur ces techniques statiques dans [Heineman00]. Plus récemment, plusieurs techniques de composition dynamique ont vu le jour. Elles sont basées sur la notion de types des composants [Gschwind02] et sur les langages

[2] La programmation par composition est appelée aussi macro programmation ou programmation dans le large.

d'interaction [Charfi03] (cf. section 4.4.5), etc. Dans l'ensemble, l'adaptation pour l'intégration peut être vue comme un cas particulier de l'adaptation évolutive.

1.3 Localisation de l'adaptation (Où ?)

L'adaptation peut être prise en charge par l'application elle-même qui est alors qualifiée *d'auto-adaptable*. Elle peut être aussi réalisée au niveau de l'infrastructure logicielle sous-jacente comme le système d'exploitation, la pile des protocoles réseaux ou l'intergiciel. De plus, l'intervention de l'administrateur ou de l'utilisateur de l'application dans l'adaptation est envisageable voire souhaitable. Mais, rien n'empêche que tous ces acteurs participent ensemble à la réalisation de l'adaptation. En fait, ces différentes possibilités correspondent à l'évolution historique de la façon dont l'adaptation a été implantée. Cette évolution est passée par trois étapes principales :

Adaptation à la charge des applications

Initialement l'adaptation était entièrement laissée à la charge des applications. C'était notamment le cas des premiers programmes destinés aux environnements mobiles. Ces programmes devaient gérer eux-mêmes les changements de contexte dus à la mobilité comme la diminution de la source d'énergie, la variation de la bande passante et du taux d'erreurs de transmission du réseau sans-fil, les déconnexions imprévues, etc. De façon générale, développer une application adaptable revenait à écrire en plus du code métier, le code qui observe l'environnement (p. ex. surveillance des paramètres du réseau), le code qui prend la décision d'adaptation (p. ex. augmenter ou diminuer la quantité d'information transmise à travers le réseau) et le code qui concrétise ces décisions (p. ex. filtrer, compresser ou mettre en attente les informations transitant par le réseau, etc.). Ainsi, l'écriture et la maintenance d'une application adaptable étaient des tâches difficiles et complexes pour les développeurs. Ceux-ci devaient posséder des compétences multiples et variées, allant de leur domaine d'expertise spécifique à la maîtrise des technologies utilisées dans les infrastructures sur lesquelles leurs applications sont déployées.

Adaptation transparente aux applications

Pour palier l'inconvénient précédent, les travaux de recherche se sont orientés vers l'intégration de l'adaptation dans l'infrastructure logicielle afin de la rendre transparente aux applications [Satyanar96a]. Cette approche est attrayante car elle permet d'exploiter les applications existantes sans modification. Elle est même désirable dans le cas où les conditions de l'adaptation seraient connues a priori [Dowling01]. Cependant, l'adaptation transparente est fondamentalement limitée car, compte tenu de la diversité des applications, aucun système (aussi intelligent soit-il) ne peut prévoir toutes les conditions et les stratégies d'adaptation possibles. Pire encore, cette approche peut être inadéquate ou peut avoir des contre-performances dans certaines situations importantes [Pitoura98].

Applications conscientes de l'adaptation

Les deux approches précédentes (l'adaptation entièrement à la charge des applications et l'adaptation complètement transparente) constituent deux extrêmes présentant chacun des inconvénients importants. Pour éviter ces inconvénients, la majorité des chercheurs admet que la meilleure approche est de rendre les applications conscientes de l'adaptation [Satyanar96b, Noble97, Pitoura98, Joseph97, Sudame99]. Pour cela il faut mettre en œuvre :

1. Une observation des ressources de la plate-forme d'exécution (p. ex. cycles CPU, mémoire, réseau, etc.) et une remontée des informations concernant les variations de ces ressources jusqu'aux applications.
2. Une collaboration entre les applications et les couches système pour prendre des décisions d'adaptation cohérentes et globalement optimales.

Cette approche s'explique par le fait que dans certaines situations, les applications sont mieux placées que l'infrastructure sous-jacente pour prendre les décisions d'adaptation. A titre d'exemple, deux techniques possibles pour s'adapter à la chute de la bande passante du réseau sont la compression des données transmises ou l'effacement d'une partie de ces données. Dans ce cas, le système ne sait pas qu'elle technique est la plus appropriée, ni quelle partie de données il faut effacer s'il choisit la deuxième technique. En fait seule l'application qui traite ces données est en mesure de faire le bon choix. La deuxième raison qui justifie cette approche est due à la concurrence entre plusieurs applications non conscientes les unes des autres. Dans ce cas le système peut jouer le rôle d'arbitre pour éviter des décisions d'adaptation individuelles qui sont contradictoires (conflictuelles) ou inéquitables.

Ainsi, le traitement de l'adaptabilité nécessite à la fois des supports système appropriés [Friday96, Davies96] et une participation importante de la part des applications (voir même de l'utilisateur, en dernier ressort). En d'autres termes, l'adaptation aux variations de l'environnement doit être *multi-niveaux* [Friday99, Pitoura98]. Le tableau 1.1 donne quelques exemples de techniques d'adaptation avec leurs emplacements par rapport aux différents niveaux logiciels.

1.4 Sujets de l'adaptation (Quoi ?)

Nous appelons sujets de l'adaptation les concepts de programmation qu'il faut adapter. Ils sont donc définis par l'approche de programmation de l'application ou de l'infrastructure logicielle à adapter. Des exemples de ces concepts sont les modules, les objets, les composants, les connecteurs, les agents, les processus, etc. Comme à la base l'adaptation est une opération de modification il est plus facile de la mettre en œuvre si l'approche de programmation suivie est modulaire car il est plus aisé d'isoler les sujets de l'adaptation et de contrôler ainsi l'étendue des modifications qu'ils subissent. C'est notamment la raison pour laquelle la majorité des chercheurs intéressés par l'adaptabilité aborde ce problème sous l'angle de l'approche composants logiciels qui est actuellement l'approche la plus modulaire.

Niveau	Technique d'adaptation	Description
Utilisateur	Changer la méthode de travail	L'utilisateur peut alléger les demandes sur le réseau, p. ex. en retardant des requêtes non essentielles gourmandes en bande passante.
Application	Changer le mode de communication	L'application peut basculer alternativement du mode RPC vers un mode asynchrone.
Intergiciel	Filtrer ou compresser	Le volume des informations échangées via le réseau peut être réduit par compression des trames de données.
Système et Réseau	Changer ou introduire de nouveau protocoles	Le système peut sélectionner une pile de protocole appropriée à un type de réseau particulier. Il peut aussi introduire une couche de protocole spécifique (p. ex. une couche liaison de données fiable).

TAB. 1.1 – Exemples d'adaptations dans différents niveaux logiciels [Friday99].

1.5 Moments de l'adaptation (Quand ?)

L'adaptation d'une application peut avoir lieu à n'importe quel moment de son cycle de vie : à la conception, à la compilation, au déploiement et/ou à l'exécution.

Phase de développement

Lors de la phase de développement qui va de la conception jusqu'à la compilation, une application peut être adaptée en utilisant des techniques de programmation et/ou de compilation appropriées. La technique la plus simple mais la plus coûteuse est la réécriture de tout ou d'une partie du code pour accommoder l'application aux conditions particulières de son environnement. C'est notamment le cas quand il s'agit de porter une application vers une plate-forme différente de celle prévue initialement. L'héritage et le polymorphisme des langages objets sont des exemples de techniques plus évoluées qui peuvent aussi être utilisées à cette fin. Cependant, bien que celles-ci soient moins fastidieuses pour le développeur, l'effort global de la modification requise pour adapter l'application reste très important.

Par ailleurs, l'une des techniques d'adaptation des applications à leur environnement est la compilation vers un code intermédiaire qui sera interprété par une machine virtuelle installée au préalable sur chacune des plates-formes hétérogènes. L'exemple le plus connu de cette technique est la machine virtuelle Java.

Phase de déploiement

Pour pallier le problème d'hétérogénéité des plates-formes, il est aussi possible de développer plusieurs versions de l'application qui sont adaptées chacune à des conditions d'exécution particulières. Ensuite le choix de la version adéquate est retardé jusqu'au déploiement de l'application sur la plate-forme d'accueil. Comparé à l'utilisation d'une machine virtuelle, cette approche présente l'avantage d'avoir de meilleures performances d'exécution des applications mais l'effort de programmation est beaucoup plus important.

Phase d'exécution

De manière générale, les adaptations faites aux moments du développement ou du déploiement des applications sont qualifiées de *statiques*. Après compilation, les modifications qu'elles apportent à l'application seront définitives et celle-ci ne pourra s'exécuter que dans les conditions prévues initialement. Si ces conditions changent, il faut reprendre le cycle de développement dès le début, ce qui n'est pas possible si le temps de réaction est limité.

Aussi afin d'assurer le bon fonctionnement des applications en dépit des variations de leurs environnements il faut les adapter en cours d'exécution. Cette adaptation qualifiée de *dynamique* peut être difficile à mettre en œuvre car, en plus d'écrire le code métier de l'application, les développeurs doivent écrire le code qui observe les variations des ressources et réalise les modifications nécessaires tout en préservant l'intégrité de l'application.

1.6 Etapes et mécanismes de l'adaptation (Comment ?)

On peut distinguer trois étapes principales par lesquelles passe toute opération d'adaptation, associées chacune à des mécanismes de mise en œuvre. Ces étapes sont :

Détection/notification

Pour qu'une application s'adapte (ou soit adaptée) aux variations pertinentes de son environnement, il faut un mécanisme qui détecte et notifie ces variations. A titre d'exemple, la chute de la bande passante réseau, l'action directe d'un utilisateur humain sur une interface homme-machine sont des variations susceptibles de déclencher une adaptation d'une ou plusieurs applications. Dans ce cas on utilise un mécanisme de démon/moniteur associé à chaque élément à surveiller. Dans le cas particulier où une application s'observe elle-même on parle *d'introspection* et les ressources surveillées sont dites *applicatives* (p. ex. la valeur d'une variable d'état, le nombre de transactions effectuées, etc.). Dans le cas où la ressource est fournie par la plate-forme d'exécution on parle de ressource *système*. Son surveillant peut être implanté par un intergiciel spécifique, par le système d'exploitation ou par un pilote du périphérique fournissant cette ressource (p. ex. le pilote d'une interface réseaux est capable de donner le taux de paquets perdus lors d'une transmission).

Prise de décision

Après la notification d'une variation, l'adaptateur doit décider selon une politique (logique) d'adaptation quelles actions entreprendre. Cette politique peut être confinée dans l'adaptateur auquel cas seules les adaptations prévues au moment de la conception seront possibles. Dans le cas contraire, si la politique d'adaptation est externalisée, elle peut être modifiée en cours d'exécution par l'utilisateur afin de réaliser des adaptations non anticipées. Comme nous le verrons dans l'état de l'art sur l'adaptabilité des applications, la politique d'adaptation peut être exprimée sous forme impérative ou déclarative. Elle peut être aussi basée sur des modèles formels comme les automates ou le calcul d'événements. Cependant, elle reste toujours liée à la nature du traitement réalisé par l'application en question.

Exécution

Dans cette étape, l'adaptateur exécute les actions de modification décidées. Ces actions portent sur l'application concernée par les variations (sur ses composants et/ou sa structure), sur l'infrastructure sous-jacente et/ou sur les applications concurrentes. Si une application se modifie elle-même on parle alors *d'intercession*. Dans ce cas particulier, la technique de réflexion (cf. section 2.1) est souvent utilisée pour mettre en œuvre l'adaptation.

1.7 Critères d'analyse des techniques d'adaptation

Pour faire un état de l'art des applications et des systèmes adaptables existants, il est intéressant d'établir les bases d'une (de) classification(s) des techniques d'adaptation. Il est facile de constater que chacune des questions précédentes constitue un axe d'analyse naturel des systèmes adaptables. Par conséquent, chacune peut donner lieu à une classification possible. À titre d'exemple, une adaptation peut être classée à la fois comme dynamique, réactive, structurelle et à la charge de l'application.

Pour dresser un état de l'art sur l'adaptabilité des systèmes logiciels, nous avons opté pour une double classification des techniques d'adaptation : par rapport aux niveaux logiciels *où* est implantée l'adaptation (infrastructure ou application), et par rapport aux approches de programmation qui déterminent *comment* mettre en œuvre l'adaptabilité.

Chaptire 2

Principaux concepts de programmation utilisés pour l'adaptabilité

Ce chapitre présente les techniques, approches et concepts de programmation utilisés pour la mise en œuvre de l'adaptabilité des systèmes logiciels, comme nous le verrons dans les travaux mentionnés dans l'état de l'art.

2.1 Technique de la réflexion

La réflexion a été initialement utilisée dans les langages de programmation pour supporter la conception de langages ouverts et extensibles [Kiczales91]. Elle a été aussi appliquée dans d'autres domaines incluant les systèmes d'exploitation [Yokote92] et plus récemment les systèmes distribués [McAffer96]. Un système logiciel est dit *réflexif* s'il est capable d'accéder à son propre comportement, de raisonner sur celui-ci et de le modifier. Pour ce faire, il doit manipuler une représentation de lui-même, c'est à dire des structures et des opérations qui interprètent son domaine d'application [Smith82].

Un système réflexif peut être décomposé en deux niveaux distincts : le *niveau de base* contenant le code métier qui traite le domaine d'application et le *méta-niveau* qui manipule la représentation de soi. Ces deux niveaux sont liés de façon causale, c'est à dire que tout changement dans l'un d'eux se répercute sur l'autre [Maes87]. L'accès au méta-niveau se fait via le concept de *réification*. La réification signifie l'exposition de certains aspects de la représentation de soi de telle sorte qu'ils puissent être accédés par l'application (le niveau de base). Une approche classique pour définir les services du méta-niveau consiste à utiliser un protocole méta-objet (MOP) [Kiczales91].

Les services du méta-niveau sont généralement appelés par l'application elle-même (le niveau de base) ou par un mécanisme de détection externe. Ainsi, un système réflexif peut adapter dynamiquement son comportement en réaction aux changements de son environnement. L'avantage majeur apporté par la technique de réflexion est la séparation nette entre le code métier des applications et le code qui gère l'adaptation. Cependant

cette technique engendre un surcoût en temps d'exécution à cause de la réification.

2.2 Programmation orientée aspects (AOP)

La programmation orientée aspects (AOP) [Kiczales97] vise à séparer les propriétés des programmes qui peuvent être encapsulées entièrement dans des procédures, des classes ou des modules, des propriétés (appelées *aspects*) qui ne peuvent être encapsulées car transversales à plusieurs éléments. Les aspects correspondent à des propriétés non fonctionnelles d'usage général qui se retrouvent dans de nombreux composants du programme de base, telles que la synchronisation, la récupération des erreurs et la persistance. Dans l'AOP les aspects sont écrits séparément dans des langages appropriés tel AspectJ [Kiczales01] qui est une extension de Java. Ensuite, un compilateur spécial appelé *tisseur (weaver)* compose (tisse) ces aspects avec le programme de base pour produire un seul exécutable.

L'AOP est très proche de la réflexivité dans le sens où le tissage des aspects peut être implanté à l'aide d'un MOP. Dans ce cas, les fonctionnalités de l'application sont programmées au niveau de base et les aspects sont implantés sous forme de méta-objets. L'intérêt de l'AOP pour les adaptations dynamiques des applications est que le tissage peut être fait en cours d'exécution (cf. section 4.2).

2.3 Programmation basée sur les langages de description d'architectures logicielles

L'architecture logicielle a émergée comme un domaine important du génie logiciel qui traite la conception et la construction de systèmes complexes. Une *architecture logicielle* définie la structure d'une application en précisant les parties (composants) qui la constituent, les *connecteurs* qui modélisent les interactions entre ces parties et les règles qui gouvernent ces interactions [Garlan00]. Des exemples de connecteurs incluent des formes d'interaction simples comme un *pipe*, un appel de procédure ou une diffusion d'événements. Mais un connecteur peut aussi représenter une forme d'interaction plus complexe comme un protocole client-serveur ou un lien SQL avec une base de données.

Le moyen le plus répandu pour représenter et analyser les architectures logicielles sont les langages de description d'architectures (ADL, Architecture Description Languages). Ce sont des notations formelles qui fournissent un cadre conceptuel et une syntaxe concrète explicitant *les composants, les connecteurs* et *la configuration architecturale (structure)* d'une application [Medvidov00]. L'intérêt de l'architecture logicielle réside dans le fait qu'il est possible d'adapter une application en reconfigurant dynamiquement son architecture (cf. section 4.3).

2.4 Programmation orientée composants

L'une des évolutions majeures en matière de développement logiciel est l'approche de programmation orientée composants [Riveill00]. Elle facilite la construction d'appli-

cations complexes, leur déploiement, leur administration et la maîtrise de leur évolution. D'après Bertrand Meyer [Meyer99a] il n'y a pas de définition qui soit largement acceptable pour le terme « composant logiciel ». Clemens Szyperski explique que ceci est dû à la diversité et la concurrence entre les modèles de composants existants, comme CCM (CORBA Component Model), EJB (Entreprise Java Beans), COM/COM+ (Component Object Model), etc. [Szypersk98]. Pour notre étude, nous avons opté pour la définition présentée dans la section 2.4.1 qui regroupe les points communs aux modèles existants et qui reste neutre par rapport aux divergences sur les autres caractéristiques qu'un composant doit avoir[1].

2.4.1 Composant logiciel

Un *composant logiciel* est une unité d'encapsulation et de réutilisation. Il possède un *état* constitué par les données qu'il traite, une *implantation* qui est le code réalisant ce traitement et des *interfaces* fournies et requises qui sont des ports d'entrée/sortie pour interagir avec le monde extérieur. Un modèle de composant est dit hiérarchique (p. ex. Fractal, Darwin) ou plat (p. ex. EJB, .Net) selon que les composants assemblés soient des composites (c.-à-d. constitués d'autres composants) ou primitifs.

Grâce à l'encapsulation, un composant peut être réutilisé dans plusieurs applications. Celles-ci sont alors construites, non plus à partir de rien, mais par assemblage de composants existants développés par des tiers. Cet assemblage (appelé aussi composition, intégration, macro-programmation ou programmation dans le large) consiste à connecter les composants fournisseurs de services aux composants clients en respectant une architecture logicielle précise.

Il est aussi possible de réutiliser une partie de l'architecture d'une application par le moyen des patrons de conception et des canevas que nous présentons dans les deux sections suivantes.

2.4.2 Les patrons de conception

Les *patrons de conception (Design Patterns)* sont des artifices qui permettent de capturer les solutions (conceptions ou micro-architectures) avérées à des problèmes récurrents. Plus précisément, chaque patron de conception nomme, explique et évalue de façon systématique une conception importante qui se reproduit dans les systèmes orientés objets [Gamma95]. Les patrons de conceptions sont une discipline récente du génie logiciel. Ils ont des racines dans d'autres disciplines, notamment dans les travaux de C. Alexander (1977) sur l'urbanisme et les architectures des bâtiments [Appleton00]. Les patrons *observer*, *strategy* et *memento* du catalogue de Gamma et coll. sont trois exemples de patrons couramment utilisés pour implanter les mécanismes de surveillance de l'environnement et de réalisation des adaptations.

[1] Par exemple, est ce qu'un composant doit être fourni sous forme binaire ? Est ce seulement pour une raison d'encapsulation de l'information [Meyer99b] ou aussi pour l'interopérabilité des composants écrits dans différents langages [Szypersk00] ?

Le patron *observer* définit une dépendance de type *un-à-plusieurs* entre un objet sujet et un nombre quelconque d'objets observateurs. Quand le sujet change, ses dépendants sont notifiés pour exécuter les mises à jours requises. La figure 2.1 montre le schéma de classes UML du patron *observer*. Les notes attachées aux méthodes donnent une idée sur leurs implantations.

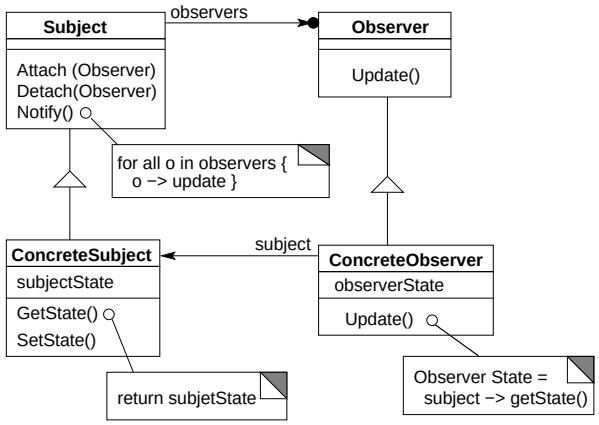


FIG. 2.1 – Structure du patron de conception *observer*.

Le patron *strategy* encapsule les variantes d'un algorithme (c.-à-d. ses implantations alternatives) dans une hiérarchie de classes et les rend interchangeables. Ceci permet la modification du comportement de cet algorithme en cours d'exécution. Ces variantes sont implantées par les classes `ConcreteStrategy` (cf. figure 2.2) qui possèdent la même interface héritée de leur classe mère `Strategy`. L'objet `Context` maintient une référence vers l'implantation sélectionnée afin de lui déléguer les requêtes des clients. Eventuellement, il maintient aussi les données résiduelles manipulées par les implantations dans l'objet membre `State`.

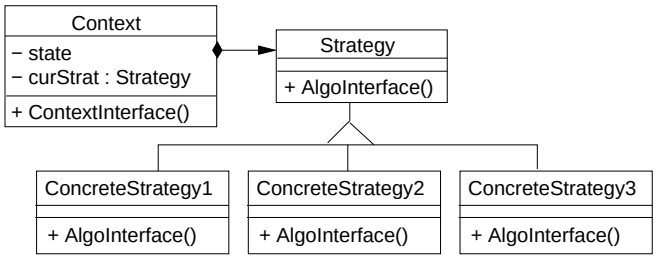


FIG. 2.2 – Structure du patron de conception *strategy*.

Le patron *memento* externalise et sauvegarde l'état interne d'un objet afin de pouvoir le restaurer ultérieurement. La figure 2.1 montre la structure de ce patron où la classe `Originator` peut créer des objets `Memento` afin de sauvegarder son état interne à des moments cruciaux de son exécution. Les objets `Memento` autorisent seulement la classe `Originator` à accéder à leur information interne. Ils sont eux mêmes stockés par une

tierce partie (le `Caretaker`) qui ne peut explorer ni modifier leurs contenus. Ainsi le patron *memento* décrit une méthode d'externalisation de l'état d'un objet sans violer le principe d'encapsulation.

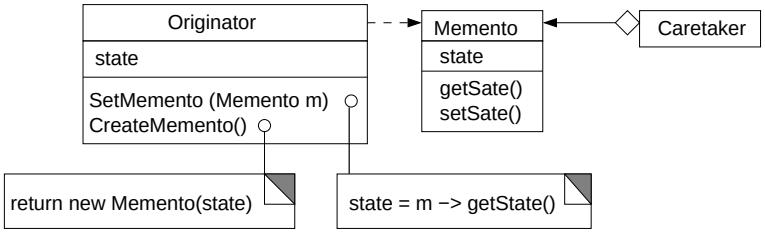


FIG. 2.3 – Structure du patron de conception *memento*.

2.4.3 Les canevas de composants

La construction d'applications suivant l'approche objet et l'approche composant se fait généralement par spécialisation d'un *canevas (framework)* propre à une classe spécifique de logiciels [Gamma95, Szypersk98, Schmidt03a]. Un canevas est une implantation partielle d'un sous système qui doit être complétée par le code spécifique à l'application. Un canevas (appelé aussi schéma de conception) peut contenir plusieurs patrons de conception. De ce fait il permet la réutilisation des conceptions (en plus des implantations) en définissant l'architecture de l'application [Gamma95].

Dans le cas de l'approche objet, l'utilisation avec succès d'un canevas passe en général par une phase d'apprentissage qui peut être difficile. Ceci est du au fait que l'architecture des applications n'est pas explicite car ces canevas exposent seulement les hiérarchies des classes et non pas les interactions entre les objets [Schneide99]. Cet inconvénient est éliminé dans le cas des canevas de composants grâce à une définition explicite de l'architecture de l'application, notamment à l'aide des ADLs. Ainsi, un *canevas de composants* est une collection de composants logiciels et de styles architecturaux[2] qui déterminent les interfaces de ces composants, les connecteurs utilisables et les règles de composition [Szypersk98, Schneide99]

Les canevas de composants sont utilisés de plus en plus pour le développement des applications et des intergiciels implantant leurs besoins non fonctionnels (cf. sections 4.4.1, 4.4.2). Comme nous l'avons déjà évoqué, l'intérêt de ces canevas pour l'adaptabilité est dû à la modularité de l'approche composant qui permet de localiser les modifications nécessaires lors d'une adaptation et donc de maîtriser leur complexité, leur effet et leur coût.

2.5 Intergiciel

L'intergiciel est la couche logicielle résidant entre les applications d'une part et le sys-

[2] Comme exemples de styles architecturaux répandus il y a : le style blackboard, client-serveur, pipeline ou le style basé sur les événements.

tème d'exploitation, la pile des protocoles réseaux d'autre part [Schantz02, Geihs01] (cf. figure 2.4). Son rôle est de réduire les coûts et simplifier le développement des applications, notamment les applications distribuées, en gérant la complexité inhérente à la distribution et en masquant l'hétérogénéité des plates-formes [Campbell99, Emmerich00]. Les intergiciels offrent aux applications des services de haut niveau qui cachent les détails de l'infrastructure sur laquelle elles reposent tels que les détails des protocoles de communication. Ces services implantent la distribution des appels de procédures distantes (RPC) et les appels d'objet distants (ORB). Ils implantent aussi une multitude de propriétés non fonctionnelles comme la tolérance aux fautes, les transactions, la sécurité, le nommage, la réplication des données, les mémoires partagées, etc.

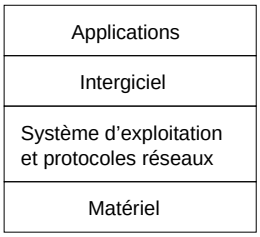


FIG. 2.4 – La couche intergicielle.

2.5.1 Intergiciel de composants

Traditionnellement, les modèles de composants comme CCM, COM et EJB sont bâtis au-dessus des intergiciels comme CORBA, DCOM (Distributed COM) et J2EE (Java 2 Enterprise Edition), respectivement. Ceux-ci fournissent un modèle de programmation orienté objet qui permet l'interconnexion (via les appels de méthodes distantes) des composants distribués.

De nos jours, la technologie des composants est utilisée pour construire les intergiciels eux-mêmes [Clarke01]. Ceci présente un grand intérêt pour l'adaptation car la flexibilité des canevas de composants permet d'augmenter la configurabilité de ces intergiciels. Ainsi il est possible d'adapter plus facilement les besoins non fonctionnels des applications distribuées par la reconfiguration dynamique des intergiciels qui les fournissent.

2.6 Conclusion

Les concepts de programmation présentés dans ce chapitre jouent un rôle important dans l'adaptabilité des applications. Nous présentons dans la partie suivante un état de l'art sur les diverses manières dont ces concepts sont utilisés pour supporter l'adaptation dynamique aux variations de l'environnement.

Deuxième partie

Etat de l'art sur l'adaptabilité du logiciel

Chaptire 3

Adaptabilité au niveau des infrastructures logicielles

Dans ce chapitre nous présentons brièvement les travaux de recherche qui implantent l'adaptation au niveau des infrastructures logicielles, soit de façon transparente aux applications ou en impliquant celles-ci dans le processus d'adaptation. Nous avons regroupé ces travaux en trois catégories distinctes en fonction du type de l'infrastructure concernée : les supports système à l'adaptabilité, les intergiciels adaptables et les proxies.

3.1 Les supports système pour l'adaptabilité

La volonté d'implanter l'adaptation au niveau des couches logicielles basses a conduit au développement de systèmes d'exploitation (SEs) flexibles et extensibles, et à l'émergence de services distribués adaptables appropriés pour les environnements dynamiques, en particulier, les environnements mobiles.

3.1.1 Systèmes d'exploitation flexibles et extensibles

Comme exemples de noyaux de SEs flexibles et extensibles on peut citer : l'exo-noyau *Aegis* [Engler95] qui permet aux applications de spécialiser ou remplacer les implantations des abstractions système traditionnelles comme la mémoire virtuelle ou la communication inter-processus, le micronoyau *SPIN* [Bershad95] qui autorise les applications à charger des *extensions* définies par l'utilisateur dans le noyau et le système réflexif *Apertos* [Yokote92]. Ce dernier adapte les objets applicatifs en leur associant des *méta-espaces* différents, ces *méta-espaces* implantent les aspects système comme l'ordonnancement ou le protocole de communication (cf. figure 3.1). Tous ces moyens permettent donc d'adapter le comportement des logiciels.

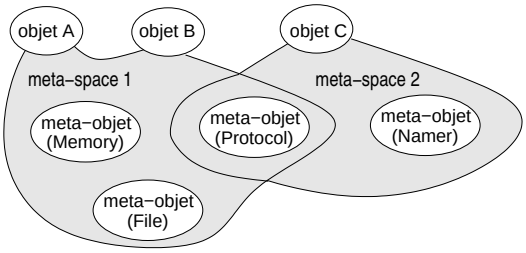


FIG. 3.1 – Méta-espaces du système *Apertos*.

3.1.2 Services distribués adaptables

Les services distribués répondent aux besoins de la communication, de l'accès aux données distantes, de la sécurité, du partage de mémoires, etc. Ces services peuvent être vus comme des intergiciels spécifiques par contraste avec les intergiciels de distribution d'objets ou les intergiciels fournisseurs de services génériques (cf. section 3.2). Nous détaillons dans les sous-sections suivantes les deux types de services qui sont les plus utilisés par les applications distribuées dans les environnements mobiles, à savoir la communication et l'accès aux données.

3.1.2.1 Services de communication adaptables

Le système Ensemble [van Rene98] un est exemple représentatif des travaux sur l'adaptation des protocoles de communications (p. ex. DiPS [Janssens01], CANS [Fu01], x-kernel [Hutchins91]). Ensemble permet la construction des protocoles réseaux par empilement de plusieurs micro-protocoles choisis parmi une vaste collection d'algorithmes. Chaque micro-protocole réalise une tache particulière (p. ex. ordonnancement des messages, élection d'un coordinateur). Le protocole obtenu (p. ex. un protocole multicast ordonné et fiable) peut être adapté dynamiquement en reconfigurant la pile des micro-protocoles, soit par remplacement d'un micro-protocole par un autre, soit par ajout d'un nouveau micro-protocole (p. ex. le cryptage des messages). Cette reconfiguration veille entre autres à la garantie de la synchronisation entre les nouveaux micro-protocoles.

3.1.2.2 Services d'accès aux données adaptables

Plusieurs travaux se sont intéressés au problème d'accès aux données distribuées dans le cas de déconnexion ou de faible connectivité réseau (p. ex. Bayou [Terry98], Rover [Joseph97], Coda [Braam98], Odyssey [Noble99]). Les techniques proposées sont principalement basées sur la réplication et le préchargement des données sur les clients mobiles. Ainsi, comme le montre la figure 3.2, le système de fichiers distribués *Coda* s'adapte aux différentes conditions de connectivité (forte connectivité, faible connectivité et déconnexion) en préchargeant les fichiers susceptibles d'être utilisés dans un cache sur le client mobile. La gestion de cette adaptation est entièrement transparente aux applications. Quand la connexion est rétablie, une mise à jour des copies est entreprise. Celle-ci peut nécessiter une intervention humaine pour résoudre les conflits dus à

plusieurs modifications des mêmes données par plusieurs clients.

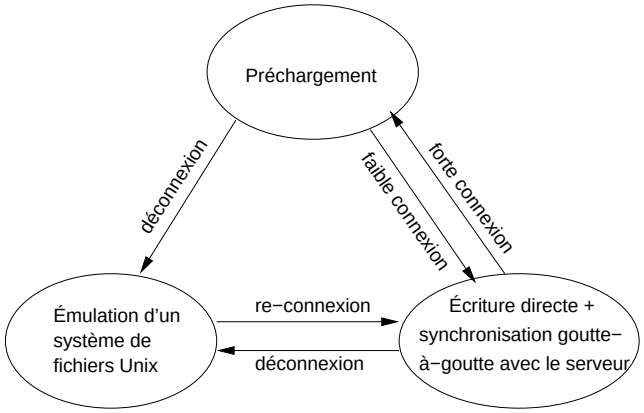


FIG. 3.2 – Gestion du cache dans le système de fichiers distribués *Coda*.

Contrairement à *Coda*, son successeur *Odyssey* [Noble99, Noble97] rend les applications conscientes de l'adaptation (c.-à-d. la gestion du cache). *Odyssey* est constitué d'un « vice-roi » (viceroy) qui s'exécute sur chaque client mobile et d'un « gardien » (warden) pour chaque type de données échangées sur le réseau. Le vice-roi est contrôlé par l'application. Il surveille les ressources et déclenche les actions d'adaptation quand c'est nécessaire. Le gardien prend en charge la modification d'un flux de données particulier.

3.2 Les intergiciels adaptables

Les intergiciels constituent l'infrastructure logicielle la plus utilisée pour mettre en œuvre l'adaptabilité aux variations de l'environnement [Mascolo02, Kon02]. Contrairement à certains intergiciels spécifiques qui s'adaptent de façon transparente aux applications (p. ex. *Coda*), beaucoup d'intergiciels exécutent eux mêmes les actions d'adaptation tout en laissant la prise de décision à la charge des applications (p. ex. *QuO* [Loyall98], *m3* [Rakotoni01], *Carisma* [Capra03]). D'autres encore, fournissent les mécanismes de surveillance et gèrent la prise de décision d'adaptation, mais délèguent l'exécution des actions d'adaptation aux applications (p. ex. intergiciel d'Efstratiou [Efstrati02]). Nous présentons ci-après des exemples représentatifs de ces intergiciels conçus suivant des approches différentes.

3.2.1 Approche basée sur les contrats de qualité de service

Quality Objects (*QuO*) [Loyall98, Schantz02] est un canevas d'objets qui supporte : (1) la spécification des contrats de qualité de service (QdS) entre les objets clients et leurs fournisseurs, (2) la surveillance de ces contrats en cours d'exécution et (3) l'adaptation aux changements des conditions système. Cette adaptation est assurée par une couche intergicielle (nommée aussi *QuO*, cf. figure 3.3) qui s'exécute au-dessus des

ORBs traditionnels comme CORBA ou Java RMI. Le canevas *QuO* est constitué des

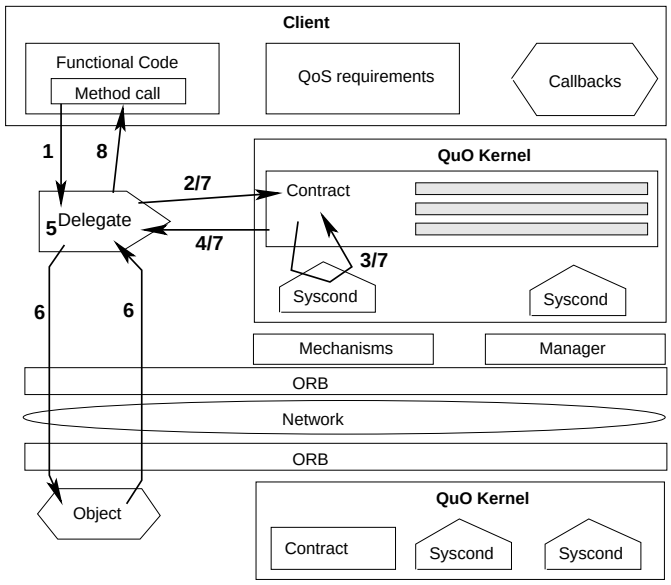


FIG. 3.3 – *QuO* s'exécutant au-dessus de CORBA.

éléments suivants :

- Les *contrats* qui spécifient le niveau de service désiré par un client, le niveau de service qu'un objet compte fournir (p. ex. le nombre d'appels de méthodes par seconde), les régions indiquant les valeurs possibles de la QdS, et les actions à entreprendre quand le niveau de la QdS change. Ils sont écrits par les spécialistes de la QdS dans le langage QDL (Quality Description Languages).
- Les *délégués* qui sont des représentants locaux pour les objets distants. À chaque appel client vers l'objet distant, le délégué vérifie l'état du système et sélectionne un comportement approprié.
- Les *objets condition du système* qui mesurent et contrôlent les objets applicatifs, les ORBs et les ressources système spécifiés dans les contrats.
- Les *gestionnaires de propriétés* particulières de la QdS telle la disponibilité des données via un mécanisme de réplication.
- Les *objets callback* qui fournissent des interfaces de notification pour les clients et les objets de l'application.

Une adaptation *QuO* peut être déclenchée de deux façons. Premièrement, les délégués réalisent des adaptations liées aux variations du réseau en interceptant les appels clients vers les objets distants. Ils peuvent ainsi agir sur l'interaction de ces objets en fonction de l'état du réseau. A titre d'exemple, le système d'avionique décrit dans [Loyall01] s'adapte aux paramètres d'un lien réseau restreint entre deux avions en modifiant la taille des données échangées sur ce lien.

Deuxièmement, les contrats *QuO* peuvent déclencher des adaptations indépendantes de l'état du réseau. Ces changements causent des transitions dans les régions de QdS

détectées par les objets condition du système. Dans ce cas l'adaptation est utilisée pour surveiller la charge CPU et aider à planifier les tâches temps réel d'avionique.

Discussion

Contrairement à la majorité des mécanismes de QdS, *QuO* ne se contente pas de satisfaire les besoins des applications en termes de QdS, mais il fournit une interface de callback qui informe les objets de l'application du niveau actuel de la QdS accomplie par rapport à la QdS désirée. Cela permet à l'application d'adapter son comportement suivant la logique spécifiée dans les contrats QDL. Ceux-ci jouent donc le rôle de politiques d'adaptation mais ils exécutent aussi eux même les actions d'adaptation indépendantes du réseau. Ceci constitue un handicap pour *QuO* car, bien que les adaptations liées au réseau sont faciles à mettre en ouvre grâce à une séparation entre les délégués et les contrats, la programmation des adaptations indépendantes du réseau n'est pas bien décrite. Ceci s'explique sans doute par le fait qu'elles n'étaient pas dans le modèle initial.

Les adaptations qui dépendent du réseau et celles qui n'en dépendent pas sont complètement orthogonales étant donné qu'elles sont contrôlées par deux entités distinctes (les délégués et les contrats). Dans ce cas, il n'est pas clair de savoir comment spécifier des adaptations sophistiquées qui prennent en compte à la fois l'état du réseau et d'autres ressources (p. ex. choisir un encodeur ayant un débit et une consommation CPU faible quand le réseau et la CPU sont très sollicités). De façon plus générale, la coordination des adaptations multi-applications (c.-à-d. de plusieurs délégués s'exécutant sur le même site) n'est pas abordée.

Enfin, notons que les objets condition du système sont assimilables à un service de surveillance de l'environnement. Mais même si l'interface de ces objets est définie par *QuO*, leurs implantations sont à la charge du développeur.

3.2.2 Approche basée sur les rôles des composants

L'intergiciel *m3* [Rakotoni01, Indulska01] adapte les applications clients-serveurs basées sur le modèle de référence RM-ODP [Rm-odp99] qui s'adresse à la gestion des entreprises. Les serveurs sont encapsulés par l'exécutif *m3* qui modifie et coordonne les messages échangés avec les clients en fonction du contexte d'exécution. L'architecture de *m3* repose sur les trois concepts suivants :

1. Les *composants* logiciels réactifs qui ont des permissions de rôles, des pré/post-conditions associées à leurs interfaces et des types d'événements qui déclenchent leur réaction.
2. Les *événements* qui sont des concepts primitifs à l'aide desquels peuvent être exprimés des protocoles d'interaction comme le passage de messages, les RPCs et les flux.
3. Les *rôles des composants* qui expriment les tâches que les composants doivent remplir dans l'organisation d'une entreprise (p. ex. directeur). Chaque rôle est un emplacement qui peut être occupé par une ou plusieurs interfaces.

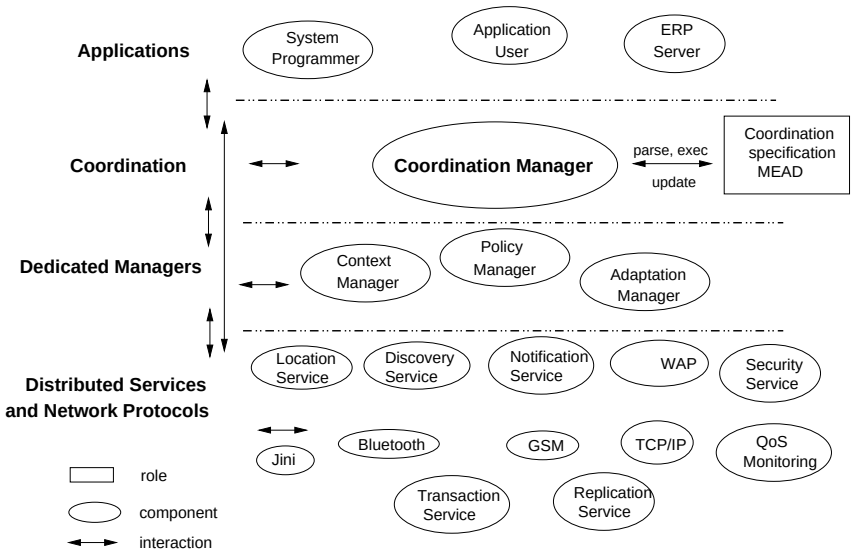


FIG. 3.4 – Intergiciel *m3*.

