Duplication et cohérence configurables dans les applications réparties à base de composants

Vania Marangozova

▶ To cite this version:

Vania Marangozova. Duplication et cohérence configurables dans les applications réparties à base de composants. Réseaux et télécommunications [cs.NI]. Université Joseph-Fourier - Grenoble I, 2003. Français. NNT : . tel-00004374

HAL Id: tel-00004374
https://theses.hal.science/tel-00004374

Submitted on 29 Jan 2004

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

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

UNIVERSITÉ JOSEPH FOURIER - GRENOBLE I

THÈSE DE DOCTORAT

pour obtenir le grade de

DOCTEUR DE L'UNIVERSITÉ JOSEPH FOURIER

Discipline : Informatique, Systèmes et Communication

présentée et soutenue publiquement par

VANIA MARANGOZOVA

Le 17 juin 2003

DUPLICATION ET COHÉRENCE CONFIGURABLES DANS LES APPLICATIONS RÉPARTIES À BASE DE COMPOSANTS

Directeur de thèse : Daniel HAGIMONT

JURY

Président :	S. Krakowiak, Professeur à l'Université Joseph Fourier
Rapporteurs :	J.M. Geib, Professeur à l'Université des Sciences et Technologies de Lille
	M. Riveill, Professeur à l'Université de Nice-Sophia Antipolis
Examinateurs :	J. Mossière, Professeur à l'Institut National Polytechnique de Grenoble
	P. Dechamboux, Responsable d'Unité à France Telecom R&D, Grenoble
Directeur de thèse :	D. Hagimont, Chargé de recherche à l'INRIA Rhône-Alpes

À mes parents

Remerciements

Je tiens à remercier tous les membres du jury pour m'avoir fait l'honneur d'accepter de juger ce travail de thèse :

- Sacha KRAKOWIAK, professeur à l'Université Joseph Fourier - Grenoble I, président du jury.
- Michel RIVEILL, professeur à l'Université de Nice-Sophia Antipolis, et Jean-Marc GEIB, professeur à l'Université des Sciences et Technologies de Lille, rapporteurs de thèse.
- Jacques MOSSIÈRE, professeur à l'Institut National Polytechnique de Grenoble et Pascal DECHAMBOUX, responsable d'unité à France Télécom R&D, examinateurs.
- Daniel HAGIMONT, Chargé de recherche à l'INRIA Rhône-Alpes, directeur de thèse et examinateur.

Je voudrais remercier Roland BALTER, directeur du projet SIRAC, ainsi que Jean-Bernard STEFANI, directeur du projet SARDES, de m'avoir permis d'évoluer dans leurs équipes de recherche et de travailler avec des personnes remarquables. Je pense tout particulièrement aux professeurs Sacha KRAKOWIAK et Jacques MOSSIÈRE qui, avec générosité et patience, guident les pas des apprentis de la science.

Je ne sais comment remercier à Daniel HAGIMONT, mon directeur de thèse, pour ses conseils, sa tolérance et son soutien tout au long de ces trois ans. Avec sa passion et sa curiosité, il a su m'entraîner et me motiver lors des moments d'hésitation. Avec perspicacité, il a su choisir les voies les plus valorisantes.

Je voudrais exprimer ma gratitude à Jacques MOSSIÈRE qui a été le principal témoin de mon apprentissage d'écriture de cette thèse. Je le remercie pour ses relectures détaillées, pour les leçons de français, pour les nombreux conseils et, surtout, pour son sourire d'encouragement.

Je tiens également à remercier les membres de l'équipe SARDES pour leur présence lors des moments difficiles. Merci à Sara BOUCHENAK, à Noël DE PALMA, à Emmanuel CECCHET et à Eric BRUNETON, déjà docteurs, pour leurs encouragements et leur exemple. Merci également à Olivier CHARRA, à Philippe LAYMAY, à Aline SENART et à tous les autres pour les moments de rire et de détente.

Mes pensées sont aussi dirigées vers l'équipe ADELE et tout particulièrement vers Pierre-Yves CUNIN et Jacky ESTUBLIER qui ont été les premiers à croire en moi et à m'encourager à continuer sur la route de la recherche. Je les en remercie de tout cœur.

Une seule ligne ne saurait transcrire ma gratitude envers mes parents et mon frère qui m'ont offert leur amour et leur soutien pendant toutes ces années d'études. Sans eux, cette thèse n'aurait pas été possible.

Je tiens à remercier, enfin, Stéphane, qui a toujours su m'apaiser et me faire rire, même pendant la remise en question qu'est la rédaction de la thèse.

Résumé

Cette thèse s'intéresse à la gestion configurable de la duplication et de la cohérence dans les applications réparties. Elle vise la définition de mécanismes de configuration qui n'impliquent pas de modification du code métier des applications et qui permettent la réutilisation du code de gestion de la duplication et de la cohérence. Le premier objectif permet d'exécuter les applications dans différents environnements sans les réimplémenter. Le deuxième objectif permet de réutiliser les protocoles de duplication et de cohérence dans différents domaines d'application.

Cette thèse se place dans le contexte des applications réparties à base de composants. Elle permet la configuration non intrusive de la duplication et de la cohérence en se basant sur la séparation entre le code métier et le code système des composants. Elle permet la réutilisation du code de gestion de la duplication et de la cohérence en modélisant les protocoles en termes de composants qui sont indépendants des applications métier.

Les modèles de programmation, de composition et d'exécution proposés sont validés à l'aide d'un prototype Java. Le coût de gestion de la duplication et de la cohérence est évalué et l'applicabilité de l'approche aux environnements CCM et EJB est étudiée. Les capacités de configuration et de réutilisation sont démontrées par des expérimentations avec plusieurs applications et plusieurs protocoles.

Mots-clés : duplication, cohérence, configuration, composant, réutilisation

Abstract

This thesis focuses on replication and consistency configuration in distributed applications. It aims to define configuration mechanisms that do not impose modifications to the business code of applications and that allow reuse of replication and consistency management code. The first point responds to the need to execute applications in different contexts. The second point allows replication protocols to be used in different applications.

The thesis objectives are considered in the case of component-based applications. Non intrusive replication configuration is based on the separation between the components' business code and their system code. Replication code reuse is achieved through modeling of protocols in terms of application-independent components.

The proposed models for programming, composition and execution of protocols and applications are validated using a Java prototype implementation. We measure the cost of our replication management and analyze the way its principles can be applied to the EJB and CCM environments. We show the configuration and reuse facilities of our prototype through several experiments involving different applications, as well as different replication protocols.

Keywords: replication, consistency, configuration, component, reuse

Table des matières

Introduction **17**

I Problématique **21**

1 Gestion système à base de composants **23**

1.1 Principes de l'approche composant . . . 23
1.1.1 Motivations . . . 23
1.1.2 Gestion des applications à base de composants . . . 24
1.1.3 L'approche composant et la séparation des aspects . . . 26
1.1.4 L'approche composant et l'administration . . . 27
1.2 Environnements à composants . . . 27
1.2.1 EJB . . . 28
1.2.2 CCM . . . 31
1.2.3 Systèmes réflexifs . . . 34
1.2.4 Systèmes de gestion de configurations . . . 35
1.2.5 Systèmes de duplication et de cohérence . . . 36
1.2.6 Synthèse . . . 38

2 Duplication et cohérence **39**

2.1 Définitions . . . 39
2.2 Applications de la duplication et de la cohérence . . . 40
2.3 Classification de la duplication et de la cohérence . . . 41
2.3.1 Classification des travaux relatifs à la duplication . . . 41
2.3.2 Classification des travaux relatifs à la cohérence . . . 45
2.3.3 Synthèse . . . 50
2.4 Configuration de la duplication et de la cohérence . . . 52
2.4.1 Niveau de mise en œuvre . . . 52
2.4.2 Transparence de la duplication et de la cohérence . . . 55
2.4.3 Moment de configuration . . . 56
2.4.4 Synthèse . . . 58

3 Positionnement du travail de thèse **61**

4 Proposition **65**

4.1 Aperçu global de la proposition . . . 65
4.2 Modèle de programmation d'applications métier . . . 66
4.2.1 Composants et interfaces . . . 66
4.2.2 Identités, références et noms . . . 67
4.2.3 Assemblage et passage de références . . . 67
4.3 Protocoles de duplication et de cohérence . . . 70
4.3.1 Composants et duplication . . . 70
4.3.2 Composants et cohérence . . . 74
4.3.3 Configuration des protocoles . . . 77
4.3.4 Synthèse . . . 78
4.4 Composition entre applications métier et protocoles . . . 79
4.4.1 Relations entre application métier et protocole . . . 79
4.4.2 Instrumentation de l'application métier . . . 83
4.4.3 Spécialisation des protocoles de duplication et de cohérence . . . 84
4.5 Architecture à l'exécution . . . 85
4.5.1 Architecture générale . . . 85
4.5.2 Gestion de la duplication et de la cohérence . . . 87
4.6 Synthèse de proposition . . . 89

II Mise en œuvre **91**

5 FAR et les applications sans duplication ni cohérence **93**

5.1 Motivations du prototype FAR . . . 93
5.2 Organisation générale de FAR . . . 94
5.3 Description et programmation de composants . . . 95
5.3.1 Déclaration de composants . . . 95
5.3.2 La classe `Component` . . . 97
5.3.3 Implémentation métier d'un composant . . . 98
5.4 Structure de composant . . . 99
5.4.1 Talons et squelettes . . . 99
5.4.2 Conteneur . . . 104
5.5 Environnement d'exécution . . . 108
5.5.1 Gestion des composants locaux . . . 108
5.5.2 Nommage de composants . . . 108
5.5.3 Service de transport de messages . . . 109
5.5.4 Déploiement . . . 109

6 Les protocoles de duplication et de cohérence dans FAR 111

6.1 Instrumentation des applications métier 111
6.1.1 Accès à l'état 111
6.1.2 Contrôle d'état stable 113
6.1.3 Synthèse 117
6.2 Composants de protocole 118
6.2.1 Composants services 118
6.2.2 Composants gestionnaires de configuration 119
6.2.3 Composants copies et clients 127
6.2.4 Composants de protocole : synthèse. 129
6.3 Intégration dans les applications métier 129
6.3.1 Relations entre interfaces 129
6.3.2 Relations entre composants 130
6.3.3 Génération des structures de protocole 132
6.3.4 Intégration des composants de protocole 134

7 Mise en œuvre dans CCM 139

7.1 Applications CCM sans duplication et cohérence 139
7.1.1 Modèle de composants 139
7.1.2 Programmation de composants 140
7.1.3 Structure de composant 141
7.1.4 Environnement d'exécution et déploiement 142
7.2 Duplication et cohérence dans CCM 142
7.2.1 Instrumentation des applications CCM 143
7.2.2 Construction de protocoles dans CCM 144
7.2.3 Intégration des protocoles dans CCM 145
7.3 Mise en œuvre dans OpenCCM 146
7.4 Synthèse 147

8 Mise en œuvre dans EJB **149**
8.1 Les applications EJB sans duplication et cohérence 149
8.1.1 Modèle de composants 149
8.1.2 Programmation de composants 151
8.1.3 Structure de composant 152
8.1.4 Environnement d'exécution et déploiement 153
8.2 Les protocoles de duplication et de cohérence dans EJB 154
8.2.1 Instrumentation des applications EJB 154
8.2.2 Construction de protocoles dans EJB 155
8.2.3 Intégration des protocoles dans EJB 156
8.3 Mise en œuvre dans JOnAS 157
8.4 Synthèse 157

9 Expérimentations **159**
9.1 Applications métier 160
9.1.1 L'application de gestion de réservations Agenda 160
9.1.2 L'application de gestion de listes de courses ShoppingList . . . 161
9.1.3 L'application de gestion de comptes bancaires Banking 163
9.2 Protocoles de duplication et de cohérence 164
9.2.1 Le protocole de gestion de déconnexions DisconnectP 164
9.2.2 Le protocole de cohérence à l'entrée ECP 167
9.3 Intégration de DisconnectP dans ShoppingList. Non-intrusion. 168
9.3.1 ShoppingList avec serveur monolithique 168
9.3.2 ShoppingList avec serveur composé 172
9.4 Intégration d'ECP dans ShoppingList. Souplesse. 174
9.4.1 Modélisation de la ShoppingList 174
9.4.2 Procédure d'intégration 174
9.5 Intégration de DisconnectP dans Agenda. Réutilisation. 176
9.5.1 Procédure d'intégration 176
9.5.2 Performances 177
9.6 Coût de FAR 177
9.6.1 Configuration de test 177
9.6.2 Coût des appels de FAR 178
9.6.3 Taille des composants FAR 180
9.7 Synthèse 181

Conclusion **183**

Bibliographie **187**

Annexes **199**

A Mise en œuvre dans OpenCCM **199**

A.1 Construction de l'application "Hello World" 199
A.1.1 Description IDL des composants de "Hello World" 199
A.1.2 Programmation des composants 200
A.1.3 Déploiement de l'application "Hello World" 202
A.2 Construction du protocole de tolérance aux pannes 203
A.2.1 Instrumentation de l'application "Hello World" 203
A.2.2 Programmation du protocole de tolérance aux fautes 204
A.2.3 Intégration du protocole 206

B Mise en œuvre dans JOnAS **207**

B.1 Construction de l'application "Hello World" 207
B.1.1 Modélisation de l'application "Hello World" 208
B.1.2 Description XML des composants de "Hello World" 208
B.1.3 Programmation des composants de "Hello World" 209
B.1.4 Génération des structures de conteneur 210
B.1.5 Déploiement 211
B.2 Construction du protocole de tolérance aux fautes 212
B.2.1 Instrumentation de l'application "Hello World" 212
B.2.2 Programmation du protocole de tolérance aux fautes 212
B.2.3 Intégration du protocole 213

Introduction

Motivations et objectifs

Cette thèse s'intéresse à la gestion configurable de la duplication et de la cohérence dans les applications réparties. Elle vise la définition de mécanismes de gestion qui permettent aux applications de choisir leur protocole de duplication et de cohérence en fonction de leurs besoins spécifiques de disponibilité et de leurs environnements d'exécution. Cet objectif global peut être décomposé de la manière suivante :

- *Réutilisation du code métier des applications*. Nous voulons fournir des mécanismes qui permettent de configurer la duplication et la cohérence d'une application sans modifier son code métier. En effet, la diversification des plates-formes d'exécution, amenée par le développement important des technologies et des réseaux, fait apparaître le besoin d'exécuter les mêmes applications dans différents environnements. La possibilité de réutilisation permettrait d'exécuter les applications dans différents environnements en configurant leurs cohérence et duplication et sans un processus coûteux de réimplémentation.
- *Réutilisation du code de gestion des protocoles de duplication et de cohérence*. Nous voulons fournir des mécanismes qui permettent de réutiliser, sans modification, pour les besoins de différentes applications, le code des protocoles existants. En effet, la diversification des plates-formes d'exécution fait apparaître des besoins non seulement au niveau des applications mais également au niveau des protocoles : ils doivent pouvoir être utilisés dans un nombre croissant d'environnements différents. Une réutilisation du code des protocoles existants permettrait de facilement mettre en place le même schéma de gestion dans différents domaines d'application. De plus, la réutilisation de parties de protocoles permettrait l'adaptation des protocoles existants et la construction rapide de nouveaux protocoles pour les besoins des environnements émergents.