La figure 3.4 montre l'architecture de l'intergiciel *m3* organisée en trois couches : la couche coordination, la couche des gestionnaires dédiés et la couche service.

Le *gestionnaire de la coordination* exécute les scripts *MEADL* (Mobile Enterprise Architecture Description Language) qui coordonnent les messages (événements) échangés par les composants. Ces scripts contiennent des règles qui associent un ensemble d'opérations à un événement. Ceci permet de configurer dynamiquement les interactions entre ces composants en modifiant les messages échangés et le protocole de communication utilisé. Par exemple la règle de la figure 3.5 réagit à l'événement `http :get(balance, account)` émis par le rôle `Director`. Si le contexte courant est sûr et la balance est supérieure à 100000 alors cette règle émet un nouvel événement indiquant la valeur de la balance et appelle la fonction `foo`.

```
ON EVENT Director:http:get(balance, account)
   WHERE (balance > 10,000)
   IN CONTEXT   "secure_environment"
 {
   EMIT employee_chat_line("DSTC has", balance);
   CALL foo(account, 23);
 }
```

FIG. 3.5 – Exemple de script *MEADL*.

La *couche des gestionnaires dédiés* contient :

1. Le *gestionnaire de contexte (CM)* qui rassemble et interprète les informations contextuelles à partir de capteurs matériels et logiciels, et les présente au *gestionnaire de l'adaptation (AM)*. Pour cela, le CM utilise les graphes RDF (Resource Description Framework) basés sur le protocole W3C CC/PP pour la description et la transmission des profils de périphériques.

2. L'AM prend les décisions d'adaptation et invoque ou installe les méthodes d'adaptation appropriées. Il utilise les règles du type «Evénement–Condition–Action » (ECA) où le champ `Action` référence une méthode d'adaptation implantée par l'AM lui-même ou par l'application (p. ex. insertion de filtres, migration d'objets). Les règles d'adaptation peuvent être ajoutées, enlevées, inhibées ou activées dynamiquement. Elles peuvent être aussi coordonnées de façon séquentielle à l'aide d'une chaîne de dépendances (c.-à-d. exécuter une adaptation après une autre). Cette chaîne de dépendances est exprimée sous la forme d'un graphe, ce qui permet d'éviter les adaptations de type ping-pong en utilisant la détection de cycles.
3. Le *gestionnaire de politiques (PM)* assure que les politiques décrivant les obligations, interdictions et permissions associées à un rôle sont respectées avant et après chaque adaptation.

La *couche service* rend les services distribués et les protocoles réseaux accessibles aux couches supérieures de façon uniforme basée sur le paradigme événement. En particulier, elle permet d'utiliser les protocoles de communications de façon transparente à travers une interface générique inspirée du modèle Linda (avec les méthodes `in(from,data,attr)` et `out(to,data,attr)`).

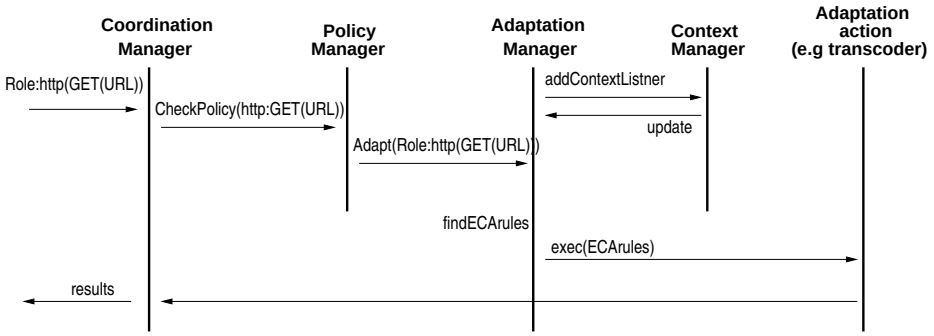


FIG. 3.6 – Interaction entre les composants *m3*.

La figure 3.6 illustre un exemple de traitement d'une requête `http(GET(URL))` d'une application par un exécutif *m3*. Le *gestionnaire de la coordination* reçoit la requête et la fait suivre au *PM* pour contrôler si son exécution est autorisée. Si l'autorisation est accordée la requête est transmise à l'*AM* qui est constamment informé par le *CM* sur le contexte de l'application. L'*AM* sélectionne une règle d'adaptation en fonction du contexte et exécute l'action spécifiée (p. ex. transcoder une image). La réponse est envoyée immédiatement au *gestionnaire de la coordination* qui la fait suivre vers l'application.

Discussion

L'architecture *m3* n'aborde pas les adaptations intra-composant mais elle considère seulement les adaptations inter-composants dans le cas des applications d'entreprise de

type clients-serveurs. *M3* facilite la programmation de ces adaptations grâce à la séparation entre la surveillance de l'environnement assurée par le *CM*, la prise de décision gérée par le *gestionnaire de la coordination* et *l'AM*, l'exécution des modifications implantées par l'*AM* ou par l'application, et la vérification de la cohérence des adaptations par rapport aux rôles des composants qui est effectuée par le *PM*. Ce dernier aspect, qui est souvent négligé dans les travaux concernant l'adaptabilité, est une contribution importante de *m3*.

Par ailleurs, la méthode d'adaptation proposée est basée sur l'interception des messages du coté du serveur, et ce de façon transparente pour le client. Bien que des modifications unilatérales du contenu de ces messages soient possibles, le changement dynamique du protocole d'interaction sans modification du client ne semble pas envisagé.

Enfin, dans *m3* une première forme de coordination de plusieurs adaptations a été introduite, mais elle est limitée à une mise en séquence de ces adaptations. La coordination de plusieurs adaptations d'applications concurrentes n'a pas encore été abordée.

3.2.3 Approche gérant les conflits d'adaptation

Carisma est un intergiciel réflexif qui rend les applications mobiles conscientes de leur contexte [Capra03]. Il leur fournit des services (p. ex. affichage d'images) qui sont dynamiquement spécialisables en fonction du contexte courant. Chaque application peut spécialiser un service à travers un *profil d'application* qui associe les différents modes (politiques) de livraison de ce service aux différentes configurations du contexte. Dans l'exemple de la figure 3.7, l'application définie deux associations demandant au service `DisplayPicture` d'afficher les images en noir et blanc si l'énergie de la batterie est faible ou en couleur si la batterie le permet. À chaque invocation d'un service par une application, l'intergiciel consulte son profil pour déterminer la politique à appliquer en fonction du contexte courant.

Le comportement de l'intergiciel par rapport à un service particulier est déterminé par une et une seule politique à la fois. Les profils sont passés à l'intergiciel au moment de l'installation des applications mais celles-ci peuvent les consulter et les modifier dynamiquement en utilisant un API réflexif. Par exemple, une application peut restreindre l'utilisation de la politique `FullColor` dans un contexte où seule l'énergie de la batterie est haute à un contexte où la mémoire disponible est grande.

Quand un service est utilisé par plusieurs applications, un conflit peut exister si, dans le même contexte, plusieurs politiques différentes peuvent être appliquées pour délivrer ce service. L'intergiciel ne sait pas dans ce cas quelle politique appliquer. Pour résoudre ce conflit dynamiquement, Mascolo et coll. ont proposé une méthode basée sur un protocole d'enchères où l'intergiciel joue le rôle du commissaire-priseur, les applications sont les enchérisseurs, et les politiques sont les biens mis aux enchères. Toutefois, l'objectif de l'intergiciel n'est pas de sélectionner la politique qui a reçu l'enchère la plus élevée mais celle qui satisfait le plus grand nombre d'applications impliquées dans le conflit.

Le calcul des offres est basé sur l'intérêt porté par l'utilisateur pour les besoins

```
<SERVICE name="displayPicture">
    <POLICY name="Black&WhiteLoad">
        <RESOURCE name="battery">
           <STATUS operator="less" value="x"/>
        </RESOURCE>
    </POLICY>
    <POLICY name="FragmentedLoad">
        <RESOURCE name="memory">
           <STATUS operator="less" value="y"/>
        </RESOURCE>
    </POLICY>
</SERVICE>
```

FIG. 3.7 – Exemple de profil d'application s'exécutant sur *Carisma*.

non fonctionnels des applications (p. ex. la sécurité, la performance, la disponibilité, etc.) Pour cela, des fonctions d'utilité sont utilisées pour traduire cet intérêt en valeurs numériques. Ces fonctions d'utilité peuvent être modifiées dynamiquement pour refléter d'éventuels changements dans les intérêts de l'utilisateur.

La résolution d'un conflit distribué est basée sur un protocole de communication à trois pas :

- **Pas 1** : À chaque invocation d'un service par une application sur un site donné (appelé ci-après site initiateur), l'intergiciel de ce site envoie une requête vers les autres sites impliqués dans l'exécution de ce service. En parallèle, chacun de ces sites évalue son contexte local, détermine les politiques autorisées et calcule son offre pour chacune de ces politiques, bien qu'aucun conflit n'ait été encore détecté.
- **Pas 2** : Tous les sites renvoient au site initiateur leurs ensembles de politiques autorisées avec les offres correspondantes. Si l'intersection de ces ensembles est vide, alors le service demandé ne sera pas délivré faute d'accord entre les applications sur la politique à appliquer. Sinon, l'initiateur détermine la politique gagnante sans communication supplémentaire.
- **Pas 3** : Le site initiateur envoie son résultat vers les autres sites et enfin le service requis peut être délivré.

Discussion

L'intergiciel *Carisma* supporte des applications qui ne sont pas forcément basées sur les composants logiciels. Ainsi la seule adaptation possible est la spécialisation des services fournis par l'intergiciel en sélectionnant le mode de fonctionnement le plus appropriée au contexte courant. *Carisma* ne dispose pas d'un mécanisme de notification des variations des ressources. Il se contente de consulter l'état de celles-ci uniquement au moment où il sélectionne le mode d'un service quand ce dernier est invoqué. Dans ce cas, des questions importantes restent suspendues quant à la possibilité de changer un mode de fonctionnement en cours d'exécution, par exemple suite au changement du contexte courant, et la possibilité d'ajouter dynamiquement un nouveau mode à un service existant.

Par ailleurs, dans tous les travaux que nous avons présentés précédemment, les décisions d'adaptation étaient prises au niveau local à un seul site, même si elles prenaient

en compte l'état des liens réseaux de ce site. *Carisma* est l'un des premiers travaux (avec notamment *Conductor*, cf. section 3.3.1) à considérer le cas plus général où la décision d'adaptation est une procédure distribuée qui tient compte des environnements d'exécution sur plusieurs sites. Un avantage important de *Carisma* est le recours à un protocole à trois pas qui minimise les communications nécessaires pour une prise de décision distribuée, ce qui est non négligeable dans le cadre des réseaux mobiles. Toutefois, si le succès de ce protocole n'est pas compromis par la défaillance de l'un des sites participants (tant qu'il y a un certain nombre de sites connectés), il est totalement voué à l'échec si le site initiateur tombe en panne.

Sur un autre plan, la prise en compte des préférences de l'utilisateur au moment de la résolution d'un conflit et la possibilité d'adapter le mécanisme de résolution (c.-à-d. les fonctions d'utilité) si ces préférences change est une contribution majeure de *Carisma*. Notons que cette procédure de résolution de conflit est une coordination des adaptations de plusieurs applications qui utilisent le même service. Le cas plus général de la coordination des adaptations d'applications partageant seulement des ressources n'est pas abordé. Enfin, *Carisma* respecte le principe de séparation des différents types de codes impliqués dans l'adaptation.

3.2.4 Approche basée sur le calcul d'événements

La plate-forme proposée dans [Efstrati02] est un intergiciel qui supporte la coordination des actions d'adaptation au niveau multi-applications. Elle est basée sur la notion de politiques système écrites dans un langage dérivé du *calcul d'événements* de Kowalsky [Kowalsky92]. La figure 3.8 montre la conception de cette plate-forme.

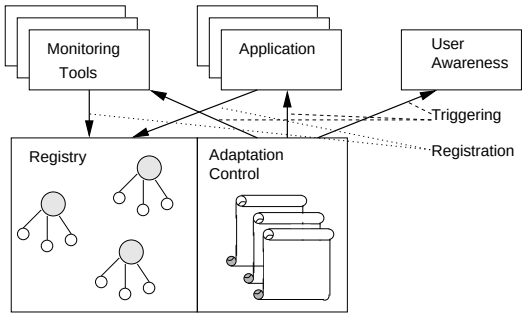


FIG. 3.8 – Plate-forme d'Efstratiou et coll. pour l'adaptation coordonnée.

Le registre agit comme un entrepôt pour les outils de surveillance et pour toutes les applications adaptables s'exécutant sur le système. À son démarrage, chaque application adaptable s'enregistre auprès de la plate-forme en lui fournissant une description XML contenant : (1) ses différents modes d'adaptation, (2) l'ensemble de ses variables d'état accessibles par la plate-forme, et (3) l'ensemble des politiques par défaut qui décrivent son comportement adaptatif. En cours d'exécution, de nouvelles politiques peuvent être introduites par l'utilisateur de l'application.

Le registre utilise ces informations en associant à chaque application un objet souche

qui traque ses variables d'état ainsi que l'état courant du système. Quand un changement ce produit, la souche émet un événement vers le contrôleur de l'adaptation pour décider si une réaction adaptative est nécessaire. Comme les applications, les outils de surveillance sont gérés à la fois comme des sources d'informations et des modules adaptables. Ils peuvent fournir des mécanismes d'adaptation spécifiques à un périphérique particulier, p. ex. mettre le périphérique dans un mode de consommation d'énergie faible.

Pour la spécification des politiques d'adaptation Efstratiou et coll. ont développé un langage dérivé du calcul d'événements [Kowalsky92]. Ce formalisme de la programmation logique permet de raisonner sur les effets des événements durant le temps en introduisant la notion de *fluent*. Un fluent peut décrire une situation au sein d'un système (p. ex. batterie faible) qui a une durée dans le temps et qui est initiée et terminée par des événements. Les règles des politiques d'adaptation basées sur ce calcul sont de la forme *événement–fluent–condition–action*. La figure 3.9 montre l'exemple d'une politique qui préconise d'utiliser une connexion réseau GSM quand l'utilisateur est à l'extérieur.

```
Events LeftHome, LeftOffice, InHome, InOffice
Fluent Outdoors :
  initiated by events: LeftHome, LeftOffice
  terminated by events: InHome, InOffice
Condition :
  Initiated(Outdoors)
Action:
  Switch network to GSM
```

FIG. 3.9 – Exemple de politique d'adaptation basée sur les *fluents*.

Le contrôleur de l'adaptation surveille le comportement de toutes les applications s'exécutant sur un seul hôte et déclenche les adaptations nécessaires décrites dans les politiques enregistrées auprès de la plate-forme. Il évalue les règles d'adaptation de façon incrémentale à chaque changement dans l'environnement. Quand tous les prédicats d'une règle sont évalués et que la condition qu'ils forment est vraie, le contrôleur d'adaptation déclenche la méthode appropriée de l'application telle que spécifiée par la règle.

Le module *user awareness* est implanté séparément comme une application qui s'enregistre auprès de la plate-forme. Le contrôleur d'adaptation utilise ce module pour notifier à l'utilisateur les décisions d'adaptation ou pour demander son assistance quand il détecte des situations conflictuelles.

Cette plate-forme a été utilisée pour développer un client vidéo (basée sur RealPlayer) et un navigateur Web adaptables. Le client vidéo permet à la plate-forme de demander des adaptations du flux vidéo en fonction des conditions réseau, comme le changement de la qualité du flux vidéo ou le changement de son taux de bits. Le navigateur Web peut contrôler le taux de transfert de données sur le réseau en filtrant les images présentes dans les pages Web. Les adaptations de ces deux applications peuvent être coordonnées par la plate-forme qui alloue plus de bande passante à l'une ou à l'autre selon les préférences de l'utilisateur.

Discussion

L'intergiciel d'Efstratiou et coll. met en œuvre une séparation des différents types de codes impliqués dans l'adaptation. Comme *Carisma* cet intergiciel n'est pas spécifique aux applications basées sur les composants logiciels. Par contre, ses actions d'adaptation sont déléguées aux applications via un mécanisme de callback. Ceci rend ces adaptations arbitraires et donc impossibles à analyser (p. ex. pour vérifier leur cohérence). Un autre point commun de ces deux intergiciels est l'implication de l'utilisateur dans la résolution des situations conflictuelles. Cependant, la plate-forme d'Efstratiou permet de plus d'informer continuellement l'utilisateur des adaptations qu'elle entreprend.

Par ailleurs, l'une des contribution clé de la plate-forme d'Efstratiou est l'externalisation des décisions d'adaptation (par rapport aux applications) et leur centralisation au niveau intergiciel. Ceci permet d'utiliser les événements générés par une application pour déclencher l'adaptation d'une autre, et surtout de coordonner les adaptations de plusieurs applications et résoudre des conflits éventuels. Mais à ce stade le mécanisme de résolution de conflit n'a pas encore été développé.

Parmi tous les travaux présentés dans cet état de l'art, les politiques d'adaptation d'Efstratiou et coll. sont les plus sophistiquées. Ce sont les seules qui permettent de raisonner sur l'effet des événements durant le temps et pas uniquement à des instants précis. Notamment, il est possible de coordonner des adaptations multi-applications ayant des relations temporelles quelconque (parallèles ou séquentielles).

3.2.5 Bilan sur les intergiciels adaptables

Les intergiciels participent à la gestion de l'adaptabilité du logiciel en mettant en œuvre une partie des mécanismes d'adaptation. Ils offrent un service de surveillance des ressources prêt à l'usage et coopèrent avec les applications dans la prise de décision d'adaptation et/ou l'exécution des actions de modification requises. Dans le cas d'adaptations multi-applications distribuées, le rôle de l'intergiciel devient capital quand il s'agit de coordonner de façon efficace, équitable et sans conflit ces adaptations (p. ex. intergiciel d'Efstratiou et *Carisma*). En effet, aucune application n'est apte à jouer ce rôle d'arbitre neutre sans connaître au moins les politiques d'adaptation de ses concurrentes. A notre avis, l'intergiciel est donc le meilleur candidat pour assurer ce rôle de façon naturelle.

3.3 Les proxies

Les travaux basés sur l'approche proxy concernent principalement l'adaptation aux variations des conditions réseau pour les applications clients-serveurs [Fox98b, Fox97, Yarvis01a, Layaida02]. L'idée de base consiste à injecter dans le chemin séparant le client et le serveur un ou plusieurs proxies qui adaptent le flux des données transitant à travers le réseau. Ces proxies modifient les données échangées ou les chemins qu'elles empruntent en réalisant des opérations comme le filtrage, le transcodage, la compression, le cryptage, le préchargement, la mise en tampon, la redirection ou l'ordonnancement

des transmissions basé sur la définition de classes de priorité entre ces données. Ces opérations permettent de palier les variations des paramètres réseaux tels que la bande passante, la latence, la gigue, le taux d'erreurs de transmission, etc.

Parmi les approches proxy nous choisissons de détailler les systèmes *Conductor* [Yarvis01a] et *Puppeteer* [de Lara02a] car ils abordent le problème de la coordination de plusieurs adaptations.

3.3.1 Proxies sur les nœuds intermédiaires du réseau

Conductor [Yarvis01a, Yarvis01b] est un système de proxies supportant l'adaptation transparente coordonnée et distribuée des protocoles applicatifs par rapport aux conditions du réseau. Il se déploie sur un sous-ensemble de nœuds d'un réseau (appelés *nœuds Conductor*) qui sont typiquement des clients, des serveurs ou des passerelles entre des sous-réseaux hétérogènes. *Conductor* se compose d'un canevas d'adaptation et

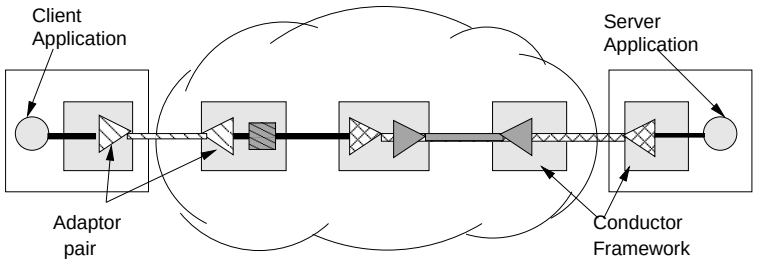


FIG. 3.10 – Architecture de *Conductor*.

de plusieurs modules adaptateurs (cf. figure 3.10). Le canevas d'adaptation fournit au niveau de chaque nœud *Conductor* un environnement d'exécution (cf. figure 3.11) pour accueillir un ou plusieurs modules adaptateurs. Cet exécutif réalise (1) l'interception des connexions client-serveur (à partir du numéro de port au niveau Socket), (2) la surveillance des ressources du nœud et de ses liens réseaux, et (3) la planification et le déploiement des modules adaptateurs.

Les modules adaptateurs sont des modules Java écrit entièrement par le développeur pour opérer des modifications sur le flux de données transitant à travers les nœuds *Conductor*. Ils sont généralement déployés par paires à l'entrée et à la sortie d'un sous-réseau pour adapter le protocole de communication aux caractéristiques particulières de celui-ci.

Processus d'adaptation

Quand un nœud *Conductor* intercepte une connexion il identifie les autres nœuds *Conductor* se trouvant sur le chemin de la destination. Ensuite une connexion TCP est établie entre chaque paire de nœuds adjacents et la planification est lancée pour sélectionner les nœuds d'adaptation et les adaptateurs à déployer. Cette planification se fait en trois étapes : collecte d'informations, formulation d'un plan et déploiement des adaptateurs. Elle prend en considération les préférences de l'utilisateur (p. ex. qualité

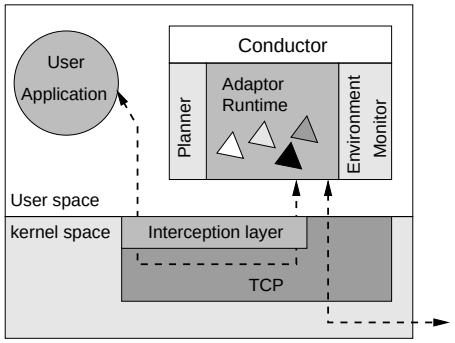


FIG. 3.11 – Nœud *Conductor*.

des données, budget, etc.), les ressources du nœud local et les caractéristiques du lien réseau. L'algorithme de planification utilisé dans *Conductor* est un algorithme centralisé avec un seul aller-retour (cf. figure 3.12).

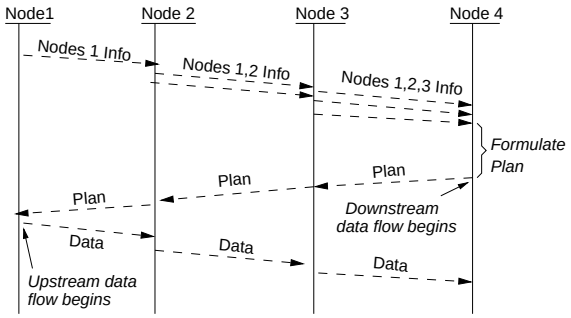


FIG. 3.12 – Planification du redéploiement des adaptateurs dans *Conductor*.

Quand la planification commence, tous les nœuds sur le chemin des données transmettent leurs informations locales vers un nœud planificateur unique (le nœud le plus proche du serveur). Ce dernier formule un plan qu'il transmet en un aller-retour vers le nœud client, en passant par tous les autres nœuds du chemin. Cette transmission s'accompagne du déploiement des adaptateurs sur chacun des nœuds sélectionnés. Dès qu'un adaptateur est déployé sur un nœud les données de l'application peuvent transiter sur ce nœud dans la direction du déploiement.

En cours d'exécution, les variations mineures dans l'environnement sont gérées par les modules adaptateurs. Si une variation est trop importante, l'adaptateur qui ne peut la gérer initie une nouvelle planification pour redéployer les modules adaptateurs.

Conductor supporte la tolérance aux fautes en utilisant un mécanisme de retransmission à la TCP. Toutefois, comme les données transmises peuvent être modifiées par les adaptateurs, *Conductor* n'utilise pas l'octet comme unité de retransmission mais un *segment de données*. Ce dernier est une unité de données qui préserve la correspondance entre les flux en entrée et en sortie d'un adaptateur (p. ex. une trame vidéo pour un adaptateur qui réalise la compression vidéo). *Conductor* inclut un mécanisme d'authen-

tification et de cryptage des données échangées lors de la planification des adaptations afin d'éviter les intrusions qui visent à rediriger les données vers des nœuds non sûrs.

Conductor a été utilisé pour adapter une connexion, reliant un PDA à un serveur, formée de trois liens successifs : sans-fil, modem et Internet. Dans cette situation deux adaptations ont été mises en place. La première consiste à compresser les données transitant par le modem pour minimiser le temps de transmission. La seconde est basée sur une estimation du temps nécessaire au traitement par le serveur d'une requête émise par le PDA et elle consiste à mettre en veille l'interface sans-fil du PDA en attendant la réception de la réponse du serveur afin d'économiser l'énergie.

Discussion

Une contribution importante de *Conductor* est la prise en compte des connexions réseaux composées de plusieurs segments hétérogènes. Il offre aussi un exécutif qui assure la surveillance des ressources sur chaque nœud du réseau. Cependant, la technique d'adaptation proposée est limitée à la modification des flux de données transitant sur le réseau. Ces modifications doivent être écrites en Java par les développeurs de l'application. Par conséquent, elles sont arbitraires et donc impossibles à analyser.

Par ailleurs *Conductor* a l'avantage de supporter la tolérance aux fautes et la sécurisation des échanges durant la prise de décision d'adaptation. Cette décision est réalisée par une procédure de planification centralisée. Il tient compte de l'état global du système distribué, limite les échanges nécessaires mais reste vulnérable à la défaillance du site planificateur.

Sur un autre plan, la planification des adaptateurs constitue une forme de coordination au niveau application car elle est limitée à un seul flux. Le cas plus général de coordination de plusieurs adaptateurs déployés sur le même nœud mais intervenant chacun sur un flux de données différent n'est pas abordé.

Enfin, indépendamment de l'adaptabilité, l'idée de permettre aux nœuds intermédiaires du réseau (les routeurs) d'intervenir aux niveaux des couches supérieures de la pile des protocoles est sujet à controverse. Elle présente plusieurs inconvénients sur les plans de la performance et de la fiabilité.

3.3.2 Proxies aux extrémités du réseau

Puppeteer [de Lara01, de Lara02a] est un système de proxies qui adapte les applications bureautiques (p. ex. éditeur de documents distants et navigateur Web) à la variation de la bande passante dans les environnements mobiles. Quand la bande passante devient peu abondante, *Puppeteer* permet aux applications manipulant des documents stockés sur un serveur distant soit d'accéder seulement à une partie d'un document (p. ex. la première diapositive d'une présentation), soit d'accéder à une version complète de ce document mais de qualité réduite (p. ex. une version faible-fidélité d'une image). La figure 3.13 montre l'architecture de *Puppeteer* constituée de proxies locaux et distants qui gèrent l'adaptation. Un proxy *Puppeteer* est constitué d'un noyau indépendant des applications, de pilotes spécifiques aux éléments d'un document, de transcodeurs et

de politiques d'adaptation spécifiques à chaque application. L'exécution des politiques d'adaptation est à la charge du proxy local. Le proxy distant est responsable de l'analyse des documents, de l'exposition de leurs structures et du transcodage de leurs éléments suivant les requêtes du proxy local.

Pour être adaptée avec *Puppeteer*, les applications doivent exporter une interface (API) qui permet de découvrir, de construire de façon incrémentale et d'afficher des sous-ensembles des éléments d'un document ainsi que leurs différentes versions. Elles doivent aussi exporter via leur API un mécanisme d'enregistrement et de notification des événements afin d'avertir le système d'adaptation des variations importantes dans l'application ou dans le comportement de l'utilisateur.

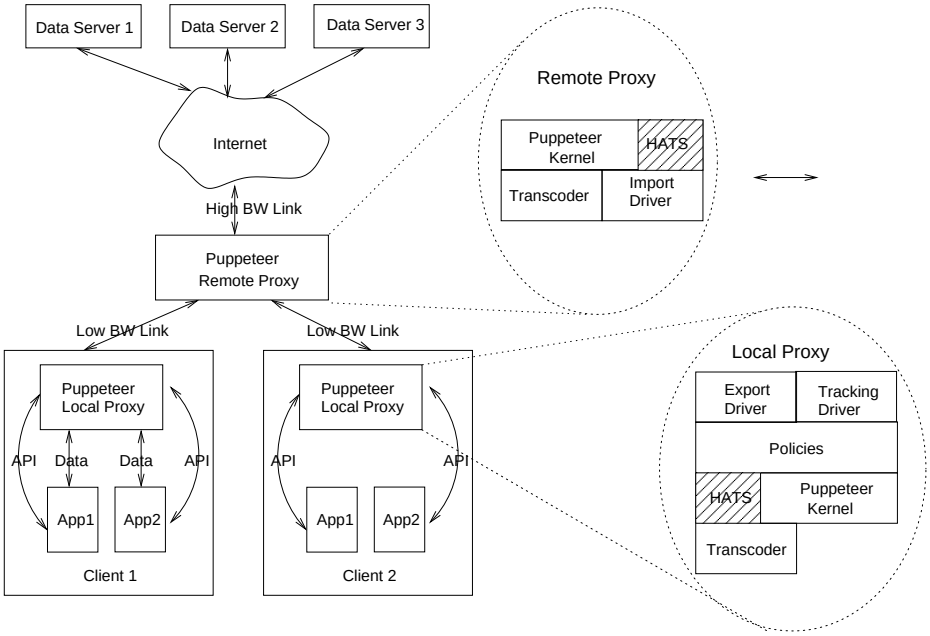


FIG. 3.13 – Architecture de *Puppeteer*.

Processus d'adaptation

Le processus d'adaptation dans *Puppeteer* est divisé en trois phases : (1) analyse du document et découverte de la structure des données, (2) récupération des éléments sélectionnés à des niveaux de fidélité spécifiques et (3) mise à jour de l'application avec les nouvelles données récupérées. Quand un document stocké sur un serveur est ouvert, le proxy *Puppeteer* distant créé un pilote d'import approprié. Ce pilote parcours le document, extrait son squelette sous la forme d'un arbre, et transfert ce dernier vers le proxy local. Les politiques d'adaptation s'exécutant dans le proxy local récupèrent alors un ensemble initial des éléments du squelette avec une fidélité spécifiée. Une fois reçu, cet ensemble initial est passé à l'application par le proxy local qui utilise pour cela un pilote d'export approprié. À ce moment l'application retourne le contrôle à l'utilisateur.

Ensuite, la politique d'adaptation utilise l'API de l'application pour l'alimenter de façon incrémentale avec le reste des éléments du document ou avec des versions de plus grande fidélité.

Les politiques d'adaptation peuvent être statiques ou peuvent dépendre de pilotes traqueurs qui détectent l'occurrence d'événements particuliers. Par exemple le passage de la souris sur une image peut être un événement qui déclenche le chargement d'une version haute-fidélité de celle-ci.

Adaptation multi-applications

Dans [de Lara02b] *Puppeteer* a été étendu pour supporter l'adaptation de plusieurs applications concurrentes sur un hôte. Cette extension appelée *HATS (Hierarchical Adaptive Transmission Scheduler)* implante des politiques d'adaptation qui contrôlent dynamiquement l'ordonnancement des transmissions réseau. Cet ordonnancement est basé sur l'affectation de priorités différentes aux applications, aux documents manipulés par chaque application et aux éléments de chaque document. Des exemples de ces politiques incluent la transmission du texte avant les images pour toutes les applications, l'allocation de la plus grande partie de la bande passante à l'application (au document) qui est en avant plan et la réduction de la fidélité des applications qui sont en arrière plan.

HATS est incorporé principalement dans le proxy *Puppeteer* distant. La figure 3.14 montre son architecture constituée par :

- Une hiérarchie de transmission qui décrit la structure des données à transmettre. Elle est constituée à raison d'une hiérarchie par hôte client en réponse aux requêtes provenant des politiques s'exécutant sur le proxy local.
- Les ordonnanceurs qui déterminent l'ordre de transmission des nœuds de la hiérarchie. Il y a un ordonnanceur pour le nœud racine afin de choisir quelle application servir, un ordonnanceur pour chaque application afin de choisir quel document transmettre et un ordonnanceur pour chaque composant dans le document. La bande passante est distribuée du haut de la hiérarchie vers le bas ce qui donne la priorité aux ordonnanceurs des nœuds parents sur ceux des descendants.
- Le multiplexeur et le démultiplexeur organisent la transmission concurrente des données associées à plusieurs nœuds de la hiérarchie.

HATS supporte la reconfiguration dynamique des priorités de transmission par des requêtes explicites émanant des politiques d'adaptation sur les proxies locaux ou émanant des ordonnanceurs s'exécutant sur le proxy distant.

Discussion

Le fait que *Puppeteer* soit transparent aux applications (clients et serveurs) lui impose des restrictions sévères. Les adaptations supportées sont restreintes à la transmission sélective des données du serveur vers le client et ne tiennent compte que des variations de la bande passante. La surveillance des autres variations de l'environnement est laissée à la charge des applications. Par ailleurs, les politiques d'adaptation

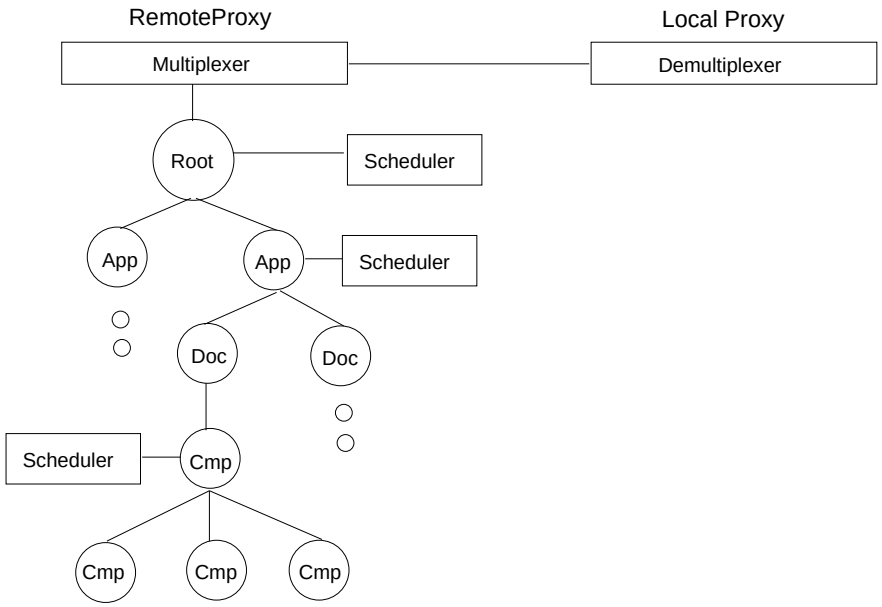


FIG. 3.14 – Adaptation multi-applications avec *HATS*.

sont spécifiques à des types de documents précis et ne sont donc pas écrites ni modifiées par les développeurs des applications.

Sur un autre plan, les applications adaptées par *Puppeteer* doivent exporter une interface spécifique permettant d'exposer la structure de leurs documents. Or la majorité des applications légataires (p. ex. MS Word) ne fournit pas une telle interface et son ajout requiert un effort de modification non négligeable.

La contribution majeure de *Puppeteer* est apportée par l'extension *HATS* qui coordonne les adaptations de plusieurs applications concurrentes par rapport à la bande passante. La solution proposée est basée sur la définition de priorités entre ces applications comme dans les systèmes d'exploitation. Cependant, elle n'est pas généralisable à plusieurs ressources car les priorités dans ce cas peuvent entraîner des situations d'interblocage.

3.3.3 Bilan sur l'approche proxy

Les systèmes à base de proxies réalisent, pour la grande majorité, une adaptation transparente aux applications. Toutefois, ils diffèrent par le nombre de proxies déployés, par l'emplacement de ceux-ci et par la nature des modifications qu'ils opèrent. Les plus simples utilisent un seul proxy du coté client comme [Layaida02]. D'autres travaux plus sophistiqués, tels que *Conductor* et *Puppeteer*, déploient plusieurs proxies entre les clients et les serveurs.

Les techniques d'adaptation basées sur l'approche proxy ont été conçues pour répondre aux variations des conditions réseaux. La plupart sont des techniques *ad hoc* qui ont été développées pour des applications spécifiques (p. ex. Web Browser [Fox98a])

et des environnements d'exécution particuliers (p. ex. les réseaux sans fil [Katz96]). Par conséquent, elles ne sont pas généralisables à d'autres types d'applications et sont difficilement transposables à d'autres plates-formes d'exécution.

3.4 Conclusion

Les infrastructures logicielles adaptables de type proxies, intergiciels, services distribués spécifiques et systèmes d'exploitation flexibles et extensibles, permettent d'adapter les aspects d'exécution de bas niveau (p. ex. ordonnancement, gestion de la mémoire) et les aspects non fonctionnels (p. ex. communication, accès aux données). Bien que ces supports soient nécessaires, ils sont néanmoins insuffisants quand il s'agit d'adapter les besoins fonctionnels des applications. Ces besoins spécifiques et variés relèvent du code métier des applications et donc ne peuvent être implantés au niveau système. Ainsi, plusieurs travaux de recherche se sont intéressés aux développement d'applications qui sont elles-mêmes adaptables. Ces travaux que nous présentons dans le chapitre suivant confirment l'idée que l'adaptabilité doit être implantée à tous les niveaux logiciels.

Chaptire 4

Adaptabilité au niveau des applications

Dans ce chapitre nous dressons un panorama des techniques d'adaptation des applications distribuées. Nous avons classé les travaux de recherche utilisant ces techniques en fonction des approches de programmation : la théorie du contrôle, les aspects, l'architecture logicielle et les composants logiciel. Pour chaque approche nous donnons un ou plusieurs exemples de travaux représentatifs. Nous concluons cet état de l'art par une synthèse qui compare les avantages et les inconvénients des travaux cités, discute les apports des différentes approches de programmation pour l'adaptabilité et justifie le choix de l'approche que nous avons fait pour traiter ce problème.

4.1 Adaptabilité suivant l'approche contrôle

La *théorie du contrôle* a été appliquée avec succès dans plusieurs disciplines d'ingénierie [Dutton97]. Elle a été aussi utilisée pour la construction de systèmes logiciels auto-adaptables [Karsai01, Meng00, Goel99]. De façon générale, les ingénieurs qui conçoivent un contrôleur font des suppositions concernant les propriétés dynamiques du système à contrôler. Mais ces suppositions peuvent ne pas être valides tout au long du cycle de vie de ce contrôleur. Donc, il risque de mal fonctionner quand les circonstances changent. Pour cela, un composant d'adaptation est introduit pour ajuster le contrôleur. La figure 4.1 montre l'architecture générique d'un contrôleur auto-adaptable. Elle est constituée de deux boucles de rétroaction : la première relie le système et le contrôleur et la seconde contient le composant d'adaptation. Ce dernier reçoit les données du système et recalcule les paramètres du contrôleur en utilisant un modèle mathématique.

Discussion

Les modèles mathématiques adoptés par les systèmes de contrôle ne conviennent pas généralement pour le contrôle des processus et des produits logiciel [Aksit03]. Une

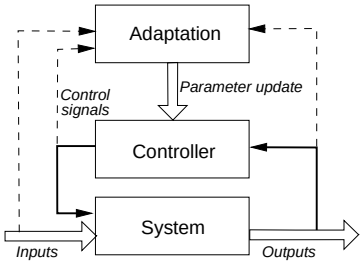


FIG. 4.1 – Architecture générique d'un contrôleur auto-adaptable.

grande partie des systèmes de contrôle supposent que les variations de l'environnement d'exécution sont des exceptions qui causent des erreurs faciles à corriger durant la phase opérationnelle du système. A partir de là, ils ne peuvent faire des adaptations non anticipées et les seules modifications qu'ils réalisent sont paramétriques (c.-à-d. non structurelles). Ceci limite considérablement leur utilité pour la construction de logiciels auto-adaptables.

4.2 Adaptabilité suivant la programmation orientée aspect

Plusieurs travaux de recherche ont développé des techniques de tissage et de composition dynamique des aspects pour adapter les applications en cours d'exécution [Truyen02b, Pawlak02, Heuzerot01, Bergmans01, Aksit96]. À titre d'exemple nous présentons le modèle *Lasagne* qui met en œuvre cette technique pour l'adaptation des applications clients-serveurs.

Lasagne [Truyen02b, Truyen01] est un mécanisme d'extension des composants par la notion d'aspect dynamique. Ces composants collaborent, en s'appelant les uns les autres, pour implanter les propriétés fonctionnelles des applications. Comme le montre la figure 4.2, chaque aspect concerne plusieurs composants et est conçu comme un ensemble d'enveloppes (wrappers) qui implantent un besoin non fonctionnel (p. ex. compression des données). À leur tour, ces enveloppes peuvent être aussi encapsulées par d'autres enveloppes. En fonction du contexte d'exécution, le comportement d'un système basé sur *Lasagne* est adapté en sélectionnant dynamiquement les enveloppes qui encapsulent les composants sans toucher au code de ces derniers.

Lasagne supporte la combinaison sélective de plusieurs aspects pour chaque collaboration (suite d'appels initiée par un client) entre plusieurs composants. Une *politique de composition* précise le sous-ensemble d'aspects qui doit être appliqué pour une collaboration spécifique. Elle prend la forme d'une liste contenant les noms des enveloppes sélectionnées avec, éventuellement un ordre partiel entre elles (p. ex. une enveloppe doit être appelée avant une autre). Cette politique est contrôlée par des *intercepteurs* placés dans l'application et dans son environnement d'exécution. Les intercepteurs déclenchent l'exécution d'une enveloppe si une expression, définie par le programmeur, sur

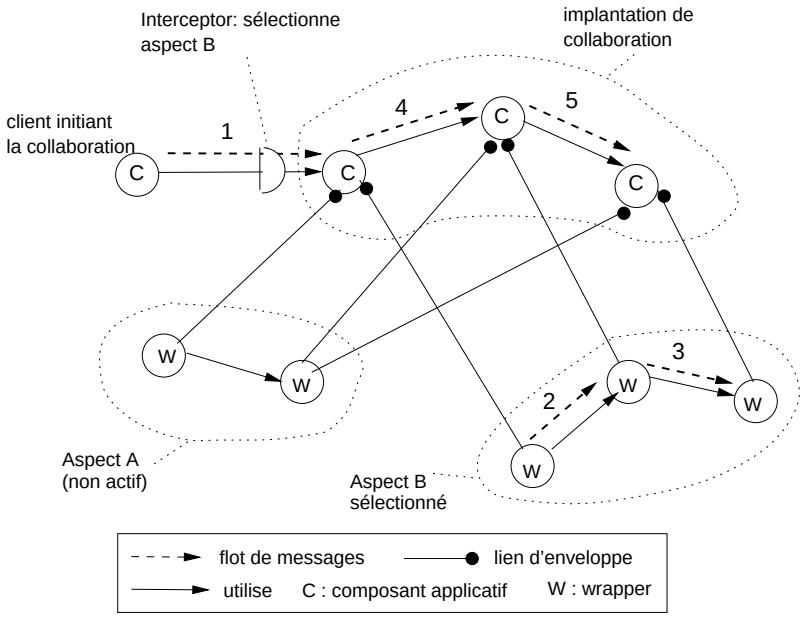


FIG. 4.2 – Combinaison sélective des aspects dans *Lasagne*.

le contexte courant devient vraie. *Lasagne* a été implanté au-dessus de JAC (Java Aspect Components) [Pawlak02], un canevas qui permet de réaliser le tissage dynamique des aspects sur des classes Java en modifiant leur Bytecode.

Discussion

Lasagne permet d'étendre un composant serveur à chaque collaboration, en fonction du contexte courant, par une combinaison d'aspects spécifiques à chaque client. Dans ce sens, il réalise une forme d'interception sophistiquée des messages en leur appliquant une combinaison de traitements qui peuvent changer d'un appel à un autre et d'un client à un autre. *Lasagne* opère une séparation entre le code fonctionnel (les composants), le code non fonctionnel (les aspects), et le code gérant l'adaptation (les intercepteurs). Toutefois, les trois mécanismes de l'adaptation (surveillance, décision et exécution) sont mélangés au sein des intercepteurs dont la mise en œuvre reste à la charge du développeur.

L'inconvénient majeur de *Lasagne* réside dans le surcoût en temps très important introduit par la combinaison dynamique des aspects à chaque collaboration entre les composants. Ceci limite l'utilisation de *Lasagne* à un niveau de granularité élevé dans l'architecture d'un système. À un niveau plus fin, la sélection dynamique des aspects pour chaque message causerait des dégradations de performance inacceptables pour certains types d'applications notamment dans le domaine multimédia.

Par ailleurs, il est possible en principe de réaliser des adaptations *algorithmiques* de chaque aspect, c.-à-d. par le remplacement de son algorithme, mais cette piste n'a pas été explorée. Il est aussi possible pour l'administrateur du système d'opérer des adaptations non anticipées par l'injection de nouveaux aspects. Cet administrateur (humain) a la

charge de vérifier l'absence de conflits entre les aspects qui peuvent être sélectionnés simultanément pour l'extension d'un composant [Truyen02a].

Notons enfin, que les intercepteurs réalisent une forme de coordination implicite centralisée entre plusieurs composants participants à une collaboration, étant donné que les aspects qu'ils sélectionnent sont déployés autour de tous ces composants.

4.3 Adaptabilité suivant la programmation basée sur les ADLs

De nombreux travaux menés sur « l'architecture logicielle » adressent le problème d'adaptation des applications à travers la modification dynamique de leurs architectures. Ces travaux reposent principalement sur les ADLs dynamiques [Oreizy99, Wermelin99, Metayer98, Allen98, Luckham95] et sur les modèles réflexifs (méta-modèles) [Cuesta02, Blair01, Cazzola00] qui incluent notamment certains modèles de composants logiciels que nous présenterons dans la section 4.4.

L'idée commune aux ADLs dynamiques consiste à réagir à l'évolution des besoins logiciels et aux changements de l'environnement d'exécution par la création et la destruction dynamiques des composants et des connecteurs qui les relient. Ces opérations de *reconfiguration* dynamique de l'architecture ont pour effet : (1) la modification de la qualité de service offerte par les composants d'une application (p. ex. rééquilibrage de la charge par le rajout de nouveaux serveurs, augmentation de la tolérance aux fautes par la réplication d'un service sur plusieurs sites) et (2) la modification des règles d'interaction entre les composants par la modification de leurs connecteurs (p. ex. cryptage des données transmises via un connecteur suite au changement du lien réseau sous-jacent vers un lien non sûre). Nous avons choisi de décrire en détail le canevas *Rainbow* [Cheng04] car il est représentatif des techniques d'adaptation basées sur l'architecture logicielle dynamique.

L'objectif de *Rainbow* est de fournir une approche qui minimise le temps et le coût de l'ajout des capacités d'auto-adaptation pour une grande variété de systèmes. *Rainbow* est constitué d'une *infrastructure d'adaptation* qui fournit les fonctionnalités communes (et donc réutilisables) aux systèmes auto-adaptables et qui est spécialisée par la *connaissance d'adaptations spécifiques à chaque système* (cf. figure 4.3).

L'infrastructure d'adaptation est divisée en trois couches :

1. **La couche système** est formée d'une interface (API) comportant des sondes qui surveillent l'état du système, un service de découverte des ressources et des *Effectors* (cf. figure 4.3) qui exécutent les modifications effectives sur le système.
2. **La couche architecture**. Les *jauges* rassemblent les informations provenant des sondes et mettent à jour les propriétés correspondantes du *modèle architectural* du système maintenu par le *gestionnaire du modèle*. Ce modèle architectural prend la forme d'un graphe de composants interagissants (p. ex. clients, serveurs, bases de données, interfaces utilisateur, etc.) Les arcs du graphe sont des connecteurs qui représentent les interactions entre les composants et qui sont généralement réalisés par des intergiciels et des supports de la distribution. *L'évaluateur des contraintes*

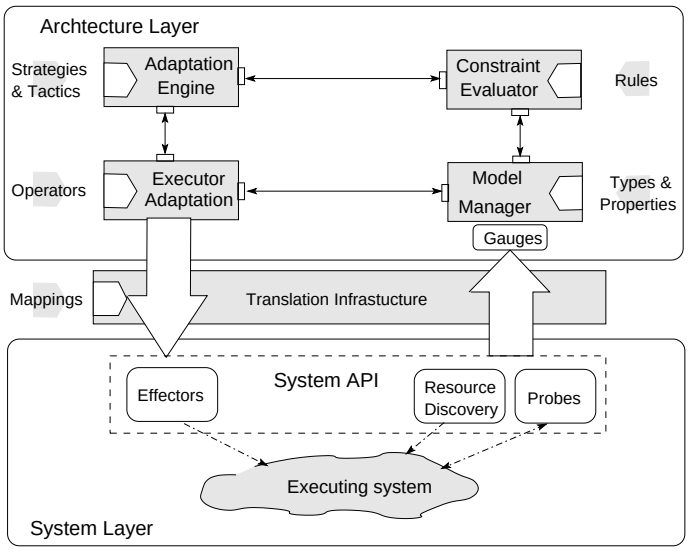


FIG. 4.3 – Le canevas *Rainbow*.

vérifie périodiquement les violations des contraintes du modèle et déclenche le *moteur d'adaptation* si nécessaire. Dans ce cas, le moteur et l'exécuteur effectuent des adaptations requises sur le système.

3. **La couche translation** établit les correspondances entre le modèle abstrait et le système. Par exemple, elle traduit un identificateur du niveau architectural vers une adresse IP.

Afin de rendre un système auto-adaptable, l'infrastructure d'adaptation doit être spécialisée en utilisant la connaissance spécifique au système. Celle-ci repose sur la notion de *style architectural* qui caractérise une famille de systèmes partageant la même structure et les mêmes propriétés sémantiques. Traditionnellement, un style architectural capture les attributs statiques d'un système (c.-à-d. les types et les propriétés des composants et des connecteurs, les règles de composition, et les analyses propres au style), mais pour supporter l'auto-adaptation, Garlan et coll. ont augmenté la notion de style par des attributs dynamiques qui sont les opérateurs et les stratégies d'adaptation. Nous présentons ci-après une brève description de l'ensemble des ces attributs.

- **Les types** définissent un vocabulaire pour les types de composants, comme Database, Client, Filter, et les types de connecteurs, comme Sql, Http, Rpc, Pipe.
- **Les règles** déterminent les compositions possibles à partir des types, comme l'interdiction de cycle dans une architecture du style pipeline.
- **Les propriétés** caractérisent les types pour fournir des informations analytiques ou sémantiques telles la charge et le temps de service pour le type Serveur dans un style client-serveur.
- **Les analyses** sont réalisées en fonction des styles et utilisent leurs propriétés. Un exemple d'analyse pour les styles orientés temps réel est l'ordonnançabilité.
- **Les opérateurs** sont de deux types : adaptation ou interrogation. Les premiers

modifient la configuration d'un style spécifique comme le rajout ou le retrait d'un service. Les seconds permettent de consulter les valeurs des propriétés du système en cours d'exécution.

– **Les stratégies** précisent, en utilisant les opérateurs et les propriétés, comment adapter un système en réponse à une condition indésirable. Chaque stratégie est écrite pour assurer un besoin particulier du système (p. ex. la performance, le coût ou la fiabilité) et met l'accent sur les propriétés liées à ce besoin. La figure 4.4 montre l'exemple d'une stratégie d'adaptation pour un système Web composé de plusieurs clients et plusieurs serveurs constituant des groupes. Cette stratégie vise à limiter le temps de réponse des requêtes des clients en rajoutant un nouveau serveur pour le groupe courant si la charge de ce dernier est importante, et en déplaçant vers un nouveau groupe de serveurs un client dont la bande passante de son lien réseau avec le groupe courant est très faible.

```
invariant C : responseTime <= maxResponseTime
! -> responseTimeStrategy(C);

strategy responseTimeStrategy (C : ClientT)
  let G : ServerGroupT = findConnectedServerGroup(C);
  if( query("load", G) > maServerLoad) {
     G.addServer();
     return true;
  }
  let conn : LinkT = findConnector(C, G);
  if (query("bandwidth", conn) < minBandwidth) {
     let G : ServerGroupT = findBestServerGroup(C);
     C.move(G);
     return true ;
  }
  return false;
}
```

FIG. 4.4 – Exemple d'une stratégie d'adaptation *Rainbow*.

Discussion

L'un des avantages du canevas *Rainbow* est la séparation des préoccupations entre le code métier de l'application, le code qui surveille l'environnement, le code qui réalise les adaptations et le code encapsulant la politique (stratégie) d'adaptation. Ce dernier code est exprimé dans un ADL qui est déclaratif et donc facile à utiliser par les développeurs.

De façon générale, les adaptations basées sur l'architecture logicielle sont un type particulier d'adaptations au niveau multi-composants. Typiquement, la modification d'un connecteur est une forme d'adaptation coordonnée des composants interagissant à travers ce connecteur. Toutefois, les adaptations multi-applications ne peuvent être coordonnées en utilisant la notion d'architecture logicielle car celle-ci ne décrit que les dépendances entres les composants d'une même application.

Par ailleurs, l'un des avantages de ce type d'adaptation est la possibilité de vérifier leur cohérence grâce à la base formelle des ADLs (p. ex. théorie des graphes dans Rainbow [Cheng04], algèbre des processus dans Wright [Allen98], grammaires de graphes

[Metayer98], ensembles d'événements partiellement ordonnés dans Rapide [Luckham95] et [Oreizy99]).

Notons enfin que ces adaptations sont parfois qualifiées de *structurelles* par contraste aux adaptations *comportementales* présentées dans la section suivante et qui concernent les aspects internes des composants.

4.4 Adaptabilité suivant l'approche composants logiciels

En raison de son caractère modulaire, un grand nombre de travaux ont adopté l'approche de programmation orientée composants pour le développement d'applications adaptables. En particulier, cette modularité permet de distinguer trois niveaux de complexité de l'adaptation avec chacun des questions différentes qu'il faut traiter. Le premier niveau est celui de l'adaptation individuelle d'un composant que nous qualifions d'adaptation *intra-composant* ; le second concerne l'adaptation de plusieurs composants interagissants appartenant à une même application et que nous qualifions d'adaptation *multi-composants* ; le troisième est lié à l'adaptation de plusieurs applications concurrentes et que nous qualifions d'adaptation *multi-applications*.

Les adaptations au niveau *intra-composant* concernent uniquement l'implantation et l'état d'un composant et ont pour effet de modifier son comportement. Les adaptations *multi-composants* concernent les composants interagissants dont les comportements doivent être modifiés ensemble de façon coordonnée. Par conséquent, ces adaptations portent aussi sur les connecteurs reliant ces composants et sont donc assimilables aux adaptations structurelles présentées dans la section précédente. Dans les modèles de composants hiérarchiques, le comportement d'un composant composite peut être adapté en modifiant la structure interne de ce dernier(e). Dans ce cas, en élevant le niveau d'abstraction, cette modification structurelle peut être vue comme une adaptation comportementale.

Les adaptations *multi-applications* concernent les composants appartenant à des applications concurrentes qui ne sont pas interagissants mais qui sont liés par le fait qu'ils partagent les ressources du système. Dans ce cas il faut coordonner les différents composants en résolvant de façon équitable les adaptations conflictuelles et en utilisant les ressources de manière efficace.

Nous présentons ci-après les différents modèles de composants (auto-)adaptables existants.

4.4.1 Modèle réifiant les dépendances entre les composants

Kon et coll. proposent un canevas pour développer des applications distribuées adaptables par le remplacement de l'implantation d'un composant ou par la reconfiguration dynamique des dépendances entres plusieurs composants [Kon00b]. A chaque composant est associé un dérivé de la classe abstraite `ComponentConfigurator` (cf. figure 4.5) qui maintient une référence vers l'implantation de ce composant et réifie ses dépendances vers les autres composants appartenant à l'application ou au système. Cette réification représentée dans la figure 4.6 est réalisée à l'aide : (1) d'une liste des clients de `C` qui

sont enregistrés avec la méthode `registerClient()`, et (2) d'un ensemble de *crochets* auxquels les fournisseurs du composant `C` peuvent être attachés en utilisant la méthode `hook()`. Chaque *crochet* définie un service requis par le composant `C` (p. ex. une pile TCP/IP, une machine virtuelle Java) et comporte un pointeur vers le composant qui fournit ce service.

```
class ComponentConfigurator {
 public:
  ComponentConfigurator(Object *implementation);
  ~ComponentConfigurator ();

  int hook (const char *hookName,
    ComponentConfigurator *component);
  int unhook (const char *hookName);
  int registerClient (ComponentConfigurator *client,
      const char *hookNameInClient = NULL);
  int unregisterClient (ComponentConfigurator *client);

  int eventOnHookedComponent (ComponentConfigurator *hookedComponent, Event e);
  int eventOnClient (ComponentConfigurator *client, Event e);

  char *name ();
  char *info ();
  DependencyList *listHooks ();
  DependencyList *listClients ();
  ComponentConfigurator * getHookedComponent (const char *hookName);

  Object *implementation ();
}
```

FIG. 4.5 – La classe abstraite `ComponentConfigurator`.

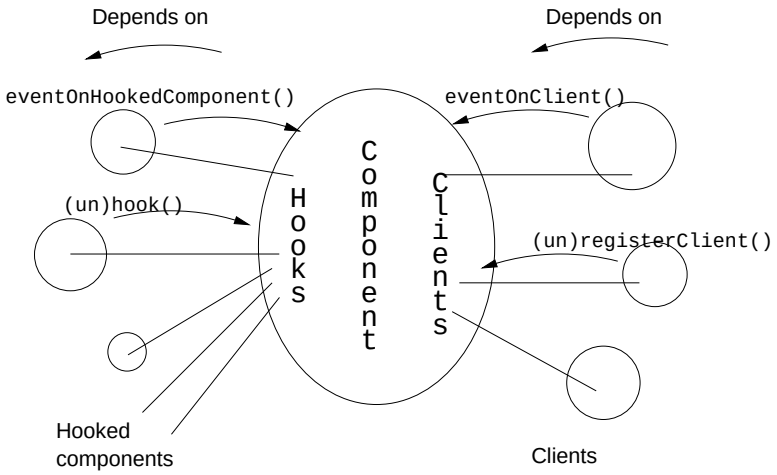


FIG. 4.6 – Réification des dépendances d'un composant.

4.4.1.1 Adaptation intra-composant

Kon et coll. utilisent le patron de conception *strategy* pour l'encapsulation des implantation alternatives d'un composant. Dans le cas de remplacement d'une implantation (S_{old}) par une autre (S_{new}) il faut s'assurer, avant de décharger S_{old}, qu'elle n'est pas en cours d'appel par aucune entité. Eventuellement, il faut aussi transférer (une partie de) l'état interne de S_{old} vers S_{new}. Kon et coll. ont adressé ces deux problèmes en proposant des mécanismes à implanter par le développeur dans les sous-classes de `ComponentConfigurator`. En l'occurrence, pour le premier problème, il est possible :

- D'utiliser `ComponentConfigurator` pour notifier tous les composants ayant une référence vers S_{old}.
- D'utiliser `ComponentConfigurator` comme une indirection sur les appels vers les composants remplaçables.
- De laisser un pointeur de renvois à la place de l'ancien composant.
- De faire en sorte que chaque accès à l'ancien composant génère une exception capturée par le client qui lui permet d'obtenir une référence vers le nouveau composant en contactant par exemple un service de nommage.
- De garder l'ancienne implantation accessible pour les anciens clients et rediriger les nouveaux clients vers la nouvelle implantation.

En ce qui concerne le problème de transfert d'état, tous les composants d'un certain type doivent a priori se mettre d'accord sur une représentation externe de l'état interne de ce type. Ensuite, chaque composant doit implanter une paire de méthodes `export_state()` et `import_state()` qui sont utilisées par le mécanisme de reconfiguration pour transférer l'état du composant sans l'interpréter.

4.4.1.2 Adaptation multi-composants

L'adaptation d'une application par la reconfiguration des dépendances entre composants repose sur les deux méthodes `eventOnHookedComponent()` et `eventOnClient()` de la classe `componentConfigurator`. La première annonce qu'un composant attaché a généré un événement et la deuxième fait la même chose pour un composant client. L'événement en question peut être la destruction de ce composant, le remplacement de son implantation ou sa reconfiguration interne. Le développeur du composant implante dans les sous-classes de `ComponentConfigurator()` les comportements nécessaires pour traiter ces événements. A titre d'exemple, un composant représentant un navigateur Web dépend généralement du composant encapsulant une machine virtuelle java (JVM). Si la JVM est remplacée en cours d'exécution du navigateur, ce dernier peut s'adapter à ce changement en spécialisant la classe `ComponentConfigurator()`. Précisément, le développeur redéfinit la méthode `eventOnHookedComponent()` dans une sous classe appelée `WebBrowserConfigurator` comme le montre la figure 4.7. Cette méthode intercepte l'événement `REPLACED` émit par le `JVMConfigurator`, gèle tous les objets dans la JVM à remplacer, met à jour la référence de la JVM courante, et intègre les objets gelés dans la nouvelle JVM.

```
int WebBrowserConfigurator::eventOnHookedComponent
(ComponentConfigurator *cc, Event e)
{
  if (cc == JVMConfigurator)
   { if (e == REPLACED)
       try {
         FrozenObjs fo = currentJVM->freezeAllObjs ();
         currentJVM = JVMConfigurator->implementation ();
         currentJVM->meltObjects (fo);
       } catch (Exception exp)
         throw ReconfigurationFailed(exp);
     }
     else...
 }
```

FIG. 4.7 – Spécialisation de la méthode `eventOnHookedComponent()`.

4.4.1.3 Illustration de l'adaptation avec *dynamicTAO*

Dans [Kon00a] le canevas *ComponentConfigurator* a été utilisé pour l'adaptation dynamique de l'ORB *TAO*. *TAO* est un ORB CORBA portable, configurable et extensible basé sur les patrons de conception [Schmidt03b]. Il emploie le patron *strategy* afin de proposer des implantations alternatives (appelées stratégies) pour différents aspects internes de l'ORB (p. ex. la concurrence, le démultiplexage des requêtes, l'ordonnancement). En outre, *TAO* offre trois modèles de concurrence différents pour gérer les connexions des clients (en utilisant un seul thread pour toutes les connexions, un thread par connexion ou un pool de threads). Pour chaque aspect interne de l'ORB une seule stratégie est sélectionnée au moment du démarrage de l'ORB et ne peut plus être changée en cours d'exécution.

DynamicTAO est une extension réflexive de *TAO* qui utilise le modèle *ComponentConfigurator* pour reconfigurer les stratégies de cet ORB en cours d'exécution. Chaque processus utilisant *dynamicTAO* contient une instance du `ComponentConfigurator` appelée `DomainConfigurator`. Elle est responsable de maintenir les références vers les instances de l'ORB et les composants fournisseurs de service s'exécutant dans ce processus. En plus, chaque instance de l'ORB contient une sous-classe de `ComponentConfigurator` appelée `TAOConfigurator` qui maintient des crochets auxquels sont attachées les stratégies de *dynamicTAO*. Les entités distantes peuvent se connecter au processus pour inspecter et modifier la configuration de *dynamicTAO* à travers un protocole basé sur TCP. Les composants fournisseurs et clients locaux peuvent aussi le faire à travers une interface de programmation CORBA. La figure 4.8 montre ce mécanisme dans le cas ou une seule instance de l'ORB est présente (`TAOConfigurator`).

4.4.1.4 Discussion

Kon et coll. ont été parmi les premiers à s'attaquer à l'adaptation interne d'un composant par le remplacement de son implantation et l'impact de celle-ci sur les composants dépendants. Les mécanismes qu'ils ont proposés pour traiter le problème du déchargement de l'ancienne implantation et le problème de transfert d'état vers la nouvelle implantation sont suffisamment génériques pour convenir à différents cas possibles.

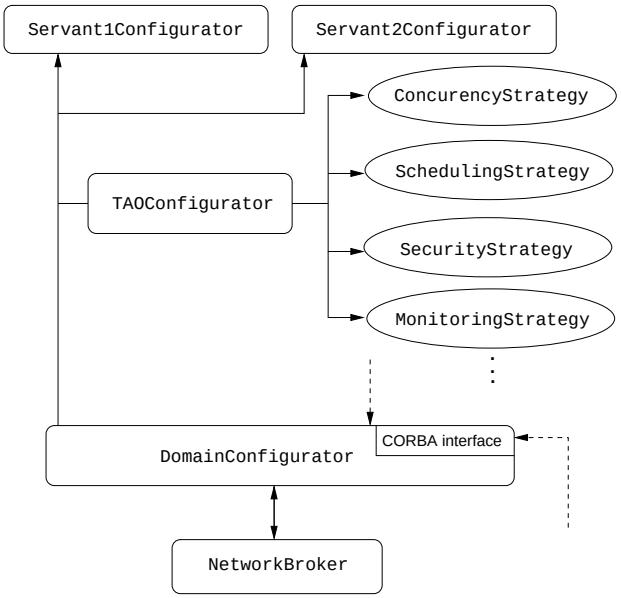


FIG. 4.8 – Reconfiguration des stratégies de *dynamicTAO*.

Toutefois, l'implantation de ces mécanismes abstraits est laissée entièrement à la charge du développeur. Ceci augmente le risque d'introduction d'erreurs pouvant être fatales lors de l'exécution de l'application (p. ex. déchargement d'une implantation en cours d'utilisation).

Par ailleurs, les dépendances entre les composants fournisseurs et clients sont réifiées de manière décentralisée au niveau de chaque composant. Cette approche pose les problèmes de redondance des informations entre les crochets et les listes des clients et d'absence d'une visibilité globale des dépendances indirectes (transitives). Le premier problème implique un surcoût lié aux mises à jour. Le second problème rend très difficile le raisonnement sur l'impact de l'adaptation d'un composant sur ses dépendants indirects (p. ex. la détection des cycles de dépendances pour stopper une adaptation potentiellement infinie). Un autre inconvénient est la non considération, au niveau multi-applications, de l'adaptation des composants qui sont liés non pas par des connexions entre leurs interfaces mais par le fait qu'ils partagent les mêmes ressources du système.

L'approche illustrée par *dynamicTAO* semble plutôt favoriser le remplacement de composants d'un ORB partagés par plusieurs fournisseurs. La mise en œuvre d'adaptations plus fines comme le changement de l'ordonnancement d'un thread gérant une connexion spécifique semble impossible.

Enfin notons l'absence de tout mécanisme générique de surveillance de l'environnement qui est aussi laissé à la charge des développeurs de composants. Ceci implique souvent le mélange du code qui produit les événements déclenchant les adaptations et du code qui réalise ces adaptations.

4.4.2 Modèle réflexif utilisant les méta-espaces

OpenCOM est un modèle de composant léger qui enrichi le standard COM avec les techniques réflexives d'introspection et d'adaptation développées par Blair et coll. [Clarke01]. Pour une raison de performance, *OpenCOM* repose sur un sous-ensemble de COM (le noyau) mais ne dépend pas de ses autres dispositifs comme la distribution (via DCOM), la persistance, la sécurité, et les transactions. Le noyau de COM comporte : (1) la norme d'interopérabilité au niveau binaire permettant d'assembler dynamiquement plusieurs composants dans un seul espace d'adressage (c.-à-d. un processus), (2) le langage de définition d'interface de Microsoft (IDL), (3) les identificateurs globaux uniques (GUIDs), et (4) l'interface *IUnknown* permettant la découverte d'autres interfaces d'un composant et le comptage du nombre de clients qui le référencent.

Par ailleurs, en plus de la notion *d'interface* (fournie) d'un composant COM, *OpenCOM* rajoute la notion de *réceptacle*[1] pour définir les interfaces requises d'un composant. Cela rend explicites les *connexions* (dépendances) existant entre les composants d'un processus.

Dans chaque processus, *OpenCOM* déploie un exécutif implanté par un composant COM appelé `OpenCOM` et qui exporte une interface `IOpenCOM` (cf. figure 4.9). `IOpenCOM` est le point central pour soumettre les requêtes de connexion ou de déconnexion des réceptacles avec les interfaces. Le développeur d'un composant *OpenCOM* doit implanter les interfaces (1) `IReceptacles` pour offrir des opérations qui altèrent les interfaces connectées aux réceptacles du composant, et (2) `ILifeCycle` pour offrir des opérations qui seront appelées par l'exécutif aux moments de la création et la destruction du composant.

4.4.2.1 Adaptation intra-composant

Chaque composant *OpenCOM* intègre un *méta-espace* qui implante les capacités réflexives permettant d'inspecter et d'adapter le composant. Ce *méta-espace* est formé de trois sous-composants standards (cf. figure 4.9) `MetaInterception`, `MetaArchitecture` et `MetaInterface`, accessibles respectivement via les trois méta-interfaces suivantes :

- `IMetaInterception` permet au programmeur d'associer (dissocier) des *intercepteurs* à une interface particulière. Un *intercepteur* est un composant avec des méthodes qui sont appelées avant et/ou après chaque invocation de méthode de l'interface spécifiée.
- `IMetaArchitecture` permet au programmeur d'obtenir les identificateurs de toutes les connexions courantes entre les réceptacles du composant cible et les interfaces externes.
- `IMetaInterface` supporte l'inspection des types des interfaces et des réceptacles déclarés par le composant cible.

[1] Le terme « réceptacle » est aussi utilisé dans le modèle CCM. Le concept lui-même apparaît dans d'autres modèles sous d'autres noms (p. ex. port d'entrée).

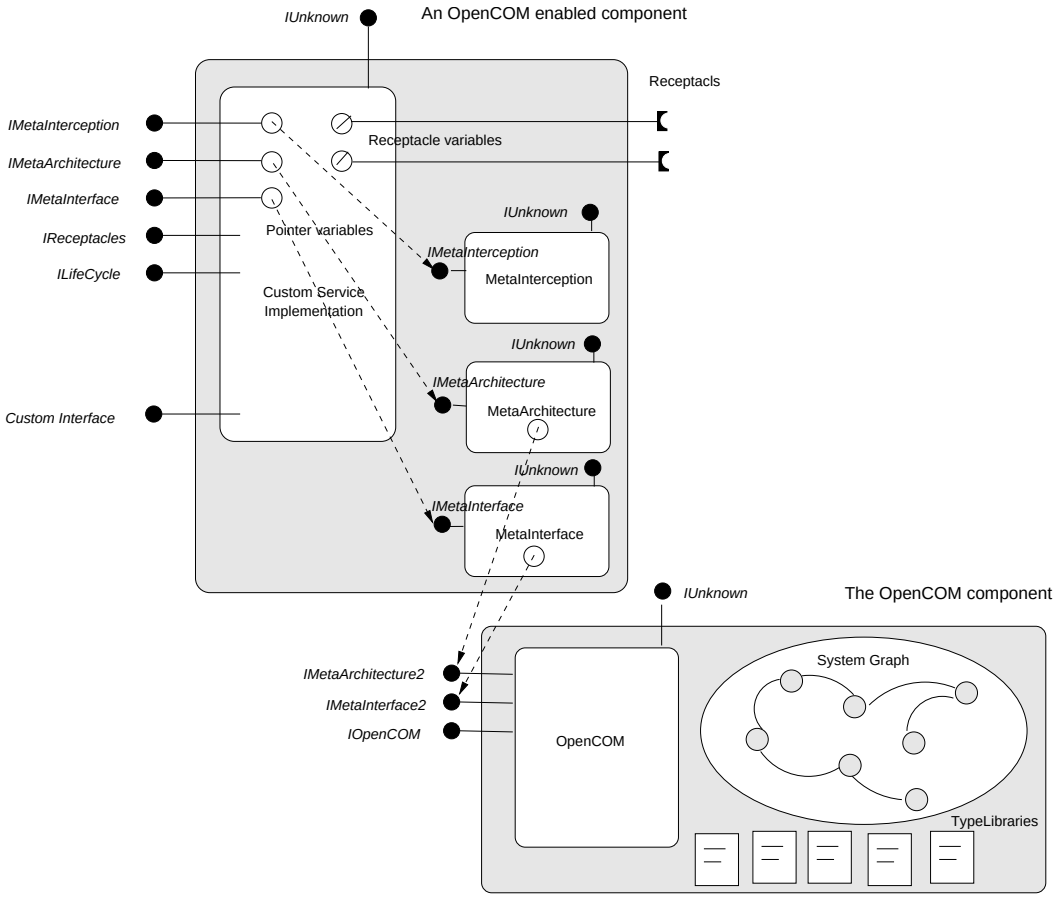


FIG. 4.9 – Architecture d'*OpenCOM*.

4.4.2.2 Adaptation multi-composants

OpenCOM supporte la reconfiguration dynamique des connexions grâce à (1) la notion de réceptacle qui rend explicites les dépendances entre composants, et (2) la structure de graphe `graph system` maintenue par l'exécutif *OpenCOM* et qui représente l'état de ces dépendances en cours d'exécution. Chaque réceptacle contient un verrou qui permet de bloquer toute reconfiguration de la connexion associée si des invocations à travers ce réceptacle sont en cours. Si une opération acquiert le verrou en premier, toutes les invocations futures seront annulées jusqu'à ce que le verrou soit relâché par un logiciel de plus haut niveau (vraisemblablement, après que le réceptacle soit connecté avec succès ailleurs).

4.4.2.3 Illustration avec OpenORB

L'objectif principal derrière le développement d'*OpenCOM* était la ré-ingénierie (avec la technologie composant) de l'intergiciel *OpenORB* v1 [Costa00] en réutilisant ses capacités réflexives (c.-à-d. la notion de méta-espace). Avec la version 2 d'*OpenORB* [Coulson02], Blair et coll. ont voulu rendre l'intergiciel plus flexible en utilisant la notion de canevas de composants (ci-après, CFs) instanciés à l'exécution. La figure 4.10 montre

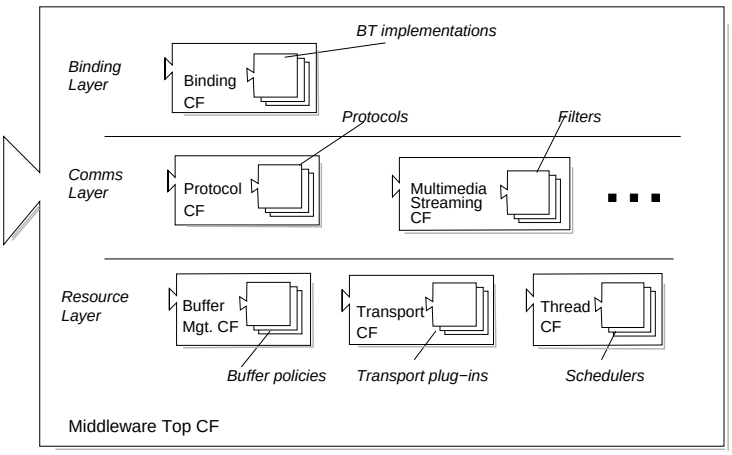


FIG. 4.10 – Architecture d'*OpenORB*.

l'architecture d'*OpenORB* v2. Il est structuré comme un CF de niveau supérieur composé de trois couches d'autres CFs appelées : *binding*, *communication* et *resource*. Le CF supérieur impose des politiques de composition concernant les changements dynamiques dans chaque couche (p. ex. interdiction de la suppression de certains CFs). Chacun des CFs dans les trois couches s'adresse à un domaine spécifique de la fonctionnalité de l'intergiciel (p. ex. protocoles de communication, ordonnancement des threads).

Le CF supérieur peut être adapté de deux façons.

1. Il peut être configuré en mandatant une politique de composition particulière et un ensemble de CFs initiaux pour peupler les couches. Ceci permet de créer une

architecture intergicielle spécifique à une plate-forme ou à un domaine d'application particulier. Par exemple, une architecture peut inclure des CFs qui gèrent les transactions et la sécurité, etc.

2. Une instance d'un intergiciel peut être configurée dynamiquement par introduction de nouveaux CFs (s'ils respectent la politique de composition) ou par remplacement ou personnalisation des CFs existants. A titre d'exemples, un CF de traitement des signaux peut être ajouté dans la couche communication, le CF de gestion des threads peut également être étendu avec un nouveau composant d'ordonnancement.

Les utilisateurs interagissent avec les CFs à travers les APIs qu'ils exportent. Chaque API regroupe les services offerts par les composants constituants le CF et inclut des opérations de (re)configuration restreintes de ce CF. De cette façon, seules les CFs peuvent utiliser l'exécutif *OpenCOM* pour l'adaptation. Les entités externes doivent passer par l'API du CF en question. Par exemple, le domaine de la communication peut accepter seulement les reconfigurations sur les sous-composants qui supportent une interface spécifique telle `IProtocol`. Ainsi, ce domaine restreint sa propre reconfiguration aux unités protocoles de communication au lieu d'autoriser le remplacement de tout le domaine.

4.4.2.4 Discussion

Parmi tous les travaux que nous avons étudiés, *OpenCOM* utilise la réflexion de manière intensive pour mettre en œuvre des adaptations du type rajout des intercepteurs pour réifier les appels aux méthodes d'une interface, inspection du type d'une interface et reconfiguration des connexions inter-composants. Toutefois, les problèmes liés à l'adaptation par remplacement à chaud d'un composant (c.-à-d. transfert d'état et déchargement de l'ancienne version) ne sont pas abordés.

Par ailleurs, la réification des connexions entre les composants d'un processus sous la forme d'un graphe maintenu à l'extérieur de ceux-ci présente deux avantages importants : (1) elle offre une visibilité globale des dépendances existantes étant donné que ces informations sont centralisées au niveau de l'exécutif *OpenCOM*, et (2) elle permet d'analyser les adaptations par reconfiguration des connexions en utilisant les opérations de manipulation de graphe.

Par contre le canevas *OpenORB* v2 n'en est pas vraiment un, car non conceptuel ; il s'apparente plus à un composant composite puisqu'il a des interfaces (APIs), des composants constituants et une politique de composition. Enfin, notons que le problème de coordination des adaptations au niveau multi-applications n'a pas été adressé.

4.4.3 Modèle réflexif et hiérarchique utilisant un MOP

Dans [David03] Ledoux et coll. proposent un canevas de composants auto-adaptables basé sur le modèle *Fractal* [Bruneton02]. Ce dernier est un modèle de composant Java réflexif et générique qui supporte la définition des composants et des liens entre leurs interfaces, et la composition hiérarchique (y compris le partage de composants). Chaque

composant *Fractal* est constitué (1) d'un *contrôleur* par lequel passent toutes les interactions avec les autres composants et (2) d'un *contenu* qui peut être une classe Java dans le cas d'un composant primitif ou d'autres composants dans le cas d'un composite. Les applications *Fractal* peuvent être reconfigurées par la modification des liens (reconfiguration structurelle) et par le paramétrage d'un composant via une interface particulière.

Afin de supporter les adaptations non anticipées au moment du développement, Ledoux et coll. ont étendu *Fractal* avec un protocole méta-objet (MOP), un service de connaissance du contexte et des politiques d'adaptation. Pour cela au contrôleur par défaut des composants *Fractal* a été rajoutée la réification des messages reçus (cf. figure 4.11). Les messages réifiés sont envoyés à un sous composant du méta-niveau pour réaliser des pré- et post-traitements génériques ou remplacer le traitement par défaut.

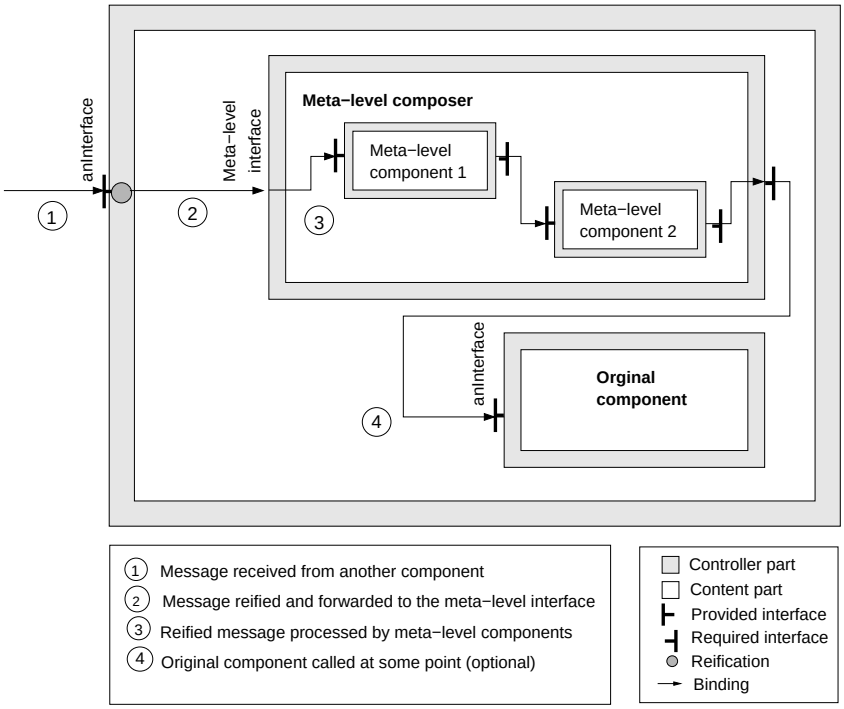


FIG. 4.11 – Extension de type MOP pour *Fractal*.

4.4.3.1 Politiques d'adaptation

A chaque composant *Fractal*, une ou plusieurs politiques d'adaptation peuvent être attachées (ou détachées) dynamiquement. Ces politiques sont contenues dans la partie contrôleur du composant et sont accessibles via une interface spécifique. Une politique d'adaptation est un ensemble de règles de la forme *événement–condition–action*. La partie *événement* décrit les circonstances dans les quelles une règle peut être déclenchée. Elle est exprimée en utilisant des événements primitifs ou des événements composites

formés avec des opérateurs pour détecter des séquences, des alternatives ou des conjonctions d'événements. Les conditions sont des expressions booléennes. La partie *action* est une séquence d'actions de reconfiguration structurelle, de paramétrage, d'ajout ou de retrait de services génériques via l'extension MOP.