Les protocoles de duplication et de cohérence existants n'ont pas été conçus pour répondre aux besoins de facilité de construction, d'adaptation et de réutilisation. En effet, les protocoles sont implémentés de manière spécifique à leurs domaines d'application et leur processus de construction requiert la connaissance de ces domaines en plus des concepts de cohérence et de duplication. Leur spécificité rend également difficile la réutilisation. Pour appliquer les mêmes principes de duplication et de cohérence dans différents contextes d'utilisation, les protocoles nécessitent une réimplémentation. Quant à l'adaptation de protocoles déjà existants, étant donné que leurs implantations sont fortement intégrées dans le code des applications les utilisant, leurs adaptations nécessitent une révision globale de ces applications. L'objectif d'adaptation des protocoles sans modification des applications est d'autant plus ambitieux que l'adaptation doit être faite à l'exécution.

Cadre du travail

Cette thèse a été effectuée au sein du projet SARDES[1] (IMAG-LSR et INRIA Rhône-Alpes).

Les travaux de recherche dans SARDES portent sur la construction d'infrastructures réparties adaptables. En particulier, ils s'intéressent aux environnements à grande échelle dont les ressources de calcul changent dynamiquement et étudient les mécanismes d'adaptation préservant la qualité de service des applications s'exécutant dans de tels environnements. Ils portent également sur l'activité d'administration qui inclut la surveillance des applications à l'exécution et leur reconfiguration lors d'une dégradation de performances.

La gestion configurable de la duplication et de la cohérence est directement liée aux thèmes de recherche cités. D'une part, la configuration de la duplication et de la cohérence est un mécanisme d'adaptation visant l'optimalité des performances des applications. D'autre part, la gestion de la duplication et de la cohérence peut être vue comme faisant partie de l'administration puisqu'elle surveille les applications et effectue des reconfigurations pour améliorer leur disponibilité.

Démarche suivie

Nous nous sommes basés sur les principes de l'approche composant. En effet, l'approche composant assure la facilité de construction et la réutilisation en préservant le principe de modularité introduit par la programmation orientée-objet. Elle traite l'adaptation en fonction de l'environnement d'exécution en considérant explicitement la gestion des services système lors des étapes de déploiement des applications et de leur exécution. En adhérant aux principes de séparation des aspects [78], elle permet la configuration des services système sans modification du code métier des applications. Elle permet, par conséquent, la réutilisation de code métier non seulement au sein de différentes applications, mais également dans des contextes d'exécution différents.

Le paradigme composant propose un canevas générique qui peut faciliter la construction, l'adaptation et la réutilisation des protocoles de duplication et de cohérence. En effet, avec le principe d'encapsulation, il peut permettre la construction de protocoles qui s'abstraient de la spécificité des domaines d'application. Son principe de séparation des aspects peut éviter le couplage fort entre protocoles et applications et donc faciliter leur adaptation et réutilisation.

Nous nous sommes intéressés, par conséquent, à l'application des principes de l'approche composant au domaine de la duplication et de la cohérence. Plus particulièrement, nous avons travaillé sur la gestion configurable de la duplication et de la cohérence dans les applications réparties à base de composants. Notre contribution porte plus particulièrement sur la définition d'un modèle à composants pour la gestion de la duplication et de la cohérence, la définition d'un modèle d'intégration entre applications et protocoles et la mise en œuvre d'une plate-forme d'exécution.

- *Modèle à composants pour la gestion de la duplication et de la cohérence*. Dans notre modèle de gestion de la duplication et de la cohérence, nous considérons de manière séparée les applications, construites sans duplication ni cohérence, et les protocoles qui fournissent la gestion de la duplication et de la cohérence. Cette approche est motivée

1. SARDES = System Architecture for Reflexive Distributed EnvironmentS.

par le principe de la séparation des aspects et vise à faciliter la réutilisation des applications, la réutilisation des protocoles et leur intégration.

Les applications sans duplication ni cohérence sont basées sur un modèle à composants minimal qui permet de spécifier les types et les assemblages des composants. Les protocoles de duplication et de cohérence sont basés sur ce même modèle à composants puisque nous considérons que les protocoles sont eux-mêmes des applications. Toutefois, leur modélisation est indépendante des applications et utilise une spécialisation du modèle afin de ne refléter que les aspects liés à la duplication et à la cohérence.

- *Modèle d'intégration entre applications et protocoles de duplication et de cohérence*. La composition entre les applications sans duplication ni cohérence et les protocoles de duplication et de cohérence est basée sur la logique des systèmes réflexifs. Les protocoles jouent le rôle d'un niveau de contrôle pour les applications et les adaptent afin de répondre à des objectifs de disponibilité.

  Au niveau de l'architecture à l'exécution, nous nous inspirons des architectures EJB et CCM et intégrons les traitements des protocoles dans les composants des applications. Les traitements de duplication et de cohérence sont structurés pour former une couche de gestion, un conteneur, qui encapsule le code métier des composants. Ainsi, en changeant de conteneur, le code métier peut être réutilisé dans différents contextes d'exécution et satisfaire différents besoins en duplication et en cohérence.

- *Mise en œuvre d'une plate-forme d'exécution*. Nous avons réalisé un prototype en Java et avons effectué des expérimentations dans les environnements CCM et EJB. Le prototype Java, nommé FAR, vise à valider l'approche proposée. Il est utilisé pour modéliser plusieurs applications métier, ainsi que plusieurs protocoles de duplication et de cohérence. Ces modélisations sont utilisées pour évaluer l'approche proposée vis à vis des objectifs de configuration et de réutilisation. Quant aux expérimentations dans CCM et EJB, elles démontrent l'applicabilité des principes proposés à des environnements standards.

Plan du document

Ce document est organisé en deux parties :

1. La première partie traite du problème de cette thèse. Elle est composée de quatre chapitres qui fournissent l'état de l'art associé, positionnent notre travail de thèse et spécifient notre proposition. Plus particulièrement, le chapitre 1 présente l'état de l'art concernant les environnements à composants. Le chapitre 2 fournit l'état de l'art sur les travaux relatifs à la duplication et à la cohérence. Le chapitre 3 explique le positionnement de notre travail qui se situe à l'intersection des domaines précédemment présentés. Enfin, le chapitre 4 présente l'approche que nous adoptons pour répondre aux problèmes énoncés.
2. La deuxième partie présente la mise en œuvre de notre proposition et son évaluation. Notamment, les chapitres 5 et 6 portent sur notre prototype FAR en décrivent, respectivement, la gestion des applications sans duplication ni cohérence et la gestion des protocoles de duplication et de cohérence. Les chapitres 7 et 8 discutent de l'applicabilité des principes de FAR dans les environnements CCM et EJB. Le chapitre 9 présente les expérimentations pour l'évaluation de FAR. La partie termine par une conclusion sur nos résultats et sur les perspectives de ce travail.

PREMIÈRE PARTIE

PROBLÉMATIQUE

La problématique de gestion configurable de la duplication et de la cohérence, considérée dans cette thèse, couvre deux aspects. D'une part, elle porte sur les mécanismes de gestion de la duplication et de la cohérence et, d'autre part, elle traite des techniques de gestion configurable de services système.

Cette partie présente l'état de l'art relatif aux deux aspects de notre problématique. Son premier chapitre est consacré à la gestion de services système dans les environnements à composant, alors que son deuxième chapitre considère les travaux portant sur la duplication et la cohérence. Le chapitre sur les composants détaille leurs principes de configuration et met en avant leur manque de considération des aspects de duplication et de cohérence. Le chapitre sur la duplication et la cohérence présente une classification des solutions de gestion existantes et discute leur rigidité.

La présentation de l'état de l'art est suivie par un chapitre qui positionne nos travaux, définis à l'intersection des domaines de gestion de la duplication et de la cohérence et de la gestion de services système à base de composants. La partie termine par un chapitre qui décrit notre proposition de solution aux problèmes présentés précédemment.

Chapitre 1

Gestion système
à base de composants

Après une présentation des motivations et des principes du paradigme composant, ce chapitre se concentre sur quelques travaux représentatifs du domaine. Nous décrivons leurs mécanismes de gestion de services système tout en prêtant une attention particulière aux services de duplication et de cohérence. Nous concluons le chapitre par une synthèse sur l'utilité et l'applicabilité des mécanismes composant pour la gestion configurable de la duplication et de la cohérence.

1.1 Principes de l'approche composant

Le paradigme composant vient en réponse au problème de complexité de gestion des logiciels. Il considère toutes les étapes du cycle de vie des applications et s'intéresse non seulement à leur programmation, mais également à leur administration. Dans la suite, nous présentons les motivations du paradigme composant (Section 1.1.1), son modèle du cycle de vie des applications (Section 1.1.2) et son lien avec l'administration (Sections 1.1.3 et 1.1.4).

1.1.1 Motivations

Popularisée par le paradigme *objet*, la programmation modulaire permettant la réutilisation de code existant a été un premier pas vers la simplification du processus de gestion de logiciels complexes. Elle a permis de considérablement réduire les coûts de développement en se concentrant sur l'organisation du code métier des applications, appelé encore *code fonctionnel*.

Le paradigme composant préserve les avantages de la programmation orientée-objet tout en adressant les activités complexes de déploiement et de contrôle à l'exécution des applications. Il s'intéresse à la gestion du *code non fonctionnel* des applications qui correspond aux services système utilisés par celles-ci comme la persistance, la sécurité ou les transactions. S'intéressant à la configuration du code non fonctionnel en fonction des environnements d'exécution, le paradigme composant vise, en plus de la réutilisation de code métier dans différentes applications, la réutilisation de ce code dans différents contextes d'exécution.

1.1.2 Gestion des applications à base de composants

Pour faciliter la gestion du code fonctionnel et non fonctionnel des logiciels, le paradigme composant considère toutes les étapes de leur cycle de vie. Notamment, il définit des principes de gestion pour leur conception, leur implantation, leur déploiement et leur exécution.

Conception composant

Les entités de base qui constituent les logiciels sont les *composants*. Ce sont des entités indépendantes qui sont des unités d'encapsulation et de réutilisation. Ils encapsulent un état et exhibent un comportement à travers des interfaces qui spécifient les *types* des composant (Figure 1-1.a). Des exemples de composants sont les composants CORBA, définis dans le modèle CCM[1] [102], ou encore les composants EJB[2] [136].

De par leur encapsulation, les composants peuvent être réutilisés dans différentes applications. Ils sont à la base de l'approche de construction d'applications par *composition* selon laquelle les logiciels ne sont plus programmés mais sont composés à partir de composants qui sont des briques de base de code existant. La composition est basée sur l'interconnexion des composants à l'aide de leurs interfaces (Figure 1-1.b).

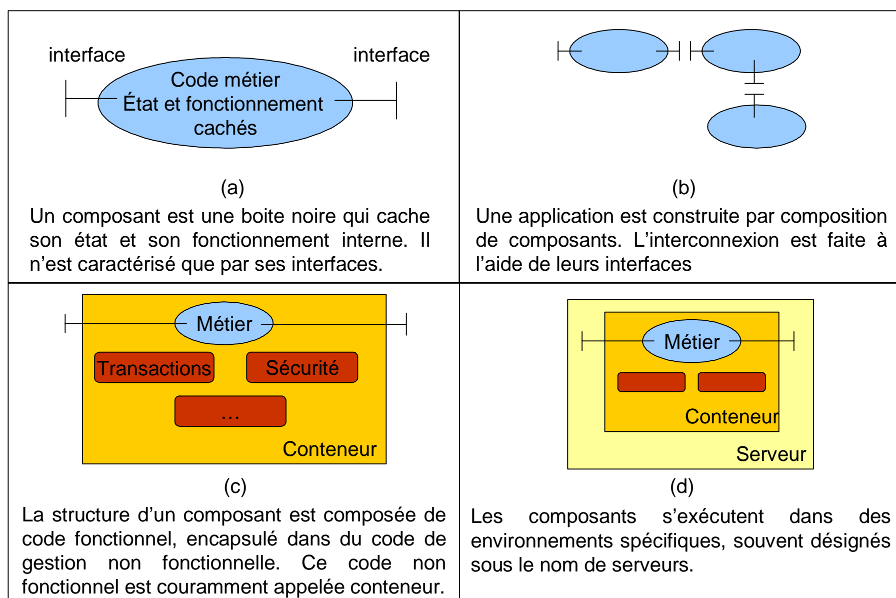


Figure 1-1. Composants, applications et structures à l'exécution.

Implantation composant

La phase d'implantation concerne la programmation de composants et la construction d'applications à base de composants.

La construction d'applications correspond à une phase d'assemblage de composants selon un modèle de composition plat ou hiérarchique. Dans le cas des modèles plats, tels que EJB [136], .NET [95] et UniCon [129], les applications sont basées sur des composants indivisibles, appelés composants primitifs. Dans le cas des modèles hiérarchiques, tels que RM-

1. CCM = CORBA Component Model, le modèle proposé par l'Object Management Group.
2. EJB =Entreprise Java Beans, le modèle proposé par Sun Mycrosystems.

ODP [99], Darwin [82] et DCUP [113], il est possible d'utiliser des composants constitués d'autres composants. Ces composants, dites composites, permettent la construction incrémentale des applications en suivant le principe de décomposition et peuvent être réutilisés dans leur intégrité. Ils permettent, également, la factorisation des traitements de gestion des sous-composants.