4.4.3.2 Service de connaissance du contexte

Le service de connaissance du contexte est disponible pour le reste du système via une interface de requêtes et une interface de notification asynchrone. Ce service assure la surveillance de l'environnement selon les trois phases suivantes :

1. *Acquisition* : elle est réalisée par des objets Java (appelés sondes) qui renvoient des mesures datées avec leurs moments d'acquisition. Le système maintient un ensemble de sondes instanciées avec pour chacune un taux d'échantillonnage particulier. Ces sondes sont invoquées par un ordonnanceur cyclique qui collecte les mesures à consommer par les autres parties du canevas.
2. *Représentation* : dans cette seconde phase, les informations collectées sont structurées dans un modèle de l'environnement en utilisant une ontologie appropriée. Le contexte est représenté comme un arbre de ressources identifié par un nom. Le système supporte plusieurs arbres de ressources en parallèle, chacun dédié à un domaine de contexte spécifique (p. ex. la topologie du réseau).
3. *Raisonnement* sur la connaissance qui résulte de la phase précédente pour détecter des situations qui requièrent une action d'adaptation. Lors de cette phase des attributs synthétiques peuvent être créés à partir d'expressions formées avec les informations renvoyées par les sondes.

4.4.3.3 Illustration

Ledoux et coll. ont utilisé ce canevas pour développer un navigateur visualiseur d'images auto-adaptable. Le principal composant de cette application est un décodeur d'images qui interprète leurs contenus (JPEG, PNG, etc.) en un bitmap directement affichable sur écran. La politique d'adaptation de ce composant permet (1) d'instancier composant cache générique et l'attacher au fournisseur d'images quand le temps moyen de chargement et de décodage devient long, (2) de modifier la taille du cache en fonction de la mémoire disponible, et (3) de modifier la politique de remplacement du cache (LRU, RMU, etc.) en fonction de l'historique des requêtes de chargement d'images.

4.4.3.4 Discussion

L'un des atouts du canevas proposé par Ledoux et coll. est la séparation du reste de l'application, du code qui surveille l'environnement et du code qui dicte la logique des adaptations. Ceci permet d'alléger la tâche du développeur et simplifie la maintenance et le portage vers des plates-formes hétérogènes des différents codes nécessaires à l'adaptabilité. Toutefois, dans ce canevas basé sur *Fractal*, les politiques d'adaptation sont écrites en Java ce qui les rends *impératives*. Or dans une politique d'adaptation on

aimerait spécifier uniquement ce qu'il faut faire sans dire comment le faire ; un langage *déclaratif* serait donc plus approprié.

Comme pour *OpenCOM*, l'adaptation intra-composant est limitée à un paramétrage du composant via une interface spécifique et les problèmes liés au remplacement d'un composant ne sont pas abordés. Par ailleurs, comme le composant méta-niveau peut être lui-même un composite, il est possible d'ajouter et d'enlever des comportements génériques dynamiquement. Mais, à ce stade le problème de la composition correcte des comportements méta-niveau n'a pas été adressé.

Enfin, dans un travail précédent indépendant de *Fractal* [David02] Ledoux et coll. ont utilisé le terme « politiques système » pour désigner des politiques d'adaptation des services intergiciels classiques comme la distribution des appels objets ou les appels asynchrones. Bien que ces services puissent être partagés par plusieurs composants applicatifs (comme pour *dynamicTAO*), leur adaptation ne dépend que des ressources système. Elle ne prend pas en compte les besoins spécifiques des applications qui n'ont que le choix d'utiliser ou non de tels services. La coordination des adaptations au niveau multi-applications n'est donc pas traitée.

4.4.4 Modèle utilisant un protocole d'adaptation graduel

Cactus [Chen01] est un canevas supportant l'adaptation graduelle et coordonnée des services et protocoles dans les systèmes distribués. La figure 4.12 montre l'architecture en couches adaptables d'un système basé sur *Cactus*. Ces couches sont construites comme une collection de composants adaptables (ACs). Le contrôleur d'adaptation dans la partie gauche de la figure coordonne les adaptations multicouches et les adaptations multi-composants sur cet hôte. Chaque collection de composants adaptables répartis sur plusieurs machines et qui coopèrent pour implanter un service distribué (p. ex. fiabilité et réplication) est appelée *composant distribué adaptable (DAC)*.

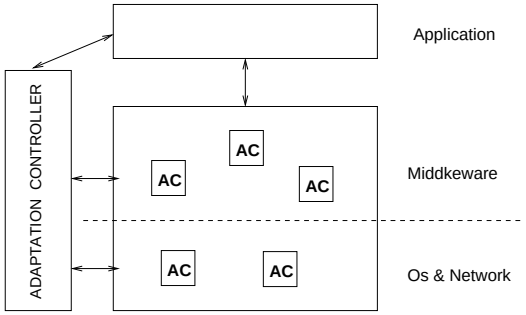


FIG. 4.12 – Architecture de *Cactus*.

La figure 4.13 montre la structure interne d'un AC qui est constituée de deux types de modules : un module adaptateur du composant (CAM) et des modules algorithmiques alternatifs conscients de l'adaptation (AAMs). Chaque AAM fournit un algorithme différent qui implante la fonctionnalité du composant, tandis que le CAM contrôle l'adaptation du comportement du composant.

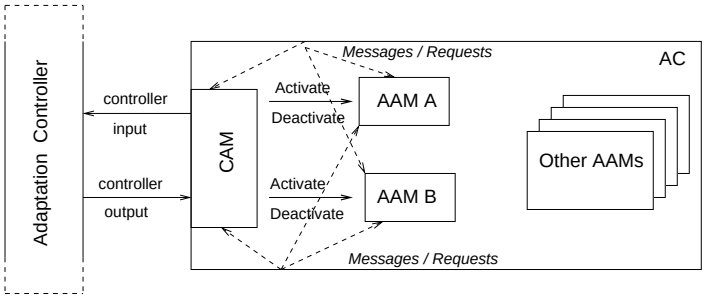


FIG. 4.13 – Structure d'un composant *Cactus*.

4.4.4.1 Processus d'adaptation

L'adaptation d'un DAC peut être divisée en trois phases :

1. **Détection des changements** : 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. *Cactus* ne dicte pas le module responsable de cette détection mais c'est au développeur de choisir entre le CAM ou l'AAM courant en fonction du type du service à développer.

   Chaque AAM doit exporter une fonction `Fitness()` qui prend comme argument un vecteur `sysState[]` reflétant l'état du système et répond si cet AAM est approprié pour cet état. Le CAM doit exporter une fonction `BestFit()` qui utilise les résultats des fonctions `Fitness()` pour sélectionner l'AAM le mieux approprié pour l'état courant du système. Cette sélection est spécifique à chaque service et doit tenir compte du coût de l'adaptation pour optimiser le comportement global du composant et doit empêcher l'oscillation entre les AAMs alternatifs. Si le choix réalisé par `BestFit()` est différent de l'AAM courant, le CAM initie la phase suivante d'accord.

2. **Accord** : la deuxième phase est un processus d'accord dans lequel tous les ACs parviennent à un consensus sur l'exécution on non d'une action d'adaptation. Encore une fois, *Cactus* ne dicte pas l'approche à suivre pour atteindre un tel consensus mais laisse le choix au développeur. Une manière possible est l'emploi de fonctions déterministes pour disséminer les vecteurs `sysState[]` locaux et calculer un vecteur d'état global du système, puis utiliser ce vecteur au niveau de chaque CAM pour choisir l'AAM optimal.

   Une autre façon est de permettre à chaque CAM de choisir localement un AAM optimum, puis d'utiliser un processus de consensus (avec une fonction de sélection déterministe) pour sélectionner un AAM parmi tous les candidats. Dans les deux cas, si le nouvel AAM est jugé meilleur que le courant, une action d'adaptation est exécutée.

3. **Action d'adaptation** : La phase finale implante un processus qui remplace un AAM de façon graduelle de telle sorte que le DAC continue de fonctionner correctement durant l'adaptation. Ce processus est exécuté par l'exécutif *Cactus* en

trois étapes : préparation, commutation des sorties et commutation des entrées. En fonction des sémantiques du service, un transfert d'état de AAM_{old} vers AAM_{new} peut avoir lieu dans les étapes de commutations des entrées-sorties. Le flot de contrôle associé à ce protocole d'adaptation graduelle est implanté par une API exportée par les AAMs, comme le montre la figure 4.14. L'étape

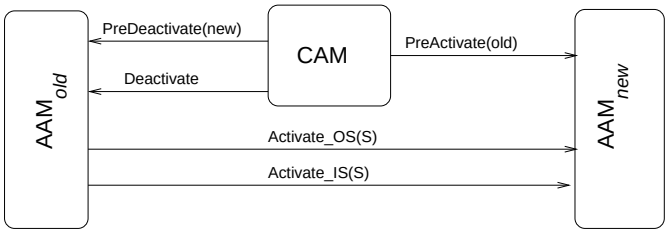


FIG. 4.14 – Protocole d'adaptation graduelle de *Cactus*.

de préparation est implantée par `PreActivate()` et `PreDeactivate()`. La fonction `Deactivate()` est utilisée pour initier la commutation, tandis que `Activate_OS()` et `Activate_IS()` implantent la commutation des sorties et des entrées respectivement. L'argument `S` représente un éventuel transfert d'état entre les deux AAMs.

4.4.4.2 Illustration

Cactus a été utilisé entre autres pour développer un protocole de transmission fiable dans un système de communication de groupe [Chen01]. Ce protocole est implanté avec deux algorithmes alternatifs ACK et NACK basés respectivement sur les accusés de réception positifs et les accusés négatifs. Le choix de l'un de ces algorithmes affecte le nombre de messages transmis et la latence de bout en bout. Quand les erreurs de transmissions sont rares, l'algorithme NACK est sélectionné afin d'utiliser un nombre faible de messages supplémentaires. Dans le cas contraire, si le taux d'erreurs de transmission est important ou si la fréquence de transmission est faible, la latence augmente et il convient d'utiliser l'algorithme ACK. L'état transféré entre les deux algorithmes inclut notamment la queue des messages en attente de transmission ou retransmission et le numéro de séquence du dernier message reçu de chaque hôte.

4.4.4.3 Discussion

Cactus supporte un seul type d'adaptation qui est le remplacement des implantations de plusieurs composants participant à un même service distribué. La solution qu'il apporte au problème du remplacement d'une implantation est l'une des plus élégantes que nous avons étudiées. Elle repose sur un protocole graduel qui, moyennant une participation du développeur, assure le transfert de l'état du composant vers la nouvelle implantation en permettant une continuité du service fournis par le(s) composant(s) durant l'adaptation.

Comme pour *Carisma*, la prise de décision d'adaptation est une procédure distribuée qui tient compte de l'état global du système. Cependant, elle doit être implantée par le

développeur et son objectif n'est pas de résoudre un conflit d'allocation de ressources mais seulement de choisir un algorithme approprié pour un service distribué.

En dernière remarque, notons l'absence d'un service de surveillance de l'environnement qui est laissée à la charge du développeur.

4.4.5 Modèle basé sur un langage d'interaction

Riveill et coll. proposent un langage d'interaction qui permet l'adaptation dynamique des composants par la pose et la destruction de règles d'interaction à l'exécution [Blay-For04]. Cette adaptation est réalisée par l'utilisateur final de l'application pour satisfaire de nouveaux besoins fonctionnels ou intégrer des propriétés non fonctionnelles (p. ex. persistance, sécurité, transaction) [Charfi03]. L'utilisateur final peut utiliser les schémas d'interactions définis par le développeur et fournis avec l'application, ou écrire lui-même de nouveaux schémas dans le langage *ISL (Interaction Specification Language)*.

Un schéma comporte des règles d'interactions qui sont composées chacune d'une partie gauche qui précise l'appel de méthode (message) auquel la règle est associée, et d'une partie droite qui exprime les contrôles à opérer sur les composants liés. Par exemple, la règle du schéma d'interaction suivant lie un composant agenda à un afficheur afin de notifier l'utilisateur chaque fois qu'un rendez-vous est ajouté.

```
interaction notification(Object O, Display){
        O.* -> O._call // display.notify(_reifiedCall)
}
```

Les schémas d'interactions sont enregistrés auprès d'un serveur d'interactions dont l'implantation Java est appelée *Noah*. Tous les appels entre composants interagissants passe par ce serveur ; si une ou plusieurs règles sont associées à un appel alors elles seront exécutées. Une instrumentation réflexive du code des composants interagissants permet l'interception des appels.

Le langage ISL autorise la pose de plusieurs interactions simultanément. Pour cela il repose sur une algèbre qui assure la fusion de ces règles de façon commutative et transitive. Ainsi, quel que soit l'ordre de pose le résultat de la fusion sera équivalent. Si une incohérence entre les règles est détectée, la fusion sera refusée.

4.4.5.1 Discussion

Le modèle d'interaction de Riveill et coll. permet l'adaptation dynamique par la modification des interactions entre les composants sans changer ceux-ci. L'apport majeur de ce modèle est le langage ISL qui, comparé aux techniques complexes de métaprogrammation, offre une méthode simple pour la réification des appels entre composants par la réécriture de code. Un autre avantage de ce modèle est la possibilité de combiner de façon cohérente plusieurs interactions qui, éventuellement, n'ont pas été prévues par le développeur des composants.

Les adaptations portent uniquement sur les interactions et sont évolutives ; elles sont contrôlées manuellement par l'utilisateur final ; il n'y pas de service de surveillance de

l'environnement. Par ailleurs, le serveur d'interactions joue le rôle de l'adaptateur tandis que les schémas d'interactions sont assimilables à des politiques d'adaptation simples.

Enfin, notons que l'interception des appels entre composants interagissants basée sur la réflexion entraîne un surcoût même si aucune règle d'interaction n'est associée à l'appel.

4.4.6 Modèle réactif utilisant le patron *strategy*

La plate-forme *MolèNE* (MObiLE Networking Environment) supporte le développement d'applications mobiles basées sur les composants adaptables [Segarra00]. Elle est constituée : (1) d'une boite à outils offrant des fonctionnalités spécifiques aux réseaux sans-fil et indépendantes des applications, et (2) d'un canevas fournissant des services génériques spécialisables par les applications (cf. figure 4.15). À titre d'exemple, l'outil communication offre des liaisons fiables même en présence de déconnexions transitoires. L'outil de migration permet de déporter l'exécution de calculs depuis les stations mobiles vers les stations fixes. Au niveau service on trouve par exemple, le service de gestion du cache implanté par deux composants *MolèNE* responsables respectivement du vidage et du chargement des données dans le cache. Le service de distribution *AeDEN* s'occupe de la répartition des calculs et des données en fonction des caractéristiques des environnements sans-fil comme l'autonomie des portables, leur localisation géographique et la bande passante disponible [Mouël01].

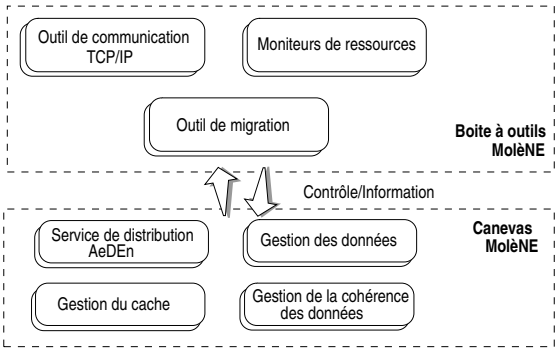


FIG. 4.15 – Architecture de la plate-forme *MolèNE*.

Chaque service du canevas de *MolèNE* est conçu de façon modulaire en implantant chacune des fonctionnalités qu'il fournit sous forme d'un *composant réactif*. Ces fonctionnalités sont adaptables dynamiquement grâce aux trois mécanismes suivants : le système de détection-notification, l'interface d'adaptation et le modèle de composants réactifs.

4.4.6.1 Système de détection-notification

Le système de détection-notification (SDN) des variations de l'environnement est structuré en deux niveaux. Les moniteurs de base (MB) surveillent l'environnement et

recueillent des informations sous forme « brute » (p. ex. niveau de bande passante, temps de latence et taux de perte). Les moniteurs de haut niveau (MHN) synthétisent les mesures provenant des MBs ou d'autres MHNs en fonction des besoins exprimés par les applications (p. ex. qualité globale d'une connexion et mémoire disponible).

4.4.6.2 Interface d'adaptation

L'interface (API) d'adaptation permet de décrire la politique d'adaptation d'un composant *MolèNE* en spécifiant les variations significatives de l'environnement et les réactions qui leurs sont associées [André00]. *MolèNE* implante une politique[2] d'adaptation avec un automate dont les états correspondent aux conditions d'exécution et à l'implantation courante du comportement et les transitions indiquent la réaction à effectuer. Il y a deux types de réactions possibles : (1) le changement des paramètres d'un composant (p. ex. le nombre d'éléments à charger en avance dans un cache) et, (2) le changement de l'implantation du composant (p. ex. basculer entre le chargement à la demande et le chargement en avance selon la bande passante).

Les politiques d'adaptation sont écrites dans le langage Java par spécialisation de la classe `AdaptationStrategy` et des classes associées comme indiqué par la figure 4.16. En particulier, la classe `InitialState` indique l'état initial de l'automate d'adaptation qui précise quel comportement doit être chargé au lancement du composant.

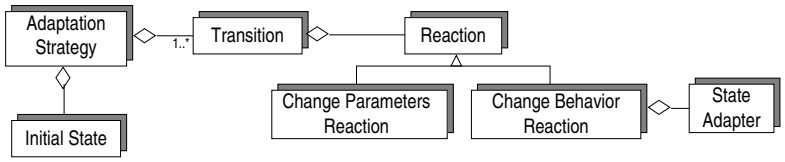


FIG. 4.16 – Politique d'adaptation d'un composant *MolèNE*.

4.4.6.3 Composants réactifs

La figure 4.17 montre l'architecture d'un composant *MolèNE* réactif basée sur le patron *strategy*. Dans la partie fonctionnelle, l'objet `Interaction` reçoit les appels de méthodes et les objets `Implementation` exécutent effectivement les fonctions appelées. Le comportement du composant est rendu adaptable par l'ajout de la partie réactive. Elle contient l'objet `Controller` qui assure que la politique d'adaptation du composant se déroule comme spécifiée par `AdaptationStrategy`. En particulier, lors d'une réaction de changement d'implantation, `Controller` appelle l'objet `StateAdapter` associée à cette réaction (cf. figure 4.16) pour qu'il prépare l'état du composant au transfert vers la nouvelle implantation. Pour déclencher une réaction l'objet `Controller` utilise le système de détection-notification qui lui notifie des variations des conditions d'exécution.

[2] Dans [Segarra00] le terme stratégie est employé à la place du terme politique.

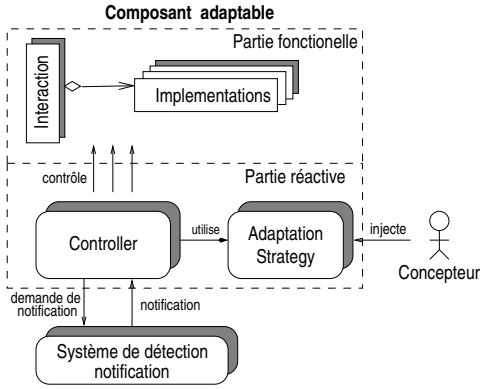


FIG. 4.17 – Architecture d'un composant *MolèNE*.

4.4.6.4 Discussion

MolèNE supporte deux types d'adaptation intra-composant avec une gestion semi-automatique du problème de transfert d'état dans le cas de remplacement d'une implantation. Le développeur doit en effet définir dans `StateAdapter` comment modifier l'état interne du composant pour qu'il soit utilisable par l'implantation remplaçante.

La plate-forme *MolèNE* offre une séparation des différents codes jouant un rôle dans l'adaptation. En particulier, son système de surveillance est l'un des plus sophistiqué car il permet de synthétiser les informations de bas niveau sur l'état de l'environnement en objets Java (MHNs) réutilisables par plusieurs applications. Cependant, dans le cas où il faudrait suivre l'évolution d'une variation par intervalles de temps courts, l'organisation hiérarchique de ce système peut poser des problèmes de performance et de rapidité de recueil des informations s'il faut traverser plusieurs niveaux de la hiérarchie. Par ailleurs, les politiques d'adaptation des composants *MolèNE* sont exprimées sous forme d'automates à l'aide d'une API orienté objet. Pour les développeurs, cette forme impérative est sans doute plus difficile à utiliser qu'une abstraction déclarative du type événement–condition–action.

Sur un autre plan, l'un des atouts de *MolèNE* est la distinction entre les outils indépendants des applications et les services spécialisables ce qui permet de maximiser la réutilisation et d'accroître la flexibilité de la plate-forme *MolèNE*.

L'aspect adaptation multi-composants et multi-applications n'a pas été abordé.

4.5 Synthèse

Le tableau 4.1 résume les avantages et les inconvénients des travaux que nous avons présentés dans cet état de l'art. L'étude de ces travaux nous a permis d'identifier les différentes manières avec lesquelles les trois étapes de l'adaptation (surveillance, décision et exécution) ont été implantées. La plupart de ces travaux adhèrent au principe de séparation des préoccupations en isolant les différents codes qui implantent ces étapes (mécanismes). Le mélange de ces codes (ou d'une partie d'eux) constitue un handicap

pour le développeur et limite les possibilités d'adaptation de son application. À titre d'exemple, la distinction entre le code de l'adaptateur et la politique d'adaptation rend possible la modification dynamique de cette politique sans toucher à l'adaptateur. Ceci permet de réaliser des adaptations non anticipées au moment du développement de l'application.

Nous présentons ci-après une synthèse des réponses à la question posée précédemment : *comment mettre en œuvre les mécanismes de l'adaptation ?* (cf. section 1.6). Ensuite, nous argumentons notre choix pour les concepts qu'il nous semble intéressant de retenir pour développer des applications auto-adaptables ainsi que pour les méthodes d'implantation associées. Enfin, nous concluons cet état de l'art par une présentation des problèmes importants qui n'ont pas encore trouvés de solutions satisfaisantes.

4.5.1 Revue des mises en œuvre des mécanismes de l'adaptation

Surveillance de l'environnement

La surveillance de l'environnement est généralement réalisée par un service distinct au niveau intergiciel. Parmi les services de surveillance existants, seuls quelques-uns permettent de détecter et de notifier les événements complexes qui dénotent plusieurs variations simultanées ou ordonnées suivant des relations temporelles particulières.

Politique d'adaptation

Concernant la spécification des politiques d'adaptation, les langages déclaratifs sont plus simples à utiliser que les langages impératifs. L'écriture des règles d'adaptation de la forme « événements–conditions–actions » s'exprime facilement, notamment dans un dialecte XML. Parmi les travaux présentés, seuls quelques-uns considèrent l'état de l'environnement au niveau global et distribué, en impliquant plusieurs sites dans la prise de décision d'adaptation. Egalement, certains travaux consultent l'utilisateur de l'application au moment de cette prise de décision, notamment afin de résoudre des choix d'adaptation conflictuels.

Cependant, la quasi-totalité des systèmes supportant l'adaptabilité ne tient pas compte du facteur temps dans leurs décisions d'adaptation. Ils ne considèrent que l'état immédiat de l'environnement mais pas l'état passé, ni l'état futur qui dépend entre autres du coût de l'adaptation. Pourtant, les informations sur l'historique des variations des ressources et sur le coût des différentes adaptations possibles sont très utiles, en particulier pour éviter les adaptations en chaînes non stables (les effets ping-pong).

Actions d'adaptation

Il existe différents types d'actions d'adaptation qui sont plus au moins supportées par les travaux existants. Dans le cas des approches de programmation modulaires, l'adaptation d'une application peut être réalisée par les opérations suivantes, listées selon l'ordre croissant de leur complexité :

		QuO	M3	Carisma	Efstratiou	Conductor	Puppeteer	Lasagne	Rainbow	C.Configurator	OpenCOM	Fractal adapt.	Cactus	Noah	MolèNE
Séparation des préoccupations		m s a p	m s a p	m s a p	m s a p	- s a,p p	- s a p	m - a,p	m s a p	m - a,p	m - a,p	m s a p	m,p - a	m - a p	m s a p
Notification des évts complexes		+	-	+	-	-	-	-	-	-	-	+	-	-	+
Politique d'adaptation	Spécification	déc	déc	déc	déc	imp	imp	déc	déc	imp	imp	déc	imp	imp	imp
	Dynamiquement modifiable	-	+	+	+	-	-	?	-	-	-	+	-	+	-
	Considère l'état passé	-	-	-	-	-	-	-	-	-	-	-	-	+	-
	Consid. l'état (distribué) global	-	-	+	-	+	+	-	+	-	-	-	+	-	-
	Consid. le coût de l'adaptation	-	-	-	-	-	-	-	-	-	-	-	-	-	-
	Consulte l'utilisateur	-	-	-	+	-	-	-	-	-	-	-	-	-	-
Types d'actions d'adaptation	Paramétrage	-	-	-	-	-	-	-	-	-	-	+	-	-	+
	Remplacement d'algorithmes	-	-	+	-	-	-	-	-	+	-	-	+	-	+
	- support du transfert d'état			-						+			+		+
	- implantations concurrentes			-						-			+		-
	Réification des appels	+	+	-	-	-	-	+	-	-	+	+	-	+	-
	Reconfig. architecture	-	-	-	-	-	-	-	+	+	+	+	-	-	-
	Interposition (proxy)	-	-	-	+	-	+	-	-	-	-	-	-	-	-
	Appel fct applicative (callback)	-	-	-	+	+	-	-	-	-	-	-	-	-	-

Légende a : code adaptateur, m : code métier, s : code de surveillance
p : politique d'adaptation, déc : déclarative, imp : impérative

TAB. 4.1 – Comparaison des différents travaux sur l'adaptabilité du logiciel.

1. Paramétrage d'un composant (ou d'un module), qui permet d'adapter son implantation à plusieurs contextes différents en changeant seulement certains de ses attributs.
2. Remplacement de l'implantation d'un composant avec, si besoin, la gestion du transfert de son état vers la nouvelle implantation et le déchargement différé de l'ancienne implantation.
3. Interception des appels de méthodes afin de réaliser des pré/post-traitements spécifiques, et ce par une réification vers le méta-niveau ou par une redirection vers un aspect, ou encore en utilisant un langage d'interaction.
4. Reconfiguration des connexions entre plusieurs composants par ajout ou suppression des composants et des connexions qui les relient. Ces connexions sont représentées explicitement dans les composants ou à l'aide d'un ADL dynamique ou d'un méta-modèle.

La majorité des supports à l'adaptabilité sont des canevas qui fournissent des implantations partielles d'un sous-ensemble de ces opérations, soit au niveau intergiciel soit au niveau application. La part du travail qui est laissée à la charge du développeur détermine le *degré d'automatisation de l'adaptation*. Ainsi, ce qui est générique est pris en charge automatiquement par un canevas (et/ou par un intergiciel sous-jacent) et ce qui spécifique est laissé à la charge du développeur. Par conséquent, les applications basées sur ces canevas peuvent être conscientes de l'adaptation.

Quand un canevas n'est pas utilisé, l'adaptation peut être gérée de façon transparente aux applications, généralement par l'interposition de proxies spécifiques entre les clients et les serveurs.

Les supports à l'adaptabilité peuvent aussi laisser l'exécution des actions d'adaptation sous l'entière responsabilité des applications. Ceci offre des possibilités d'adaptation illimitées, mais ces actions d'adaptations sont plus difficiles à contrôler que lorsqu'elles sont définies dans un cadre précis.

4.5.2 Guide pour le développement d'un support de l'adaptation

Idéalement, le développeur d'une application (auto-)adaptable ne doit fournir que le code métier et la politique qui spécifie ce qu'il faut adapter et quelles actions entreprendre en fonction des variations de l'environnement. L'étude des travaux existants montre que l'approche de programmation orientée composant est très bien appropriée pour atteindre un tel objectif. Combinée avec les patrons de conception *strategy* et *observer* et avec la technique de réflexion, cette approche offre à la fois la séparation entre le code métier, le code de surveillance, le code de l'adaptateur et la politique de l'adaptation ainsi que la possibilité de réaliser chacune des actions d'adaptation énoncées dans la section 4.5.1.

Aucune autre approche parmi toutes celles que nous avons étudiées ne permet un tel choix d'actions. L'approche aspect dynamique ne permet de réaliser que les interceptions des appels entre les composants interagissants. Ces interceptions trouvent leur

équivalent dans la réification des appels dans les modèles de composants réflexifs. Toutefois, dans ces modèles il n'est pas nécessaire d'instrumenter le code de l'application avec des outils spécifiques comme c'est le cas dans l'approche aspect. L'approche architecture logicielle dynamique ne permet que les adaptations par reconfiguration des connexions des composants. La cohérence de ces adaptations peut être vérifiée grâce à la base formelle des ADLs, mais on retrouve aussi cet avantage avec les modèles de composant réflexifs. En effet, il possible de représenter les connexions entre les composants à l'aide de méta-informations basées sur un modèle formel comme les graphes de dépendances.

Le développement d'applications suivant l'approche composants se fait généralement à partir d'un canevas générique. Pour qu'il soit efficace, le canevas de composants adaptables doit avoir les propriétés clés suivantes :

- Les moniteurs doivent être dédiés chacun à un type de ressources donné et s'exécuter au niveau intergiciel sur chaque hôte, et ce afin qu'ils soient partagés par toutes les applications déployées sur cet hôte. Ils doivent assurer deux modes de surveillance qui sont la consultation directe des valeurs des ressources et la notification asynchrone des variations. Il est préférable, pour des raisons de performance, que la structure du système de surveillance soit la plus simple possible. Le code de surveillance doit être facilement extensible, par exemple à l'aide de la technique orientée objet de l'héritage, pour prendre en compte les caractéristiques propres aux nouvelles technologies qui apparaissent au fur et à mesure.
- L'adaptateur de chaque composant doit pouvoir réagir à des événements complexes. Il doit être indépendant de tout code métier et doit pouvoir exécuter les différentes actions d'adaptation intra et inter-composants : paramétrage, remplacement d'implantation, interception des appels et reconfiguration des connexions. En particulier, pour les remplacements d'implantations, il doit fournir des supports pour automatiser le transfert de l'état du composant de l'ancienne implantation vers la nouvelle, et éviter la perte des messages en transit en les redirigeant vers la bonne implantation. En d'autres termes, il doit garantir l'intégrité de l'application au moment de l'adaptation.
  Le code de l'adaptateur doit être réutilisable sans modifications, exception faite de la partie qui interprète la politique d'adaptation. Cette partie pourrait évoluer si le langage d'expression des politiques d'adaptation est modifié pour prendre en compte des besoins spécifiques à des domaines d'applications particuliers. Eventuellement, l'adaptateur pourrait aussi consulter de façon interactive l'utilisateur humain afin de finaliser une décision d'adaptation (p. ex. demander une confirmation ou un choisir une action parmi plusieurs).
- La politique d'adaptation doit être exprimée dans un langage déclaratif et facilement extensible, en utilisant des règles ayant des formes simples telle que la forme événement–condition–action. De plus, la cohérence de ces règles doit être vérifiable de façon rapide. La politique d'adaptation doit être séparée de l'adaptateur afin de pouvoir la modifier dynamiquement pour réaliser des adaptations non anticipées au moment du développement de l'application. Idéalement, pour la prise de décision, les règles d'adaptation doivent prendre en compte l'état glo-

bal du système distribué, l'historique des variations et le coût de l'exécution des adaptations envisagées.

Les différentes propriétés que nous avons déduites des travaux existants ne suffisent pas à assurer une adaptation efficace au niveau multi-composants et au niveau multi-applications. Nous verrons dans la partie suivante de cette thèse quels sont les problèmes liés aux adaptations multiples et comment les résoudre à travers la coordination de ces adaptations.

Troisième partie

Plate-forme de composants adaptables *Aceel*

Chaptire 5

Problématique : la coordination des adaptations multiples

Ce chapitre expose le problème traité dans cette thèse, qui est la coordination des adaptations de plusieurs applications distribuées. Après une identification concise du problème, nous montrons que seuls certains aspects ont été traités par les rares travaux existants. Nous justifions aussi pourquoi est-il important de proposer une solution plus complète.

5.1 Problèmes des adaptations multiples des composants logiciels

A cause des dépendances existant entre les composants logiciels, l'adaptation de l'un d'eux peut avoir un impact non négligeable sur les autres. Pour cela, l'entité qui prend la décision d'adaptation doit considérer ces dépendances afin que les actions choisies soient coordonnées entre les composants concernés.

Dans leur étude interdisciplinaire sur la théorie de la coordination [Malone94], Malone et Crowston définissent celle-ci comme la gestion des dépendances entre plusieurs activités. En poursuivant cette étude dans le cadre spécifique de l'intégration des composants logiciels, Dellarocas a proposé une classification des dépendances entre composants en deux familles [Dellaroc96] : les dépendances de flots entre les composants qui interagissent et les dépendances de partage de ressources entre les composants concurrents.

Dans notre étude sur la coordination des adaptations multiples des composants, nous avons adopté la classification de Dellarocas car elle procure un cadre générique pour analyser les problèmes liés à chaque famille de dépendances. En particulier elle permet de distinguer les protocoles de coordination qui s'appliquent le mieux à chaque famille. Par exemple, la notification, la mise en séquence, la standardisation (des formats) et l'interrogation de l'utilisateur sont des protocoles bien adaptés à la gestion des dépendances de flots. Tandis que les files FIFO, les classes de priorités, les budgets et les enchères sont des exemples de protocoles qui conviennent à la gestion des dépendances de partage de ressources [Malone94]. Cette classification permet aussi d'identifier les

primitives qui servent à implanter ces protocoles (p. ex. primitive de synchronisation, d'exclusion mutuelle, etc.).

Nous présentons ci-après une analyse des problèmes liés aux dépendances entre les composants qui s'adaptent aux variations de l'environnement. Notons que chaque variation de l'environnement concerne un seul attribut d'une seule ressource, qu'une variation peut être notifiée à plusieurs composants simultanément et que plusieurs variations peuvent avoir lieu et peuvent donc être notifiées en même temps. Notons aussi que les composants interagissants appartiennent à une même application mais peuvent être répartis sur plusieurs sites, tandis que les composants concurrents n'appartiennent pas forcément à la même application mais sont déployés dans le même environnement.

En fonction de sa politique d'adaptation, un composant qui reçoit une ou plusieurs notifications simultanées peut avoir le choix d'exécuter une action d'adaptation parmi plusieurs possibles. Idéalement, ce composant devrait choisir l'action la plus profitable pour lui. C'est à dire, celle qui maximise la qualité du service qu'il offre à son utilisateur. Cependant, si ce composant possède des dépendances envers d'autres composants de son environnement, son adaptation individuelle risque de causer des conflits ou des contre-performances au niveau du système global.

Effectivement, l'adaptation d'un composant peut modifier sa façon d'interagir avec les autres, par exemple en modifiant le format des données échangées, la fréquence des échanges ou le protocole de communication utilisé, ou encore en rompant carrément cette interaction.

L'adaptation peut aussi amener le composant à consommer et/ou à libérer d'autres ressources en plus de celle qui, par sa variation, a déclenché l'adaptation. Ces ressources peuvent être convoitées par plusieurs composants concurrents (interagissants ou non). Si ces derniers s'adaptent en même temps en réaction à la même variation de l'environnement ou à plusieurs variations simultanées, il peut y avoir des conflits de partage ou d'accès aux ressources (chacun voulant profiter de l'augmentation d'une ressource) ou, au contraire, une sous-exploitation de celles-ci (chacun voulant libérer une partie d'une ressource devenue rare).

Par ailleurs, la consommation et la libération d'une ressource sont elles-mêmes des variations qui peuvent déclencher l'adaptation d'autres composants. Dans ce cas, il y a un risque d'instabilité du système si une adaptation particulière provoque des réactions en chaîne ou en boucle de plusieurs composants. Dans le premier cas, l'adaptation d'un composant peut provoquer celle d'un autre, qui déclencherait à son tour l'adaptation d'un troisième et ainsi de suite. Dans le second cas, deux composants (ou plus) partageant plusieurs ressources peuvent chacun réagir à l'adaptation de l'autre tout en annulant l'effet de celle-ci. Ceci obligerait donc l'autre composant à recommencer de nouveau son adaptation ce qui conduit à un effet « ping-pong ».

Dans le cas des dépendances de flot, il est nécessaire pour les composants interagissants de se concerter pour convenir des actions d'adaptation cohérentes qui préservent leur intégrité et assurent la continuité de leur interaction. A titre d'exemple, dans une application de vidéo streaming entre un serveur s'exécutant sur une station fixe et un client déployé sur un équipement mobile muni d'une connexion sans-fil, si le serveur s'adapte à la croissance des erreurs de transmission en sélectionnant un format de co-

dage moins sensible à celles-ci (p. ex. MJPEG), il doit consulter le client pour voir s'il dispose du décodeur approprié et s'il est prêt à l'employer.

Dans le cas des dépendances de partage de ressources, il faut résoudre les conflits qui peuvent avoir lieu entre les composants concurrents, assurer une exploitation efficace des ressources disponibles et éviter l'instabilité du système.

5.2 Solutions existantes pour la coordination des adaptations multiples

Parmi tous les travaux sur l'adaptabilité du logiciel que nous avons présentés dans la partie précédente, quelques uns ont plus ou moins abordé le problème de la coordination des adaptations multiples. Il s'agit des systèmes *M3*, *Conductor*, *DynamicTAO*, *Cactus*, *Carisma*, *HATS* et l'*intergiciel d'Efestratiou* (cf. partie II). Cependant, aucun de ces systèmes n'a traité l'ensemble des problèmes de coordination précédents.

Les systèmes *M3* et *Conductor* ne proposent qu'une solution consistant à mettre en séquence les actions d'adaptation pour le premier et à définir un ordre de déploiement des modules adaptateurs pour le second. Cette solution est une forme simple de coordination entre entités interagissantes. Elle est implantée par le développeur de façon ad hoc — au niveau du code métier — dans le cas de *Conductor* et de façon générique — au niveau d'un script d'adaptation — dans le cas de *M3*.

Les systèmes *DynamicTAO*, *Cactus* et *Carisma* adressent de façon plus générique le problème de coordination des entités interagissantes (les composants pour les deux premiers et les applications pour le troisième). *DynamicTAO* propose une interface à implanter par le développeur du composant pour gérer de façon décentralisée et explicite les dépendances envers les autres composants. *Cactus* propose un protocole d'accord élaboré qui permet à deux composants de remplacer leurs implantations de façon cohérente et graduelle. *Carisma* permet à plusieurs applications distribuées, qui sont indépendantes mais qui utilisent en commun le même service intergiciel, de choisir le mode de fonctionnement de celui-ci en employant un protocole basé sur les enchères. Cependant, *DynamicTAO*, *Cactus* et *Carisma* ne traitent pas les adaptations conflictuelles entre plusieurs composants ou plusieurs applications en concurrence pour les mêmes ressources.

Tout à fait à l'opposé des trois systèmes précédents, l'intergiciel d'Efestratiou traite les problèmes liés au partage des ressources entre plusieurs applications adaptables, mais ne considère par le cas des entités interagissantes. Cet intergiciel utilise un langage évolué pour formuler des règles d'adaptation qui tiennent compte des relations temporelles entre plusieurs événements, au niveau de plusieurs applications. Mais, à l'instant de rédaction de ces lignes, aucun mécanisme de résolution de conflit n'a été proposé.

Le système *HATS* est le seul qui considère à la fois la coordination des composants interagissants et des composants concurrents mais seulement dans le cadre spécifique des applications clients-serveurs qui échangent des documents et en considérant uniquement la variation de la bande passante du réseau. La solution proposée permet au client et au serveur de convenir des parties du document qui sont réellement transmises grâce à

l'exportation du format du document et de l'API qui le modifie. *HATS* permet aussi d'éviter les conflits d'utilisation de la bande passante entre les applications concurrentes en ordonnançant la transmission des données suivant un système de priorités.

Enfin, notons qu'aucune des solutions existantes ne traite le problème d'instabilité du système dû aux adaptations en chaîne ou en boucle.

5.3 Conclusion

Le vrai potentiel de l'adaptation ne peut être atteint qu'en coordonnant les adaptations des composants s'exécutant dans le même environnement. Ceci permettra une utilisation plus intelligente des ressources afin d'optimiser, notamment en fonction des préférences de l'utilisateur, la qualité de service offerte par les applications. La coordination consiste à gérer les problèmes dus à tous les types de dépendances entre les composants logiciels. Il s'agit d'un besoin vital qui nécessite des mécanismes génériques en plus de ceux qui réalisent les adaptations individuelles des composants.

Quand plusieurs composants peuvent s'adapter ensemble, il est nécessaire de coordonner leur actions afin :

- de préserver l'intégrité des applications et de garantir la continuité des interactions en place.
- d'éviter les conflits et d'optimiser l'utilisation des ressources de l'environnement,
- d'assurer la stabilité du système distribué.

Nous avons vu que les travaux qui ont abordé le problème de la coordination des adaptations n'ont pas considéré l'ensemble de ces aspects. Notre travail, présenté dans le reste de cette thèse, vise à faire une proposition plus complète.

Chaptire 6

Conception de la plate-forme *Aceel*

Ce chapitre présente la plate-forme de composants *Aceel*, notre solution pour faciliter le développement d'applications distribuées adaptables. Cette plate-forme est constituée d'un canevas de composants réflexifs, d'un service de surveillance de l'environnement et d'un intergiciel pour la coordination des adaptations multi-composants et multi-applications.