L'implantation des composants est prise en charge par le programmeur de composants, ainsi que par le système à composants. Le programmeur fournit le code fonctionnel, alors que le système se charge de la génération du code non fonctionnel. La structure de composant fait coopérer ses deux parties de code en suivant un modèle dans lequel le code non fonctionnel encapsule le code fonctionnel (Figure 1-1.c). Ainsi, le code métier peut être réutilisé non uniquement, comme dans les approches à objets, au sein de différentes applications, mais également dans différents environnements d'exécution.

Les modèles à composants existants instancient de différentes manières le modèle abstrait de structuration. Par exemple, dans EJB ou CCM, le code de gestion non fonctionnelle est appelé *conteneur* et gère les propriétés de persistance, de sécurité et de transactions. Dans RM-ODP, il est appelé *capsule* et a des fonctions de gestion de transactions, de duplication et de sécurité.

Déploiement

Le déploiement d'une application inclut l'installation de ses composants sur des machines d'exécution, sa configuration en fonction de l'environnement concret d'exécution et son lancement. Contrairement à l'approche objet où le code de déploiement fait partie du code des applications et un lancement demande une intervention sur tous les sites d'exécution, dans le monde à composants, le code de déploiement est défini de manière groupée et séparément de l'application. L'interprétation des directives de déploiement est automatisée et est laissée à charge aux environnements d'exécution. Dans EJB et CCM, par exemple, le déploiement est dirigé à l'aide de descripteurs XML[1].

L'unité sur laquelle portent les règles de déploiement est encore le composant. Les règles configurent les conteneurs des composants de manière à ce que le code fonctionnel de ces composants puisse être exécuté dans l'environnement d'exécution considéré.

Exécution

L'exécution des applications à base de composants est liée à deux aspects qui sont, d'une part, le support à l'exécution des instances de composant et, d'autre part, les mécanismes de contrôle du fonctionnement des applications pendant leur exécution.

- *Support d'exécution des instances de composant*. Les instances de composant sont, comme dans le paradigme objet, des entités d'exécution concrètes créées à partir de types de composants. Leur exécution est rendue possible grâce à des environnements qui leur fournissent tous les services dont elles ont besoin. Souvent désignés sous le nom de *serveurs* [136], ces environnements se chargent de fournir des services système dont les indispensables services de nommage et de communication inter-composant, ainsi que des services de persistance, de sécurité, de transactions, etc. Ces services sont accessibles aux conteneurs qui les adaptent aux besoins des composants (Figure 1-1.d).
- *Contrôle des applications à l'exécution*. Les traitements de contrôle à l'exécution, ap-

1. XML = eXtended Markup Language.

pelés traitements d'*administration*, comprennent l'observation et l'éventuelle reconfiguration des applications [40]. Ces reconfigurations assurent le fonctionnement optimal des applications en les adaptant en fonction de variations de l'environnement d'exécution ou en fonction d'évolutions de besoins. Des modèles à composants qui prêtent attention aux aspects d'administration sont par exemple CCM et RM-ODP.

1.1.3 L'approche composant et la séparation des aspects

La conception de logiciels a traditionnellement été faite en utilisant la méthode de décomposition *fonctionnelle* selon laquelle les fonctionnalités des logiciels sont structurées en procédures généralisées et encapsulées dans des modules séparés. Cette organisation facilite l'évolution de fonctionnalités puisqu'elle ne requiert que la modification des modules concernés. Toutefois, l'approche ne considère pas les traitements *non fonctionnels* tels que la gestion de la distribution, l'optimisation des ressources de calcul ou la persistance. Le code de ces traitements, dispersés dans différents modules, est difficile à maintenir et à modifier.

La programmation par aspects (AOP[1]) [78] a récemment été introduite comme solution aux problèmes de gestion des traitements non fonctionnels. Les désignant sous le nom d'*aspects,* l'approche propose leur modularisation au moyen d'une programmation indépendante et séparée. Son objectif principal est d'isoler les traitements dispersés et ainsi de faciliter la manipulation d'un aspect donné. La construction d'une application est ainsi basée sur l'entrelacement (*weaving*) du code des modules indépendants qui implémentent les aspects.

La plupart des projets s'étant intéressés à l'AOP se placent au niveau langage ou au niveau intergiciel. Alors que les approches langage s'intéressent plus à la programmation des aspects, les solutions intergicielles considèrent l'AOP plutôt du point de vue de la composition d'aspects. Ainsi, AspectJ [77] et JAC [108] sont des outils langage qui définissent des constructions de programmation d'aspects sans faire la distinction fonctionnel - non fonctionnel. DART [119] et FlexiNet [57], sont des systèmes intergiciels considérant la composition d'aspects à base de réflexivité [131]. Dans DART, les aspects considérés sont la distribution, la duplication et la persistance, alors que dans FlexiNet ce sont différents types de communication distribuée.

L'approche composant est également une instanciation de l'AOP. En effet, elle met en œuvre une séparation entre le code fonctionnel (code métier) et le code non fonctionnel (services système) d'un composant. Elle permet la programmation séparée en laissant le programmeur se charger de la programmation fonctionnelle et en fournissant des aspects non fonctionnels préprogrammés. Elle *tisse*[2] les aspects en suivant le modèle structurel du conteneur encapsulant le code métier (cf. Figure 1-1.). En permettant l'évolution de différents aspects sans entraîner un redéveloppement complet de toute la structure logicielle, le paradigme composant répond aux objectifs de l'AOP.

La gestion des aspects dans le paradigme composant est soumise à de nombreuses contraintes. Alors que le code fonctionnel peut être modifié en toute liberté, les aspects non fonctionnels sont généralement limités en ce qui concerne leur nombre, leur type et leur composition. La plupart des plates-formes à composants ne considèrent qu'un ensemble fini d'aspects et fixent d'avance leurs règles de composition. Les aspects les plus courants sont la distribution, la persistance, les transactions et la sécurité. Les aspects de duplication et de cohérence, qui sont l'objet de cette thèse, ne sont que rarement considérés.

1. AOP = Aspect-Oriented Programming
2. Le terme *tisser* est utilisé pour désigner les traitements qui organisent le code final d'une application bénéficiant d'aspects programmés séparément.

1.1.4 L'approche composant et l'administration

L'administration est l'activité assurant le fonctionnement optimal des applications dans leurs environnements d'exécution [79]. Elle se charge de l'initialisation des applications au moment du déploiement, ainsi que de leurs évolutions à l'exécution.

L'initialisation d'une application n'inclut pas uniquement la création et le lancement des entités applicatives. Elle concerne également le paramétrage de l'application et des services système utilisés pour refléter au mieux l'environnement d'exécution. Ce paramétrage peut concerner, par exemple, la décision de l'ensemble de machines sur lesquelles l'application va s'exécuter ou la définition d'exigences en termes de qualité de service du réseau.

L'évolution des applications est rendue nécessaire dans des cas d'environnements d'exécutions instables pouvant entraîner des dégradations des performances des applications. L'administration se charge d'observer les applications et de les reconfigurer. Par exemple, dans le cas de panne, les traitements d'administration diffuseront l'information de panne, choisiront une entité opérationnelle de remplacement et redirigeront les communications vers cette entité.

Le modèle général d'administration des applications [132] est représenté sur la Figure 1-2. Il fait apparaître deux niveaux qui correspondent respectivement aux applications et à leur administration. Les applications sont instrumentés pour pouvoir être surveillées et éventuellement reconfigurées. Le niveau d'administration reçoit les informations de surveillance et prend les décisions de contrôle des applications.

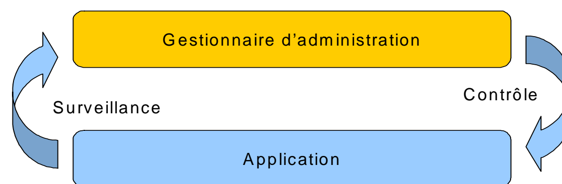


Figure 1-2. Modèle général d'administration d'applications

L'administration est explicitement adressée par le paradigme composant. En ce qui concerne l'initialisation des applications, elle est représentée par les traitements de déploiement qui se chargent de la définition de la configuration initiale d'une application. En ce qui concerne l'évolution des applications à l'exécution, elle est facilitée par la structure des composants. En effet, les performances à l'exécution peuvent être observées grâce à l'encapsulation du code fonctionnel dans les conteneurs. Le contrôle de ces performances peut être fait grâce aux liens que ces conteneurs établissent entre ce code et les services système.

Même si le paradigme composant dispose des mécanismes de base pour gérer les adaptations dynamiques des applications, la complexité de gestion fait que la plupart des travaux existants, comme par exemple EJB, ignorent les problèmes de reconfiguration. De rares projets, tels que DCUP [119] ou META [153], adoptent l'approche contraire qui consiste à ne considérer que les problèmes de reconfiguration.

1.2 Environnements à composants

Cette section présente un tour d'horizon des travaux dans le domaine à composants. Elle décrit les modèles Entreprise JavaBeans (EJB) [136] et CORBA Component Model (CCM)

[102], avant de discuter les propriétés de systèmes réflexifs, des systèmes de configuration et des projets à composants s'intéressant à la duplication et à la cohérence.

Le plan détaillé de notre présentation est le suivant :

- *EJB et CCM*. Dans les sections 1.2.1 et 1.2.2, nous présentons respectivement les modèles EJB, et CCM. Pour chaque modèle nous donnons une brève description générale et décrivons le cycle de vie en prêtant une attention particulière aux aspects liés à l'AOP et à l'administration. Notamment, au niveau de la programmation, nous évaluons le support des aspects, au niveau du déploiement, nous analysons les caractéristiques de paramétrisation et de configuration et, finalement, au niveau de l'exécution, nous présentons les capacités d'observation et de reconfiguration. Dans tous les cas, nous mettons en avant les techniques utilisées ou utiles pour la gestion des deux services considérés dans cette thèse : la duplication et la cohérence.
- *Systèmes réflexifs*. Dans la section 1.2.3, nous présentons les systèmes réflexifs qui s'intéressent à l'intégration de nouveaux aspects dans les applications. L'approche réflexive est, en effet, une solution possible aux problèmes de gestion de la duplication et de la cohérence.
- *Systèmes de configuration*. Dans la section 1.2.4, nous nous intéressons aux techniques de gestion de configuration proposées dans les environnements à composants. Nous analysons l'applicabilité de ces techniques dans le contexte de la duplication et de la cohérence.
- *Systèmes de duplication et de cohérence*. Finalement, dans la section 1.2.5, nous considérons les travaux dans les environnements à composants qui se sont intéressés à la gestion de la duplication et de la cohérence.

1.2.1 EJB

Après une présentation générale de l'architecture de l'environnement EJB, cette section se focalise sur ses mécanismes de gestion d'aspects non fonctionnels et ses techniques de contrôle du déploiement et de l'exécution des applications. Elle discute également les différentes implémentations existant de la spécification.

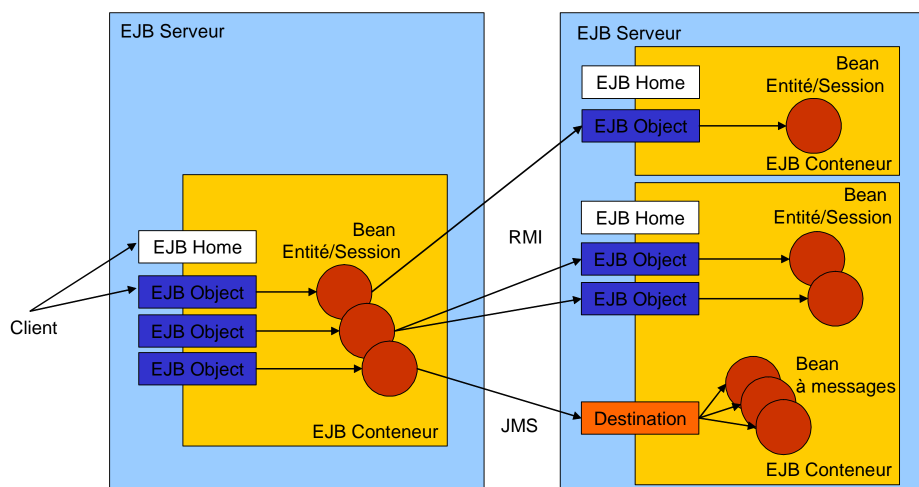


Figure 1-3. Architecture générale du modèle EJB

Présentation générale

Selon le modèle EJB (Figure 1-3.), spécifié par Sun Microsystems [136], une application répartie est constituée de composants appelés *Beans* qui sont implémentés en Java. Les Beans n'ont qu'une seule interface d'entrée et peuvent être de trois types : session, entité et à messages. Au sein des applications, les Beans de type session et entité sont accessibles à travers une communication synchrone basée sur Java RMI [139]. Les Beans à messages, introduits par la dernière version de la spécification, sont un moyen d'intégration d'un service d'événements dans les EJB. Ils communiquent de manière asynchrone en utilisant JMS [137].

L'établissement d'un lien vers un Bean consiste en la récupération de sa référence. Elle peut être passée en paramètre par un autre composant ou obtenue à l'aide d'un service de nommage.

Les instances de Beans s'exécutent dans des *conteneurs EJB* qui sont chargés de leurs aspects non fonctionnels. Les conteneurs peuvent héberger plusieurs instances de plusieurs types de Bean. Ils gèrent les transactions, la sécurité et la persistance selon trois types de configurations prédéfinies correspondant aux trois types de composants. Les conteneurs sont inclus dans des environnements d'exécution, des *serveurs EJB*, qui fournissent les services système nécessaires.

Structure de conteneur et gestion d'aspects non fonctionnels

Selon les types de composants, les conteneurs ont les configurations suivantes. Pour les Beans *session* qui représentant un dialogue avec des clients et qui sont non persistants et non concurrents, les conteneurs ne gèrent que l'état de ce dialogue. Pour les Beans *à messages,* définis en tant que composants sans état, pouvant être partagés entre clients, les conteneurs gèrent la concurrence. Pour les Beans *entité,* représentant des données persistantes, les conteneurs gèrent la concurrence et la persistance. Les conteneurs fournissent une gestion de la persistance par défaut (CMP[1]), mais offrent aux Beans la possibilité de la gérer eux-mêmes (BMP). Pour tous les types de Beans, les conteneurs fournissent les transactions et la sécurité.