6.1 Introduction : objectifs de développement d'*Aceel*

Notre proposition s'inscrit dans le cadre des travaux sur l'adaptabilité des applications menés à l'IRISA, qui ont été commencés par Segarra et coll. en 1997 et qui ont abouti en 2001 au développement de la plate-forme *MolèNE* (cf. section 4.4.6). *MolèNE* faisait partie des premières solutions ayant adopté l'approche composants logiciels pour traiter le problème de l'adaptabilité (cf. section 4.5). L'objectif de notre travail a été de résoudre les problèmes liés à la coordination des adaptations de plusieurs composants logiciels. Pour atteindre cet objectif, nous avons bâti un nouveau modèle de composants dans lequel l'adaptation réactive coordonnée est une propriété clé considérée dès le départ.

Ce nouveau modèle de composant appelé *Aceel (Adaptive ComponEnt modEL)* est basé sur le modèle réactif de *MolèNE* et tient compte des récentes avancées en matière d'adaptabilité du logiciel, identifiées dans notre étude de l'existant. En particulier, nous avons poussé la prise des décisions d'adaptation à l'extérieur des composants vers un niveau logiciel plus bas qui est l'intergiciel de coordination des adaptations. Nous avons aussi réorganisé l'architecture du service de surveillance de l'environnement en une architecture plate et distribuée afin, respectivement, d'augmenter ses performances et de prendre en compte l'état global du système lors des prises de décisions.

Ainsi, en partant de la plate-forme *MolèNE*, nous avons développé la nouvelle plate-forme de composants *Aceel* illustrée dans la figure 6.1 [Chefrour03a, Chefrour03b, Chefrour05]. Les sections suivantes présentent en détail les trois pierres angulaires de cette plate-forme : le canevas de composants *Aceel*, le service de surveillance de l'environnement et l'intergiciel de coordination des adaptations.

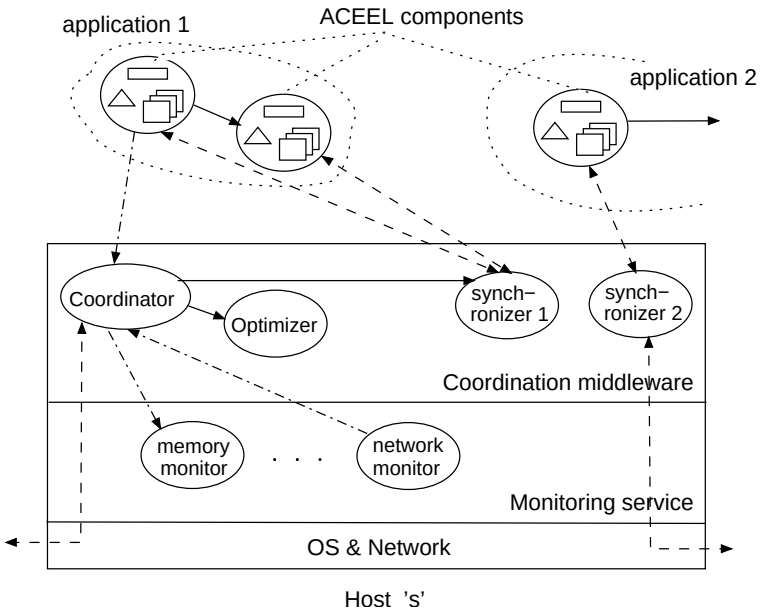


FIG. 6.1 – Plate-forme de composants adaptables *Aceel*.

6.2 Modèle de composant réflexif adaptable *Aceel*

Dans cette section nous n'aborderons pas en détails la programmation orientée composant, mais nous nous concentrons seulement sur les aspects structurels et dynamiques des composants qui ont un rapport direct avec l'adaptabilité. En particulier, nous ne statuons pas sur certaines questions conceptuelles liées aux modèles de composants (p. ex. le modèle est-il plat ou hiérarchique, les connecteurs sont-ils décrits explicitement dans un ADL ou déduits à partir d'un IDL) mais nous spécifions comment supporter l'adaptabilité dans chaque cas. Notre objectif a été de concevoir un modèle d'adaptation générique qui peut être transposé sur différentes plates-formes de composants.

6.2.1 Aspects structurels

Pour concevoir le modèle *Aceel*, nous avons repris les idées de base de *MolèNE*, à savoir la séparation de la partie fonctionnelle et de la partie réactive d'un composant (cf. figure 4.17) et le recours au patron de conception *strategy*.

Par définition, le patron *strategy* encapsule les variantes d'un algorithme donné dans une hiérarchie de classes afin de les rendre interchangeables (cf. section 2.4.2). Ainsi, il permet de mettre en œuvre les adaptations algorithmiques consistant à remplacer l'implantation d'un composant par une autre. Ce patron présente notamment l'avantage de pouvoir modifier l'une des implantations alternatives sans toucher aux autres. Toutefois, tel que ce patron est décrit dans [Gamma95], ces implantations doivent être toutes prédéfinies. De plus, le remplacement de l'une d'entre elles par une autre est réalisé par

les clients du composant.

Or, dans le cas d'*Aceel*, nous voulons que le composant réalise lui-même ce remplacement de son implantation en réaction aux variations de l'environnement, y compris en utilisant une implantation non anticipée. Pour cela nous avons eu recours à la technique de la réflexion.

La réflexion permet de gérer l'adaptation à un niveau externe aux implantations du composant, par le moyen d'un méta-objet. Elle offre aussi la possibilité de surveiller l'état du composant (introspection), que nous considérons comme une ressource pouvant déclencher une adaptation, et la possibilité de modifier cet état (intercession) si nécessaire. La réflexion permet également l'interception des appels de méthodes grâce à la réification et permet la reconfiguration de l'architecture d'un composant composite par la manipulation des connecteurs au méta-niveau.

Les sous-sections suivantes détaillent les notions de base, l'architecture et la politique d'adaptation d'un composant *Aceel*.

6.2.1.1 Notion de dimension d'adaptation

En comparant l'adaptation d'un composant au mouvement d'un corps, nous pouvons dire que chaque sujet de l'adaptation (p. ex. une implantation, une variable d'état, etc.) définit une *dimension d'adaptation* dans laquelle le composant peut évoluer entre plusieurs positions (p. ex. plusieurs implantations, plusieurs valeurs d'une variable d'état, etc.). Par exemple, un composant d'affichage peut s'adapter en variant la taille (longueur X largeur) ou la résolution couleur des images rendues. Dans le cadre d'*Aceel*, nous disons qu'il possède deux *dimensions d'adaptation* appelées respectivement « taille image » et « couleur image ».

L'ensemble des *dimensions d'adaptation* d'un composant définit son *espace d'adaptation*. Concrètement, chaque vecteur de cet espace correspond à un *mode de fonctionnement* possible du composant. A chaque vecteur de *l'espace d'adaptation* correspond à la fois un vecteur (à estimer) des ressources utilisées par le composant et une valeur (à définir) de la qualité de service rendue à l'utilisateur. Ces deux correspondances permettent de choisir, en fonction des variations de l'environnement, les actions d'adaptation qui amènent le composant dans le mode de fonctionnement qui fournit la meilleure qualité de service. Ces deux correspondances sont spécifiées dans la politique d'adaptation du composant.

Si un composant possède plusieurs sujets d'adaptation, son utilisateur peut vouloir favoriser l'adaptation de l'un de ces sujets par rapport aux autres. Pour tenir compte de cet aspect, nous avons associé à chaque *dimension d'adaptation* un poids qui représente l'importance de son sujet pour l'utilisateur. Par exemple, si un composant envoie des images sur le réseau, il peut s'adapter à la chute de la bande passante en réduisant leurs tailles. Pour cela, il peut soit redimensionner ces images, soit diminuer leur résolution couleur. Bien que ces deux actions affectent deux sujets différents, du point de vue du composant, elles sont équivalentes car elles ont le même effet. Le choix de l'une ou l'autre dépend avant tout de l'importance des sujets d'adaptation « taille image » et « couleur image » pour l'utilisateur. Pour l'utilisateur d'un PDA, il serait préférable

de réduire les dimensions des images au lieu de leurs couleurs. Au contraire, pour un utilisateur qui dispose d'un grand écran et qui est intéressé par les détails des images, le choix inverse est plus judicieux.

6.2.1.2 Notion de qualité de service

La qualité d'un service est une notion perceptuelle qui dépend des intérêts (ou des préférences) de l'utilisateur concernant un ou plusieurs aspects de ce service. A titre d'exemple, l'intérêt de l'utilisateur peut être de sécuriser les données, de minimiser le coût d'un service payant, de garantir la disponibilité d'un autre service, d'avoir des présentations multimédia de haute fidélité, etc. Ces intérêts se traduisent au niveau des applications par le cryptage des données, par la diminution des périodes de connexion à un réseau payant, par la création d'une copie locale d'une base de données distante, par le choix d'une grande résolution d'image, etc.

Afin de choisir les adaptations qui maximisent la qualité de service, il est nécessaire de quantifier les intérêts de l'utilisateur. Pour cela, nous associons à chaque *dimension d'adaptation* un *indice de qualité* qui dénote la satisfaction perçue par l'utilisateur pour les différentes positions du composant dans cette dimension. Ces valeurs dépendent de la nature du traitement réalisé (p. ex. présentation multimédia) et de la façon dont la qualité de service est mesurée (p. ex. taille d'une image, fréquences des trames vidéo, etc.). La valeur maximale de cet indice correspond à la meilleure position du point de vue de l'utilisateur.

La définition des valeurs de l'*indice de qualité* en fonction des niveaux perceptibles de la qualité de service est une tâche laissée au développeur. Celui-ci propose des valeurs par défaut qui peuvent être modifiées dynamiquement par l'utilisateur du composant. Ces valeurs peuvent donc être différentes d'un utilisateur à un autre.

Par exemple, un composant d'affichage peut s'adapter en variant la taille des images rendues. Dans le cadre d'*Aceel*, nous disons qu'il possède une *dimension d'adaptation* appelée « taille image ». Un utilisateur qui ne fait que visionner des images sur son écran, pourrait affecter à l'*indice de qualité* de cette *dimension d'adaptation* des valeurs qui augmentent avec la taille des images. Un autre utilisateur qui souhaite laisser de la place pour d'autres fenêtres prioritaires sur son écran, pourrait définir un *indice de qualité* qui est inversement proportionnel à la taille des images. Un troisième utilisateur pourrait affecter les valeurs maximales de l'*indice de qualité* aux tailles moyennes et donner des valeurs inférieurs pour les autres tailles, etc.

Étant donnée que l'*indice de qualité* est utilisée pour modéliser une préférence ou un intérêt relatif, alors ses valeurs n'ont qu'une signification relative. Par conséquent, nous pouvons normaliser les *indices de qualité* de toutes les *dimensions d'adaptation* des composants sous la forme d'un pourcentage. Cette normalisation permet de définir la qualité de service d'un composant comme la moyenne pondérée des *indices de qualité* de l'ensemble des *dimensions d'adaptation* de ce composant. Les *indices de qualité* sont pondérés en les multipliant par des facteurs (des nombres entiers) qui dénotent chacun l'importance donnée à chaque *dimension d'adaptation* par rapport aux autres. Cette pondération permet de choisir une adaptation si, suite à une variation de l'environne-

ment, plusieurs adaptations deviennent possibles.

De la même façon, nous définissons la qualité de service sur un site comme la moyenne pondérée des qualités de service des composants *Aceel* qui y sont déployés. La qualité de service de chaque composant est pondérée par un facteur qui représente la priorité de l'application à laquelle il appartient par rapport aux autres. Ces priorités permettent de favoriser certains composants lors des adaptations qui provoquent des conflits de partages des ressources (cf. section 6.4). Ces priorités sont définies par l'utilisateur de l'application à laquelle ces composants appartiennent. Nous supposons que les machines qui accueillent les applications adaptables (souvent des équipements portables) sont utilisées par un seul utilisateur à la fois. Dans le cas contraire, les priorités des applications appartenant à des utilisateurs différents doivent être définie par l'administrateur de la machine hôte.

6.2.1.3 Architecture d'un composant *Aceel*

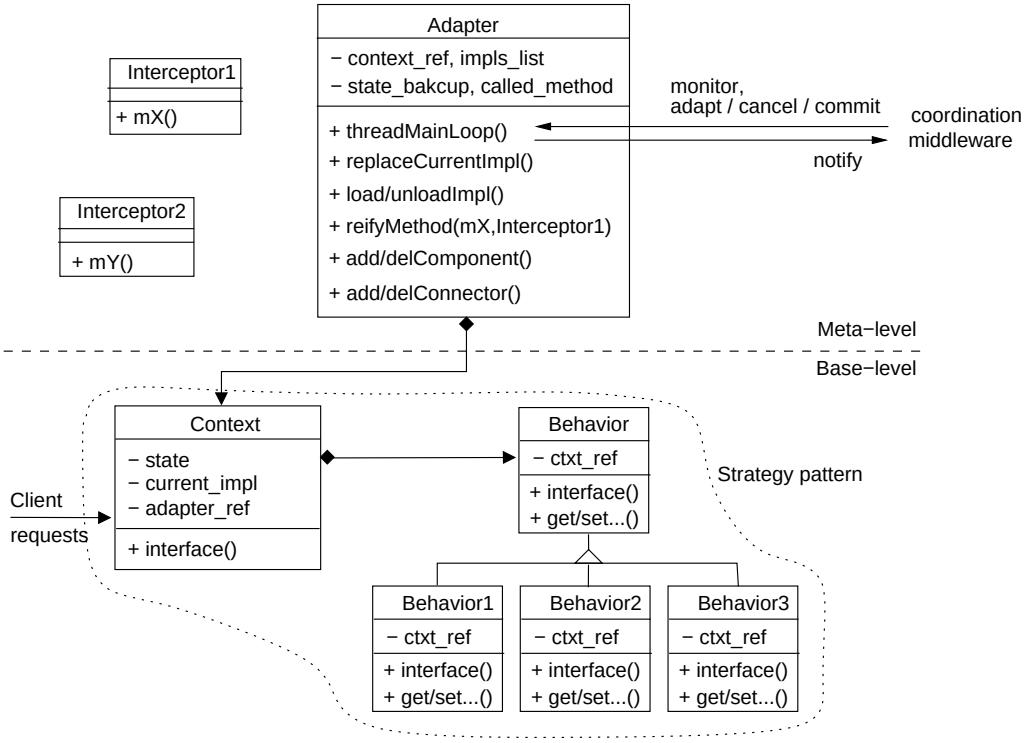


FIG. 6.2 – Modèle de composant adaptable *Aceel*.

La figure 6.2 montre l'architecture à deux niveaux d'un composant *Aceel*. Le niveau de base est conçu suivant le patron *strategy*. Les implantations alternatives du composant sont encapsulées dans les classes dérivées de la classe abstraite `Behavior`. Celle-ci définit l'interface du composant qui est la même que celle de l'objet `Context`. Seul visible de

l'extérieur, l'objet `Context` joue le rôle d'intermédiaire entre le composant et ses clients en déléguant leurs appels vers l'implantation courante. Il encapsule dans son attribut `state` un éventuel état rémanent du composant qui est séparé des implantations. Cet état est constitué des attributs du composant qui persistent entre deux appels successifs aux méthodes de l'interface.

Le méta-niveau contient le méta-objet `Adapter` qui exécute effectivement l'adaptation du composant. Il possède plusieurs méthodes dont chacune effectue une action d'adaptation particulière, tel que le remplacement d'implantation, le paramétrage d'une variable d'état, etc. (cf. section 7.1.2.1). Ces méthodes sont exécutées quand `Adapter` reçoit un ordre émanant de l'intergiciel de coordination des adaptations. `Adapter` réalise aussi une introspection du niveau de base du composant et notifie cet intergiciel quand il détecte des variations pertinentes.

Dans le cas d'un modèle de composant hiérarchique, le méta-objet `Adapter` d'un composant composite maintient une structure de graphe orienté qui représente son architecture. Les sommets et les arcs de ce graphe correspondent respectivement aux sous-composants interagissants et aux connecteurs qui les relient.

Le méta-niveau peut éventuellement contenir des intercepteurs qui sont des objets sans état utilisés pour réifier les appels aux méthodes du composant. Les intercepteurs peuvent réaliser des traitements spécifiques avant et/ou après l'appel à une méthode particulière.

Pour choisir les actions d'adaptation à exécuter, l'intergiciel de coordination utilise une politique d'adaptation écrite par le développeur du composant dans un script séparé. La séparation de la politique d'adaptation du méta-objet `Adapter` permet de découpler l'entité qui choisit l'action d'adaptation adéquate et l'entité qui exécute effectivement cette action. Avec ce découplage, il est possible de modifier la politique d'adaptation en cours d'exécution en soumettant un nouveau script à l'intergiciel de coordination . Cette modification dynamique peut être vue comme une forme de réflexion à un « méta-méta-niveau ». Ce dernier contrôle le comportement de `Adapter` qui contrôle, à son tour, le comportement du composant.

La modification dynamique de la politique d'adaptation peut être utilisée pour rajouter une nouvelle implantation non anticipée ou pour changer les règles d'adaptation en cours d'exécution. En particulier, une règle d'adaptation peut être changée pour exploiter les nouvelles ressources qui apparaissent comme les périphériques branchés à chaud (p. ex. clés USB). En plus du développeur, l'utilisateur final de l'application peut aussi changer les *indices de qualité* des *dimensions d'adaptation* selon ses préférences.

6.2.1.4 Politique d'adaptation d'un composant *Aceel*

La figure 6.3 présente la grammaire du langage déclaratif que nous avons définie pour exprimer les *politiques d'adaptation* des composants *Aceel*. Une politique d'adaptation comporte un entête et un corps. L'entête spécifie la liste des implantations alternatives du composant, le type de ces implantations (interruptibles ou non, cf. section 7.1.2), les endroits où elles sont stockées et les moments de leurs créations et de leurs destructions. L'entête spécifie aussi si le composant possède un état rémanent séparé

de ses implantations ou si chaque implantation maintient un état interne qui peut être transféré vers les autres.

La création *initiale* des implantations alternatives consomme plus de mémoire que la création *différée*. Toutefois, après la phase d'initialisation du composant, elle assure un temps de réponse rapide lors des adaptations par changement d'algorithmes. Par conséquent, le choix entre ces deux modes de création se fait en fonction de la disponibilité des ressources (p. ex. la mémoire libre, la latence d'un lien réseau quand l'implantation doit être téléchargée) et en fonction de la nature de l'application (p. ex. les applications multimédia exigent un temps de réponse court).

```
1:        adap_policy     ::= header  body
2:        header          ::= state_type  altern_impls  impls_type
3:        body            ::= adap_dimensions
4:
5:        state_type      ::= 'separated'  |  'transferable'
6:        altern_impls    ::= impl  |  impl  altern_impls
7:        impls_type      ::= 'atomic'  |  'restartable'  |  'cancelable'
8:        impl            ::= name  url  creation  destruction
9:        name            ::= <string>
10:       url             ::= <string>
11:       creation        ::= 'initial'  |  'differed'
12:       destruction     ::= 'immediate'  |  'differed'
13:
14:       adap_dimensions ::= adapt_dimension  |  adapt_dimension  adap_dimensions
15:       adapt_dimension ::= name  weight  polling_periods  adap_rules
16:       weight          ::= <positive_real>
17:       polling_periods ::= resource period  |  resource period  polling_periods
18:       resource        ::= <string>
19:       period          ::= <positive_integer>
20:       adap_rules      ::= adapt_rule  |  adapt_rule  adap_rules
21:       adapt_rule      ::= condition  adapt_actions  quality_index
22:       condition       ::= term  |  'not' condition  |  condition  'or'  condition
23:                           |  condition  'and'  condition
24:       term            ::= resource  operator  threshold
25:       threshold       ::= <real>  unit
26:       operator        ::= '>'  |  '>=' |  '='  |  '<' |  '<='
27:       unit            ::= ''  |  'Mb'  |  'Kbps'  |  '%'  |  ...
28:       adap_actions    ::= action  params  cost |  action params cost  adap_actions
29:       action          ::= tuneParameter  |  replaceImpl  |  reifyMethod
30:                           |  addComponent  |  delComponent
31:                           |  addConnector  |  delConnector
32:       params          ::= <string>
33:       cost            ::= <positive_real>
34:       quality_index   ::= <percentage>
```

FIG. 6.3 – Grammaire des politiques d'adaptation.

Le corps de la politique d'adaptation comporte les *dimensions d'adaptation* du composant. Chacune d'elles est définie par un nom, un poids, une liste de périodes de test des ressources et une liste de *règles d'adaptation*. Ces règles indiquent comment modifier le composant en fonction des variations des ressources. Chaque règle d'adaptation définit une position possible du composant dans sa dimension. Ainsi, pour choisir le *mode de fonctionnement* du composant, il faut spécifier sa position en sélectionnant une règle

d'adaptation dans chacune de ses dimensions.

Les règles d'adaptation sont sous la forme `condition adapt_actions quality_index`. Le champ `condition` dénote une configuration particulière de l'environnement. Il s'agit d'une expression logique dont les termes sont des comparaisons d'attributs de ressources par rapport à des seuils. Quand cette expression devient vraie, la règle correspondante est qualifiée *d'applicable*. Les actions d'adaptation `adapt_actions` peuvent être alors exécutées en séquence afin de modifier la position du composant dans la dimension d'adaptation associée.

Une action d'adaptation est spécifiée par les champs `action`, `params` et `cost`. Le premier fait référence à l'une des méthodes du méta-objet `Adapter` qui modifie le sujet de l'adaptation. Le second précise les informations nécessaires à l'exécution de cette méthode (p. ex. valeur d'un attribut, nom d'une implantation, etc.). Le troisième spécifie le coût de l'exécution de cette méthode. Ce coût peut inclure le temps d'exécution de la méthode d'adaptation et les ressources qu'elle consomme.

La qualité de service qui résulte de l'application d'une règle d'adaptation est spécifiée dans son champ `quality_index`. Cette valeur permet de sélectionner, dans chaque dimension d'adaptation, la meilleure règle si plusieurs sont applicables.

L'extrait de code suivant montre un exemple d'une *dimension d'adaptation* avec une règle qui dépend des variations de deux ressources.

```
adpat_dimension: 'Reception buffer size', weight = 2
    polling_periods:
        poll memory every 30s
        poll network[10.2.13.34].bandwidth every 240s
    adapt_rules:
        rule:
            when memory.free >= 10MB and network[10.2.13.34].bandwidth >= 512Kbps:
            tuneParameter state.bufferSize = 8MB
            quality_index: 80%
```

6.2.2 Aspects dynamiques

Quand l'intergiciel de coordination prend la décision d'appliquer une règle d'adaptation donnée, il envoie au composant concerné l'ordre d'exécuter la ou les actions cette règle. Cet ordre est traité par le méta-objet `Adapter` qui entame alors le processus d'adaptation suivant :

6.2.2.1 Processus d'adaptation

1. Geler l'interface du composant afin de préserver l'intégrité de son état durant l'adaptation.
2. Sauvegarder les informations nécessaires pour éventuellement annuler les actions d'adaptation qui vont suivre.
3. Exécuter les actions d'adaptation spécifiées en appelant pour chaque action la méthode correspondante de `Adapter`.
4. Notifier le succès où l'échec de cette exécution à l'intergiciel de coordination.

5. Attendre l'ordre de valider ou d'annuler les actions d'adaptation réalisées.
6. Dégeler l'interface du composant pour reprendre le traitement des nouveaux appels clients.

Le premier pas du processus d'adaptation est réalisé par intercession de l'interface de l'objet `Context`. Il a pour but de mettre en attente les appels clients de façon transparente. Ceux-ci sont délégués ultérieurement à l'implantation courante, après le dégel de l'interface du composant. Chaque action d'adaptation modifie la position du composant dans la *dimension d'adaptation* à laquelle elle appartient. Ceci conduit au changement du mode de fonctionnement du composant, et donc au changement de sa consommation des ressources et de la qualité du service qu'il fournit.

Avant d'exécuter une action adaptation, le méta-objet sauvegarde les informations nécessaires pour réaliser une reprise en cas de l'échec de cette action ou pour pouvoir l'annuler sur demande de l'intergiciel de coordination. En particulier, si l'action en question modifie une variable d'état, `Adapter` préserve une copie de celle-ci. De même, si `Adapter` remplace l'implantation courante, il garde celle-ci en mémoire jusqu'à validation de l'adaptation par l'intergiciel de coordination. L'échec d'une adaptation peut être dû à la dynamicité de l'environnement qui change avant que la réaction à une variation ultérieure ne soit terminée. Ce cas de figure est probable si la période entre deux variations successives de l'environnement est très courte.

Par ailleurs, comme les variations de l'environnement sont asynchrones par nature, elles peuvent déclencher une adaptation durant le traitement d'un appel client par l'implantation courante. Dans ce cas, avant d'exécuter le troisième pas du processus d'adaptation, le méta-objet peut soit attendre la fin du traitement en cours, soit l'arrêter puis le reprendre ultérieurement ou encore l'abandonner et renvoyer un message d'erreur au client. Ce choix incombe au développeur car il dépend avant tout du code métier du composant, notamment si ce code modifie l'état du composant de façon réversible ou non. Il peut aussi dépendre de contraintes sur le temps de réponse du composant, qui sont imposées par ses clients. Dans le modèle *Aceel*, nous offrons au développeur la possibilité de choisir l'une ou l'autre de ces trois alternatives via la politique d'adaptation du composant. La mise en œuvre de cette possibilité est détaillée dans le chapitre suivant (cf. section 7.1.2).

Dans le cas d'une adaptation algorithmique d'un composant qui ne possède pas un état séparé, il peut s'avérer nécessaire de transférer l'état interne de l'implantation courante vers celle qui la remplace. Ce cas spécifique est détaillé dans la section suivante.

6.2.2.2 Gestion du transfert d'état entre implantations

Le transfert d'état entre des implantations alternatives constitue un problème important pour les développeurs de composants adaptables[1]. Il ne peut être implanté de façon générique (et donc ne peut être automatisé à l'aide d'un support à l'adaptation) car il dépend étroitement de la nature du traitement réalisé par le composant et des

[1] Cette discussion est aussi valable pour le remplacement dynamique d'un composant par un autre composant du même type.

types des données qui constituent son état. De plus, il peut avoir un surcoût important (en mémoire et en temps d'exécution) si ces données sont volumineuses. Dans la littérature on trouve quatre approches différentes pour résoudre ce problème.

1. La première approche, qui est proposée par Szyperski, contourne ce problème en définissant les composants comme des entités sans état [Szypersk00]. Cette restriction ne peut s'appliquer à toutes les applications. En effet, dans beaucoup de cas il est nécessaire de préserver les données manipulées bien au-delà d'un seul appel de méthode. L'idée de réaliser cela par le passage des données rémanentes comme arguments à chaque appel de méthode implique un surcoût important si ces données sont volumineuses et si l'appel est distant.
2. La deuxième approche est celle adoptée par la majorité des travaux qui ont adressé ce problème (p. ex. *MolèNE*). Elle consiste à utiliser une représentation de l'état propre à chaque implantation et laisser au développeur le soin de programmer le transfert d'état entre les implantations alternatives. Cet effort de programmation doit être répété à chaque fois qu'une nouvelle implantation est rajoutée. Il requiert une bonne connaissance des détails internes des différentes implantations et empêche donc l'utilisation d'implantations « boîtes noires » fournies par des tiers.
3. La troisième approche pallie l'inconvénient de la deuxième en fournissant un support qui, à partir d'une description donnée par le développeur, génère automatiquement le code qui réalise le transfert d'état. En d'autres termes, à l'instar de l'écriture des politiques d'adaptation, le développeur programme le transfert d'état de façon déclarative et non plus de manière procédurale impérative. À titre d'exemple, pour le transfert d'état entre des composants du même type, [Azzouz03] utilise des méta-informations décrivant l'état de chaque composant et des règles de correspondance entre les attributs équivalents. Ainsi, l'attribut `heat` et l'attribut `temp` appartenant à deux composants distincts sont tous deux décrits par la méta-information `température` et une règle de conversion des degrés Celsius vers Fahrenheit et vice versa.

   Cette approche est très utilisée dans les adaptations évolutives (p. ex. lors d'une mise à jour logicielle) où les différences entre les représentations de l'état du composant peuvent être importantes. Le transfert d'état est alors effectué une seule fois depuis l'ancienne implantation (ou version du composant), qui sera abandonnée, vers la nouvelle. Par conséquent la performance de ce transfert est moins cruciale que dans le cas des adaptations réactives où les remplacements d'implantation sont nombreux et dynamiques.
4. La quatrième approche est capturée par le patron *strategy* qui instaure une séparation entre l'état rémanent du composant et ses différentes implantations. Cet état est alors représenté avec des structures de données communes à toutes les implantations. Contrairement à la première approche, ici un composant peut avoir un état encapsulé dans son objet `Context`, seules les implantations alternatives sont sans état [Gamma95]. La séparation de l'état permet d'éviter le transfert de celui-ci à chaque remplacement d'implantation. Par rapport aux deux approches

précédentes celle-ci simplifie la tâche du développeur qui a seulement à choisir une structure de données adéquate pour représenter l'état de son composant. Il n'a pas à programmer plusieurs conversions ou règles de correspondances entre des structures de données différentes. Toutefois, comme la deuxième approche, celle-ci empêche l'utilisation d'implantations « boîtes noires » fournies par des tiers.

Dans le cas du modèle *Aceel*, comme nous nous intéressons en premier lieu aux adaptations réactives, nous avons opté pour la quatrième approche car sans transfert d'état les adaptations par remplacement d'implantation sont plus rapides. Toutefois, il subsiste des cas particuliers où il est impossible de trouver une représentation de l'état du composant qui est commune à toutes ses implantations.

Un exemple de ces cas est la gestion de l'allocation mémoire par un composant adaptable. Ce dernier peut avoir un état rémanent constitué d'une liste des zones mémoire libres et plusieurs implantations correspondant aux algorithmes BestFit, FirstFit, WorstFit, etc. Ce composant peut s'adapter en changeant son algorithme d'allocation pour optimiser l'utilisation de la mémoire ou pour diminuer son temps de réponse. En fonction de l'algorithme sélectionné, la liste des zones libres doit être triée selon les adresses mémoire ou selon la taille des zones libres et donc elle ne peut être utilisée directement par tous ces algorithmes.

Pour pallier ce problème, nous offrons au développeur le choix d'outre passer l'utilisation d'un état séparé, en adoptant la deuxième approche quand c'est nécessaire. Pour cela, il suffit de fixer le champ `state_type` dans la politique d'adaptation du composant à `transferable` et surcharger la méthode `transferState` de la classe `Behavior`. Cette méthode sera automatiquement appelée par `Adapter` à chaque remplacement de l'implantation courante.

6.3 Service de surveillance de l'environnement

La plate-forme *Aceel* fournit sur chaque site qui accueille des composants adaptables un service pour la surveillance de l'environnement d'exécution. Comme le montre la figure 6.4, le service de surveillance est constitué d'un *moniteur maître* qui contrôle plusieurs moniteurs dédiés chacun à une ressource système particulière (p. ex. mémoire, cpu, réseau, batterie). Rappelons ici que chaque ressource applicative est surveillée directement par le méta-objet du composant qui possède cette ressource (cf. section 6.2.1.3).

L'emploi d'un service de surveillance permet de partager les moniteurs des ressources entre plusieurs applications. Ce service leur évite de gérer elles-mêmes le suivi des variations des ressources qu'elles utilisent. En particulier, l'utilisation d'une interface homogène pour communiquer avec l'intergiciel de coordination (cf. section 7.2.2) permet de masquer l'hétérogénéité des plates-formes d'exécution aux applications adaptables.

Le moniteur maître reçoit les requêtes émises par l'intergiciel de coordination et les redirige vers les moniteurs des ressources concernées. L'intergiciel de coordination peut *consulter* l'attribut d'une ressource donnée à tout moment (p. ex. taille de la mémoire disponible). En particulier, avant la prise de décisions d'adaptations, il consulte les niveaux de disponibilité des ressources référencées dans plus d'une politique d'adaptation

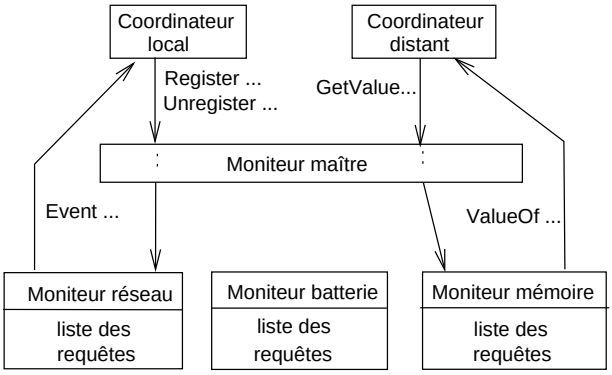


FIG. 6.4 – Service de surveillance de l'environnement.

afin d'éviter les conflits de partage de ressources.

L'intergiciel de coordination peut aussi *s'enregistrer* auprès d'un moniteur de ressource pour qu'il le notifie à chaque variation pertinente de celle-ci. Dans ce cas, il spécifie dans sa requête de surveillance les seuils de variation et la fréquence de test (polling) de cette ressource, tels que définis dans la politique d'adaptation du composant intéressé.

Les moniteurs des ressources notifient uniquement des événements simples dont chacun dénote la variation d'un attribut par rapport à un seuil. Un événement peut être destiné à plusieurs composants intéressés par la même variation. Le choix de ne pas gérer la notification d'événements complexes facilite la mise en œuvre des moniteurs efficaces, qui ont une consommation de ressources et un temps de réponse minimaux (cf. section 7.2.1).

La notification rapide d'événements simples augmente la réactivité des composants aux variations de l'environnement sans les empêcher d'avoir des règles d'adaptation complexes. En effet, grâce à la combinaison logique de plusieurs termes, les `conditions` des règles d'adaptation des composants sont assimilables à des événements complexes. Ces derniers ne sont pas notifiés en entier par un seul surveillant, mais ils sont notifiés et évalués de façon incrémentale au fur et à mesure de l'occurrence des événements simples qui les composent.

6.4 Intergiciel de coordination des adaptations

Coordonner les actions d'adaptation de plusieurs composants nécessite de connaître l'ensemble de leurs politiques d'adaptation et d'avoir une vision globale des ressources qu'ils utilisent. Ces informations ne peuvent être gérées de façon efficace par les composants eux-mêmes. Mais elles doivent être gérées à l'extérieur de ceux-ci, à un niveau logiciel inférieur.

Dans le cas contraire, on aurait eu une structure de composant complexe qui contient, en plus du code métier et du code adaptatif, le code qui gère les dépendances avec les autres composants. Une telle structure présente plusieurs inconvénients parmi lesquels la

difficulté de réutiliser et de maintenir les trois types de codes, le risque d'incohérence des informations redondantes — risque qui est amplifié dans le cadre distribué, le gaspillage d'une partie de la mémoire qui est une ressource précieuse sur les équipements mobiles, et la surcharge du réseau avec l'échange de messages dont le nombre croit de façon combinatoire en fonction du nombre de composants.

En ce qui concerne le système d'exploitation, si la gestion des ressources fait partie de son rôle principal, l'interprétation des politiques d'adaptation des composants et la prise en compte de leurs éventuelles interactions sont des tâches de plus haut niveau qu'il n'est pas destiné à gérer. De façon plus générale, l'évolution des systèmes d'exploitation depuis les systèmes monolithiques vers les micros, puis les exo-noyaux montre que ces derniers relègue de plus en plus de tâches vers l'espace utilisateur pour se limiter à la gestion des composants matériels de l'ordinateur.

Par conséquent, nous pensons que le meilleur niveau pour gérer la coordination des composants adaptables est le niveau intergiciel. La plate-forme *Aceel* fournit sur chaque site accueillant des composants un tel intergiciel. Afin de considérer les dépendances dues aux interactions entre les composants, l'intergiciel de coordination utilise une politique propre à chaque application adaptable. Cette *politique de coordination* est écrite par le développeur de l'application, en se basant sur les politiques d'adaptation des composants assemblés et sur la nature de leurs interactions.

Nous présentons ci-après l'architecture de l'intergiciel de coordination ainsi que la grammaire des politiques de coordination. Ensuite, nous détaillons le fonctionnement de cet intergiciel qui passe par l'interprétation des politiques des composants, puis la prise de décision et l'exécution des actions d'adaptation.

6.4.1 Architecture de l'intergiciel de coordination

La figure 6.5 montre l'architecture orientée composants de l'intergiciel de coordination. Sur chaque site cet intergiciel est constitué d'un composant *coordinateur*, d'un composant *optimiseur* et d'un ou plusieurs composants *synchroniseurs*.

Le *coordinateur* gère l'enregistrement des composants et des applications adaptables et interprète leurs différentes politiques (cf. section 6.4.3). Il représente le cœur névralgique de l'intergiciel car c'est lui qui reçoit les événements notifiant les variations de l'environnement, met à jour les conditions des règles d'adaptation, utilise *l'optimiseur* pour choisir les actions d'adaptation à appliquer, communique avec les coordinateurs des autres sites pour gérer les dépendances entre les composants distants et lance les *synchroniseurs* qui exécutent les actions choisies.

L'optimiseur représente le cerveau de l'intergiciel qui est responsable de la prise de décision des adaptations. Son rôle est de sélectionner les meilleurs modes de fonctionnement des composants afin d'optimiser leur qualité de service, en fonction des contraintes de l'environnement (cf. section 6.4.4).

Les *synchroniseurs* sont les bras de l'intergiciel de coordination car ils contrôlent l'exécution des actions d'adaptation. Il y a un *synchroniseur* pour chaque application adaptable sur chacun des sites qui l'accueillent (cf. section 6.4.5).

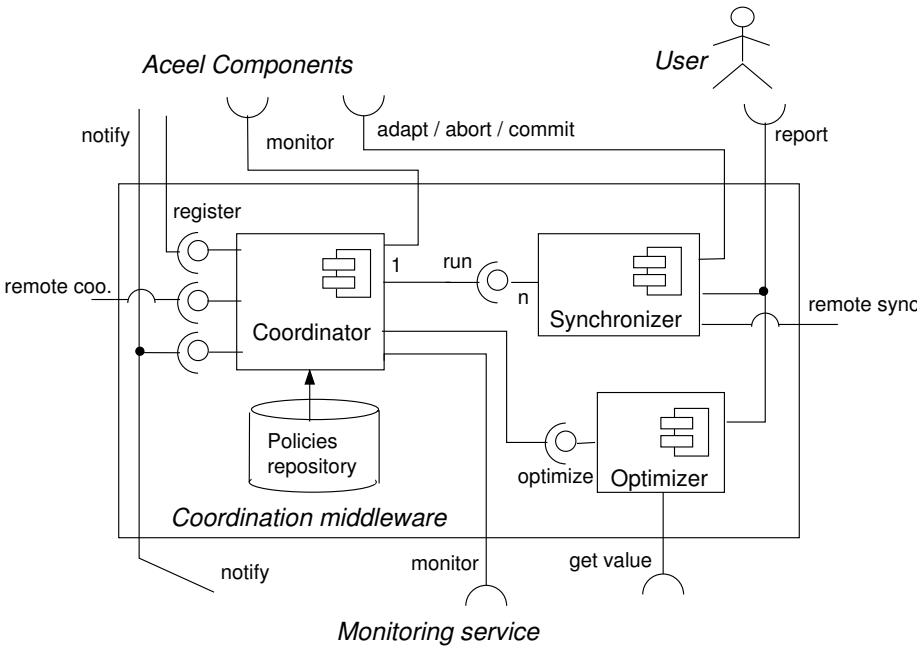


FIG. 6.5 – Architecture de l'intergiciel de coordination des adaptations.

6.4.2 Politique de coordination des composants interagissants

Une politique de coordination est écrite dans un script séparé en respectant la grammaire illustrée par la figure 6.6.

Une politique de coordination est composée d'un entête et d'un corps. L'entête permet d'identifier les composants adaptables de l'application ainsi que les sites où ils sont déployés. Le corps est formé par une suite de *règles de coordination*. Ces règles garantissent la cohérence des adaptations des composants d'une même application et assurent ainsi la continuité de leur interaction après-coup. Chaque règle spécifie quels composants doivent toujours être adaptés ensemble. Une règle de coordination est appliquée quand l'un des composants spécifiés dans son champ `condition` doit exécuter une adaptation spécifique. Dans ce cas, le champ `sync_actions` de cette règle spécifie les actions d'adaptation à appliquer aux autres composants. Ce champ précise aussi quelles sont les actions qui doivent être exécutées en séquence et quelles sont celles qui doivent être exéctuées en parallèle (cf. section 6.4.5).

Lors du déploiement d'une application distribuée, le *coordinateur* de chaque site d'accueil enregistre sa politique de coordination et traduit ses règles sous une forme interne. Plus tard, quand un composant de l'application est créé, sa politique d'adaptation est à son tour interprétée telle que le décrit dans la section suivante.

```
-------------------------------------------------------------------------------
1:          coord_policy  ::= header body
2:          header        ::= appli_uid comp_list
3:          comp_list     ::= comp_uid host | comp_uid host  comp_list
4:          appli_uid     ::= <string>
5:          comp_uid      ::= <string>
6:          host          ::= <string>
7:
8:          body          ::= rules_list
9:          rules_list    ::= coord_rule | coord_rule  rules_list
10:         coord_rule    ::= condition sync_actions
11:         condition     ::= term | term 'or' condition
12:         term          ::= comp_uid 'runs' action
13:         sync_actions  ::= operator actions_list
14:         operator      ::= 'parallel' | 'sequential'
15:         actions_list  ::= adapt_actions | adapt_actions  sync_actions
16:         adapt_actions ::= comp_action | comp_action  adapt_actions
17:         comp_action   ::= comp_uid action params
18:         action        ::= <string>
19:         params        ::= <string>
-------------------------------------------------------------------------------
```

FIG. 6.6 – Grammaire des politiques de coordination.