Le conteneur est constitué d'objets d'interposition qui interceptent les invocations sur les Beans et qui gèrent des propriétés non fonctionnelles au moyen de traitements avant et après ces invocations. Les prétraitements peuvent inclure la vérification des droits d'accès qui est propre à la gestion de sécurité, le démarrage des transactions ou la propagation des contextes transactionnels. Les post traitements se chargent de l'enregistrement éventuel des beans sur support persistant et de la terminaison des transactions démarrées par un prétraitement.

Dans le cas de composants entité ou session, les conteneurs fournissent deux objets : un *EJB Home* et un *EJB Object*. Les EJB Home sont des usines de composants qui permettent la création, la recherche et la destruction d'instances. Ils sont rattachés à un type de composant. Les EJB Object sont des objets d'interposition qui sont rattachés aux instances d'un type de composant et représentent le seul moyen pour accéder aux composants encapsulés à l'intérieur des conteneurs. Ce sont eux qui contiennent, en plus des fonctions d'emballage et de déballage de messages réseau, les pré et post traitements nécessaires à la gestion des aspects non fonctionnels.

Dans le cas de composants à messages, dont les instances n'ont pas d'état ni d'identité, les conteneurs fournissent un objet d'interposition qui définit un objet de destination. Les messages des clients sont adressés à cet objet qui délègue au conteneur l'envoi d'un accusé de réception et la transmission des messages à une des instances des composants à messages.

---

1. CMP et BMP sont respectivement des abréviations pour Container-Managed Persistence et Bean-Managed Persistence

L'environnement EJB propose une gestion d'aspects non fonctionnels qui suit les principes de la séparation des aspects. Cependant, il ne prévoit la gestion que des aspects non fonctionnels déjà mentionnés : persistance, transactions, sécurité et concurrence. La spécification définit de manière précise les liens entre ces aspects, la manière dont ils sont intégrés dans les conteneurs et l'ordre de leur traitement.

Contrôle au déploiement et à l'exécution

Les EJB prévoient l'utilisation de descripteurs de déploiement pour la génération automatique des conteneurs. Ces descripteurs contiennent des informations sur le code fonctionnel, ainsi que sur le code non fonctionnel des composants. En ce qui concerne les fonctionnalités métier des composants, ils déclarent leurs classes d'implémentation, ainsi que leurs interfaces d'accès à distance. En ce qui concerne les aspects non fonctionnels, les descripteurs définissent le type de composant (session, entité, à messages) et donnent des précisions sur la gestion des services système utilisés.

La dernière version de la spécification introduit également la possibilité de décrire, en plus des informations de gestion de composants, des informations relatives aux assemblages des applications. Le déployeur peut définir les dépendances entre Beans en spécifiant les types de Beans qui seront référencés par un Bean donné.

L'environnement EJB ne fournit aucune spécification concernant l'observation et le contrôle d'une application pendant son exécution. Il n'est pas possible d'obtenir de vision globale de l'application, ni d'informations sur l'état d'exécution d'un composant. Une solution possible est d'utiliser une autre spécification fournie par Sun Microsystems qui est l'extension de la plateforme Java pour l'administration (JMX) [138]. Elle permet la programmation d'objets de contrôle pour les besoins applicatifs spécifiques. Dans le cas des EJB, ces objets pourraient faire partie des conteneurs pour fournir des fonctionnalités de gestion additionnelles. Cependant, JMX se place au niveau applicatif et il n'existe pas de spécification de son intégration dans les EJB au niveau conteneur.

Implémentations existantes

La liste des projets implémentant la spécification EJB est longue [41]. Elle inclut aussi bien des travaux en source libre que des produits commerciaux. Parmi les projets en source libre les plus utilisés sont JOnAS [69] et JBoss [67]. Les fournisseurs principaux d'implémentations commerciales sont BEA avec BEA WebLogic [12], IBM avec WebSphere [63] et Oracle avec Oracle9iAS [105].

Une grande majorité des projets existants fournissent, en plus des caractéristiques requises par la spécification EJB, des services additionnels. En font partie des services génériques d'administration ainsi que des solutions de gestion de la duplication et de la cohérence. En ce qui concerne la gestion de l'administration, elle est abordée dans des projets comme WebSphere, JBoss ou BEA WebLogic et est basée sur des outils tels que JMX [138] ou SNMP [22]. Toutefois, elle ne s'adresse qu'aux aspects non fonctionnels de base : persistance, sécurité et transactions.

En ce qui concerne la gestion de la duplication et de la cohérence, de nombreux projets la considèrent sous l'ongle de la répartition de charge et de la tolérance aux fautes. Ils proposent des extensions de gestion de *clusters*, des groupes de copies pour un Bean, intégrés à l'aide d'objets EJB Home et EJB Object spéciaux. Toutefois, outre le fait que les solutions proposées ne font pas partie de la spécification et sont donc fournies de manière ad hoc, la gestion de la

duplication et de la cohérence est figée et le développeur d'application ne peut que choisir parmi un nombre limité de protocoles.

Synthèse

La spécification EJB ne considère pas la duplication et la cohérence. Toutefois, elle dispose d'un ensemble de caractéristiques intéressantes comme la séparation des aspects et la gestion de certaines informations globales concernant les applications. Le principe de séparation des aspects est applicable pour les besoins d'une gestion flexible de la duplication et de la cohérence. Quand aux informations d'assemblage disponibles, elles sont utiles du point de vue de l'administration d'une application dont la duplication et la cohérence font partie. L'intégration des aspects de duplication et cohérence peut s'appuyer sur ces éléments mais elle nécessite en plus des moyens d'expression, de génération et de configuration des traitements reliés. Elle pourrait s'inspirer des solutions ad hoc des implémentations EJB existants mais devrait les étendre pour répondre aux besoins de gestion flexible et de construction facile de nouvelles solutions de duplication et de cohérence.

1.2.2 CCM

Spécifié par l'Object Management Group en 2002, CCM [102] est un modèle jeune qui n'a pas l'étendue d'utilisation des EJB ou de sa plate-forme de base CORBA. Toutefois, il a l'avantage de considérer des environnements d'exécution hétérogènes et ses caractéristiques reflètent les dernières recherches autour des composants.

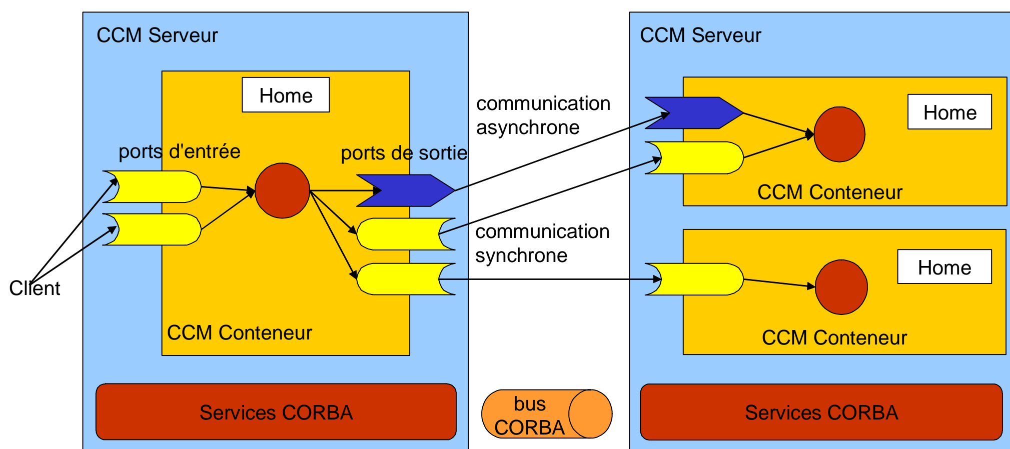


Figure 1-4. Architecture générale du modèle CCM

Présentation générale

Les composants CCM sont définis par un ensemble d'attributs et par leurs interfaces fournies et requises, appelés respectivement *ports d'entrée* et *ports de sortie*. Les attributs de composants sont utilisés pour la configuration au déploiement ou à l'exécution. Les ports définissent les points de communication des composants et sont utilisés pour leur invocation et leur interconnexion. En effet, il n'est pas possible d'appeler un composant sans référencer un de ses ports d'entrée.

Pour que les composants d'une application puissent communiquer, ils doivent être interconnectés à l'aide de leurs ports. Les connexions sont établies entre des ports de sortie et des ports

d'entrée qui représentent des interfaces identiques et qui sont de la même nature synchrone ou asynchrone. En d'autres termes, les ports ne peuvent pas être connectés si l'interface qui est requise par le composant avec le port de sortie n'est pas fournie par le composant avec le port d'entrée. L'acheminement des communications est pris en charge par le bus CORBA sous-jacent.

CCM définit les types de composants, ainsi que leurs implantations, à l'aide de langages déclaratifs. Les types de composants sont décrits en utilisant une extension de l'IDL[1] de CORBA [101] ce qui permet la programmation des composants en différents langages et leur exécution dans des environnements hétérogènes. Quant aux implantations des composants, elles sont décrites à l'aide du langage de description CIDL[2]. Contenant des détails sur la structure logicielle et sur la gestion non fonctionnelle des composants, les descriptions CIDL sont utilisées pour la génération des leurs *conteneurs*. Comme dans les EJB, ces conteneurs s'exécutent dans de *serveurs* qui leur fournissent les services système nécessaires. Étant donné que CCM est défini comme une couche au-dessus de CORBA, les services système proposés sont les services de transactions, de sécurité, de tolérance aux fautes, etc. de CORBA (Figure 1-4.).

Structure de conteneur et gestion d'aspects non fonctionnels

Contrairement aux conteneurs EJB, un conteneur CCM ne gère les instances que d'un type de composant. Les conteneurs contiennent des usines de composants (des *Home)* qui se chargent de la création et de la destruction des instances de composant. Ils disposent également d'objets d'interposition pour les interfaces fournies et requises des instances de composant.

Les conteneurs proposent deux interfaces particulières de gestion des objets d'interposition : une d'introspection et une de gestion de ports. L'interface d'introspection fournit des informations sur le type de composant, sur ses interfaces fournies et requises, ainsi que sur l'état de ses connexions à d'autres composants. L'interface de gestion de ports est utilisée pour établir ou détruire des connexions entre composants. En effet, les interconnexions entre composants sont gérées à l'aide de primitives explicites et ne sont pas établies par simple passage de référence comme dans le cas des EJB.

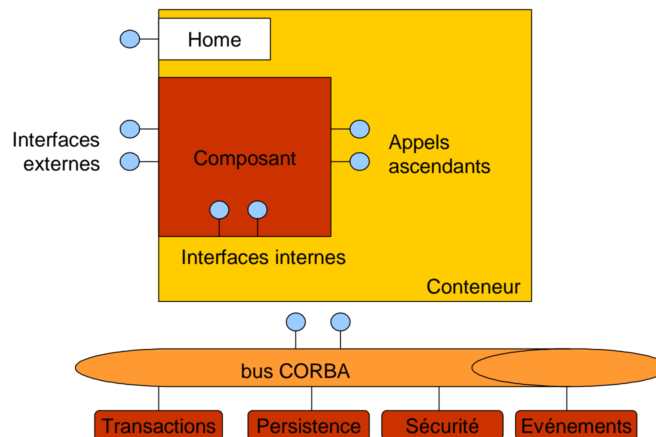


Figure 1-5. Structure d'un conteneur CCM

Les interfaces d'introspection et de gestion de ports font partie des interfaces *externes* des conteneurs (Figure 1-5.) qui incluent également les interfaces fonctionnelles des composants,

1. IDL = Interface Definition Language
2. CIDL = Component Implémentation Definition Language

ainsi que l'interface de l'objet Home. En plus des interfaces externes, les conteneurs disposent d'interfaces *internes* et d'*appels ascendants*. Les interfaces internes sont fournies par les conteneurs et utilisées par les composants pour paramétrer la gestion des services système CORBA. Les interfaces d'appels ascendants sont fournies par les composants et utilisées par les conteneurs et interviennent dans les cas où la gestion des services système est à la charge des composants. Ces cas sont analogues à la persistance gérée par les Beans (BMP) dans les EJB.

La spécification CCM définit quatre instanciations de la structure générale des conteneurs. Ce sont quatre configurations prédéfinies de gestion d'aspects non fonctionnels correspondant à quatre catégories de composant : service, session, processus et entité. Comme dans le cas des EJB, la gestion des aspects non fonctionnels inclut la persistance, les transactions et la sécurité, avec possibilité de gestion de la persistance par le conteneur ou par le composant. Dans le cas des composants *service*, qui sont des composants sans état, et des composants *session*, qui ont un état non persistant, les conteneurs ne fournissent que les transactions et la sécurité. Pour les besoins des états persistants des composants *processus* et *entité* dont la différence réside dans l'utilisation de clés primaires pour l'identification des instances *entité*, les conteneurs se chargent, en plus des transactions et de la sécurité, de la gestion de la persistance.

Contrôle au déploiement et à l'exécution

CCM suit l'exemple des EJB et fournit des descripteurs de déploiement. Toutefois, ces descripteurs concernent non seulement les composants isolés, mais également les assemblages de composants pour former des applications. En ce qui concerne les descripteurs de composants, ils décrivent leur structure logicielle, leur type en termes d'interfaces fournies et requises, ainsi que leur catégorie (processus, entité, session, service). Ces descripteurs sont générés à la compilation de la description des composants[1], mais peuvent être modifiés pour paramétrer la gestion des aspects non fonctionnels. Quant aux descripteurs d'assemblages, ils décrivent les architectures initiales des applications. Ils spécifient les instances de composants constituant une application, définissent des règles de placement sur des sites d'exécution et donnent les connexions à établir entre instances.

CCM ne gère pas explicitement le contrôle des applications à l'exécution. La plate-forme ne définit pas de gestionnaire global qui peut fournir des renseignements sur les paramètres d'exécution et sur les évolutions des applications par rapport à leur définition initiale dans les descripteurs. Elle ne fournit pas non plus de gestion spécifique de reconfiguration. Toutefois, les interfaces d'introspection et de gestion de ports sont des briques de base permettant l'observation et le contrôle à l'exécution de manière partielle. Par introspection, il est possible de naviguer et de connaître les connexions entre composants. Par l'interface de gestion de ports il est possible de reconfigurer les liens entre composants.