6.4.3 Interprétation des politiques d'adaptation des composants

Lors de la création d'un composant *Aceel*, le *coordinateur* parcours sa politique d'adaptation afin d'extraire les informations nécessaires à la prise de décisions d'adaptation. Pour chacun des termes qui forment les conditions des règles d'adaptation, le *coordinateur* envoie une requête d'enregistrement au surveillant de la ressource référencée. Le surveillant d'une ressource est soit le service de surveillance de l'environnement si cette ressource est fournie par une plate-forme d'exécution, soit le mécanisme d'introspection d'un composant *Aceel*. Dans le dernier cas, le composant surveillé et le composant intéressé par ses variations doivent appartenir à la même application. A la fin de l'analyse de la politique d'adaptation, le *coordinateur* initialise à faux les conditions de toutes les règles d'adaptation du composant.

Les tests des ressources par les surveillants dédiés commencent dès la réception des requêtes d'enregistrement. Les notifications émises suite aux premiers tests permettent à l'intergiciel de coordination de déterminer quelles règles d'adaptation sont applicables juste après l'analyse de la politique d'adaptation d'un composant. Ces règles ont pour effet d'initialiser le composant en fonction de l'état actuel de l'environnement. En particulier, il est possible de sélectionner l'implantation courante du composant si ce dernier en possède plusieurs. Dans ce cas, l'intergiciel de coordination ordonne au composant d'exécuter l'action d'adaptation qui charge en mémoire l'implantation sélectionnée même si la création de celle-ci est spécifiée comme différée dans la politique d'adaptation.

De façon générale, quand le *coordinateur* reçoit une notification, il met à jour puis réévalue les conditions des règles d'adaptation concernées pour voir si elles ont changées. Le changement de l'une de ces conditions signifie que le mode de fonctionnement courant du composant correspondant n'est plus viable ou n'est plus optimal. La non viabilité est la conséquence de la diminution des ressources de l'environnement qui de-

viennent insuffisantes pour continuer le mode de fonctionnement courant. Au contraire, la sous optimalité est due à l'augmentation des ressources telle que l'exécution d'un autre mode de fonctionnement qui offre une qualité de service meilleur devient possible. Dans les deux cas, le *coordinateur* doit prendre la décision d'adapter ou non chacun des composants concernés en utilisant *l'optimiseur*.

6.4.4 Prise des décisions d'adaptations

Pour réagir à une variation de l'environnement, l'intergiciel de coordination pourrait adapter tous les composants de son site en les configurant dans leurs meilleurs modes de fonctionnement viables. Pour cela il suffit de sélectionner, dans chaque dimension d'adaptation de chaque composant, la règle qui a le plus grand indice de qualité parmi toutes les règles applicables. Toutefois, ce choix réalisé sur une base individuelle pour chaque composant pourrait conduire à des adaptations conflictuelles entres les composants concurrents et/ou à des adaptations incohérentes entre les composants interagissants (cf. chapitre 5).

Pour éviter ces problèmes, l'intergiciel de coordination doit plutôt sélectionner la *combinaison adéquate* parmi les modes de fonctionnements viables des composants afin de maximiser leur qualité de service. Nous montrons ci-après, comment choisir la *combinaison adéquate* des adaptations multi-composants en modélisant la prise de décision comme un problème *d'optimisation combinatoire*.

6.4.4.1 Modélisation de la coordination des adaptations multiples

Pour modéliser la coordination des adaptations multiples nous avons utilisé les variables et les fonctions suivantes :

- n : le nombre des composants *Aceel* en cours d'exécution sur un site.
- c_i : la classe de priorité du composant i ($i \in [1, n]$).
- m_i : le nombre des dimensions d'adaptation du composant i ($m_i \geq 1$).
- p_{ij} : le poids de la dimension j du composant i ($j \in [1, m_i]$).
- o_{ij} : le nombre des règles d'adaptation dans la dimension j.
- D_{ij} : l'ensemble des positions possibles du composant i dans la dimension j.
- x_{ij} : la position actuelle du composant i dans la dimension j, donnée par le numéro de la règle d'adaptation sélectionnée ($x_{ij} \in D_{ij} = \{1, 2 \ldots, o_{ij}\}$).
- $q(x_{ij})$: l'indice de qualité associé à chaque règle de la dimension j ($q(x_{ij}) \in [0, 100]$).
- E_i : l'ensemble des ressources référencées dans les règles d'adaptation du composant i. Soit $E = \cup_{i=1}^{n} E_i$.
- P : l'ensemble des ressources partagées par les composants concurrents et référencées dans leurs règles d'adaptation ($P = \{E_i \cap E_{i'} \mid \forall i, i' \in [1, n] \text{ et } i \neq i'\}$).
- $dispo(r)$: le niveau de disponibilité de la ressource $r \in E$ à un instant donné.
- $seuil_i(r)$: la part de la ressource r utilisée par le composant i. Elle correspond au seuil maximum par rapport auquel cette ressource est comparée dans les conditions des règles d'adaptation sélectionnées.

Le mode de fonctionnement et la qualité de service d'un composant i sont représentés, respectivement, par le vecteur X_i et la fonction $Q(X_i)$ suivants :

$$X_i = [x_{i1} \dots x_{ij} \dots x_{im_i}], \quad X_i \in D_i \quad \text{et} \quad D_i = \prod_{j=1}^{m_i} D_{ij} \tag{6.1}$$

$$Q(X_i) = \frac{\sum_{j=1}^{m_i} p_{ij}\ q(x_{ij})}{\sum_{j=1}^{m_i} p_{ij}} \tag{6.2}$$

De façon plus générale, les modes de fonctionnement des n composants déployés sur un site et la somme de leurs qualités de service sont représentés, respectivement, par :

$$X = [X_1 \dots X_i \dots X_n], \quad X \in D \quad \text{et} \quad D = \prod_{i=1}^{n} D_i = \prod_{i=1}^{n} (\prod_{j=1}^{m_i} D_{ij}) \tag{6.3}$$

$$Q(X) = \sum_{i=1}^{n} c_i\, Q(X_i) = \sum_{i=1}^{n} c_i \left(\frac{\sum_{j=1}^{m_i} p_{ij}\, q(x_{ij})}{\sum_{j=1}^{m_i} p_{ij}} \right) \tag{6.4}$$

Au vu de cette notation, le rôle du composant *optimiseur* revient à trouver le vecteur $X \in D$ qui maximise la fonction $Q(X)$ en respectant les contraintes (1) de disponibilité des ressources, (2) de partage de ces ressources sans conflits et (3) de cohérence des interactions entre les composants.

Les contraintes de premier type sont vérifiées si les conditions des règles d'adaptation sélectionnées sont vraies. Elles sont exprimées par :

$$\forall r \in E, \quad \forall i \in [1, n] : \quad seuil_i(r)\ <=\ dispo(r) \tag{6.5}$$

Les contraintes du second type évitent les adaptations conflictuelles dues au partage des ressources. Elles sont sous la forme :

$$\forall r \in P : \quad \sum_{i \in P} seuil_i(r)\ <\ dispo(r) \tag{6.6}$$

Les contraintes du troisième type sont déduites à partir de la politique de coordination des composants de chaque application. Soient :

- rc : une règle de coordination des composants de l'application notée *App*.
- C_{rc} : l'ensemble des composants référencés dans le champ `condition` de rc.
- S_{rc} : l'ensemble des composants référencés dans le champ `sync_actions` de rc.
- v : le numéro de la règle d'adaptation contenant l'action associée au composant i dans le champ `sync_actions` de rc.

Chaque règle de coordination rc définie une fonction discrete f qui fixe certains éléments des vecteurs $X_{i'}$ où $i' \in S_{rc}$ en fonction de certains éléments des vecteurs X_i où $i \in C_{rc}$.

$$\begin{array}{rcccl} f & : & D_{ij} & \longrightarrow & D_{i'j'} \\ & : & v & \longrightarrow & v' \end{array}$$

Ainsi, la règle de coordination rc impose une contrainte sur les vecteurs $X_{i,i\in App}$ qui est exprimée par :

$$\forall i' \in S_{rc} : \quad \exists i \in C_{rc}, \quad \exists j \in [1, m_i], \quad \exists j' \in [1, m_{i'}], \quad \text{tels que} \quad x_{i'j'} = f(x_{ij}) \tag{6.7}$$

Cette formulation permet de voir l'adaptation de plusieurs composants comme un *problème d'optimisation distribuée sous contraintes (PODC)* [Nocedal99]. Précisément, il s'agit d'un problème de *programmation non linéaire* où :

- La fonction objectif est la qualité de service $Q(X)$ qui est non linéaire.
- Les variables de décision sont les éléments du vecteur X qui représentent les numéros des règles d'adaptation à appliquer.
- Le domaine de définition de ces variables est l'ensemble fini et discret D.
- Les contraintes du problème sont les (in)équations 6.5, 6.6 et 6.7 dues aux dépendances de partage de ressources et aux dépendances d'interaction entre les composants adaptables.
- Les contraintes du troisième type sont des contraintes distribuées car elles peuvent lier des variables appartenant à des composants distants.

Dans la section suivante nous analysons les solutions existantes pour ce problème d'optimisation combinatoire appartenant à la classe NP-difficile. Ensuite, nous argumentons le choix de la solution retenue, d'abord dans le cas centralisé où il n'y a pas de contraintes distribuées, puis dans le cas distribué où ces contraintes existent.

6.4.4.2 Résolution du problème dans le cas centralisé

Pour résoudre un problème centralisé d'optimisation combinatoire sous contraintes, de nombreuses méthodes ont été développées en recherche opérationnelle [Hillier01] et en intelligence artificielle [Russell02]. Ces méthodes peuvent être classées sommairement en deux grandes catégories : les *algorithmes exacts* et les *métaheuristiques*.

Les algorithmes exacts sont basés sur une énumération systématique des éléments du domaine de définition. Ils diffèrent par la façon avec laquelle ils parcourent ce domaine souvent représenté avec un arbre. Bien que ces algorithmes garantissent d'atteindre la solution optimale, leurs temps de calcul combinatoire les rend inefficaces quand la taille du domaine de définition est très grande. Dans notre cas, cette taille est donnée par : $s = \mid D \mid = \prod_{n}^{i=1}(\prod_{j=1}^{m_i} o_{ij})$.

Quand la taille du domaine de définition est très grande, il convient d'utiliser une métaheuristique pour s'assurer que le temps nécessaire à la prise de décision d'adaptation reste court.

Les métaheuristiques sont basées sur une amélioration itérative de la solution courante par une recherche dans le voisinage ou une recherche évolutive. Pour échapper au optima locaux, certaines métaheuristiques incorporent une composante stochastique qui permet de poursuivre une marche aléatoire vers l'optimum global. Les principales métaheuristiques utilisées sont : le recuit simulé, les algorithmes génétiques et la recherche tabou [Charon96].

1. Le recuit simulé minimise de manière stochastique une fonction de coût. C'est une méthode de recherche dans le voisinage où l'algorithme procède par de petites modifications de la solution courante, en autorisant éventuellement une dégradation temporaire de la qualité de la solution.
2. Les algorithmes génétiques nécessitent un codage des solutions potentielles sous la forme de chaînes semblables aux chromosomes. L'algorithme procède par mutation et croisement de ces codes en sélectionnant les solutions de qualité élevée à chaque nouvelle génération (évolution).
3. La recherche tabou est une méthode déterministe qui conserve en mémoire les k dernières configurations visitées. Celles-ci sont dites tabou, c'est à dire que l'on s'interdit pour les futures solutions. L'objectif premier est d'éviter une exploration cyclique pour augmenter l'efficacité de la recherche.

Contrairement aux algorithmes exacts, les métaheuristiques peuvent approcher une solution optimale en un temps polynomial. Parmi les trois métaheuristiques précédentes, le recuit simulé est le plus adapté à notre problème. En effet, d'un coté, la recherche tabou et les algorithmes génétiques sont plus coûteux en ressources. La recherche tabou consomme beaucoup de mémoire car elle conserve les configurations visitées, tandis que les algorithmes génétiques nécessitent un temps d'exécution important car à chaque itération ils évaluent toute une population au lieu d'une seule solution. Or les mécanismes d'adaptation doivent utiliser le minimum de ressources possible.

D'un autre coté, le recuit simulé est facile à implanter et sa convergence a été démontrée avec les chaînes de Markov [Press92]. De plus, cette métaheuristique possède des paramètres intrinsèques (p. ex. la température initiale, le nombre de paliers, la fonction de calcul du voisin) qu'il est possible d'adapter afin de contrôler son temps de calcul et donc la qualité de la solution obtenue. Ceci permet de réaliser des compromis entre le temps nécessaires pour prendre une décision d'adaptations coordonnées et la qualité de cette décision. Notamment, il est possible de favoriser l'un ou l'autre de ces aspects en fonction de la granularité des composants, par exemple en réalisant pour les composants applicatifs à gros grains, des adaptations moins rapides mais de meilleure qualité.

6.4.4.3 Résolution du problème dans le cas distribué

Afin d'optimiser les modes de fonctionnement des composants interagissants distants, il faut prendre en compte les contraintes distribuées lors de la recherche des valeurs adéquates pour les variables de décision x_{ij}. Pour cela, il est nécessaire de considérer l'importance de chacun des sites d'un système distribué par rapport aux autres. L'importance d'un site est définie par son administrateur en fonction de plusieurs critères tels que le type des applications qui y sont déployées, la connectivité du site, sa puissance de calcul, sa capacité de stockage, etc.

De façon générale, dans le cadre de l'adaptabilité, les sites d'un système distribué peuvent être considérés commes des sites « pairs » accueillant des composants qui ont éventuellement des priorités différentes. Quand l'intergiciel de coordination sur un site

pair réagit à une variation de l'environnement, son *coordinateur* doit chercher, avec les *coordinateurs* des autres sites, un accord sur les valeurs à affecter aux variables de décision soumises aux contraintes distribuées.

Dans certains cas spécifiques, un système distribué peut être constitué d'un seul site « maître » et de plusieurs sites « esclaves » (p. ex. les réseaux nomades formés par plusieurs terminaux mobiles reliés à un seul point d'accès). Typiquement, le site maître accueille la partie « serveur » (ou proxy) de plusieurs applications distribuées (p. ex. serveur Web, serveur de fichiers, serveur vidéo, etc.), tandis que les sites esclaves accueillent chacun la partie « client » de ces applications (p. ex. navigateur Web, client Ftp, player vidéo, etc.). Afin de favoriser les adaptations sur le site maître, le *coordinateur* de ce dernier peut imposer son choix aux sites esclaves sans chercher à trouver un accord avec eux quand il faut affecter les variables soumises aux contraintes distribuées.

A titre d'exemple, un serveur multicast peut imposer certaines propriétés des informations qu'il diffuse aux clients, telles que le format d'encodage d'une vidéo, le taux de bits transmis, etc. Les raisons derrière une telle démarche peuvent être diverses. En l'occurrence, dans un système distribué asynchrone, il est possible que les clients ne tombent jamais d'accord si on les laissait choisir les propriétés des informations diffusées par le serveur multicast. En outre, il peut être préférable de favoriser d'abord l'amélioration de la qualité de service sur le site maître par rapport aux sites esclaves, par exemple, pour servir un nombre maximum de clients.

Ci-après, nous présentons des exemples d'algorithmes pour l'optimisation distribuée dans le cas général des sites pairs, puis dans le cas spécifique des sites maître-esclave.

Cas de sites pairs

Dans le cas de sites pairs, l'adaptation doit viser à maximiser la qualité de service globale en tenant compte de l'état du système distribué. Soit N le nombre des sites qui accueillent des composant *Aceel*. La qualité de service globale Qg peut être définie comme la somme des qualités de service sur ces N sites.

$$Qg = \sum_{s=1}^{N} Q_s(X_s) \tag{6.8}$$

Les *coordinateurs* des sites pairs peuvent utiliser deux méthodes pour affecter les valeurs adéquates aux variables soumises aux contraintes distribuées. La méthode la plus triviale consiste à sélectionner un site particulier (p. ex. par une élection ou une désignation en fonction de la puissance de calcul, de la connectivité, etc.), de rassembler sur ce site toutes les données distribuées du problème (c.-à-d. les variables de décision de chaque site, leurs domaines de définition, les contraintes, etc.), de les soumettre à *l'optimiseur* pour les traiter de façon centralisée et de renvoyer la solution globale calculée vers les autres sites. Ces étapes sont répétées pour chaque variation pertinente sur chacun des sites pairs.

La deuxième méthode consiste à satisfaire en premier les contraintes locales à chaque site en utilisant un algorithme d'optimisation centralisé, puis satisfaire les contraintes distribuées en mettant en œuvre un protocole d'accord entre les sites pairs.

En détails, le *coordinateur* de chaque site s lance d'abord son *optimiseur* pour calculer, en fonction des contraintes locales, les k meilleures solutions du vecteur X_s et les k qualités de service Q_s correspondantes. Ensuite, pour chaque contrainte distribuée qui lie une variable locale x_{ij} à une variable distante $x_{i'j'}$ (cf. 6.7), le *coordinateur* de s diffuse vers son homologue qui maintient $x_{i'j'}$ les k meilleures valeurs calculées pour x_{ij}, accompagnées par les qualités de service Q_s correspondantes. Ainsi, le *coordinateur* de chaque site s va disposer en local des informations nécessaires pour sélectionner de façon déterministe le vecteur optimal X_s parmi les k vecteurs qu'il a calculés, tel que la qualité de service globale Qg soit optimale.

Cas d'un site maître

Dans le cas d'une adaptation déclenchée sur un site maître, le *coordinateur* relaxe les contraintes distribuées, lance *l'optimiseur* pour trouver le meilleur vecteur X en utilisant un algorithme centralisé, puis transmet la valeur optimale de chaque variable x_{ij} liée à une variable distante $x_{i'j'}$ vers le site esclave concerné. Chaque site esclave recalcule alors son vecteur optimal X' en commençant par fixer les variables $x_{i'j'}$ soumises aux contraintes distribuées. La valeur de chacune de ces variables est déterminée en fonction de la variable distante x_{ij} avec laquelle elle est liée. Ensuite, le *coordinateur* appelle *l'optimiseur* pour calculer les éléments restants du vecteur X' en utilisant un algorithme centralisé.

Cas de sites esclaves

Dans le cas d'une adaptation déclenchée sur un site esclave, le *coordinateur* commence par fixer les variables locales $x_{i'j'}$ liées par des contraintes distribuées aux variables x_{ij} du site maître, en fonction des valeurs actuelles de ces dernières. Ensuite, il appelle *l'optimiseur* pour calculer les éléments restants du vecteur local X' en utilisant un algorithme centralisé.

Discussion

Bien qu'elle soit facile à implanter, la première méthode exposée ci-dessus pour le cas des sites pairs doit être évitée car elle est vouée à l'échec si le site central tombe en panne. De plus, elle peut nécessiter un coût de communication prohibitif lors de la collecte des données distribuées car tous les domaines de définition D_{ij} de toutes les variables x_{ij} de chaque site pair doivent être transmis (entre autres) vers le site central. En particulier, si cette collecte est lente, les données du problème pourraient changer, à cause de la dynamicité de l'environnement, avant même leur centralisation.

Contrairement à la méthode centralisée, la deuxième méthode proposée pour les sites pairs est moins sensible aux pannes et ne nécessite pas un site qui possède une importante puissance de calcul. Cette méthode réduit le coût de la communication car seules les informations qui concernent les variables soumises aux contraintes distribuées sont échangées. De plus, au lieu d'envoyer l'ensemble de définition de chacune de ces

variables, seules les k meilleures valeurs sont transmises. La section 7.3 présente une mise en œuvre du *coordinateur* qui utilise la deuxième méthode proposée.

Concernant la prise de décisions d'adaptations sur les sites de type maître-esclave, les deux méthodes proposées ci-dessus sont simples à implanter car elles éliminent le besoin d'un protocole d'accord distribué. L'approche adoptée dans ce cas spécifique favorise les sites maîtres au détriment des sites esclaves. Le *coordinateur* de chaque site esclave se contente de préserver le *statu quo* sur le site maître en limitant la réaction aux variations de son environnement aux seules adaptations des composants locaux.

6.4.5 Exécution des adaptations

A la réception du nouveau vecteur optimal X', le *coordinateur* le compare avec le vecteur actuel X afin de déterminer si la règle d'adaptation sélectionnée dans chaque dimension a changé. Pour appliquer les nouvelles règles d'adaptation, le *coordinateur* lance le *synchroniseur* associé à chaque application qui doit être modifiée. Chaque *synchroniseur* organise l'exécution des actions d'adaptation associées aux nouvelles règles, en parallèle ou en séquence, telle que spécifiées dans la politique de coordination de l'application.

Afin de pallier le problème classique de fiabilité des systèmes distribués, le *synchroniseur* contrôle l'exécution des actions d'adaptation de façon transactionnelle. Ceci est assuré par l'utilisation d'un protocole de *validation à deux phases (2PC)* comme décrit dans la section 7.3.3.

Le choix d'associer un *synchroniseur* à chaque application présente deux avantages. Sur un plan, il permet un contrôle plus fin des adaptations des composants d'une même application, à travers sa politique de coordination. Sur un autre plan, il permet de paralléliser et d'isoler l'exécution des adaptations de plusieurs applications distribuées. Ainsi, si un échec se produit au niveau d'une application, les adaptations des autres applications ne sont pas compromises. Dans le cas d'un échec de l'adaptation, le *synchroniseur* informe l'utilisateur afin qu'il puisse intervenir sur l'application incriminée, par exemple, en l'arrêtant.

6.5 Conclusion

Nous venons de présenter notre plate-forme de composants adaptables dont les trois briques de base sont le canevas de composants *Aceel*, le service de surveillance de l'environnement et l'intergiciel de coordination des adaptations.

Le canevas de composants est basé sur un modèle réflexif qui sépare le code métier, le code qui exécute concrètement les adaptations et le code qui spécifie quand et comment s'adapter. Le dernier code prend la forme de politiques écrites dans un langage déclaratif facile à utiliser. Plusieurs types d'adaptations sont entièrement gérés par le méta-objet de chaque composant (p. ex. le paramétrage, la réification des appels, les adaptations algorithmiques, le rajout dynamique d'une nouvelle implantation, etc.). Le méta-objet préserve l'intégrité du composant en supportant plusieurs types d'implan-

tations alternatives (p. ex. atomiques, transactionnelles, etc.) et en gérant le problème de transfert d'état de façon générique.

Le service de surveillance de l'environnement possède une architecture plate qui permet une notification rapide des variations de l'environnement sous la forme d'événements simples. Ce service supporte deux modes d'interaction avec les couches logicielles supérieures. Il s'agit des modes synchrone et asynchrone pour, respectivement, consulter le niveau d'une ressource et suivre son évolution dans le temps.

L'intergiciel de coordination est la pièce maîtresse de la plate-forme *Aceel* car c'est à son niveau que les décisions d'adaptation sont prises. Notamment, il prend en compte le coût des actions d'adaptation lors de la prise décision. Cet intergiciel, conçu lui-même avec des composants adaptables, résout les problèmes de conflits sur les ressources et les incohérences qui peuvent avoir lieu lors des adaptations multiples.

Nous avons modélisé la coordination comme un problème d'optimisation distribuée de la qualité service sous les contraintes de l'environnement. En particulier, pour gérer les composants interagissants, l'intergiciel utilise une politique de coordination propre à chaque application. Cette politique, écrite aussi dans un langage déclaratif, permet un contrôle fin de l'ordre dans lequel les actions d'adaptation sont exécutées.

Sur un autre plan, la coordination des composants concurrents repose sur les préférences de l'utilisateur humain. Celles-ci sont traduites dans notre modèle par des notions telles que la qualité perceptuelle résultant de l'application d'une règle d'adaptation, le poids d'un sujet d'adaptation, la classe de priorité d'un composant et l'importance d'un site au sein d'un système distribué.

Enfin, l'exécution de plusieurs actions d'adaptation est considérée comme un service *best effort* qui, en cas de pannes ou erreurs d'exécution, permet le retour du système à sa configuration initiale.

Dans le chapitre suivant nous présentons la mise en œuvre de la plate-forme *Aceel*.

Chaptire 7

Mise en œuvre et expérimentation de la plate-forme *Aceel*

Dans le chapitre précédent, nous avons présenté la plate-forme de composants adaptables *Aceel*. Nous nous intéressons maintenant à l'implantation de cette plate-forme afin de concrétiser les concepts proposés et les valider à travers une expérimentation. D'abord, nous présentons la mise en œuvre des aspects génériques du modèle de composant *Aceel* sous la forme d'un canevas. Puis, nous montrons comment sont implantés les services intergiciels sous-jacents pour la surveillance de l'environnement et la coordination des composants interagissants et/ou concurrents. Ensuite, nous illustrons l'utilisation de la plate-forme *Aceel* pour le développement d'un navigateur Web et d'une application de vidéo à la demande adaptables en environnement mobile.

7.1 Mise en œuvre du modèle de composants *Aceel*

Nous avons mis en œuvre un canevas générique pour écrire les composants *Aceel* en utilisant le langage Python[1] et écrire leurs politiques d'adaptation et leurs politiques de coordination en utilisant le langage XML[2]. Dans les sous-sections suivantes nous argumentons le choix de ces deux langages et nous détaillons le canevas *Aceel*.

7.1.1 Choix des langages de programmation

Python est un langage de script qui permet le prototypage rapide des nouvelles idées. Il est indépendant des différentes plates-formes d'exécution car il appartient à la famille des langages à Bytecode interprété. Il est aussi facile à interfacer avec d'autres langages comme C++ ou Java ce qui permet de les utiliser pour écrire les implantations alternatives des composants. Plus important encore, Python possède des propriétés

[1] http ://www.python.org
[2] http ://www.w3.org

utiles pour l'adaptabilité telles que sa nature orientée objet, sa réflexivité et son typage dynamique.

D'une part, les techniques objets et la réflexivité forment les pierres angulaires du modèle. En particulier, le mécanisme d'introspection des composants *Aceel* repose sur la possibilité d'inspecter la valeur et le type de n'importe quel objet Python. Il est aussi possible de charger des modules Python en cours d'exécution et d'ajouter (ou d'enlever) dynamiquement un nouveau membre (un attribut ou une méthode) à un objet existant. Ces capacités d'intercession sont directement utilisées pour implanter les adaptations par remplacement d'implantation, par réification des appels de méthodes et par rajout/suppression de sous-composants.

D'autre part, la possibilité de modifier dynamiquement le type d'une variable enrichit davantage les adaptations par paramétrage de l'état d'un composant. Par exemple, une variable d'état prévue pour recevoir des valeurs entières peut, sans problème, devenir un réel si la nature des données traitées par le composant évolue.

XML est un métalangage qui sert à définir des langages déclaratifs et extensibles. D'un côté, il permet de décrire les informations sous la forme d'une arborescence de balises qui est facile à parser et facile à lire pour l'utilisateur humain. Parmi les nombreux parseurs XML existants nous avons choisi d'utiliser le parseur répandu le plus rapide afin de minimiser le temps d'analyse des politiques. Il s'agit du parseur pyRXP[3] qui présente aussi l'avantage de valider les fichiers XML par rapport aux DTDs qui définissent leurs grammaires. L'annexe A inclut les DTDs XML qui définissent la grammaire des politiques d'adaptation et la grammaire des politiques de coordination des composants *Aceel*.

D'un autre côté, il est facile d'étendre un langage XML en rajoutant de nouvelles balises ou de nouveaux attributs aux balises existantes. Ceci permet d'élargir la réaction des composants à de nouveaux types de ressources (p. ex. nouvelles technologies de périphériques) sans modifier le méta-objet `Adapter`. Il suffit de définir les attributs de ces ressources et de rajouter les moniteurs dédiés au sein du service de surveillance.

7.1.2 Canevas *Aceel*

Le canevas *Aceel* fournit des implantations prototypes des classes `Adapter` et `Context`, ainsi que des semi-implantations des classes `State`, `Behavior` et `Interceptor` (cf. figure 6.2). Pour développer un composant adaptable, il faut d'abord écrire son code métier en spécialisant les trois dernières classes, puis définir sa politique d'adaptation. Si ce composant interagit avec d'autres composants *Aceel*, alors le développeur qui assemble ces composants doit aussi écrire leur politique de coordination.

Le canevas *Aceel* fournit également la classe générique `Monitor` qui est utilisée pour développer les moniteurs des ressources (cf. section 7.2). Les sous-sections suivantes présentent en détails le méta-objet, l'état et les implantations d'un composant adaptable.

[3] http ://www.reportlab.com/pyRXP

7.1.2.1 Implantation du méta-objet `Adapter`

Le méta-objet `Adapter` et le code du niveau de base s'exécutent dans deux threads parallèles. Dès la création du composant, le constructeur de `Adapter` enregistre le composant auprès de l'intergiciel de coordination en lui passant sa politique d'adaptation. Quand cet enregistrement est confirmé, la méthode `Adapter.thread_main_loop` prend la main et se met à l'écoute des requêtes émanant de l'intergiciel de coordination. Si elle reçoit des requêtes d'introspection, commence à surveiller périodiquement l'état du composant et notifie ses variations pertinentes à l'intergiciel sous-jacent (p. ex. une variable de `Context.state`, le nombre d'appels à une méthode de l'interface, etc.). Par contre, quand elle reçoit un ordre d'intercession, `thread_main_loop` adapte le composant ou annule sa dernière adaptation, en appelant l'une des méthodes suivantes de `Adapter` :

- `replaceCurrentImpl`. Elle implante les adaptations algorithmiques en remplaçant l'implantation courante par autre. Eventuellement, elle appelle la fonction de transfert d'état fournie par l'implantation remplaçante (cf. section 6.2.2.2).
- `load/unloadImpl`. Elles sont utilisées par `replaceCurrentImpl` pour charger en mémoire ou décharger une implantation donnée.
- `reifyMethod`. Elle met en œuvre une interception des appels et retours d'appels clients vers le méta-niveau afin de réaliser des prétraitements et/ou des post-traitements spécifiques (p. ex. compression des données, filtrage, encryptage, etc.). Ces traitements sont encapsulés dans des classes définies par le développeur à partir de la classe `Interceptor`.
- `add/delComponent`. Elle rajoute (respectivement, supprime) un nouveau composant à un composant composite et met à jour le graphe de dépendances.
- `add/delConnector`. Elle rajoute (respectivement, supprime) un nouveau connecteur entre les composants interagissants et met à jour le graphe de dépendances.

Le méta-objet `Adapter` communique avec le *coordinateur* en utilisant un protocole basé sur TCP. Dans ce protocole, les requêtes d'introspection ont un format identique à celui des requêtes utilisées entre le *coordinateur* et le service de surveillance. Ces dernières sont décrites dans la section 7.2.

7.1.2.2 Etat et implantations alternatives du composant

La tâche du développeur d'un composant *Aceel* varie en fonction du type des implantations de ce composant. Si celles-ci utilisent chacune un état interne, le développeur doit fournir une méthode de transfert de cet état d'une implantation à une autre (cf. section 6.2.2.2). Dans un tel cas, il doit aussi fournir dans chaque implantation des méthodes d'accès à son état interne (`get/set...`) pour permettre à `Adapter` de réaliser les adaptations par paramétrage.

Dans le cas où l'état du composant est séparé de ses implantations, les méthodes d'accès aux attributs de cet état doivent être définies directement dans la classe `State`.

Le développeur du composant doit indiquer si ses implantations peuvent être interrompues par `Adapter` et si un appel interrompu est abandonné ou repris après l'adaptation. Pour cela, il positionne le champ `impls_type` dans l'entête de la politique d'adaptation (cf. figure 6.3) à l'une des valeurs `atomic`, `restartable` ou `cancelable`. Ces valeurs

configurent le comportement du méta-objet `Adapter` de trois façons différentes.

Dans le premier cas, `Adapter` attend la fin du traitement applicatif en cours avant d'exécuter l'adaptation demandée. Cette attente est réalisée avec des mécanismes de synchronisation (des sémaphores) entre les méthodes de `Context` dans le niveau de base et la méthode `thread_main_loop` dans le méta-niveau. Elle présente l'avantage de préserver facilement l'intégrité de l'état du composant.

Par exemple, les implantations qui utilisent chacune un état interne combiné avec une méthode de transfert doivent être atomiques. Toutefois, ce choix n'est convenable que si l'exécution des méthodes des implantations est rapide (c.-à-d. elle prend fin avant que la variation de l'environnement à l'origine de l'adaptation ne devienne caduque) et que si cette variation n'entrave pas cette exécution. Un exemple de variation qui peut empêcher la méthode courante de se poursuivre est la perte d'une connexion réseau lors d'une transmission de données.

Dans le second cas, `thread_main_loop` arrête le traitement courant pour le relancer ultérieurement. Pour cela, lors de la redirection de chaque appel client à l'implantation courante, `thread_main_loop` mémorise le nom et les arguments effectifs de la méthode appelée.

Si le composant possède un état séparé, `thread_main_loop` sauvegarde aussi une copie de cet état en vue de le restaurer avant la relance du traitement. Cette sauvegarde est implantée suivant le patron de conception *memento* où le rôle de l'objet `Caretaker` est joué par le méta-objet `Adapter` (cf. section 2.4.2). Elle n'est effectuée que si la méthode appelée utilise l'un des accesseurs qui modifient l'état séparé du composant (`State.set...`).

Quand `Adapter` remplace l'implantation courante, la méthode interrompue qui sera relancée est celle de l'implantation remplaçante. Dans ce cas, l'adaptation est assimilable à une procédure de reprise après panne qui est transparente aux clients du composant. La « panne » en question est la variation de l'environnement qui a déclenchée l'adaptation.

La sauvegarde de l'état séparé du composant peut entraîner un surcoût important, en particulier si cet état est grand. À cause de cet inconvénient le recours à l'arrêt-reprise du traitement courant doit être limité aux seuls cas qui le justifient. De plus, cette solution n'est envisageable que si le temps de l'adaptation est court par rapport aux contraintes de temps de réponses imposées au composant.

Le troisième cas est similaire au cas précédent sauf que le traitement courant est abandonné au lieu d'être relancé après l'adaptation. Dans ce cas, seul l'état du composant est mémorisé avant l'appel client afin de le restaurer s'il a été modifié. Après l'adaptation, le méta-objet informe le client du composant qu'une erreur s'est produit lors de l'exécution de son appel.

7.2 Mise en œuvre du service de surveillance

En utilisant Python, nous avons implanté le service de surveillance de l'environnement sous la forme d'un processus démon multi-threads qui s'exécute sur chaque site accueillant des composants *Aceel*. Le thread principal de ce processus joue le rôle du

moniteur maître. Il contrôle plusieurs autres threads dont chacun représente le moniteur d'une ressource système particulière.

7.2.1 Moniteurs de ressources

Nous avons implanté les moniteurs de ressources en se basant sur les patrons de conception *singleton* et *observer* [Gamma95]. Le patron *singleton* assure qu'un seul moniteur est créé pour chaque type de ressource et que ce moniteur a un point d'accès unique et global via le moniteur maître. Chaque moniteur de ressource est dérivé de la classe générique `Monitor` fournie par le canevas *Aceel* (cf. figure 7.1). Cette classe joue le rôle du sujet dans le patron *observer* tandis que les *coordinateurs* sur les sites du système distribué jouent le rôle des observateurs (cf. section 2.4.2).

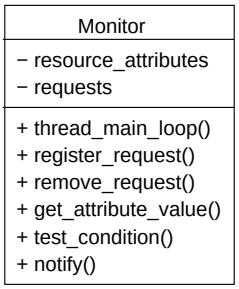


FIG. 7.1 – Classe générique pour les moniteurs de ressources.

La classe `Monitor` maintient une liste de toutes les requêtes de surveillance émises par les *coordinateurs*. Pour chaque requête, la méthode `thread_main_loop` réalise périodiquement les tests demandés et notifie le(s) *coordinateur(s)* intéressé(s) quand les conditions des testées changent.

Pour rajouter un moniteur d'une ressource spécifique, il suffit de spécialiser la classe `Monitor` en définissant sa méthode abstraite `get_attribute_value`. Cette méthode renvoie les valeurs des attributs spécifiques à chaque ressource. Elle utilise pour cela les fonctions du système d'exploitation qui permettent d'inspecter la ressource en question.Notre implantation prototype du service de surveillance fournie les moniteurs des ressources du tableau 7.1.

Ressource	attributs
CPU	vitesse et charge durant 1, 5 ou 15 minute(s) passée(s)
mémoire vive	taille totale et espace libre
mémoire secondaire	taille totale et espace libre pour chaque point de montage
réseau	latence et bande passante entre deux sites
système	toutes variables d'environnement

TAB. 7.1 – Moniteurs de ressources

7.2.2 Protocole ADMP

La communication entre le(s) *coordinateur(s)* et le service de surveillance est régie par le protocole ADMP[4] implanté au-dessus de TCP. Les requêtes émises par un *coordinateur* prennent les formes des messages ADMP suivants :

1. `GetValue Resource[(Index)?] (Attribut)+` :
   Il permet la consultation d'un ou de plusieurs attributs d'une ressource. Le champ optionnel `Index` distingue une ressource particulière parmi plusieurs appartenant au même type (p. ex. nom d'une interface réseau s'il y en a plusieurs).
2. `Monitor Resource[(Index)?] Attribut Operator Threshold (PollingPeriod)?` :
   Il permet de surveiller les variations d'un attribut d'une ressource. Le moniteur de cette ressource enregistre le test exprimé par `Attribut Operator Threshold`. Le champ `Operator` est un opérateur de comparaison et le champ `Threshold` spécifie un seuil. Le champ optionnel `PollingPeriod` précise la période de temps qui sépare deux consultations successives de l'attribut pour le comparer avec le seuil et notifier le *coordinateur* si le résultat de la comparaison est vrai. Si ce champ est absent une valeur par défaut est utilisée selon le type de la ressource.
3. `Unregister (Resource[(Index)?])? (Attribut)*` :
   Il permet d'annuler les requêtes de surveillance enregistrées auparavant pour une ou plusieurs ressources.

Les réponses renvoyées par les moniteurs vers le(s) *coordinateur(s)* sont des messages du type :

1. `ValueOf Resource[Index?] (Attribut = Val)+` :
   Il répond aux requêtes du type `GetValue` en donnant les valeurs des attributs demandés.
2. `Event Resource[Index?] Condition ... is True/False` :
   Il notifie le changement de la valeur de vérité d'une condition correspondant à une requête de surveillance `Monitor`.

Le protocole ADMP constitue une interface rapide pour surveiller l'état de l'environnement qui peut être employé indépendamment des applications adaptables. En particulier, l'utilisateur peut y accéder simplement en établissant une connexion `telnet` avec le moniteur maître. Un tel accès est souvent utile lors de la mise au point des composants adaptables.

7.3 Mise en œuvre de l'intergiciel de coordination

Nous présentons dans cette section l'implantation des trois composants de l'intergiciel de coordination des adaptations, en commençant par *l'optimiseur*. Ensuite, nous expliquons les traitements spécifiques greffés sur cette implantation de base pour traiter les situations particulières, telles que la défaillance d'un composant ou d'un lien réseau

[4] Aceel Distributed Monitoring Protocol.

au milieu d'une coordination et le lancement de coordinations simultanées sur plusieurs sites.

7.3.1 Mise en œuvre de *l'optimiseur*

L'optimiseur s'occupe de la prise des décisions d'adaptation en utilisant une méthode d'optimisation combinatoire. Au lieu d'opter de façon définitive pour une métaheuristique, nous avons choisi de garder la possibilité d'utiliser un algorithme exact afin d'avoir une solution de qualité quand les conditions de l'environnement le permettent. Ainsi, nous avons mis en œuvre *l'optimiseur* comme un composant adaptable en utilisant notre propre modèle *Aceel*.

Le composant *optimiseur* possède deux implantations alternatives, atomiques et sans état, appelées `enumerative-search` et `simulated-annealing` (cf. figure 7.2). A ce stade de notre travail, seule la première implantation a été fournie. Comme notre objectif était de développer et tester un prototype de la plate-forme *Aceel* avec quelques applications adaptables, le nombre de composants traités et, par conséquence, la taille du problème de coordination qu'il fallait résoudre sont restés petits. Dans ces conditions, il était possible d'employer un algorithme exact pour optimiser les modes de fonctionnement des composants adaptables sans risquer une explosion combinatoire.

La politique d'adaptation de *l'optimiseur* comporte deux règles qui contrôle le remplacement dynamique de son implantation (cf. annexe B). Si la taille du problème est relativement faible et si la puissance de calcul du site d'accueil est bonne, alors l'implantation énumérative est utilisée pour trouver les modes de fonctionnement optimaux. Dans les autres cas, le recuit simulé est employé pour éviter que la prise de décision ne prenne un temps exorbitant et que l'adaptation ne perde son caractère réactif.