Implémentations existantes

Les implémentations de CCM ne sont que peu nombreuses [32]. Elles comptent à ce jour plusieurs implémentations partielles en source libre comme OpenCCM [104], EJCCM [42] ou MicoCCM [146] et un premier produit commercial de iCMG [62]. Alors que les implémentations non commerciales ne se concentrent que sur la spécification CCM, le produit proposé par iCMG considère également plusieurs aspects d'administration. Il fournit des outils d'observation de l'environnement d'exécution, des composants et des ressources système et permet la

1. Il s'agit de la description CIDL qui donne des informations sur la composition du code fonctionnel et non fonctionnel des composants.

définition et le chargement dynamique de scripts décrivant des traitements de gestion. Il utilise ses capacités d'administration pour gérer la tolérance aux fautes et la répartition de charge.

Synthèse

Comme le modèle EJB, CCM ne considère qu'un nombre limité d'aspects non fonctionnels qui n'incluent pas la duplication, ni la cohérence. Néanmoins, il fournit plus de facilités pour leur intégration que les EJB. Notamment, en plus d'une architecture à base de conteneurs qui reflète le principe de séparation des aspects, il fournit de nombreuses informations sur l'architecture globale des applications. Il permet sa définition initiale dans des descripteurs de déploiement ainsi que son observation et contrôle à l'exécution. Toutes ces caractéristiques font que CCM fournit les outils de base pour l'intégration, dans les conteneurs, des règles de contrôle de l'exécution des applications que sont la duplication et la cohérence.

1.2.3 Systèmes réflexifs

L'approche réflexive [131] fournit une réponse à la complexité des applications en proposant des techniques de définition et d'intégration de nouveaux aspects au sein des ces dernières. Elle fait apparaître deux niveaux : le niveau de base et le niveau *méta* (Figure 1-6.). Le niveau de base correspond aux fonctionnalités déjà existantes, alors que le niveau méta rajoute des traitements de gestion additionnels.

Les opérations assurant le lien entre les deux niveaux sont la *réification* et la *réflexion*. La réification construit une représentation du niveau de base qui est manipulable au niveau méta. La réflexion permet de répercuter sur le niveau de base les modifications effectuées sur sa représentation réifiée.

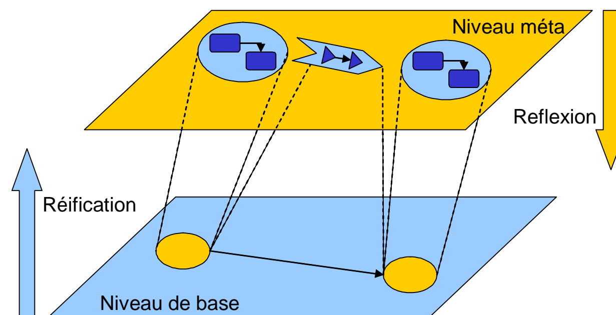


Figure 1-6. Architecture d'un système réflexif

La programmation des objets et des méta-objets étant indépendante, la réflexivité s'est imposée comme une des techniques principales pour l'AOP (cf. Section 1.1.3, p. 26). Elle permet l'intégration et la gestion de propriétés non fonctionnelles en utilisant les méta-objets mettant en œuvre ces propriétés. Par exemple, pour introduire la gestion de la concurrence ou de la sécurité, il suffit que les méta-objets effectuent les traitements de gestion appropriés entre la réification et la réflexion des appels vers les objets de base.

La composition entre objets de base et méta-objets, ainsi qu'entre plusieurs méta-objets, peut être faite de différentes manières. Des projets comme les JavaPods [18] choisissent une logique linéaire. Les méta-objets forment une liste et c'est l'ordre dans cette liste qui détermine l'ordre d'exécution des traitements qu'ils fournissent. Dans d'autres cas, comme dans DART [119] ou les travaux de Kon et al. [80], la composition est plus complexe et est faite selon une

structure de graphe. Dans Blair et al., ce graphe distingue des méta-objets pour la manipulation de la structure des objets de base, de leurs interfaces et des messages de communication.

La composition d'aspects est l'objectif principal des projets réflexifs. Elle est considérée à un niveau générique qui ne fait pas de distinction entre traitements fonctionnels et non fonctionnels. Les aspects non-fonctionnels sont souvent considérés en guise d'exemple des capacités des plates-formes proposées. Il s'agit le plus souvent de la distribution, de la sécurité ou de la persistance, gérés sous forme de post et pré traitements des appels des objets de base. Les travaux s'intéressant à la duplication et à la cohérence, tels que Guarana [100] ou DART [119], fournissent généralement des schémas de communication de groupe[1] et ne définissent pas de modèle d'organisation interne des méta-objets correspondants.

1.2.4 Systèmes de gestion de configurations

Une configuration caractérise une application à un moment de son exécution. Elle reflète la structure interne de l'application, ainsi que son utilisation de ressources et de services système. Dans le cadre composant, il s'agit respectivement de l'architecture applicative interconnectant des composants et de la gestion des propriétés non fonctionnelles telles que la persistance ou la distribution.

La définition d'une configuration initiale est faite au cours de la conception et pendant le déploiement. Elle a pour but de paramétrer une application en fonction de son environnement d'exécution afin de garantir de performances optimales. Si un autre environnement de déploiement est considéré ou si l'environnement d'exécution courant est sujet à des variations, l'optimalité d'exécution des applications est obtenue au moyen de *reconfigurations*.

Dans les travaux de recherche, la gestion de configurations est abordée de deux manières : en définissant des canevas intergiciels de gestion ou en se basant sur des langages de description d'architecture (ADL[2]).

Canevas intergiciels de gestion de configurations

Des canevas de gestion des configurations sont proposés, entre autres, dans le projet DCUP [113] et dans les travaux sur la reconfiguration dans CORBA de Paulo et al. [107]. Dans DCUP, la gestion de configurations consiste à remplacer, pendant l'exécution, des parties prédéfinies dans la structure des composants. Le protocole de reconfiguration est basé sur une architecture de gestionnaires, responsables de l'arrêt des composants, du blocage des communications entrantes, de la reconfiguration effective des composants (remplacement) et du relancement des composants reconfigurés. Le canevas de reconfiguration dans Paulo et al. utilise le même principe de gestionnaires qui agissent sur l'état d'exécution des composants et qui contrôlent leurs échanges. Il définit un service de reconfiguration dynamique pour les applications CORBA et permet la destruction, le remplacement et la migration d'objets.

Langages de description d'architectures

L'approche basée sur les ADL fournit une modélisation explicite de l'architecture des applications [130] ce qui rend les éléments applicatifs visibles et manipulables pour les besoins de

1. La communication de groupe est discutée plus en détail dans le chapitre consacré aux travaux relatifs à la duplication et à la cohérence.
2. ADL = Architecture Description Language

gestion de configurations. Des représentants de cette approche sont Darwin [82], Wright [4] et Rapide [91].

- *Darwin*. Dans Darwin, une configuration décrit les composants d'une application, leurs communications et leur contrôle. Le contrôle est un traitement de gestion de configurations basé sur des opérations de suppression et d'ajout de composants, ainsi que d'interconnexions entre composants.
- *Wright*. Dans le cas de Wright, les éléments architecturaux sont les composants ainsi que leurs interconnexions, modélisées sous forme d'entités indépendantes appelées connecteurs. L'objectif de Wright étant la vérification de propriétés de correction, il utilise comme outil linguistique de modélisation un modèle algébrique de processus. La gestion de configurations est modélisée en termes d'échanges de messages et s'appuie sur des messages de commande particuliers qui concernent, comme dans Darwin, l'instanciation et l'interconnexion de composants.
- *Rapide*. Dans Rapide, la modélisation des architectures est encore différente. Comme dans CCM, il manipule des composants ayant des interfaces requises et fournies, synchrones ou asynchrones.

La liste des ADLs est longue [64][93]. Les langages se différencient par les propriétés qu'ils attachent aux composants, par leurs moyens de modélisation et par les actions de reconfiguration qu'ils considèrent. Ils permettent l'expression, la simulation et la vérification de propriétés applicatives complexes comme la compatibilité d'interconnexion de composants ou l'absence d'interblocages. Toutefois, les ADLs ne sont que des langages : ils restent au niveau modélisation et ne considèrent pas l'instanciation des architectures en termes d'applications d'exécution.

La gestion de configurations est étroitement liée à la gestion de la duplication et de la cohérence. La gestion de configurations est responsable du paramétrage des services système et de l'adaptation des applications en fonction de l'environnement d'exécution. La gestion de la duplication et de la cohérence est, quant à elle, un service système particulier qui définit des règles d'exécution optimale dans des environnements à problèmes de disponibilité. Toutefois, les solutions d'administration définissent des mécanismes génériques de gestion de configurations et ne spécifient pas leur utilisation dans le cadre de la duplication et de la cohérence.

1.2.5 Systèmes de duplication et de cohérence

Dans leur majorité, les projets qui s'intéressent explicitement à la gestion de la duplication et de la cohérence, définissent des modèles à objets étendus. Ce sont des modèles à objets plus que des modèles à composants puisqu'ils ne considèrent pas les aspects d'assemblage, de déploiement et de contrôle à l'exécution. Ils sont étendus puisqu'ils introduisent, en plus des objets “normaux” des structures de contrôle spécifiques à la duplication et à la cohérence. D'une manière générale, ces structures concernent le nommage et la gestion des communications entre copies et par conséquent la distribution. Deux exemples de travaux marquant dans le domaine sont Globe [135] et GARF [51].

Globe

Globe [135] est un projet visant la construction d'applications répondant aux besoins de mise à l'échelle. Son modèle à objets fragmentés définit les entités applicatives en tant qu'objets répartis, ayant des représentants locaux sur différents sites. Ces représentants sont structurés en quatre sous-objets qui font apparaître de manière explicite quatre types de traitements. Notamment, il s'agit des traitements spécifiques à l'application, à la communication, à la duplication

et à la cohérence, ainsi qu'aux relations (contrôle) entre ces trois aspects. Les objets qui leur correspondent sont appelés respectivement objet sémantique, de communication, de duplication et de contrôle (Figure 1-7.).

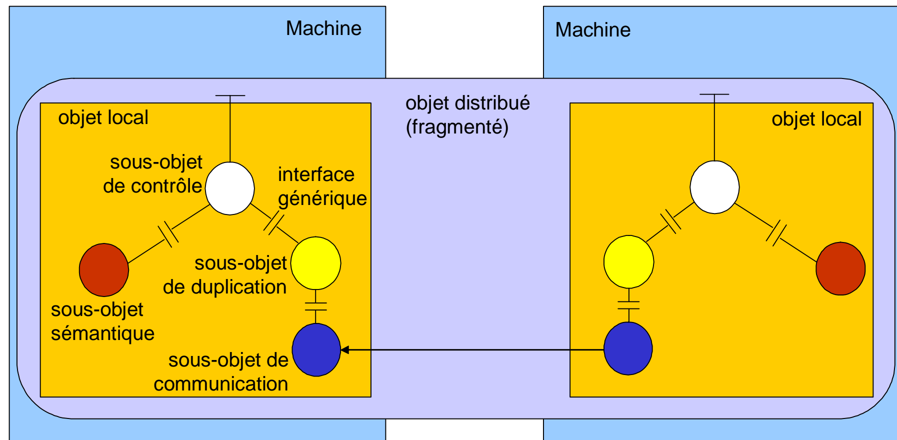


Figure 1-7. Modèle de Globe

Les opérations de gestion de la cohérence, encapsulées dans le sous-objet de duplication, sont appelées par le sous-objet de contrôle, avant et après les traitements applicatifs. Ces opérations sont définies par une interface générique et peuvent être implémentées pour satisfaire la sémantique spécifique de différents objets applicatifs, tels les documents web [110]. Globe propose actuellement deux schémas de fonctionnement, notamment un schéma sans duplication ni cohérence et un schéma maître-esclave avec gestion de cohérence pessimiste [5].

GARF

GARF [51] est un projet s'intéressant aux applications réparties tolérantes aux fautes. Le modèle proposé fait apparaître des objets données et des objets comportementaux. Les objets données décrivent les fonctionnalités de l'application dans un contexte centralisé, sans concurrence, sans persistance et sans pannes. Ces trois aspects sont pris en charge par les objets comportementaux. Dans ce sens, GARF se rapproche fortement au modèle à composants basé sur la notion de conteneur responsable de la gestion des aspects non fonctionnels. D'autant plus que les objets comportementaux sont en effet des *encapsulateurs* contrôlant les échanges des objets en utilisant des objets d'interposition appelés *messagers* (Figure 1-8.).

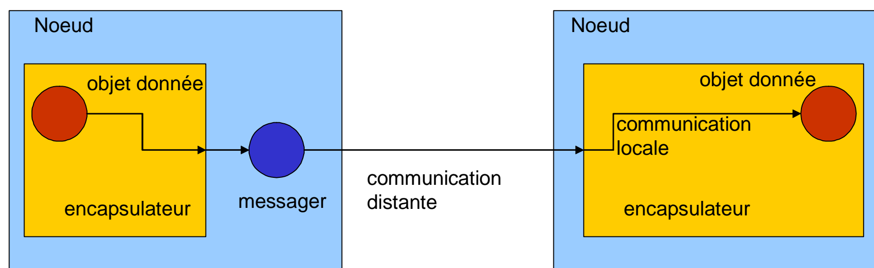


Figure 1-8. Modèle de GARF

Comme dans Globe, les encapsulateurs de GARF gèrent la duplication et la cohérence sous forme de pré et de post traitements d'un appel de méthode. Les échanges entre replicas sont assurés par les messagers qui peuvent intégrer différentes primitives de communication comme,

par exemple, une primitive de diffusion. GARF propose deux schémas de communication qui correspondent aux deux stratégies de duplication. La première est une duplication active où tous les replicas traitent les modifications à effectuer. La deuxième est une duplication passive où les modifications sont traitées par des entités choisies et ne sont répercutées qu'ultérieurement aux autres replicas.