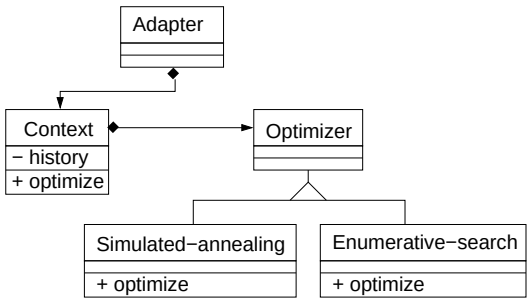


FIG. 7.2 – Architecture de *l'optimiseur*.

L'implantation courante de *l'optimiseur* est appelée par le *coordinateur* qui lui passe toute les données du problème d'optimisation (p. ex. le domaine de définition des variables de décision, les contraintes, les paramètres nécessaires pour l'évaluation de la fonction objectif, etc.). Cette implantation commence par consulter le service de surveillance pour connaître les niveaux des ressources partagées entre les composants concurrents (c.-à-d. connaître $dispo(r), \forall r \in P$). Ensuite, elle calcule le nouveau vecteur optimal, le gain en qualité de service et le coût des adaptations à réaliser.

Soit X' le nouveau vecteur optimal et Q' la qualité de service correspondante. Le gain en qualité de service est égale à $G = Q' - Q$ où Q est la qualité de service actuelle. Le coût C des adaptations est obtenu en cumulant les valeurs des champs `cost` des règles d'adaptation indiquées par X' (cf. section 6.2.1.4).

Dans le cas où X' n'existe pas (c.-à-d. aucun vecteur de D ne satisfait les nouvelles contraintes de l'environnement), *l'optimiseur* informe l'utilisateur qu'il est impossible de coordonner les adaptations des applications dans les conditions d'exécution actuelles. Dans ce cas, la liste des applications impliquées et la liste ressources qu'elles partagent sont précisées.

Si le vecteur X' existe, alors avant de le transmettre au *coordinateur*, *l'optimiseur* vérifie que le gain procuré par les adaptations à réaliser n'est pas négligeable par rapport à leur coût (c.-à-d. $G/C \gg 0$). Si ce n'est pas le cas, *l'optimiseur* décide alors d'abandonner ces adaptations et indique au *coordinateur* de ne rien faire.

Le fait d'informer l'utilisateur d'un éventuel échec de la coordination permet de l'introduire dans la boucle de prise de décision. De cette façon il peut choisir d'arrêter l'une des applications concurrentes, afin de libérer quelques ressources. Cette libération va engendrer des variations qui seront détectées et notifiées par le service de surveillance. Ainsi, une nouvelle procédure de coordination sera lancée avec plus de chance d'aboutir à une optimisation de la qualité de service.

7.3.2 Mise en œuvre du *coordinateur*

Afin de supporter les différentes relations hiérarchiques entre les sites distribués, nous avons conçu le *coordinateur* comme un composant *Aceel* qui possède trois implantations alternatives. Celles-ci sont dédiées respectivement aux sites maîtres, aux sites esclaves et aux sites pairs (cf. figure 7.3). Elles sont de type atomique et donc ne peuvent être interrompues en cours de traitement par les nouveaux événements de notification qui arrivent. Cela permet de préserver l'intégrité du *coordinateur* et par conséquence l'intégrité des composants adaptables qu'il contrôle. Les nouveaux événements reçus sont stockés et traités après la fin de la coordination en cours.

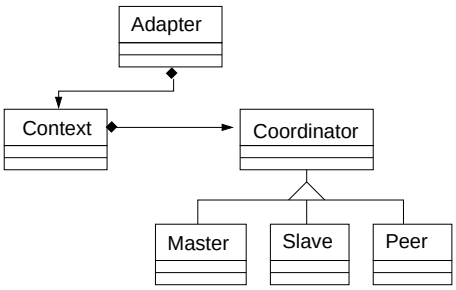


FIG. 7.3 – Architecture du *coordinateur*.

Le *coordinateur* possède un état séparé dans lequel il maintient :
– la liste des applications adaptables déployées sur le site,

- la liste des composants *Aceel* en cours d'exécution,
- la liste des ressources partagées entre les composants concurrents et référencées dans leurs politiques d'adaptation,
- le temps maximum d'attente des réponses aux requêtes de coordination,
- et des représentations internes des politiques d'adaptation et des politiques de coordination des composants.

Les représentations internes des différentes politiques englobent entre autres, les variables utilisées pour modéliser le problème de la coordination, telles que le vecteur X, la qualité de service actuelle Q, le domaine D, les classes de priorités des composants c_i, les poids des dimensions d'adaptation p_{ij}, etc. (cf. section 6.4.4.1).

Le choix de l'implantation du *coordinateur* à utiliser sur chaque site est réalisé de façon statique par l'administrateur avant de démarrer l'intergiciel de coordination. Etant donné que le type du site d'accueil ne change pas en cours d'exécution, cette implantation ne sera pas remplacée dynamiquement par l'`Adapter` du coordinateur. Comme illustré dans l'annexe B, la politique d'adaptation du *coordinateur* définie une seule dimension. Celle-ci contient les règles d'adaptation qui contrôlent le temps d'attente des réponses au requêtes de coordination (cf. section 7.3.4.1). Contrairement aux composants *Aceel* applicatifs, quand le *coordinateur* est créé son implantation est chargée avant l'interprétation de sa politique d'adaptation car c'est elle même qui réalise cette interprétation. Egalement, pour adapter dynamiquement le temps maximum d'attente, l'implantation du *coordinateur* exécute elle-même les actions nécessaires sans passer par un quelconque *synchroniseur*, ni par son méta-objet.

Une fois lancé, le *coordinateur* créé le composant *optimiseur*, l'enregistre comme un composant adaptable et parcourt sa politique d'adaptation. Lors de ce parcourt, le *coordinateur* demande au service de surveillance de lui notifier les variations de la mémoire vive libre et de la vitesse et la charge de la CPU. Il envoie aussi à son propre méta-objet `Adapter` une demande d'introspection du cardinal de l'ensemble D. Cette introspection est d'ailleurs la seule tâche que réalise ce méta-objet.

Quand le cardinal de D varie de façon pertinente suite à la création et/ou la destruction de composants *Aceel*, `Adapter` notifie l'implantation courante du *coordinateur*. Celle-ci envoie à son tour un ordre à l'`Adapter` du composant *optimiseur* pour qu'il remplace son algorithme. Notons que l'intergiciel de coordination n'a pas besoin d'une politique de coordination propre, bien qu'il soit construit avec deux composants *Aceel* interagissants. En effet, seul *l'optimiseur* peut faire une adaptation dynamique, qui plus est, contrôlée directement par le *coordinateur*.

Ci-après, nous décrivons l'implantation des *synchroniseurs* qui contrôlent l'exécution des adaptations des composants applicatifs.

7.3.3 Mise en œuvre du *synchroniseur*

Une application *Aceel* dispose sur chacun de ses sites d'accueil d'un *synchroniseur* qui gère ses composants locaux. Ce *synchroniseur* utilise la politique de coordination de l'application pour exécuter les adaptations des composants en parallèle et/ou en séquence. Si cette politique définie une synchronisation entre des composants distants, les *syn-*

chroniseurs associés à ces composant communiquent entre eux pour fixer les points de rendez-vous nécessaires afin de réaliser cette synchronisation.

Dans le cas des actions d'adaptation parallèles, chaque *synchroniseur* envoie aux composants locaux l'ordre d'exécuter les actions spécifiées et attend leurs réponses. Chaque composant suit alors le processus d'adaptation décrit dans la section 6.2.2, puis notifie le succès ou l'échec de son action au *synchroniseur*. Dès la réception des réponses des composants locaux, chaque *synchroniseur* échange avec ses semblables distants le résultat des actions d'adaptation effectuées. Si toutes les actions sont réussies, chaque *synchroniseur* envoie aux composants qu'il gère l'ordre de valider leurs adaptations. Sinon, si au moins une adaptation a échoué, il envoie aux composants ayant réussis leurs adaptations l'ordre d'annuler celles-ci.

Concernant les actions d'adaptation séquentielles, chaque composant ne reçoit l'ordre de s'adapter que si son prédécesseur a réussi son adaptation et a reçu l'ordre de la valider. Si un composant échoue, le *synchroniseur* envoie à tous ses prédécesseurs l'ordre d'annuler leurs adaptations et abandonne le reste des actions envisagées. A la fin de la synchronisation, le *coordinateur* de chaque site se met de nouveau à l'écoute des événements notifiant les variations de l'environnement.

A titre d'exemple, dans une application multicast, les adaptations des clients peuvent être lancées en parallèle avant celle du serveur. Si elles réussissent toutes, le serveur pourra alors être adapté à sont tour. Sinon, les clients qui ont réussis leurs adaptations recevront l'ordre de les annuler. Dans ce cas rien n'est fait au niveau du serveur. Si par contre c'est l'adaptation de ce dernier qui échoue alors tout le monde recevra l'ordre d'annulation.

7.3.4 Traitement des cas particuliers

Dans les sous-sections suivantes nous présentons le traitement des situations particulières qui sont dues aux problèmes classiques de tolérance aux fautes et d'établissement d'un ordre total entre les événements dans un système distribué asynchrone.

7.3.4.1 Gestion des défaillances lors d'une coordination

La tolérance aux fautes est un aspect important dans la conception des systèmes distribués. Une défaillance causée par un problème matériel ou logiciel peut affecter la coordination des adaptations multiples. En cas de pannes, le protocole de coordination se bloque et entraîne la défaillance des applications en cours d'adaptation. Dans *Aceel*, ce problème est pallié par la limitation du temps d'attente des réponses aux requêtes de coordination transmises via le réseau. Chaque réponse qui ne parvient pas à destination avant l'échéance fixée est interprétée comme une réponse négative par celui qui l'attendait (c.-à-d. le *coordinateur* ou le *synchroniseur*). Ce choix pessimiste garantit la cohérence des composants adaptables par un retour systématique à l'état initial en cas de défaillance. Dans ce sens, la coordination des adaptations peut être considérée comme un service « best effort » offert aux applications distribuées adaptables.

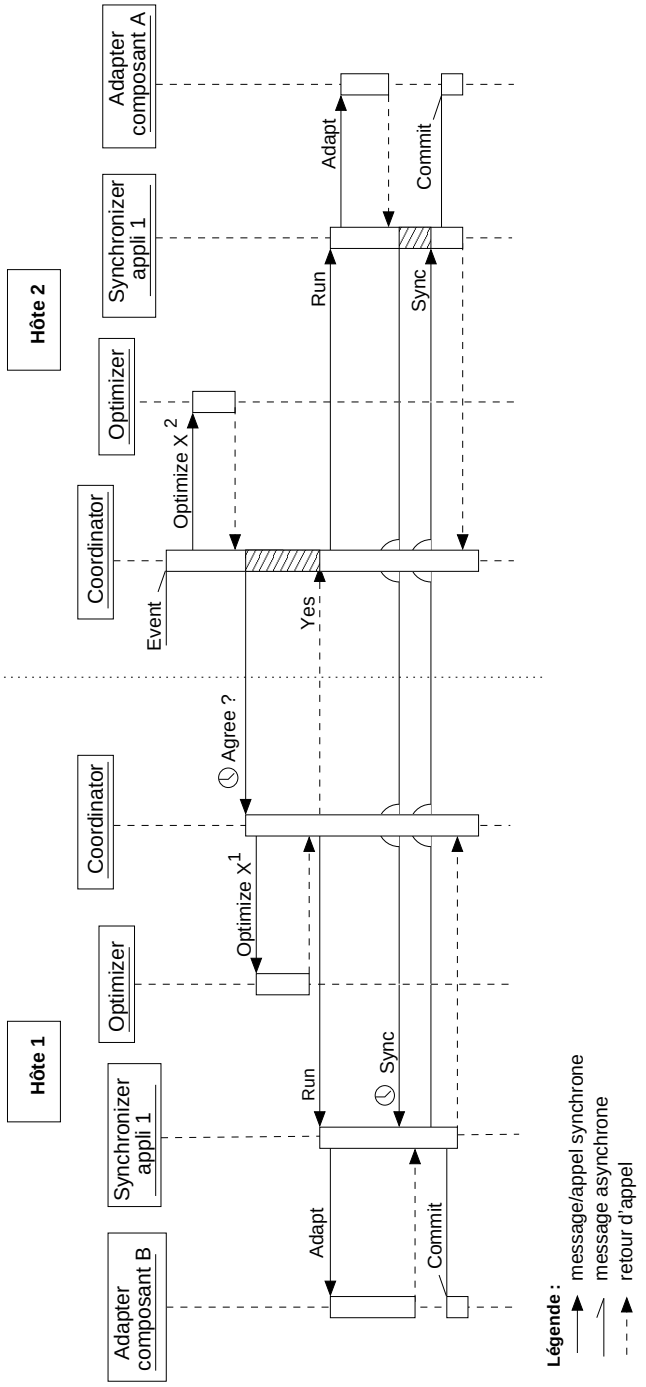


FIG. 7.4 – Coordination des adaptations sur des sites pairs.

La figure 7.4 montre un exemple de coordination des adaptations de deux composants déployés sur deux sites pairs. Les attentes qui ont lieu lors de cette coordination sont indiquées par les zones hachées dans le diagramme de séquence. Le *coordinateur* et le *synchroniseur* de chaque site attendent chacun les réponses de leurs pairs distants afin de, respectivement, se mettre d'accord sur les actions à exécuter et confirmer le bon déroulement de cette exécution. Dans les deux cas, la période maximum d'attente est définie dans la politique d'adaptation du *coordinateur* comme une variable qui peut être adaptée en fonction de la qualité des liens réseau entre les sites distants.

7.3.4.2 Gestion des coordinations simultanées

Etant donnée que les notifications des variations de l'environnement sont asynchrones, il est possible que les *coordinateurs* de plusieurs sites commencent simultanément une coordination des adaptations des composants qu'ils accueillent. Dans le cas où les composants sur ces sites interagissent les uns avec les autres, il y a un risque d'interblocage entre les *coordinateurs* quand chacun enverra aux autres une demande de consensus sur les variables de décisions soumises aux contraintes distribuées et attendra leurs réponses.

Pour empêcher l'interblocage, il faut établir un ordre logique total entre les demandes de consensus émises par les *coordinateurs* d'un système distribué. Ceci peut être réalisé par l'utilisation des horloges logiques de Lamport [Lamport78], par exemple. Ainsi, seul le *coordinateur* qui émet la première demande de consensus pourra poursuivre son traitement. Les autres *coordinateurs* abandonnent les coordinations qu'ils ont entamées, pour participer aux consensus initié par le premier.

7.4 Expérimentation d'*Aceel* en environnement mobile

Dans le cadre de notre travail sur l'adaptabilité des composants logiciels en environnement dynamiques, nous avons participé au projet RNRT Cyberté [André03a]. Ce projet vise à développer un terminal mobile muni de plusieurs interfaces réseaux de types WiFi et Ethernet. L'idée générale de Cyberté est d'exploiter ces interfaces de façon interchangeable, voir en parallèle, afin de garantir une connectivité maximale. Pour démontrer la faisabilité et l'utilité de cette idée, nous avons eu pour tâche le développement d'une application de vidéo à la demande qui s'adapte aux variations de l'environnement d'exécution sur ce terminal. Nous avons aussi développé un navigateur Web adaptable et nous l'avons déployé dans le même environnement pour tester la coordination des adaptations des applications concurrentes. Nous présentons ci-après ces deux exemples en commençant par le plus simple.

7.4.1 Navigateur Web adaptable

En utilisant *Aceel*, nous avons conçu un navigateur Web qui supporte la désactivation dynamique du téléchargement des images en fonction de la bande passante disponible. Ce navigateur est constitué d'un composant adaptable que nous avons développé en

moins d'une journée. Notre composant possède une seule implantation avec un état interne qui contient un booléen pour indiquer s'il faut ou non télécharger les images de chaque page Web consultée. L'implantation de notre composant est une simple enveloppe Python autour d'un code légataire ouvert écrit en C. Ce code est celui de *Dillo*[5], un navigateur Web léger fréquemment utilisé dans les équipements mobiles.

L'implantation de notre composant offre une méthode d'accès qui modifie son état et appelle une fonction spécifique que nous avons rajouté à *Dillo* pour activer ou désactiver dynamiquement le téléchargement des images. Cette méthode d'accès est utilisée par le méta-objet pour adapter le navigateur sur ordre du coordinateur du site d'accueil. La politique d'adaptation du navigateur est listée dans l'annexe C. Elle comporte une seule dimension d'adaptation avec des règles qui déterminent quand activer ou désactiver le téléchargement des images en fonction de la bande passante entre le navigateur et le proxy Web distant.

7.4.2 Application de vidéo à la demande adaptable

Notre deuxième exemple d'application basée sur *Aceel* est plus élaboré que le premier. Il s'agit d'une application de vidéo à la demande qui possède une architecture client-serveur [André03b]. Elle est constituée de l'*émetteur* et du *récepteur* qui sont deux composants *Aceel* déployés sur des sites pairs (cf. figure 7.4.2).

L'émetteur possède une seule implantation interruptible basée sur le code ouvert du projet *Darwing Streaming Server*[6]. Cette implantation écrite en C++ est interfacée avec l'objet `Context` par le moyen de `l'API Python/C` intégré à Python. Chacune des vidéos délivrées par l'émetteur est pré-encodée dans deux formats différents (`MPEG4` et `MJPEG`), avec deux taux de trames différents (`15fps` et `30fps`) et trois débits différents (`86kbps`, `170kbps` et `340kpbs`). Chaque combinaison de ces trois paramètres adaptables correspond à un fichier vidéo distinct. Ainsi, pour adapter l'un de ces paramètres, il suffit de choisir le bon fichier comme source du flux transmis.

L'émetteur maintient dans son état séparé le nom de la vidéo à envoyer vers le client et un pointeur temporel indiquant l'avancement dans la lecture de celle-ci. Sa politique d'adaptation définie trois dimensions d'adaptation (cf. annexe C). Les deux premières dimensions comportent les règles qui déterminent le taux de trames et le débit de la vidéo à transmettre en fonction de la puissance de calcul du site client et de la bande passante du lien réseau, respectivement. La variation de la puissance de calcul du client (qui dépend de la taille de sa mémoire libre et de la charge de son processeur) est notifiée par son propre service de surveillance vers le coordinateur du serveur distant. L'une des variations pertinentes de la bande passante est détectée par le moniteur réseau quand le terminal mobile bascule entre les interfaces WiFi et Ethernet.

La troisième dimension d'adaptation indique qu'il est possible d'utiliser le format `MPEG4` ou le format `MJPEG` de façon interchangeable, indépendamment des variations de l'environnement.

[5] http ://www.dillo.org
[6] http ://developer.apple.com/darwin/projects/streaming/

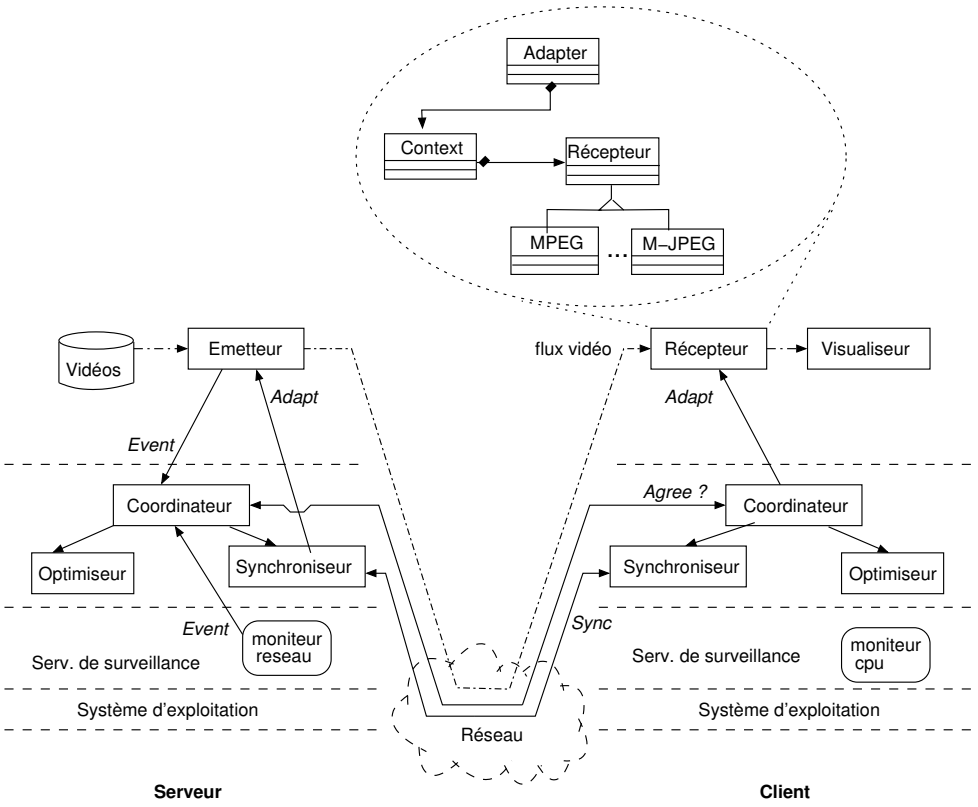


FIG. 7.5 – Application de vidéo à la demande basée sur *Aceel*.

Le récepteur possède deux implantations interruptibles qui supportent le décodage des formats `MPEG4` et `MJPEG`, respectivement. Ces implantations sont basées sur le code C ouvert du projet *MPlayer*[7]. Elles sont chargées dès la création du récepteur afin d'accélérer le basculement de l'une vers l'autre ultérieurement. Le récepteur possède aussi un pointeur temporel indiquant l'avancement dans la lecture du flux vidéo reçu.

La politique d'adaptation du récepteur définie une seule dimension correspondant au format de décodage utilisé (cf. annexe C). Elle contient les règles qui déterminent l'implantation à employer en fonction du taux de bits erronés dans le flux vidéo reçu. Quand ce taux est élevé, il convient d'utiliser un format de codage intra-trames (c.-à-d. `MJPEG`) qui est moins sensible aux erreurs que les formats inter-trames. Dans ces derniers, le codage d'une trame vidéo dépend des trames qui l'entourent, chose qui favorise la propagation des erreurs.

[7] http ://www.mplayerhq.hu

7.4.3 Coordination des composants adaptables interagissants

Pour coordonner les adaptations de l'émetteur et du récepteur vidéo, nous avons écrit la politique illustrée dans l'annexe C. Cette politique préserve la cohérence de l'application vidéo à travers une règle de coordination qui est appliquée à chaque fois que le récepteur adapte son implantation. Cette règle impose à l'émetteur de changer aussi le format de la vidéo transmise en même temps. La décision d'exécuter cette action d'adaptation est prise par le *coordinateur* du site serveur en concertation avec son pair sur le site client.

L'exécution en parallèle des deux adaptations qui changent le format d'encodage vidéo est contrôlée par les *synchroniseurs* associés à l'application sur les deux sites. Après ces deux adaptations, toujours sur ordre du *coordinateur*, l'émetteur règle son pointeur temporel sur celui du récepteur. Cette troisième adaptation est du type paramétrage d'une variable d'état. Elle permet la retransmission des trames qui ont été éventuellement perdues à cause de l'interruption du récepteur par l'adaptation qui remplace son implantation.

7.4.4 Coordination des applications adaptables concurrentes

Pour tester la coordination des composants adaptables concurrents, nous avons déployé les deux applications précédentes dans l'environnement mobile illustré par la figure 7.4.4. Ces deux applications partagent la bande passante réseau entre le terminal mobile et la station fixe.

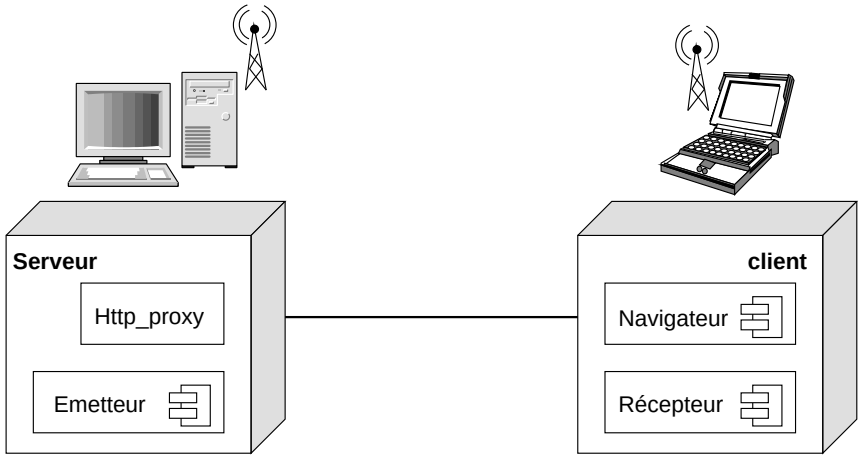


FIG. 7.6 – Applications *Aceel* en environnement mobile.

En exécutant les deux applications en même temps, nous constatons que l'intergiciel coordonne les adaptations du navigateur Web et de l'émetteur vidéo suivant différents scénarios, en fonction de leurs classes de priorité et en fonction des variations de la bande passante.

Par exemple, si la bande passante est inférieure à `128kbps`, l'application vidéo ne fonctionne pas quelque soit sa priorité. Seul le navigateur Web peut être utilisé.

Si la priorité du navigateur Web est supérieure à celles des composants vidéo et si la bande passante est entre `128kbps` et `184kbps`, l'intergiciel de coordination informe l'utilisateur qu'il n'est pas possible d'adapter les deux applications à moins d'arrêter l'une d'elles. Au delà de `184kbps`, le navigateur peut toujours télécharger les images du Web. L'application vidéo utilise alors le reste de la bande passante disponible.

Si l'émetteur vidéo est prioritaire, l'intergiciel ordonne au navigateur Web d'activer le téléchargement des images seulement si la bande passante est dans les intervalles `[184kbps, 255kbps]` et `[312kbps, 511kbps]` ou si elle est supérieure à `568kbps`.

Les priorités des composants peuvent être établies dynamiquement en fonction du focus courant de l'utilisateur. Pour cela, il suffit (1) de les définir comme des paramètres adaptables du *coordinateur*, (2) de rajouter à ce dernier une règle d'adaptation qui affecte la plus grande priorité aux composants de l'application qui a le focus de l'utilisateur et (3) de rajouter un moniteur de l'IHM qui identifie cette application.

7.5 Conclusion

Nous avons présenté dans ce chapitre une mise œuvre à vocation générique de notre plate-forme *Aceel*. Le choix d'un langage de prototypage rapide pour développer les différentes briques de notre plate-forme et le recours à un langage déclaratif et extensible pour exprimer les politiques qui contrôlent son fonctionnement ont été décisif pour valider nos idées. La mise en œuvre de la plate-forme *Aceel* est elle-même adaptable et réflexive car nous avons utilisé notre propre modèle de composants pour implanter l'intergiciel de coordination des adaptations.

Nos exemples d'expérimentation, choisis pour leurs simplicités, ont permis de montrer l'intérêt d'*Aceel* pour accélérer et faciliter le développement des applications adaptables. En particulier, nous avons montré l'utilité d'*Aceel* quand il s'agit de rendre un code légataire adaptable rapidement, de coordonner les adaptations des composants interagissants et de coordonner les adaptations des applications concurrentes.

Notre approche de coordination des adaptations multiples n'est pas restreinte au seul cas des applications web ou multimédia. Au contraire, elle peut être appliquée à toute application distribuée qui peut offrir différentes qualités de service en fonction des ressources disponibles.

Chaptire 8

Conclusion

8.1 Rappel des objectifs

Notre étude de l'état de l'art sur l'adaptabilité du logiciel a montré que (1) peu de travaux ont adressé ce problème en développant des supports génériques pour l'adaptation et que (2) rares sont ceux qui se sont attaqués au problème plus complexe de la coordination des adaptations multiples. A notre connaissance aucun système n'a considéré l'ensemble des aspects liés à la coordination des adaptations multiples, à savoir la gestion des dépendances d'interaction et des dépendances de partage des ressources entre les entités qui s'adaptent.

Après avoir identifier ce manque, nous avons eu pour objectif le développement de mécanismes génériques qui supportent les adaptations multiples des applications distribuées en environnement dynamique. Ces mécanismes doivent accélérer et faciliter la tâche des développeurs de telles applications. Les développeurs peuvent se concentrer sur leur code métier au lieu de se préoccuper des aspects techniques d'implantation de l'adaptation. Pour valider nos idées, nous nous sommes aussi fixés pour but l'expérimentation des mécanismes proposés sur des applications couramment utilisées dans un environnement dynamique réel.

8.2 Démarche suivie

Vu la complexité des objectifs fixés, nous avons procédé par séparation des préoccupations qui entrent dans l'adaptation dynamique aux variations de l'environnement. Pour cela, il a fallu choisir une approche logicielle modulaire afin d'isoler et traiter chaque préoccupation à part. Ainsi, parmi les différentes approches logicielles étudiées dans l'état de l'art, nous avons choisi l'approche orientée composants.

D'abord, nous avons commencé par traiter le problème de l'adaptabilité au niveau de la brique de base de cette approche qui est le composant logiciel. Puis, nous nous sommes intéressés aux adaptations des composants liés par des dépendances d'interaction. Ensuite, nous avons traité le cas des adaptations des composants liés par des dépendances de partage des ressources.

Cette démarche simplifie le problème complexe de l'adaptation à travers une décomposition naturelle qui permet d'isoler les questions propres à chaque niveau d'abstraction (composant, application et système distribué). Cette décomposition facilite la tâche des développeurs car elle leur permet de traiter l'adaptation parallèlement aux phases de développement dans l'approche orientée composants (décomposition, assemblage et déploiement).

Afin de valider notre proposition nous avons réalisé une plate-forme de composants constituée d'un canevas générique pour le développement de composants réactifs et de deux services intergiciels pour permettre les adaptations de ces composants. Ensuite, nous avons testé cette plate-forme en l'utilisant pour développer une application de vidéo à la demande et un navigateur Web adaptables dans un environnement mobile. Nous présentons ci-après un bilan scientifique du travail réalisé.

8.3 Bilan scientifique

Notre travail sur la coordination des adaptations dynamiques multiples a recouvert plusieurs aspects.

1. Identifier de façon précise la nature des problèmes propres à la coordination de plusieurs adaptations, à savoir la gestion des dépendances d'interaction et des dépendances de partage des ressources entre les entités qui s'adaptent, ainsi que la garantie de leur stabilité.
2. Développer des mécanismes génériques pour supporter les adaptations individuelles et les adaptations multiples en utilisant l'approche composants. Pour cela, nous avons opérer une *séparation* nette entre le code métier des applications, le code qui détecte et notifie les variations de l'environnement, le code qui traite les événements de notification, décide les adaptations et évite les conflits, le code qui dicte la politique de l'adaptation, le code qui dicte la politique de coordination des composants interagissants, le code qui exécute chaque action d'adaptation et le code qui synchronise ces exécutions. Pour la plupart, ces codes sont fournis par notre plate-forme et sont directement réutilisables. Les seuls codes fournis par le développeur sont le code métier et les politiques d'adaptation et de coordination.
3. Elaborer, via une remise à plat du modèle *MolèNE*, le nouveau modèle *Aceel* qui supporte l'intercession du code métier du composant et l'introspection de son état. L'intercession peut être réalisée à travers plusieurs types d'adaptations, notamment les adaptations algorithmiques par remplacement de l'implantation courante, les adaptations par paramétrage des variables d'état. Dans le premiers cas, nous avons proposé une solution générique au problème de transfert d'état entre les implantations alternatives du composant. Grâce à la combinaison de la réflexion et du patron de conception *strategy*, il est aussi possible de rajouter dynamiquement, à un composant, une nouvelle implantation non prévue au départ.
4. Définir un langage déclaratif et extensible pour l'expression des politiques d'adaptation des composants. L'adaptation peut être déclenchée dans des conditions

complexes. Ces conditions sont exprimées par la combinaison d'événements simples qui peuvent être générés n'importe où au sein du système distribué. L'une des originalités de notre langage est la possibilité d'exprimer les coûts estimés des actions d'adaptation. Ces coûts sont pris en compte lors de la prise de décision pour choisir les actions qui ont le meilleur rapport qualité/prix.

5. Impliquer l'utilisateur dans le processus d'adaptation et tenir compte de ses préférences. Grâce à la notion de *dimensions d'adaptation*, l'utilisateur peut favoriser l'adaptation de certains aspects parmi tous ceux qui sont modifiables au sein d'un composant. Il peut aussi donner l'avantage à certaines applications par rapport à d'autres en définissant des classes de priorité des composants. L'utilisateur est aussi informé de l'évolution de la coordination afin qu'il puisse intervenir en cas d'échec ou de conflits.
6. Définir un langage déclaratif pour l'expression des politiques de coordination des adaptations des composants interagissants. Ces politiques écrites par le développeur précisent quels composants doivent s'adapter ensemble et comment synchroniser leurs actions.
7. Développer un intergiciel pour la coordination des adaptations des composants applicatifs afin de préserver leur cohérence et d'éviter les conflits sur les ressources qu'ils partagent. Cet intergiciel est lui-même conçu avec notre modèle de composant *Aceel*, ce qui permet d'adapter son comportement en fonction des relations hiérarchiques existants entre les sites d'un système distribué et en fonction du nombre des composants adaptables déployés sur chaque site. Notre intergiciel gère la coordination des adaptations multiples en modélisant celle-ci comme un problème d'optimisation combinatoire, sous les contraintes de l'environnement, de la qualité de service rendue par les composants.
8. Développer un service de surveillance de l'environnement qui est (1) simple à interfacer avec toute couche logicielle désirant recevoir les notifications des variations pertinentes et (2) facile à étendre pour gérer de nouvelles ressources. Dans ce cas, l'emploi de XML rend inutile la modification du langage décrivant les politiques d'adaptation. Par conséquent, il n'est pas nécessaire de réécrire le code qui interprète ces politiques et qui reçoit les événements de notification (c.-à-d. le *coordinateur*).
9. Utiliser les mécanismes de la plate-forme *Aceel* pour développer un navigateur Web et une application de vidéo à la demande adaptables dans un environnement mobile. Ces expériences ont montré qu'il est simple d'interfacer *Aceel* avec des applications légataires et qu'il est donc facile de rendre adaptable les codes existants qui sont sensibles aux variations de l'environnement.

8.4 Limites et perspectives

L'expérimentation de la plate-forme *Aceel* nous a permis d'identifier quelques limites de la solution proposée et a soulevé de nouvelles questions qui peuvent constituer des axes de recherche pour les travaux futurs.

Parmi les travaux possibles à réaliser dans un moyen terme, nous citons :

1. Le développement d'un outil de vérification automatique de la couverture des règles d'adaptation. Par couverture nous entendons l'existence de règles qui considèrent tout l'intervalle de variation de chaque ressource spécifiée dans une politique d'adaptation. Sans cette couverture, il peut exister des configurations de l'environnement pour lesquelles la politique d'adaptation ne spécifie aucune action. Dans ce cas, le composant concerné est laissé tel qu'il est, ce qui peut conduire à une erreur d'exécution.
2. Le développement de profileurs pour (1) mesurer les ressources consommées par les composants et (2) estimer les coûts de leurs actions d'adaptation. De tels outils devraient accélérer et raffiner l'écriture des règles d'adaptation. A titre d'exemple, pour écrire les règles d'adaptation du récepteur vidéo (cf. annexe C) nous avons mis au point un outil capable d'évaluer la consommation mémoire du programme *MPlayer* en employant le profileur mémoire *mpatrol*[1]. Un tel procédé peut être généralisé à d'autres types de ressources en exploitant les profileurs qui deviennent de plus en plus disponible (p. ex. le module Python `profile` qui mesure le temps d'exécution d'une fonction).
3. Le développement de modèles décrivant les variations de certaines ressources dans le temps afin d'anticiper les actions d'adaptation (p. ex. le chargement d'une implantation). Par exemple, la variation de la bande passante d'un lien réseau sans fil peut être prédite, sous certaines conditions en fonction de la vitesse et de la direction de déplacement du mobile par rapport à son point d'accès. Dans la même perspective, il est possible de modéliser le comportement de l'utilisateur à travers une interface homme-machine intelligente. Ceci permettrait aussi d'anticiper les adaptations en prévoyant, avec une certaine probabilité, les prochaines actions de l'utilisateur.

A plus long terme, il convient d'aborder les aspects qui n'ont pas été traités jusqu'à maintenant et qui ont un impact important sur la longivité des applications adaptables, notamment, dans le cadre des systèmes distribués. Parmi ces aspects nous citons :

1. La validation du passage à l'échelle de l'intergiciel de coordination en déployant un nombre important de composants adaptables. Dans un tel cas, on peut par exemple utiliser le recuit simulé comme implantation alternative pour le composant *optimiseur*.
2. La résolution du problème de l'instabilité du système distribué qui peut être causée par l'exécution de plusieurs adaptations en boucle ou en chaîne.
3. L'étude de l'éventualité de prendre les décisions d'adaptation même quand on ne dispose pas de toutes les informations sur l'état global de l'environnement.
4. L'étude de la possibilité de réutiliser le code d'une politique d'adaptation quand le code métier controlé par celle-ci est réutilisé dans de nouveaux composants adaptables. Eventuellement, les dimensions d'adaptation peuvent former l'unité

[1] http ://www.cbmamiga.demon.co.uk/mpatrol/

de base dans cette réutilisation. Dans le même ordre d'idées, les règles qui forment les politiques de coordination peuvent être réutilisées dans les applications où les schémas d'interaction entre les composants se répètent.

Enfin, il est toujours possible d'apporter des améliorations ponctuelles à la mise en œuvre actuelle de la plate-forme *Aceel* telles que :

1. Le rajout d'une procédure d'authentification des entités qui génèrent les événements notifiant les variations de l'environnement. Ceci permettrait d'éviter les problèmes de sécurité dus aux fausses notifications envoyées par des intrus.
2. Le développement d'une interface graphique pour faciliter l'écriture des différentes politiques. Cette interface devrait générer les scripts XML à partir de formulaires ergonomiques remplis par le développeur.

Bibliographie

[Aksit96] Mehmet Aksit, Bedir Tekinerdogan & Lodewijk Bergmans. *Achieving adaptability through separation and composition of concerns.* In M. Muhlhause, editeur, Special Issues in Object-Oriented Programming, pages 12–23. dpunkt verlag, 1996.

[Aksit03] Mehmet Aksit & Zièd Choukair. *Dynamic, Adaptive and Reconfigurable Systems Overview and Prospective Vision.* In 23 rd International Conference on Distributed Computing Systems Workshops (ICDCSW 03). IEEE Computer society, 2003.

[Allen98] Robert J. Allen, Remi Douence, & David Garlan. *Specifying and Analyzing Dynamic Software Architectures.* In Conference on Fundamental Approaches to Software Engineering, Lisbon, Portugal, mars 1998. LNCS 1382 Springer.

[André03a] Françoise André, Jean-Marie Bonnin, Bruno Deniaud, Karine Guillouard, Nicolas Montavont, Thomas Noël & Lucian Suciu. *Optimized Support of Multiple Wireless Interfaces within an IPv6 End-Terminal.* In Smart Object Conference (SOC), Grenoble, France, mai 2003.

[André03b] Françoise André & Bruno Deniaud. *Multimedia systems adaptation in mobile environments.* In IASTED International multi-conference Applied Informatics (AI). Parallel and Distributed Computing Networks (PDCN), Innsbruck, Austria, février 2003.

[André00] Françoise André & Maria-Teresa Segarra. *MolèNE : un système générique pour la construction d'applications mobiles.* Calculateurs Parallèles. Numéro spécial : Evolution des plates-formes orientées objets répartis, vol. 12, no. 1, 2000.

[Appleton00] Brad Appleton. *Patterns and Software : Essential Concepts and Terminology.* URL : http ://www.cmcrossroads.com/bradapp/docs/patterns-intro.html, 2000.

[Azzouz03] Samir Azzouz. Adaptation dynamique pour modèles à composants logiciels. Master's thesis, Université Joseph Fourier, IMAG, 12 septembre 2003.

[Baker99] Mark Baker & Rajkumar Buyya. High performance cluster computing : Architectures and systems, chapitre Cluster Computing at a Glance. Prentice Hall PTR, New Jersey, 1999. Chap. 1.

[Bergmans01] L. Bergmans & M. Aksit. *Composing Crosscutting Concerns Using Composition Filters.* Communications of the ACM, vol. 44, no. 10, pages 51–57, octobre 2001.

[Bershad95] B. N. Bershad, S. Savage, P. Pardyak, E. G. Sirer, M. Fiuczynski, D. Becker, S. Eggers & C. Chambers. *Extensibility, safety and performance in the SPIN operating system.* In 15th ACM Symposium on Operating Systems Principles (SOSP '95), Colorado, USA, décembre 1995.

[Blair01] Gordon S. Blair & Geoff Coulson et al. *The Design and Implementation of Open ORB version 2.* IEEE Distributed Systems Online, Special Issue on Reflective Middleware, vol. 2, no. 6, 2001.

[Blay-For04] M. Blay-Fornarino, D. Emsellem, A-M. Pinna-Dery & M. Riveill. *Un service d'interactions : principes et implémentation.* RSTI - série TSI, vol. 23, no. 2, pages 175–204, 2004.

[Bosch97] Jan Bosch. *Superimposition : A component adaptation technique.* Rapport technique, Department of Computer Science and Business Administration, University of Karlskrona/Ronneby, septembre 1997.