1.2.6 Synthèse

Le paradigme composant s'impose de plus en plus dans le domaine de construction de logiciels grâce à sa vision complète du leur cycle de vie. Il s'intéresse non seulement à la facilité de programmation mais considère également les aspects d'assemblage, de déploiement et de contrôle à l'exécution. L'approche assure ainsi la gestion du code fonctionnel ainsi que du code non fonctionnel des applications qui correspondent respectivement à leur code métier et aux services système nécessaires pour leur bonne exécution.

Le paradigme composant considère la réutilisation des modules logiciels non seulement par rapport à leur logique fonctionnelle mais également par rapport à leur configuration non fonctionnelle. Pour permettre la réutilisation de fonctionnalités dans le cadre de différentes applications, il suit le principe de modularisation et d'encapsulation. Pour permettre la réutilisation de ces fonctionnalités dans différents contextes d'exécution, il adhère aux principes de gestion séparée des aspects non fonctionnels. Il définit des entités spéciales, des conteneurs, qui encapsulent le code métier des composants et qui leur fournissent une configuration spécifique des service système utilisés en fonction de leur environnement d'exécution.

L'idée de séparation des aspects système des aspects fonctionnels des applications répond bien aux objectifs de gestion configurable de la duplication et de la cohérence. Toutefois, dans une grande majorité des travaux autour des composants, tels que CCM et EJB, la duplication et la cohérence ne sont pas considérées. Les principes de séparation ne sont appliqués qu'à un ensemble limité et prédéfini d'aspects et les configurations des conteneurs sont fixées (conteneurs non extensibles).

Le problème de conteneur non extensible est adressé de manière générique dans des travaux autour de la réflexivité et de la gestion de configurations. Les systèmes réflexifs, tels que DART ou JavaPod, proposent des solutions d'intégration de nouveaux aspects au sein d'une application mais sans aborder la duplication ni la cohérence. De même, les travaux sur la gestion de configurations, qu'il s'agisse de langages de description d'architectures comme Darwin et Rapide, ou d'intergiciels comme DCUP, proposent des mécanismes génériques qui ne sont pas appliqués dans le cadre de la duplication et de la cohérence.

Finalement, il existe des projets, tels que GARF et Globe, où la duplication et la cohérence forment le centre d'intérêt. Cependant, ils définissent des modèles à objets étendus qui se concentrent uniquement sur l'étape de programmation des applications et ne considèrent pas la configuration au déploiement ni la gestion à l'exécution.

Chapitre 2

Duplication et cohérence

Ce chapitre commence par une définition des termes de duplication et de cohérence (Section 2.1), et par une présentation des principales applications de la duplication et de la cohérence (Section 2.2). Il continue avec une classification générale des travaux dans le domaine (Section 2.3) en prêtant une attention particulière aux efforts de gestion configurable (Section 2.4). Le chapitre termine par une synthèse sur les besoins en termes de configurabilité de la duplication et de la cohérence.

2.1 Définitions

Les termes de duplication et de cohérence ont des significations différentes selon leurs domaines d'application et sont souvent employés de manière confuse. La distinction, par exemple, entre "duplication active" et "duplication passive" [126][112] ne porte pas sur le processus de duplication mais sur la gestion de la cohérence entre copies. L'appellation s'adresse, en effet, non aux copies mais aux échanges entre elles.

Nous définissons les termes de duplication et de cohérence de la manière suivante.

- Duplication

La duplication est le processus de création et de placement de copies d'entités logicielles. Les entités dupliquées peuvent être des données, du code ou les deux à la fois. Des exemples sont respectivement les fichiers [8][59][89][125], les tâches de calcul scientifique [81] et les objets [9][47][56][92][135].

La création de copies consiste à reproduire la structure et l'état des entités dupliquées. La copie d'un fichier est un autre fichier de même contenu. La copie d'un programme est un autre programme qui exécute le même code et dont l'état d'exécution est celui du programme initial.

Le placement des copies consiste à choisir, en fonction des objectifs de duplication, un environnement d'exécution pour les copies. Par exemple, le placement dans un environnement accessible localement assure le travail en mode déconnecté [109] alors que le placement sur plusieurs machines permet la distribution de la charge [87].

Le processus de duplication peut être décomposé en plusieurs opérations. La mise en œuvre de ces opérations est définie dans un *protocole de duplication.*

- Cohérence

La cohérence est une relation qui définit le degré de similitude entre les copies d'une entité dupliquée. Dans le cas idéal, cette relation caractérise des copies qui ont des comportements

identiques. Dans les cas réels, où les copies évoluent de manière différente, la cohérence définit les limites de divergence autorisées entre copies.

La relation de cohérence est assurée par *synchronisation* entre copies. Considérant que les copies d'une entité se partagent son état, la synchronisation définit des règles d'accès à cet état. Elle peut être vue comme une définition indirecte de la cohérence où cette dernière n'est plus définie en termes de différences entre copies mais en termes de règles de contrôle de ces différences [71][90][148].

Le terme *protocole de cohérence* désigne l'ensemble d'opérations nécessaires à maintenir la cohérence entre copies.

2.2 Applications de la duplication et de la cohérence

La duplication et la cohérence sont largement utilisées pour répondre aux problèmes issus de la distribution : elles améliorent les performances d'accès, assurent la tolérance aux pannes et facilitent le passage à l'échelle.

- *Amélioration des performance d'accès***.** L'implication des réseaux dans les interactions distribuées introduit une différence entre les accès locaux et les accès distribués pouvant atteindre plusieurs ordres de grandeur. Les performances d'accès peuvent être améliorés par duplication des services sur des machines ayant des connexions réseau de bonne qualité. Cette technique est typiquement utilisée dans les systèmes de cache [19][36][73][147] et dans les environnements à usagers mobiles. Dans le cas de ces derniers, la duplication est guidée par les déplacements des usagers et leur fournit un accès continu à leurs environnements habituels de travail [76].
- *Tolérance aux pannes*. La tolérance aux pannes est une des utilisations principales de la duplication et de la cohérence. Les pannes de machines sont tolérées en copiant un service sur plusieurs machines [52][96][141], alors que les pannes de connexion sont gérées en copiant les services sur des machines accessibles par des chemins différents [72]. Les pannes de connexions sont également gérées dans les environnements à usagers mobiles où la duplication et la cohérence permettent le travail en mode déconnecté [60][61][70][75][144].
- *Passage à l'échelle*. La mise à l'échelle aggrave tous les problèmes issus de la distribution [97]. Les environnements à grande échelle sont plus vulnérables aux pannes, subissent un trafic qui sature les canaux de communication et doivent faire face à un nombre de requêtes qui dépasse la capacité de traitement des services. La duplication de services permet de placer des copies partout dans les grandes infrastructures [121], de diminuer ainsi le trafic [1] et de répartir la charge en aiguillonnant les clients vers les copies disposant de plus de ressources [3] .

La tolérance aux pannes, l'amélioration des performances d'accès ou encore la mise à l'échelle sont nécessaires dans de nombreux domaines. Les protocoles de duplication et de cohérence trouvent ainsi des applications dans les systèmes de fichiers, les bases de données, les mémoires partagées réparties, les processus distribués, etc.

- *Systèmes de fichiers*. Les systèmes de fichiers sont les structures fondamentales pour l'organisation des informations. Ils utilisent la duplication et la cohérence pour assurer la pérennité des informations (tolérance aux pannes), pour garantir la performance d'accès [96][141], ainsi que pour mettre en place des solutions pour des usagers

mobiles [45][83][123]. Leur application principale est le stockage des données personnelles : documents écrits, documents multimédia, courriel, etc.

- *Bases de données*. Comme les systèmes de fichiers, les bases de données gèrent le stockage, l'organisation et la manipulation de données. Leurs principales préoccupations en ce qui concerne la duplication et la cohérence sont également la tolérance aux pannes, l'amélioration de performances [74][150] et la prise en charge des usagers mobiles [143]. Omniprésentes, elles ont leur place dans le secteur financier, dans les entreprises de service, dans les offres touristiques, etc.
- *Mémoires partagées réparties*. Fournissant l'abstraction d'une mémoire centralisée, le défi pour les mémoires partagées réparties est d'assurer la performance d'accès. Se focalisant sur la gestion de caches, les protocoles de duplication et de cohérence portent surtout sur les mécanismes permettant la réduction du nombre de messages de synchronisation [20]. Typiquement utilisées pour les calculs scientifiques parallèles [152], les mémoires partagées réparties trouvent également leur place pour la gestion de caches web [23].
- *Processus distribués*. Les processus distribués ne considèrent pas les services du point de vue de leurs données, mais les représentent en termes de calculs composés de suites d'actions. Leur utilisation de la duplication et de la cohérence vise principalement la tolérance aux pannes. Les travaux existants se concentrent sur les mécanismes de gestion de groupes de processus et sur la gestion de la cohérence par diffusion ordonnée de messages [28][54]. Les processus distribués sont un modèle de base des exécutions réparties et s'appliquent par conséquent à tout genre d'applications.

2.3 Classification de la duplication et de la cohérence

Les techniques de gestion de la duplication et de la cohérence sont tellement nombreuses qu'il n'existe pas de classification générale. La plupart des classifications existantes se focalisent sur des domaines d'application particuliers comme les mémoires partagées réparties [2][71], les processus coopérants distribuées [17][31], ou encore les bases de données [25][150]. Les efforts de comparaison de solutions provenant de différents domaines sont rares et se concentrent sur un sous-ensemble de caractéristiques [29][90][122][151].

Nous avons décidé de présenter les travaux du domaine selon les deux axes que sont la duplication et la cohérence. Pour chaque axe, nous présentons les critères de classification que nous avons retenus, ainsi que les travaux qui y adhèrent. La section 2.3.1 est dédiée à la duplication, alors que la section 2.3.2 traite les travaux relatifs à la cohérence.

2.3.1 Classification des travaux relatifs à la duplication

Le critères de classification que nous avons retenus pour la duplication sont les suivants :

- *Unité de duplication (Quoi)*. Quelles sont les entités dupliquées? Quel est leur type?
- *Technique de copie (Comment)*. Comment une copie est-elle créée? Quelles sont les informations dupliquées?
- *Moment de copie (Quand)*. Quand une copie est-elle créée? Est-elle créée statiquement, avant l'exécution, ou dynamiquement, en cours d'exécution?

- *Placement de copie (Où)*. Où une copie s'exécute-t-elle? Comment le placement répond-il aux objectifs de mise à l'échelle, de performances et de tolérance aux pannes?

Dans la suite, nous détaillons chacun de ces critères et montrons les différentes approches de gestion proposées par les travaux existants.

Unité de duplication (*Quoi*)

Un protocole de duplication est caractérisé par le type des copies qu'il gère, ainsi que par les critères de sélection des entités effectivement copiées. Le type des copies est généralement choisi en fonction des domaines d'application des protocoles, alors que les critères de sélection sont définis en fonction des objectifs de performances des protocoles. Ces deux points sont détaillés dans ce qui suit.

(i) Type de copie

Le type caractérise la structure et la sémantique d'une entité. La structure définit l'organisation interne de l'entité et joue un rôle important lors du processus de création de copie. Ce processus, discuté dans la section sur les techniques de copie, consiste à reproduire la structure et le contenu de l'entité.

La sémantique définit les opérations autorisées sur une entité. Elle définit des critères de correction qui sont à la base de la gestion de la cohérence entre copies. Par exemple, la sémantique des fichiers stipule que deux opérations d'écriture sur un même fichier sont conflictuelles mais que les opérations de création et de destruction de fichiers différents dans un répertoire ne le sont pas. La manière dont ces critères de correction sont définis et préservés est discutée dans la partie dédiée aux travaux relatifs à la cohérence (cf. Section 2.3.2, p. 45).

Les protocoles de duplication considèrent principalement quatre types d'entités : les fichiers, les zones mémoire, les bases de données et les objets.

- *Fichiers*. Dans le cas des fichiers, les techniques de copie se basent sur la structure séquentielle des fichiers et sur leur organisation hiérarchique en répertoires. La cohérence de copies de fichiers est maintenue par des synchronisations des accès en lecture et en écriture. La duplication garantit la tolérance aux pannes dans des systèmes comme Harp [89] et Leases [48]. Elle assure le travail en mode déconnecté pour les usagers mobiles dans de nombreux projets comme CodA [124], Ficus [117] et l'AFS mobile [60][61]. Elle permet l'accès rapide aux fichiers dans les travaux s'intéressant aux environnements à grande échelle tels que Roam [118] et LegionFS [149].
- *Zones mémoire*. La duplication de la mémoire est appliquée dans les travaux sur les mémoires partagées réparties avec l'objectif de fournir l'abstraction d'un environnement d'exécution centralisé. Comme dans le cas des fichiers, la cohérence est gérée par rapport aux accès en lecture et en écriture, et la copie est définie en fonction des structures hiérarchiques de blocs et de pages. Des projets marquants dans le domaine sont Khazana [21], Munin [19][20] et Arias [36]. Khazana s'intéresse à l'abstraction d'un espace de stockage partagé pouvant être utilisé dans le cadre de différents types de systèmes. Munin et Arias proposent des supports d'exécution intégrant différents protocoles de cohérence.
- *Bases de données*. Comme les mémoires et les fichiers, les bases de données représentent un support essentiel pour le stockage de données et utilisent la duplication pour la tolérances aux pannes, les performances d'accès, le travail des usagers mobiles en mode déconnecté, etc. Toutefois, la possibilité de définir différents types de données impose

l'utilisation de techniques de copie plus complexes qui prennent en compte la spécificité des données. La gestion de la cohérence est également influencée et considère, au moyen des transactions, des opérations plus évoluées que les lectures et les écritures de données[1]. Des projets qui travaillent sur la duplication de bases de données sont Bayou [134] qui fournit un environnement de travail pour les usagers mobiles, ainsi que Dragon [150][151] qui se penche sur les performances d'accès.

- *Objets*. Étant une abstraction qui permet la modélisation de tout type d'entité, les objets sont largement utilisés et dupliqués pour les besoins de tolérance aux pannes, de performance d'accès et de mise à l'échelle. Des nombreux projets qui travaillent sur la duplication d'objets nous pouvons citer Globe [135] qui s'intéresse aux documents web, GARF [51] qui permet l'utilisation de différents protocoles de tolérance aux pannes ou encore Rover [70] qui permet le travail en mode déconnecté. Des travaux comme CASCADE [30] ou IRL [92] s'intéressent respectivement à la duplication pour la gestion des caches ou la tolérance aux pannes d'objets CORBA. Enfin, des projets tels que Mole [10] ou FarGo [58] se penchent sur la duplication pour les besoins de migration d'objets autonomes (objets actifs ou agents).

(ii) Sélection des entités à copier

Les protocoles de duplication appliquent des critères de sélection pour décider si une entité doit être effectivement copiée ou non. Ces critères sont principalement de trois types : ils sont définis en fonction des types des entités, portent sur les modèles d'accès à ces entités ou alors sont des critères applicatifs.

- *Critères de type*. Les critères de type sont utilisés pour distinguer les entités de type "duplicable" d'un système et pour indiquer que ces entités sont systématiquement dupliqués. Des exemples de cette approche sont les objets sérialisables de Java [140] ou encore les types-valeurs de CORBA [101].
- *Critères d'accès*. Les critères d'accès limitent les coûts de synchronisation en restreignant l'ensemble des entités à copier. Ils copient les entités uniquement si celle-ci sont effectivement utilisées et sont appliqués dans les protocoles de gestion de cache [19][86].
- *Critères applicatifs*. Alors que les critères de type et les critères d'accès sont définis au niveau des protocoles de duplication, les critères applicatifs sont définis par les applications qui utilisent ces protocoles. En prenant en compte les spécificités applicatives, ces critères visent à réduire le coût de synchronisation de copies ainsi que l'espace de stockage nécessaire à celles-ci. Ils ont typiquement été mis en place sous forme de préférences utilisateur dans les environnements à usagers mobiles [94][124] et sous forme d'abonnements dans les réseaux actifs diffusant du contenu à la demande [111].

Technique de copie (*Comment*)

Le processus de création de copie dépend de la structure et de l'état de l'entité à dupliquer. La structure de l'entité peut être indivisible ou composée, alors que l'état peut être constitué de données, de code et d'un éventuel état d'exécution.

1. La notion de transaction est considérée dans la partie dédiée aux travaux relatifs à la cohérence.

(i) Structure de copie

Dans le cas où l'entité à copier a une structure indivisible, la copie reproduit cette structure et l'initialise avec l'état capturé de l'entité. La technique de copie dépend donc de l'état de l'entité et est discutée dans le point suivant.

Dans le cas où l'entité à copier a une structure composée, la copie peut être faite à différentes niveaux de *granularité*. En effet, elle peut considérer la structure dans sa totalité ou alors travailler sur les structures qui font partie de cette dernière. Par exemple, dans le cadre des systèmes de fichiers, CVS [24] et NFS [125] permettent la copie de répertoires, Ficus [117] et PAST [121] copient des fichiers, alors que Coda [123] et InterMezzo [88] considèrent le niveau intermédiaire de volume. Au niveau des mémoires, la copie peut considérer les blocs comme dans MemNet [38], les pages comme dans Munin [20] ou les collections de pages comme les segments dans Arias [36].

La granularité de copie influence les performances des protocoles [71][96][122]. Une grande granularité permet la copie groupée de plusieurs entités et répond aux besoins de localité d'accès. Toutefois, elle interdit les accès concurrents aux entités appartenant à un tel groupe. Une granularité fine atténue le problème de faux partage mais induit d'importants coûts de synchronisation.

(ii) État de copie

La création de copie est basée sur la capture et la restauration de l'état de l'entité à dupliquer. Dans le cas d'entités données, la capture et la restauration travaillent sur les valeurs de ces données. Dans le cas d'entités objets, la capture et la restauration considèrent les données internes des objets, ainsi que leur état d'exécution. Toutefois, la complexité de capture de l'état d'exécution fait que la plupart des techniques ne considèrent que l'état "passif" des objets. C'est l'approche adoptée, par exemple, par la sérialisation Java [140] ou par le projet sur les agents mobiles Mole [10]. La capture de l'état actif d'exécution est surtout considéré dans le cadre de la migration [15] et non de la duplication.

Moment de copie (*Quand*)

La création de copies peut être faite de manière statique ou dynamique. La création statique correspond à la définition de copies d'une entité avant l'exécution de celle-ci, alors que la création dynamique permet la création de copies en cours d'exécution.

La création statique de copies a l'avantage de fixer le nombre, les identités et les placements des copies. La connaissance de ces informations permet la mise en place de protocoles de cohérence adaptés à la configuration spécifique de l'ensemble de copies. Elle peut être utilisée, par exemple, pour la détection de pannes de copies comme dans le cas de NFS [125].

La création dynamique de copies permet l'adaptation de l'ensemble de copies aux variations de l'environnement d'exécution. Les facteurs qui déclenchent de telles créations dynamiques sont typiquement les problèmes de connexion, les problèmes de surcharge ou encore les nombreux accès consécutifs aux copies. Les nouvelles copies répondent à ces problèmes en assurant respectivement la disponibilité lors de partitions, la répartition de charge et la localité d'accès. Pour la gestion de partitions, un exemple typique est Bayou [143] avec son fonctionnement en mode déconnecté. Pour la localité d'accès, nous pouvons citer Munin [19] ou encore les travaux sur les caches web [23]. Quant à la répartition dynamique de charge, elle est considérée dans le travaux de Rabinovitch et al. [114] sur les placement des copies.

Placement de copie (*Où*)

Le placement des copies a un impact direct sur le fonctionnement d'un système dupliqué. Les schémas de placement sont définis en fonction des objectifs de performance d'accès, de tolérance aux fautes ou encore de mise à l'échelle.

Pour une amélioration des performances d'accès, le placement doit prendre en compte les caractéristiques des environnements d'exécution des utilisateurs. Les travaux, par exemple, de Radoslavov et al. [114] et de Rabinovich, et al. [115] proposent des algorithmes de placements basés sur les informations de topologie disponibles dans les routeurs d'Internet.

Pour la mise à l'échelle, le placement doit assurer que les copies recouvrent la totalité de l'infrastructure considérée. Nous pouvons citer le projet PAST [121] qui fournit l'abstraction d'un espace de stockage dans un environnement *pair à pair*[1] et qui propose un algorithme de placement automatique de copies.

2.3.2 Classification des travaux relatifs à la cohérence

Les critères de classification que nous avons choisis pour la cohérence sont les suivants :

- *Critères de correction (Pourquoi).* Pourquoi les copies doivent-elles se synchroniser? Ce besoin est-il défini par rapport à un état de référence de l'entité dupliquée ou par rapport au contexte d'utilisation de celle-ci?
- *Copies de synchronisation (Qui).* Quand une opération doit être effectuée sur une copie, quelles sont les copies qui décident de la manière de traiter cette opération?
- *Techniques de synchronisation (Comment).* Comment les copies se mettent-elles d'accord sur la manière de traiter une opération? Quels sont leurs échanges? Qui déclenche une synchronisation?
- *Moment de synchronisation (Quand).* Quand le processus de synchronisation est-il déclenché? La synchronisation est-elle faite avant une modification de copie ou après plusieurs modifications?

Critères de correction (*Pourquoi*)

La correction d'exécution des copies d'une entité peut être spécifiée de deux manières différentes. La première approche consiste à définir une exécution de référence qui est imposée aux copies. Les copies sont synchronisées selon une logique de cohérence dite *forte* puisqu'elles ne sont pas autorisées à diverger. La deuxième approche n'impose pas d'exécution de référence aux copies mais considère les exigences des entités qui interagissent avec celles-ci. Les copies sont synchronisées selon une logique de cohérence dite *relâchée* puisqu'elles sont autorisées à diverger tant qu'elles fournissent l'illusion d'une exécution qui satisfait ces entités.

(i) Exécution de référence des copies. Cohérence forte.

L'approche qui consiste à définir une exécution de référence pour les copies correspond à l'approche intuitive qui tente de préserver la logique d'une exécution centralisée. Dans ce cas, toute modification de copie est interprétée comme une modification de l'entité non dupliquée et la synchronisation des copies garantit que toute modification sur une copie est répercutée sur

1. Les environnements pair à pair sont des environnements où les ressources évoluent dynamiquement. Les machines peuvent se connecter à n'importe quel moment et communiquer avec toute entité dans l'infrastructure.

toutes les autres copies. Plus précisément, elle garantit que la consultation de n'importe quelle copie reflète la dernière modification effectuée sur le groupe de copies. Ceci a été défini par Lamport comme étant la *cohérence séquentielle* [85] qui est dite encore cohérence *forte* puisqu'elle ne tolère pas de divergences entre copies.

La garantie de la cohérence forte nécessite une synchronisation qui implique l'ensemble des copies et qui interdit leurs modifications concurrentes. Elle a un coût important puisqu'elle nécessite la connaissance de cet ensemble des copies ainsi qu'une gestion des nombreuses communications entre celles-ci. Des travaux qui mettent en place une telle gestion sont, par exemple, les projets sur les mémoires partagées réparties Orca [7] ou Clouds [35] qui utilisent respectivement les mécanismes de diffusion ou de synchronisation à base de verrous. La cohérence forte est également considérée dans les travaux sur la communication de groupe tels qu'ANSA [106], le projet GARF [51] ou encore le projet DRAGON [151][1]. ANSA définit une interface standard d'utilisation des groupes, GARF utilise cette notion pour la gestion de la tolérance aux pannes, alors que DRAGON met en place la cohérence forte pour des bases de données.

Le coût important de synchronisation fait que l'approche ne peut être utilisée dans les environnements à grande échelle qui manipulent beaucoup de copies. Les approches de cohérence forte ne peuvent non plus être utilisées dans les environnements à partitions où les communications entre copies peuvent être interrompues. Pour ces raisons, de nombreux systèmes adoptent l'approche de cohérence relâchée, détaillée dans ce qui suit.

(ii) Garanties de cohérence vis à vis des clients des copies. Cohérence relâchée.

Le principe de la cohérence relâchée consiste à retarder la synchronisation entre copies et de ce fait à autoriser les divergences de celles-ci[2]. Dans ce cas, les divergences autorisées sont définies vis à vis des exigences de correction des entités qui utilisent les copies et la synchronisation ne concerne que les opérations qui influencent les exécutions de celles-ci.

Après des expériences telles que Grapevine [127] démontrant que l'absence de garanties de cohérence mène à une confusion les clients observant des phénomènes sans relation avec leurs propres exécutions, plusieurs projets ont essayé de caractériser les modes d'usage de copies. Nous pouvons citer le projet Munin [19] qui travaille dans le domaine des mémoires partagées réparties, ainsi que Bayou [142] qui considère les environnements à usagers mobiles.

Munin [19] relâche la cohérence de quatre manières différentes en définissant quatre types de variables partagées : les non modifiables, les conventionnelles, les migratoires et les partagées en écriture. Les deux premiers types correspondent aux cas les plus communs : les variables non modifiables n'ont pas besoin de synchronisation, alors que les variables conventionnelles se synchronisent de manière standard sur les accès en écriture. Les variables migratoires sont typiquement manipulées dans une section critique et sont synchronisées à l'aide de verrous qui octroient les droits de lecture et d'écriture pour plusieurs accès consécutifs. Finalement, les variables partagées en écriture permettent qu'on modifie leurs différentes sous-parties en parallèle.

Dans Bayou [142], les usagers qui travaillent en mode déconnecté ont la possibilité de synchroniser leur travail selon également quatre schémas de cohérence relâchée. Le premier

1. La notion de groupe va être discutée dans les techniques de synchronisation.
2. Cette définition n'est pas celle des travaux dans le domaine des mémoires partagées réparties qui disent gérer une cohérence relâchée dans les cas où le système n'assure pas la gestion de la cohérence de manière transparente mais implique également l'application.

schéma (*Read your Writes*) permet aux clients d'observer, lors de leurs lectures, les écritures qu'ils ont effectuées précédemment. Le deuxième schéma (*Monotonic Reads*) permet aux clients de ne pas perdre la possibilité d'observer des écritures qui ont été observées lors de lectures précédentes. Le troisième schéma (*Writes follow Reads*) garantit que si les clients ont observé un certain nombre d'écritures, alors les écritures consécutives vont être exécutées après les écritures observées. Le quatrième et dernier schéma (*Monotonic Writes*) garantit qu'une écriture est prise en compte seulement si les écritures la précédant sont également prises en compte.

Copies de synchronisation (*Qui*)

Parmi les copies d'une entité dupliquée, les copies de synchronisation sont celles qui décident de la manière dont une opération sur une copie doit être traitée. Appelées *maîtres*, ces copies détiennent l'état de référence[1] de l'entité dupliquée et sont les seules à pouvoir le faire évoluer. Les autres copies, les *esclaves*, se réfèrent aux décisions des maîtres.

Les protocoles de cohérence font usage de deux schémas de gestion des maîtres : le maître unique et le multi-maître. Dans le cas de maître unique, une seule copie détient et met à jour l'état de référence de l'entité dupliquée. Dans le cas du multi-maître, les opérations de modification sont traitées par plusieurs copies.

Le schéma à maître unique est simple à gérer. Il permet de mettre facilement en place une cohérence forte mais n'est adapté qu'aux systèmes de copies subissant peu de modifications. En effet, dans des systèmes où les modifications sont fréquentes, le maître unique représente un goulot d'étranglement. Un exemple typique de maître unique est le DNS [3] où les informations sur les domaines de noms sont mises à jour sur un serveur primaire (le maître) et répercutées sur des serveurs secondaires (les esclaves).

Le schéma multi-maître utilise, dans la plupart des cas, des protocoles à cohérence relâchée et répartit les opérations de modification entre plusieurs copies. Ainsi, il permet de mieux répondre aux besoins de mise à l'échelle et d'amélioration des performances. Le schéma est utilisé dans les projets Bayou [134] et Roam [118] qui assurent le travail en mode déconnecté des usagers mobiles, dans les systèmes de fichiers tels que NFS [125] ou encore dans les applications de travail coopératif comme le forum d'actualités USENET [133]. Dans les environnements "*pair-à-pair*" émergents, le schéma multi-maître est poussé à l'extrême. Dans ces environnements où les ressources de calcul évoluent dynamiquement, les objectifs privilégiés sont ceux de disponibilité et donc toutes les copies peuvent traiter des opérations. Des projets représentants sont PAST [121] et Globus [120] qui fournissent des systèmes de fichiers dupliqués.