[Braam98] P. J. Braam. *The Coda Distributed File System.* Linux Journal, no. 50, juin 1998.

[Bruneton02] E. Bruneton, T. Coupaye & J.B. Stefani. *Recursive and Dynamic Software Composition with Sharing.* In 7th International Workshop on Component-Oriented Programming (WCOP02), at ECOOP, Malaga, Spain, 10–14 juin 2002.

[Buisson04] Jeremy Buisson, Françoise André & Jean-Louis Pazat. *Adaptation dynamique de codes parallèles.* In Journées composants, Lille, France, mars 2004.

[Campbell99] Andrew T. Campbell, Geoff Coulson & Michael E. Kounavis. *Managing Complexity : Middleware Explained.* IT Professional Magazine, vol. 1, no. 5, pages 22–28, septembre/octobre 1999.

[Capra03] Licia Capra, Wolfgang Emmerich & Cecilia Mascolo. *CARISMA : Context-Aware Reflective mId-dleware System for Mobile Applications.* IEEE Transactions on Software Engineering, novembre 2003.

[Cazzola00] Walter Cazzola, Andrea Savigni, Andrea Sosio & Francesco Tisato. *Explicit Architecture and Architectural Reflection.* In 2nd International Workshop on Engineering Distributed Objects. LNCS, Springer-Verlag, 2000.

[Charfi03] Anis Charfi. Software interaction : Towards dynamic adaptation of .net objects. Master's thesis, Technical University of Munich, july 2003.

[Charfi04] A. Charfi, D. Emsellem & M. Riveill. *Dynamic Component Composition in .NET.* Journal of Object Technology, vol. 3, no. 2, pages 37–46, 2004.

[Charon96] Irène Charon, Anne Germa & Olivier Hudry. Méthodes d'optimisation combinatoire. Masson, Paris, 1996.

[Chefrour03a] Djalel Chefrour & Françoise André. *Auto-adaptation de composants ACEEL coopérants.* In 3ème Conférence Française sur les Systèmes d'Exploitation(CFSE'3), La Colle sur Loup, France, octobre 2003.

[Chefrour03b] Djalel Chefrour & Françoise André. *Développement d'applications en environnements mobiles à l'aide du modèle de composant adaptatif ACEEL.* In Langages et Modèles à Objets LMO'03, Vannes, France, 2003. Publié dans STI, série L'objet, vol. 9, Hermes.

[Chefrour05] Djalel Chefrour. *Developing component based adaptive applications in mobile environments.* In ACM Symposium on Applied Computing (SAC), pages 1146–1150, Santa Fe, New Mexico, USA, 13–17 mars 2005.

[Chen01] Wen-Ke Chen, Matti A. Hiltunen & Richard D. Schlichting. *Constructing Adaptive Software in Distributed Systems.* In 21st International Conference on Distributed Computing Systems (ICDCS-21), pages 635–643, Mesa, AZ, avril 2001.

[Cheng02] S.-W. Cheng, D. Garlan, B. Schmerl, P. Steenkiste & N. Hu. *Software architecture-based adaptation for grid computing.* In 11th IEEE International Symposium on High Performance Distributed Computing, pages 389–398, Edinburgh, Scotland, 2002.

[Cheng04] Shang-Wen Cheng, An-Cheng Huang, David Garlan, Bradley Schmerl & Peter Steenkiste. *Rainbow : Architecture-based Self-adaptation with Reusable Infrastructure.* Submitted for publication, 2004.

[Clarke01] M. Clarke, G. S. Blair, G. Coulson & N. Parlavantzas. *An Efficient Component Model for the Construction of Adaptive Middleware.* In IFIP / ACM International Conference on Distributed Systems Platforms (Middleware), Heidelberg, Germany, novembre 2001.

[Costa00] Fabio M. Costa & Gordon S. Blair. *Integrating Meta-Information Management and Reflection in Middleware.* In 2nd International Symposium on Distributed Objects & Applications (DOA'00), pages 133–143, Antwerp, Belgium, 21–23 septembre 2000.

[Coulouri01] George Coulouris, Jean Dollimore & Tim Kindberg. Distributed systems — concepts and design. Addison-Wesley Publishers, 3 edition, 2001.

[Coulson02] G. Coulson, G. S. Blair, M. Clarke & N Parlavantzas. *The Design of a Configurable and Reconfigurable Middleware Platform.* Distributed Computing, vol. 15, no. 2, pages 109–126, 2002.

[Cuesta02] Carlos E. Cuesta, Pablo de la Fuente, Manuel Barrio-Solórzano, & Encarnación Beato. *Coordination in a Reflective Architecture Description Language.* In F. Arbab & C. Talcott, editeurs, Coordination 2002. LNCS 2315, Springer-Verlag, 2002.

[David02] Pierre-Charles David & Thomas Ledoux. *An Infrastructure for Adaptable Middleware.* In DOA'02, Irvine, California, USA, octobre 2002.

[David03] Pierre-Charles David & Thomas Ledoux. *Towards a Framework for Self-adaptive Component-Based Applications.* In J.-B. Stefani, I. Demeure & D. Hagimont, editeurs, DAIS, pages 1–14, Paris, France, octobre 2003.

[Davies96] Nigel Davies, Adrian Friday, Gordon S. Blair & Keith Cheverst. *Distributed Systems Support for Adaptive Mobile Applications.* Mobile Networks and Applications (MONET), vol. 1, no. 4, pages 399–408, 1996.

[de Lara01] Eyal de Lara, Dan S. Wallach & Willy Zwaenepoel. *Puppeteer : Component-based Adaptation for Mobile Computing.* In 3rd USENIX Symposium on Internet Technologies and Systems (USITS), San Fransisco, CA, mars 2001.

[de Lara02a] Eyal de Lara. *Component-Based Adaptation for Mobile Computing.* PhD thesis, Rice University, Houston, TX, avril 2002.

[de Lara02b] Eyal de Lara, Dan S. Wallach & Willy Zwaenepoel. *HATS : Hierrchical adaptive transmission Scheduling for Multi-Application Adaptation.* In Multimedia Computing and Networking Conference (MMCN'02), San Jose, CA, janvier 2002.

[Dellaroc96] Chrysanthos Dellarocas. *A Coordination Perspective on Software Architecture : Towards a Design Handbook for Integrating Software Components.* PhD thesis, MIT Center for Coordination Science, février 1996.

[Dowling01] Jim Dowling & Vinny Cahill. *Dynamic Software Evolution and the K-Component Model.* In Workshop on Software Evolution, OOPSLA, 2001.

[Dutton97] Ken Dutton, William Barraclough, Steve Thompson & Bill Barraclough. The art of control engineering. Addison-Wesley, 1997.

[Efstrati02] Christos Efstratiou, Adrian Friday, Nigel Davies & Keith Cheverst. *A Platform Supporting Coordinated Adaptation in Mobile Systems.* In 4th IEEE Workshop on Mobile Computing Systems and Applications (WMCSA'02), Callicoon, New York, juin 2002.

[Emmerich00] Wolfgang Emmerich. *Software Engineering and Middleware : A Roadmap.* In The Future of Software Engineering - 22 Int. Conf. on Software Engineering, pages 117–129. ACM Press, mai 2000.

[Engler95] Dawson R. Engler, M. Frans Kaashoek & James O'Toole Jr. *Exokernel : an operating system architecture for application-level resource management.* In 15th ACM Symposium on Operating Systems Principles (SOSP '95), pages 251–266, Colorado, USA, décembre 1995.

[Fabry76] R. S. Fabry. *How to design a system in which modules can be changed on the fly.* In 2nd international conference on Software engineering, pages 470–476, San Francisco, CA, USA, 1976. IEEE Computer Society Press.

[Fox97] A. Fox, S. Gribble, Y. Chawathe, E. Brewer & P. Gauthier. *Cluster-Based Scaleable Network Services.* In 16th ACM Symposium on Operating System Principles (SOSP 97), octobre 1997.

[Fox98a] A. Fox, I. Goldberg, S. Gribble, D. Lee, A. Polito & E. Brewer. *Experience With Top Gun Wingman, A Proxy-Based Graphical Web Browser for the USR PalmPilot.* In IFIP International Conference on Distributed Systems Platforms and Open Distributed Processing (Middleware '98), Lake District, UK, septembre 1998.

[Fox98b] Armando Fox, Steven D. Gribble, Yatin Chawathe & Eric A. Brewer. *Adapting to Network and Client Variation Using Active Proxies : Lessons and Perspectives.* IEEE Personal Communications, septembre 1998.

[Friday96] Adrian Friday. *Infrastructure Support for Adaptive Mobile Applications.* PhD thesis, Computing Department, Lancaster University, U.K, Bailrigg, Lancaster, septembre 1996.

[Friday99] Adrian Friday, Nigel Davies, Gordon Blair & Keith Cheverst. *Developing Adaptive Applications : The MOST Experience.* Journal of Integrated Computer-Aided Engineering, vol. 6, no. 2, pages 143–157, 1999.

[Fu01] Xiaodong Fu, Weisong Shi, Anatoly Akkerman & Vijay Karamcheti. *CANS : Composable, Adaptive Network Services Infrastructure.* In USENIX Symposium on Internet Technologies and Systems (USITS), mars 2001.

[Gamma95] Erich Gamma, Richard Helm, Ralph Johnson & John Vlissides. Design patterns – elements of reusable object-oriented software. Addison–Wesley, 1995.

[Garlan00] David Garlan. *Software Architecture and Object-Oriented Systems.* In IPSJ Object-Oriented Symposium, Tokyo, Japan, août 2000.

[Geihs01] Kurt Geihs. *Middleware Challenges Ahead.* Computer, vol. 34, no. 6, pages 24–31, juin 2001.

[Goel99] Ashvin Goel, David Steere, Calton Pu & Jonathan Walpole. *Adaptive Resource Management Via Modular Feedback Control.* In HOTOS, 1999.

[Gschwind02] Thomas Gschwind. *Adaptation and Composition Techniques for Component-Based Software Engineering.* PhD thesis, Technichen Univesität Wien, Austria, Febrary 2002.

[Heineman98] George T. Heineman. *A Model for Designing Adaptable Software Components.* In In 22nd Annual International Computer Software and Applications Conference, pages 121–127, Vienna, Austria, août 1998.

[Heineman00] George T. Heineman. *An Evaluation of Component Adaptation Techniques.* Rapport technique WPI-CS-TR-99-04, Computer Science Department Worcester Polytechnic Institute, Worcester, MA, USA, 2000.

[Heuzerot01] Dirk Heuzeroth, Welf Löwe, Andreas Ludwig & Uwe Aßmann. *Aspect-Oriented Configuration and Adaptation of Component Communication.* In GCSE 2001, pages 58–69. Springer-Verlag, LNCS 2186, 2001.

[Hillier01] Frederick S. Hillier & Gerald J. Lieberman. Introduction to operations research, seventh edition. McGraw-Hill, New York, 2001.

[Hutchins91] Norm Hutchinson & Larry Peterson. *The x-Kernel : An architecture for implementing network protocols.* IEEE Transactions on Software Engineering, vol. 17, no. 1, pages 64–76, janvier 1991.

[Hölzle93] Urs Hölzle. *Integrating Independently-Developed Components in Object-Oriented Languages.* In O. Nierstrasz, editeur, ECOOP '93, pages 36–56, Kaiserslautern, Germany, juillet 1993. Springer-Verlag, LNCS 707.

[Indulska01] Jadwiga Indulska, Seng Wai Loke, Andry Rakotonirainy & Arkady B. Zaslavsky. *An Open Architecture for Pervasive Computing.* In International IFIP TC6/WG6.1 Conference on New Developments in Distributed Applications and interoperable Systems DAIS 01, pages 175–187, Krakow, septembre 2001. Kluwer.

[Janssens01] N. Janssens, S. Michiels & P. Verbaeten. *DiPS/CuPS : a Framework for Runtime Customizable Protocol Stacks.* Rapport technique CW328, Department of Computer Science, K. U. Leuven, nov 2001.

[Joseph97] Anthony D. Joseph, Joshua A. Tauber & M. Frans Kaashoek. *Mobile Computing with the Rover Toolkit.* IEEE Transactions on Computers : Special issue on Mobile Computing, vol. 46, no. 3, mars 1997.

[Karsai01] Gabor Karsai, Akos Ledeczi, Janos Sztipanovits, Gábor Péceli, Gyula Simon & Tamás Kovácsházy. *An Approach to Self-Adaptive Software based on Supervisory Control.* In International Workshop on Self-Adaptive Software (IWSAS), pages 25–38, Balatonfred, Hungary, 2001. Springer-Verlag, LNCS 2614.

[Katz96] R. H. Katz, E. A. Brewer, E. Amir, H. Balakrishnan, A. Fox, S. Gribble, T. Hodes, D. Jiang, G. T. Nguyen, V. Padmanabhan & M. Stemm. *The Bay Area Research Wireless Access Network (BARWAN).* In Forty-First IEEE Computer Society International Conference (COMPCON), pages 15–20, Santa Clara, CA, février 1996.

[Kiczales91] Gregor Kiczales, Jim des Rivieres & Daniel G. Bobrow. The art of the metaobject protocol. The MIT Press, 1991.

[Kiczales97] Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina V. Lopes, Jean-Marc Loingtier & John Irwin. *Aspect Oriented Programming.* In 11th European Conference on Object-Oriented Programming, ECOOP 97. Springer-Verlag, LNCS 1241, juin 1997.

[Kiczales01] Gregor Kiczales, Erik Hilsdale, Jim Hugunin, Mik Kersten, Jeffrey Palm & William Griswold. *Getting started with ASPECTJ.* Communications of the ACM, vol. 44, no. 10, pages 59–65, octobre 2001.

[Kon00a] Fabio Kon & Roy Campbell. *Dependence Management in Component-Based Distributed Systems.* IEEE Concurrency, vol. 8, no. 1, pages 26–36, -janviermars 2000.

[Kon00b] Fabio Kon, Manuel Román, Ping Liu, Jina Mao, Tomonori Yamane, Luiz Claudio Magalhães & Roy H. Campbell. *Monitoring, Security, and Dyna-*

mic Configuration with the dynamicTAO Reflective ORB. In Middleware, New York, USA, 3–7 avril 2000.

[Kon02] Fabio Kon, Fabio Costa, Roy Campbell & Gordon Blair. *The Case for Reflective Middleware.* Communications of the ACM, vol. 45, no. 6, pages 33–38, juin 2002.

[Kowalsky92] R. Kowalsky. *Database updates in event calculus.* Journal of Logic Programming, vol. 12, pages 121–146, 1992.

[Lamport78] Leslie Lamport. *Time, clocks, and the ordering of events in a distributed system.* Communications of the ACM, vol. 21, no. 7, pages 125–133, juillet 1978.

[Layaida02] Oussama Layaida & Daniel Hagimont. *Adaptation dynamique dans une application multimédia répartie.* In Journées Francophone d'Accès Intelligent aux Documents Multimédias sur l'Internet, Sousse, Tunisie, Juin 2002.

[Loyall98] Joseph P. Loyall, Richard E. Schantz, John A. Zinky & David E. Bakken. *Specifying and Measuring Quality of Service in Distributed Object Systems.* In First International Symposium on Object-Oriented Real-Time Distributed Computing (ISORC '98), Kyoto, Japan, 20–22 avril 1998.

[Loyall01] Joseph P. Loyall, Jeanna M. Gossett, Christopher D. Gill, Richard E. Schantz, John A. Zinky, Partha Pal, Rich Shapiro amd Craig Rodrigues, Michael Atighetchi & David Karr. *Comparing and Contrasting Adaptive Middleware Support in Wide-Area and Embedded Distributed Object Applications.* In 21st IEEE International Conference on Distributed Computing Systems (ICDCS-21), Phoenix, Arizona, 16–19 avril 2001.

[Luckham95] David C. Luckham & James Vera. *An Event-Based Architecture Definition Language.* IEEE Transactions on Software Engineering, vol. 21, no. 9, pages 717–734, juillet 1995.

[Maes87] Pattie Maes. *Concepts and Experiments in Computational Reflection.* In Object-Oriented Programming Systems, Languages, and Applications Conference (OOPSLA'87), pages 147–155, décembre 1987.

[Malone94] Thomas W. Malone & Kevin Crowston. *The Interdisciplinary Study of Coordination.* ACM Computing Surveys, vol. 26, no. 1, pages 87–119, mars 1994.

[Mascolo02] Cecilia Mascolo, Licia Capra & Wolfgang Emmerich. *Middleware for Mobile Computing (A Survey).* In E. Gregori, G. Anastasi & S. Basagni, editeurs, Advanced Lectures on Networking, LNCS 2497, pages 20–58, Pisa, Italy, 2002.

[McAffer96] J. McAffer. *Meta-level Architecture Support for Distributed Objects.* In Reflection'96, pages 39–62, San Francisco, USA, octobre 1996.

[Medvidov00] Nenad Medvidovic & Richard N. Taylor. *A Classification and Comparison Framework for Software Architecture Description Languages.* IEEE Transactions on Software Engineering, vol. 26, no. 1, pages 70–93, 2000.

[Meng00] Alex C. Meng. *On Evaluating Self-Adaptive Software.* In International Workshop on Self-Adaptive Software (IWSAS), pages 65–73, Oxford, UK, 2000. Springer-Verlag, LNCS 1936.

[Metayer98] D. Le Metayer. *Describing software architecture styles using graph grammars.* IEEE Transactions on Software Engineering, vol. 24, no. 7, pages 521–553, juillet 1998.

[Meyer99a] Bertrand Meyer. *On to components.* IEEE Computer, vol. 32, no. 1, pages 139–140, janvier 1999.

[Meyer99b] Bertrand Meyer. *The Significance of Components.* Software development magazine, URL : http ://www.sdmagazine.com/, novembre 1999.

[Mouël01] F. Le Mouël & F. André. *AeDEn : un cadre général pour une distribution adaptative et dynamique des applications en environnements mobile.* Revue Electronique sur les Réseaux et l'Informatique Répartie (RERIR), no. 11, mars 2001.

[Noble97] Brian Noble, M. Satyanarayanan, D. Narayanan, J. Tilton, J. Flinn & K. Walker. *Agile Application-Aware Adaptation for Mobility.* In 16th ACM Symposium on Operating System Principles, St. Malo, France, octobre 1997.

[Noble99] Brian Noble & M. Satyanarayanan. *Experience with adaptive mobile applications in Odyssey.* Mobile Networks and Applications, vol. 4, 1999.

[Nocedal99] Jorge Nocedal & Stephen J. Wright. Numerical optimization. Springer-Verlag, New York, 1999.

[Oreizy99] Peyman Oreizy & Michael M. Gorlick et al. *An Architecture-Based Approach to Self-Adaptive Software.* IEEE Intelligent Systems, vol. 14, no. 3, pages 54–62, mai/juin 1999.

[Pawlak02] R. Pawlak, L. Duchien, G. Florin & L. Seinturier. *JAC : Un Framework pour la Programmation Orientée Aspect en Java.* L'Objet, Hermes, vol. 8, no. 4, pages 145–168, 2002.

[Pitoura98] Evaggelia Pitoura. *Software Models for Mobile Wireless Computing.* Summer School on Data Management for Mobile Computing, Jyvaskyla, Finland, août 1998.

[Press92] William H. Press, William T. Vetterling & Brian P. Flannery. Numerical recipes in c, the art of scientific computing, 2nd edition. Brian P. Flannery, NY, USA, 1992.

[Rakotoni01] Andry Rakotonirainy, Jadwiga Indulska, Seng Wai Loke & Arkady B. Zaslavsky. *Middleware for Reactive Components : An Integrated Use of Context, Roles, and Event Based Coordination.* In Middleware, pages 77–98, 12–16 novembre 2001.

[Riveill00] Michel Riveill & Philippe Merle. *La programmation par composants*, 2000. Techniques de l'Ingénieur, collection Informatique, H2759.

[Rm-odp99] Rm-odp, 1999. Information Technology - Open Distributed Processing - Reference Model - Enterprise Language (ISO/IEC 15414 ITU-T Recommendation X.911) juillet.

[Russell02] Stuart Russell & Peter Norvig. Artificial intelligence : A modern approach, 2nd edition. Prentice Hall, New Jersey, 2002.

[Salkintz99] A. K. Salkintzis. *A Survey of Mobile Data Networks.* IEEE Communication Surveys, vol. 2, no. 3, 1999.

[Satyanar96a] M. Satyanarayanan. *Fundamental Challenges in Mobile Computing.* In Fifteenth ACM Symposium on Principles of Distributed Computing, Philadelphia, PA, mai 1996.

[Satyanar96b] M. Satyanarayanan. *Mobile Information Access.* IEEE Personal Communications, vol. 3, no. 1, février 1996.

[Schantz02] Richard E. Schantz & Douglas C. Schmidt. *Research Advances in Middleware for Distributed Systems : State of the Art.* In IFIP World Computer Congress, Montreal, Canada, août 2002.

[Schmidt03a] Douglas C. Schmidt & Frank Buschmann. *Patterns, Frameworks, and Middleware : Their Synergistic Relationships.* In 25th international conference on Software engineering, pages 694–704, Portland, Oregon, USA, 2003. IEEE Computer Society Press.

[Schmidt03b] Douglas C. Schmidt, Balachandran Natarajan, Aniruddha Gokhale, Chris Gill & Nanbor Wang. *TAO : A Pattern-Oriented Object Request Broker for Distributed Real-time and Embedded Systems.* IEEE Distributed Systems Online, 2003. http ://dsonline.computer.org/middleware/articles/dsonline-TAO.html.

[Schneide99] Jean-Guy Schneider & Oscar Nierstrasz. *Components, Scripts and Glue.* In Leonor Barroca, Jon Hall & Patrick Hall, editeurs, Software Architectures — Advances and Applications, pages 13–25. Springer Verlag, 1999.

[Segarra00] Maria-Teresa Segarra. *Une plate-forme à composants adaptables pour la gestion des environnements sans fil.* PhD thesis, Université de Rennes 1, novembre 2000.

[Smith82] Brian C. Smith. *Reflection and Semantics in a Procedural Programming Language.* PhD thesis, MIT, janvier 1982.

[Sudame99] Pradeep Sudame & B. Badrinath. *On Providing Support for Protocol Adaptation in Wireless Networks.* ACM Baltzer journal of Mobile Networks and Applications (MONET) special issue on Wireless Internet and Intranet Access, 1999.

[Szypersk98] Clemens Szyperski. Component software : beyond object-oriented programming. ACM Press/Addison-Wesley Publishing Co., New York, USA, 1998.

[Szypersk00] Clemens Szyperski. *Point, Counterpoint.* Software development magazine, URL : http ://www.sdmagazine.com/, février 2000.

[Tanenbau96] Andrew S. Tanenbaum. Computer networks. Prentice Hall PRT, New Jersey, 3 edition, 1996.

[Tanenbau02] Andrew S. Tanenbaum & Maarten van Steen. Distributed systems — principles and paradigms. Prentice-Hall PTR, New Jersey, 2002.

[Terry98] D. B. Terry, K. Petersen, M. J. Spreitzer & M. M. Theimer. *The Case for Non-transparent Replication : Examples from Bayou.* IEEE Data Engineering, no. 4, December 1998.

[Troya98] José M. Troya & Antonio Vallecillo. *A Refective Component Model for Open Systems.* In Workshop on Reflective OO Programs and Systems, ECOOP'98, 1998.

[Truyen01] E. Truyen, B. Vanhaute, W. Joosen, P. Verbaeten & B. N. Jørgensen. *Dynamic and Selective Combination of Extensions in Component-based Applications.* In 23rd International Conference on Software Engineering, pages 233–242, Toronto, Canada, mai 2001.

[Truyen02a] Eddy Truyen, Wouter Joosen & Pierre Verbaeten. *Consistency Management in the presence of Simultaneous Client-Specific Views.* In International Conference on Software Maintenance (ICSM), pages 501–510, Montreal, Canada, 2002.

[Truyen02b] Eddy Truyen, Wouter Joosen & Pierre Verbaeten. *Run-time Support for Aspects in Distributed System Infrastructure.* In First AOSD Workshop on Aspects, Components, and Patterns for Infrastructure Software (ACP4IS), 2002.

[van Rene98] Robbert van Renesse, Kenneth P. Birman, Mark Hayden, Alexey Vaysburd & David A. Karr. *Building Adaptive Systems Using Ensemble.* Software - Practice and Experience, vol. 28, pages 963–979, 9 1998.

[Varshney00] Upkar Varshney & Ron Vetter. *Emerging Mobile and Wireless Networks.* Communications of the ACM (CACM), vol. 43, no. 6, pages 73–81, Juin 2000.

[Warfield01] Andrew Warfield, Yvonne Coady & Norm Hutchinson. *Identifying Open Problems in Distributed Systems.* In European Research Seminar on Advances in Distributed Systems ERDAS, Bertinorom Taly, 14–18 mai 2001.

[Wermelin99] Michel Wermelinger. *Specification of Software Architecture Reconfiguration.* PhD thesis, Universidade Nova de Lisboa, Sept 1999.

[Yarvis01a] Mark Yarvis. *Conductor : Distributed Adaptation for Heterogeneous Networks.* PhD thesis, UCLA Department of Computer Science, novembre 2001.

[Yarvis01b] Mark Yarvis, Peter Reiher, Kevin Eustice & Gerald J. Popek. *Conductor : Enabling Distributed Adaptation.* Rapport technique, UCLA, juin 2001.

[Yokote92] Yasuhiko Yokote. *The Apertos Reflective Operating System : The Concept and its Implementation.* In OOPSLA'92, pages 414–434, Vancouver, Canada, octobre 1992. ACM Press.

Annexe A

DTDs *Aceel*

A.1 DTD XML pour les politiques d'adaptation

```
--------------------------------------------------------------------------------
1:        <?xml version="1.0" encoding="iso-8859-1" ?>
2:        <!-- An XML DTD for Aceel components adaptation policies -->
3:        <!-- Copyright (C) Djalel Chefrour 2002 -->
4:        <!-- Start date: 2002-12-16 -->
5:
6:        <!ELEMENT component             (header, body)>
7:        <!ELEMENT header                (implementation+)>
8:        <!ATTLIST header
9:                  id                    CDATA #REQUIRED
10:                 priority              CDATA #REQUIRED
11:                 state_type            (separated | transferable) #REQUIRED
12:                 impls_type            (atomic | restartable | cancelable) #REQUIRED
13:                 application           CDATA #REQUIRED >
14:       <!ELEMENT implementation        EMPTY>
15:       <!ATTLIST implementation
16:                 name                  CDATA #REQUIRED
17:                 url                   CDATA #REQUIRED
18:                 loading               (initial | differed) "initial"
19:                 unloading             (immediate | differed) "differed">
20:
21:       <!ELEMENT body                  (adaptation_dimension+)>
22:       <!ELEMENT adaptation_dimension  (polling_periods?, adaptation_rules)>
23:       <!ATTLIST adaptation_dimension
24:                 name                  CDATA #REQUIRED
25:                 weight                CDATA "1">
26:       <!ELEMENT polling_periods       (poll+)>
27:       <!ELEMENT poll                  EMPTY>
28:       <!ATTLIST poll
29:                 resource              CDATA #REQUIRED
30:                 host                  CDATA "localhost"
31:                 every                 CDATA #REQUIRED >
32:       <!ELEMENT adaptation_rules      (rule+)>
33:       <!ENTITY % condition            "(any_condition | when | and | or | not)">
34:       <!ELEMENT and                   (%condition; , (%condition;)+ )>
35:       <!ELEMENT or                    (%condition; , (%condition;)+ )>
36:       <!ELEMENT not                   (%condition;)>
37:       <!ELEMENT rule                  (%condition;  , adapt+)>
38:       <!ATTLIST rule
39:                 rule_id               CDATA #IMPLIED
40:                 quality_index         CDATA "100">
```

```
41:         <!ELEMENT any_condition         EMPTY>
42:         <!ELEMENT when                  EMPTY>
43:         <!ATTLIST when
44:                   resource              CDATA #REQUIRED
45:                   ge                    CDATA #IMPLIED
46:                   le                    CDATA #IMPLIED
47:                   gt                    CDATA #IMPLIED
48:                   lt                    CDATA #IMPLIED
49:                   eq                    CDATA #IMPLIED
50:                   ne                    CDATA #IMPLIED>
51:         <!ELEMENT adapt                 EMPTY>
52:         <!ATTLIST adapt
53:                   action                (tuneParameter | replaceImpl | reifyMethod |
54:                                         addComponent | delComponent |
55:                                         addConnector | delConnector) #REQUIRED
56:                   params                CDATA #REQUIRED
57:                   cost                  CDATA "1" >
```

A.2 DTD XML pour les politiques de coordination

```
1:          <?xml version="1.0" encoding="iso-8859-1" ?>
2:          <!-- An XML DTD for Aceel components coordination policies -->
3:          <!-- Copyright (C) Djalel Chefrour 2002 -->
4:          <!-- Start date: 2005-08-04 -->
5:
6:          <!ELEMENT application       (header, body)>
7:          <!ELEMENT header            (component+)>
8:          <!ATTLIST header
9:                    id                CDATA #REQUIRED>
10:         <!ELEMENT component         EMPTY>
11:         <!ATTLIST component
12:                   id                CDATA #REQUIRED
13:                   host              CDATA #REQUIRED>
14:         <!ELEMENT body              (coordination_rule+)>
15:         <!ENTITY % condition        "(when | or)">
16:         <!ELEMENT or                (%condition;, (%condition;)+)>
17:         <!ELEMENT when              EMPTY>
18:         <!ATTLIST when
19:                   comp_id           CDATA #REQUIRED
20:                   runs              CDATA #REQUIRED>
21:         <!ELEMENT coordination_rule (%condition;, synchronize)>
22:         <!ELEMENT synchronize       (parallel | sequence)>
23:         <!ENTITY % operator         "(adapt | parallel | sequence)">
24:         <!ELEMENT parallel          (%operator;, (%operator;)+)>
25:         <!ELEMENT sequence          (%operator;, (%operator;)+)>
26:         <!ELEMENT adapt EMPTY>
27:         <!ATTLIST adapt
28:                   comp_id           CDATA #REQUIRED
29:                   action            CDATA #REQUIRED
30:                   params            CDATA #REQUIRED>
```

Annexe B

Politiques de l'intergiciel de coordination

B.1 Politique d'adaptation de l'*optimisateur*

```
1:         <?xml version="1.0" encoding="iso-8859-1" standalone="no" ?>
2:         <!DOCTYPE component SYSTEM "/opt/aceel/xmldtds/adapolicy.dtd">
3:
4:         <component>
5:           <header id="Optimizer" priority="1" application="aceel_middleware"
6:                   state_type="separated" impls_type="atomic" >
7:             <implementation name="enumerative-search" url="optimizer"
8:                             loading="initial" unloading="immediate" />
9:             <implementation name="simulated-annealing" url="optimizer"
10:                            loading="differed" unloading="immediate" />
11:          </header>
12:          <body>
13:            <adaptation_dimension name="OptimizationMethod">
14:              <polling_periods>
15:                <poll resource="mem.free" every="30s"/>
16:                <poll resource="cpu.loadavg1" every="30s"/>
17:                <poll resource="comp[Coordinator].cardinal_D" every="5m"/>
18:              </polling_periods>
19:              <adaptation_rules>
20:                <rule quality_index="100%">
21:                  <and>
22:                    <when resource="mem.free" ge="20Mb" />
23:                    <when resource="cpu.loadavg1" lt="0.8" />
24:                    <when resource="comp[Coordinator].cardinal_D" lt="10000" />
25:                  </and>
26:                  <adapt action="replaceImpl" params="enumerative-search" cost="1"/>
27:                </rule>
28:                <rule quality_index="80%">
29:                  <any_condition/>
30:                  <adapt action="replaceImpl" params="simulated-annealing" cost="1"/>
31:                </rule>
32:              </adaptation_rules>
33:            </adaptation_dimension>
34:          </body>
35:        </component>
```

B.2 Politique d'adaptation du *coordinateur*

```
<?xml version="1.0" encoding="iso-8859-1" standalone="no" ?>
<!DOCTYPE component SYSTEM "/opt/aceel/xmldtds/adapolicy.dtd">

<component>
  <header id="Coordinator" priority="1" application="aceel_middleware"
          state_type="separated" impls_type="atomic" >
    <implementation name="peer" url="coordinator" />
    <implementation name="master" url="coordinator" />
    <implementation name="slave" url="coordinator" />
  </header>
  <body>
    <adaptation_dimension name="Timeout">
      <polling_periods>
        <poll resource="net[remote].latency" every="30s"/>
        <poll resource="cpu[remote].speed" every="30s"/>
      </polling_periods>
      <adaptation_rules>
        <rule>
          <and>
            <when resource="net[remote].latency" lt="0.03s" />
            <when resource="cpu[remote].speed" ge="300Mhz" />
          </and>
          <adapt action="tuneParam" params="timeout=0.5s" />
        </rule>
        <rule>
          <any_condition/>
          <adapt action="tuneParam" params="timeout=1s" />
        </rule>
      </adaptation_rules>
    </adaptation_dimension>
  </body>
</component>
```

Annexe C

Politiques des exemples *Aceel*

C.1 Politique d'adaptation du navigateur Web

```
1:         <?xml version="1.0" encoding="iso-8859-1" standalone="no" ?>
2:         <!DOCTYPE component SYSTEM "/opt/aceel/xmldtds/adapolicy.dtd">
3:
4:         <component>
5:           <header id="Browser" priority="2" application="aceel_www"
6:                   state_type="transferable" impls_type="restartable" >
7:             <implementation name="AdaptableDillo" url="browser" />
8:           </header>
9:           <body>
10:            <adaptation_dimension name="download_images" weight="1">
11:              <polling_periods>
12:                <poll resource="net[http_proxy].bandwidth" every="30s"/>
13:              </polling_periods>
14:              <adaptation_rules>
15:                <rule quality_index="100%">
16:                  <when resource="net[http_proxy].bandwidth" ge="56kbps" />
17:                  <adapt action="tuneParameter" params="load_imgs=1" cost="1"/>
18:                </rule>
19:                <rule quality_index="50%">
20:                  <any_condition/>
21:                  <adapt action="tuneParameter" params="load_imgs=0" cost="1"/>
22:                </rule>
23:              </adaptation_rules>
24:            </adaptation_dimension>
25:          </body>
26:        </component>
```

C.2 Politique d'adaptation du récepteur vidéo

```
1:          <?xml version="1.0" encoding="iso-8859-1" standalone="no" ?>
2:          <!DOCTYPE component SYSTEM "/opt/aceel/xmldtds/adapolicy.dtd">
3:
4:          <component>
5:            <header id="Receiver" priority="1" application="aceel_vod"
6:                    state_type="separated" impls_type="cancelable" >
7:              <implementation name="MJPEG" url="recv_impls"
8:                              loading="initial" unloading="differed"/>
9:              <implementation name="MPEG4" url="recv_impls"
10:                             loading="initial" unloading="differed"/>
11:           </header>
12:           <body>
13:             <adaptation_dimension name="Encoding">
14:               <polling_periods>
15:                 <poll resource="comp_state.netBER" every="120s"/>
16:               </polling_periods>
17:               <adaptation_rules>
18:                 <rule quality_index="100%">
19:                   <when resource="comp_state.netBER" ge="10e-5" />
20:                   <adapt action="replaceImpl" params="MJPEG" cost="2"/>
21:                 </rule>
22:                 <rule quality_index="100%">
23:                   <any_condition/>
24:                   <adapt action="replaceImpl" params="MPEG4" cost="2"/>
25:                 </rule>
26:               </adaptation_rules>
27:             </adaptation_dimension>
28:           </body>
29:         </component>
```

C.3 Politique d'adaptation de l'émetteur vidéo

```
1:          <?xml version="1.0" encoding="iso-8859-1" standalone="no" ?>
2:          <!DOCTYPE component SYSTEM "/opt/aceel/xmldtds/adapolicy.dtd">
3:
4:          <component>
5:            <header id="Streamer" priority="1" application="aceel_vod"
6:                    state_type="separated" impls_type="cancelable" >
7:              <implementation name="HackedDSS" url="strm_impl" />
8:            </header>
9:            <body>
10:             <adaptation_dimension name="FrameRate" weight="2">
11:               <polling_periods>
12:                 <poll resource="mem[client].free" every="2m"/>
13:                 <poll resource="cpu[client].speed" every="5m"/>
14:                 <poll resource="cpu[client].loadavg1" every="1m"/>
15:               </polling_periods>
16:               <adaptation_rules>
17:                 <rule quality_index="100%">
18:                   <and>
19:                     <when resource="mem[client].free" ge="40Mb" />
20:                     <when resource="cpu[client].speed" ge="100MHz" />
21:                     <when resource="cpu[client].loadavg1" lt="3" />
22:                   </and>
23:                   <adapt action="tuneParameter" params="framerate=30fps" cost="1"/>
24:                 </rule>
25:                 <rule quality_index="50%">
26:                   <any_condition/>
27:                   <adapt action="tuneParameter" params="framerate=15fps" cost="1"/>
28:                 </rule>
29:               </adaptation_rules>
30:             </adaptation_dimension>
31:             <adaptation_dimension name="BitRate" weight="1">
32:               <polling_periods>
33:                 <poll resource="net[client].bandwidth" every="30s"/>
34:               </polling_periods>
35:               <adaptation_rules>
36:                 <rule quality_index="60%">
37:                   <when resource="net[client].bandwith" ge="128kbps" lt="256kbps" />
38:                   <adapt action="tuneParameter" params="bitrate=86kbps" cost="1"/>
39:                 </rule>
40:                 <rule quality_index="80%">
41:                   <when resource="net[client].bandwidth" ge="256kbps" lt="512kbps" />
42:                   <adapt action="tuneParameter" params="bitrate=170kbps" cost="1"/>
43:                 </rule>
44:                 <rule quality_index="100%">
45:                   <when resource="net[client].bandwidth" ge="512kbps" />
46:                   <adapt action="tuneParameter" params="bitrate=340kbps" cost="1"/>
47:                 </rule>
48:               </adaptation_rules>
49:             </adaptation_dimension>
50:             <adaptation_dimension name="Encoding" weight="1">
51:               <adaptation_rules>
52:                 <rule quality_index="100%">
53:                   <any_condition/>
54:                   <adapt action="tuneParameter" params="format=MPEG4" cost="1"/>
55:                 </rule>
56:                 <rule quality_index="100%">
57:                    <any_condition/>
58:                    <adapt action="tuneParameter" params="format=MJPEG" cost="1"/>
59:                 </rule>
```

```
60:              </adaptation_rules>
61:            </adaptation_dimension>
62:          </body>
63:        </component>
-------------------------------------------------------------------------------------
```

C.4 Politique de coordination de l'application vidéo

```
1:          <?xml version="1.0" encoding="iso-8859-1" standalone="no" ?>
2:          <!DOCTYPE application SYSTEM "/opt/aceel/xmldtds/coopolicy.dtd">
3:
4:          <application>
5:            <header id="aceel_vod">
6:              <component id="Streamer" host="pomerol.irisa.fr" />
7:              <component id="Receiver" host="sloy.irisa.fr" />
8:            </header>
9:            <body>
10:             <coordination_rule>
11:               <when comp_id="Receiver" runs="replaceImpl"/>
12:               <synchronize>
13:                 <sequence>
14:                   <parallel>
15:                     <adapt comp_id="Receiver" action="replaceImpl" params="F"/>
16:                     <adapt comp_id="Streamer" action="tuneParamter" params="format=F"/>
17:                   </parallel>
18:                   <adapt comp_id="Streamer" action="tuneParameter"
19:                          params="time_cursor=Receiver.state.time_cursor"/>
20:                 </sequence>
21:               </synchronize>
22:             </coordination_rule>
23:           </body>
24:         </application>
```

Résumé

L'objectif de cette thèse est de contribuer au développement de méthodes et d'outils pour l'adaptation de composants logiciels et pour leur coordination en environnement dynamique.

Nous proposons une capture des mécanismes de l'adaptabilité dans un modèle de composants générique basé sur la réflexivité et les techniques objet. Ces composants reposent sur un service de surveillance de l'environnement et sur un intergiciel de coordination des adaptations. La coordination inclut la gestion des dépendances entre les composants interagissants et/ou concurrents.

En instaurant la séparation des préoccupations, notre approche facilite la tâche du développeur à qui nous offrons des langages déclaratifs pour spécifier les politiques qui contrôlent les adaptations des composants et leur coordination dans des scripts séparés.

Nous avons montré l'utilité de notre modèle, appelé *Aceel*, en l'utilisant pour développer une application vidéo et un navigateur Web adaptables en environnement mobile.

Mots clés : modèle de composant adaptable, adaptation application de vidéo, politiques XML, intergiciel de coordination, environnement mobile.

Abstract

The aim of this thesis is to contribute to the development of methods and tools for the adaptation of software components and for their coordination in dynamic environments.

We propose to capture the mechanisms of adaptability in a generic component model based on reflexivity and object-oriented techniques. The components use an underlying service for monitoring the environment and a middleware for coordinating multiple adaptations. Coordination includes the management of dependencies between interacting components and dependencies between concurrent components.

By emphasizing separation of concerns, our approach eases the developer task by offering him declarative languages to specify the policies that control the components adaptations and the components coordination in separate scripts.

We have showed the utility of our model, named *Aceel*, by using it to develop an adaptable video on demand application and an adaptable Web browser in a mobile environment.

Keywords : adaptive component model, video on demand adaptation, XML policies, coordination middleware, mobile environment.