L'organisation d'un ensemble de maîtres, ainsi que des copies esclaves qui dépendent de ces maîtres est principalement basée sur le mécanisme de groupe [6][13][28][44][106]. Ces mécanismes définissent des règles d'appartenance à un groupe et de gestion de cohérence entre les membres de celui-ci. L'appartenance peut être définie de manière statique, évoluer dynamiquement ou encore traiter les problèmes de pannes et de partitionnement [65][72]. La communication est basée sur les techniques de diffusion qui peuvent gérer différemment les ordres d'émission et de livraison de messages, ainsi que la fiabilité et l'atomicité de transmission [31].

1. Excepté les divergences dans le cas de cohérence relâchée.

Techniques de synchronisation (*Comment*)

Pour se synchroniser, les copies d'une entité dupliquée doivent se mettre d'accord sur l'ordre des opérations à traiter, sur le type des informations à échanger, ainsi que sur la manière dont ces informations sont propagées. Dans la mesure où il est considéré que les copies se partagent l'état de l'entité dupliquée, l'ordre définit les contraintes sur les accès concurrents à cet état partagé. Les informations échangées peuvent porter sur l'état actuel des copies ou sur les opérations qu'elles ont effectuées. Elles peuvent être propagées à l'initiative des copies maîtres ou alors à la demande des esclaves. Ces trois points sont discutés en détail dans ce qui suit.

(i) Ordre d'exécution

Les ordres ont été introduits indépendamment par deux domaines différents. D'une part, ils ont été proposés sous la forme de transactions dans le domaine des bases de données [50]. D'autre part, ils ont été utilisés pour définir les règles de coordination des groupes de processus coopérants [13][28][31]. Utilisés pendant longtemps dans leurs domaines respectifs, les deux mécanismes ont récemment été rapprochés par des travaux étudiant leurs aspects similaires et complémentaires [29][46][150]. Ils ont récemment été conjointement utilisés dans des systèmes hybrides tels que les systèmes transactionnels répartis à base de communication de groupe [44].

Qu'il s'agisse de définitions provenant des bases des données ou des processus distribués, les ordres peuvent être caractérisés en fonction de l'ensemble des opérations ordonnancées et du critère d'ordonnancement utilisé.

En ce qui concerne l'ensemble des opérations, les ordres peuvent porter sur la totalité des opérations dans un système (*ordre total*) ou n'en ordonnancer qu'un sous-ensemble (*ordre partiel*).

- *Ordre total.* La définition d'un ordre total pour un système dupliqué correspond à la solution intuitive qui vise à rapprocher l'exécution de ce dernier à une exécution centralisée. En effet, dans les systèmes centralisés toutes les opérations sont ordonnancées. Des relations d'ordre total sont définies, par exemple, dans les systèmes à maître unique comme le DNS ou dans les systèmes à jetons tels que la mémoire partagée répartie Pilgrim [53]. Dans le premier cas, le maître unique est responsable de l'exécution des opérations et, par conséquent, définit l'ordre global comme étant l'ordre de ces traitements. Dans le cas des jetons, l'ordre global est défini par l'ordre d'accès au jeton qui donne les droits d'exécution des opérations.
- *Ordre partiel*. Les ordres partiels réduisent le coût de synchronisation induit par les ordres totaux en n'ordonnançant, dans un système distribué, que les événements qui ont des dépendances. Des exemples de tels ordres sont l'ordre causal défini par Lamport [84] ou encore les transactions [50]. L'ordre causal concerne les échanges ayant une relation de cause à effet par rapport à une notion de temps logique, alors que les transactions définissent les séquences d'actions élémentaires qui composent les opérations applicatives.

En ce qui concerne les critères d'ordonnancement, les ordres peuvent être définis par rapport aux moments d'exécution des opérations (*ordre syntaxique*) ou en se basant sur leur sémantique (*ordre sémantique*).

- *Ordre syntaxique*. Un ordre syntaxique ordonnance un ensemble d'opérations en se référent au moments logiques de leurs exécutions (*estampilles*). L'approche a l'avantage de pouvoir ordonner tout type d'opérations et est de ce fait appliquée dans différents domaines. Des exemples sont les stratégies d'ordonnancement des lectures et des écritures

dans les bases de données tels le protocole de verrouillage à deux phases (2PL) [14] ou encore la propagation des mises à jour dans Bayou [109]. Les ordres syntaxiques sont également largement utilisés pour définir les propriétés de fiabilité et d'atomicité des diffusions de groupe [31].

- *Ordre sémantique*. Les ordres sémantiques répondent aux problèmes de faux conflits qui handicapent les ordres à base d'estampilles. En effet, ils définissent des règles de correction basées sur la sémantique des opérations et non sur des informations syntaxiques. Considérés dans des projets comme IceCube [75], ils sont utilisés dans les cas où les traitements de synchronisation et le nombre d'opérations rejetées doivent être minimisés. Il s'agit, par exemple, des cas des usagers mobiles travaillant en mode déconnecté ou des environnements à grande échelle comptant de nombreuses copies.

(ii) Type d'échanges de synchronisation

Lors d'une synchronisation, les informations échangées peuvent porter sur l'état le plus récent des copies (*échanges d'état*) ou alors sur les opérations qui ont fait évoluer cet état (*échanges d'opération*s).

Les échanges d'état sont simples à gérer puisqu'ils ne nécessitent que des primitives de capture et de restauration d'état. Toutefois, ces échanges sont coûteux dans le cas de copies volumineuses et ne sont pas adaptés aux systèmes avec de nombreux conflits. En effet, ils ne permettent pas de réconciliation entre copies mais seulement le choix d'une copie de référence. L'état de cette copie est répercuté sur les autres copies en ignorant les possibles modifications effectuées sur celles-ci. L'approche est typiquement utilisée dans la gestion de sites miroirs comme dans le cas du DNS.

Les échanges d'opérations sont plus efficaces en termes de bande passante et permettent une réconciliation de conflits en fonction de la sémantique des opérations. Toutefois, leurs résultats sont obtenus au prix d'une gestion complexe de journaux d'opérations. En effet, les copies sélectionnent les opérations à journaliser tout en optimisant la taille des enregistrements [37][116]. Un projet explorant l'approche des échanges d'opérations est IceCube [75].

(iii)Modèle de propagation

Dans un groupe de copies, la propagation des mises à jour effectuées par les copies maîtres peut être faite de deux manières différentes. Elle peut être faite à la demande des maîtres (modèle *push)* ou à la demande des copies esclaves *(*modèle *pull).*

Dans le modèle push, ce sont les maîtres qui sont responsables de transmettre les mises à jour aux esclaves. La propagation peut être faite sans délai et les communications entre copies ne sont effectuées que lorsqu'il y a de nouvelles mises à jour. Toutefois, la technique est coûteuse en termes de trafic réseau puisqu'elle transmet les mises à jour à toutes les copies sans prendre en compte leurs besoins réels de synchronisation. De plus, comme il est montré dans les travaux de Demers et al.[39], les mises à jour risquent d'être envoyées plusieurs fois. L'approche est appliquée dans les environnements fortement connectés comme le forum à messages USENET [133] ou encore la base de données relationnelle Oracle [33].

Dans le modèle pull, ce sont les esclaves qui demandent les nouvelles mises à jour. Le trafic réseau est réduit puisque seules les informations utiles sont acheminées. Toutefois, les communications sont initiées périodiquement, même en l'absence de modifications, et ce sont les esclaves qui doivent gérer la liste des mises à jour qui leur sont nécessaires. Le modèle pull est utilisé surtout dans les environnements où les ressources réseau sont limitées comme les environnements à usagers mobiles tels que Bayou [144]

Moment de synchronisation (*Quand*)

Une copie peut se synchroniser avec les autres copies avant ou après l'exécution d'une opération. Les deux approches correspondent à deux stratégies de gestion de la cohérence : la gestion pessimiste et la gestion optimiste.

Dans l'approche pessimiste, tout traitement est précédé par un accord global entre copies sur la manière dont ce traitement doit être effectué. Les divergences entre copies ne sont donc pas permises et les copies vérifient les conditions de cohérence forte. Les opérations sont exécutées de manière définitive et fournissent des résultats fiables. Toutefois, le déterminisme d'exécution nécessite un processus de synchronisation qui est trop coûteux pour les environnements à grande échelle et non réalisable dans les environnements à partitions.

La gestion optimiste [122] adopte l'approche contraire : elle n'impose pas d'accord global avant l'exécution des opérations et synchronise les copies ultérieurement. Elle repose sur l'hypothèse que les accès conflictuels sur les copies sont rares et traite les opérations sans délai. Toutefois, les résultats de traitement sont provisoires [49][145] et peuvent être invalidés lors de l'étape de synchronisation. Cette dernière est responsable non seulement de la propagation des opérations entre copies, mais également de la détection et de la résolution d'éventuels conflits.

Le retardement de l'étape de synchronisation augmente la probabilité de conflit tout en permettant l'optimisation des échanges entre copies. Par exemple, dans les travaux sur l'opération déconnectée dans AFS [61], la synchronisation est retardée afin d'attendre une connexion réseau favorable. Dans des travaux sur les applications web coopératives de Dédieu et al. [37] ou encore ceux sur la synchronisation des fichiers de Ramsey et Czirmas [116], le retardement est utilisé pour minimiser la taille du journal des opérations non synchronisées.

La gestion optimiste trouve ses applications dans les applications tolérant la cohérence relâchée comme le forum de messages USENET [133]. Elle est appliquée dans les environnements à grande échelle comme PAST [121], Globus [120] ou LegionFS [149] où une gestion pessimiste aurait eu un coût prohibitif. Elle est également la seule à pouvoir être utilisée dans les environnements à partitions [34].

2.3.3 Synthèse

La duplication et la cohérence sont d'une grande importance dans les environnements distribués. Elles répondent à des besoins de performances, de tolérance aux pannes et de mise à l'échelle. Elles sont utilisées dans le cadre des bases de données, des fichiers ou des mémoires et leurs applications couvrent tous les domaines d'activité.

Les travaux relatifs à la duplication peuvent être classifiés selon l'unité de duplication (*Quoi*), la technique de copie (*Comment*), le moment de duplication (*Quand*) et les critères de placement des copies (*Où*).

- *Quoi*. Dans leur majorité, les travaux relatifs à la duplication choisissent l'unité de duplication en fonction de leur domaine d'application et permettent ainsi la mise en place de techniques de copie et de protocoles de cohérence spécifiques. La sélection des entités effectivement copiées est faite selon trois types de critères qui portent respectivement sur les types des entités, sur les modèles de leurs accès et sur le contexte d'usage applicatif.
- *Comment*. Le processus de duplication dépend de la structure et de l'état des entités considérées. En ce qui concerne la structure, il peut être défini par rapport à des niveaux de granularité dans des structures composées ou alors porter sur des structures indivisibles.

En ce qui concerne l'état, le processus de duplication peut capturer des états données, des états contenant du code ou encore des états courants d'exécution. Le choix parmi différents niveaux de granularité permet de définir différents compromis entre les objectifs de localité d'accès et les problèmes de faux partage. Le choix de l'état à capturer permet de se situer dans un cas simple de copie de données ou alors dans le cas difficile d' objets en cours d'exécution, surtout considéré dans le cas de la mobilité.

- *Quand*. La duplication peut être faite avant le lancement d'un système ou pendant son exécution. La première technique permet la mise en place de protocoles de cohérence spécifiques aux configurations de copies, alors que la deuxième permet l'adaptation du groupe de copies aux variations de l'environnement d'exécution.
- *Où*. Les travaux existants définissent les stratégies de placement en fonction de leurs objectifs de performances et de disponibilité. Le projet PAST [121], par exemple, propose un algorithme de placement automatique visant à recouvrir la totalité d'une infrastructure à grande échelle, alors que les travaux de Rabinovitch et al. [115] placent les copies en fonction d'informations sur la topologie de l'environnement d'exécution.

Les travaux relatifs à la cohérence peuvent être classifiés selon les critères de correction choisis (*Pourquoi*), le choix des copies qui effectuent les mises à jour (*Qui*), la technique de synchronisation (*Comment*) et le moment de celle-ci (*Quand*).

- *Pourquoi*. La correction d'exécution d'un groupe de copies peut être définie par rapport à une exécution centralisée ou alors être basée sur la notion de comportement correct observable par les entités qui interagissent avec les copies. Dans le premier cas, la majorité des solutions de synchronisation ne permettent pas de divergences entre copies et assurent une cohérence dite *forte*. Dans le deuxième cas, le coûts de synchronisation sont diminués grâce à l'approche de cohérence *relâchée* qui autorise les divergences entre copies.
- *Qui*. Un groupe de copies est composé de copies *maîtres* qui décident de la manière de traiter les modifications des copies et de copies *esclaves* qui suivent les décisions des maîtres. Les *maîtres* peuvent être organisées selon deux schémas différents : le schéma à maître unique et le schéma multi-maître. La première solution est adaptée aux environnements subissant peu de modifications et permet de facilement mettre en place une cohérence forte. La solution multi-maître est surtout utilisée pour fournir une cohérence relâchée adaptée aux environnements à partitions ou à grande échelle.
- *Comment*. Lors d'une synchronisation, les copies doivent se mettre d'accord sur le type d'informations à échanger, sur la technique de leur propagation et sur l'ordre des opérations à traiter. En ce qui concerne les *informations à échanger*, elles peuvent porter sur les états des copies ou sur les opérations qu'elles ont traitées. Les deux approches sont respectivement utilisées dans les environnements subissant peu de modifications et dans ceux ayant de nombreux conflits.

  La *propagation des modifications* peut être demandée par les copies maîtres ou par les copies esclaves. La première solution est adaptée aux environnements fortement connectés, alors que la deuxième solution permet que les copies esclaves travaillent en mode déconnecté.

  Finalement, *les ordres* servent à ordonnancer toutes les opérations dans un système ou alors diminuer le coût de synchronisation en ne considérant que les opérations avec des dépendances. Ils peuvent ordonnancer les opérations de tout type en utilisant leurs moments logiques d'exécution ou alors se focaliser sur la gestion de conflits en se basant sur la sémantique des opérations.
- *Quand*. La synchronisation des copies peut être effectuée de manière pessimiste, avant

une modification, ou alors de manière optimiste, après plusieurs modifications. La première approche garantit une cohérence forte alors que la gestion optimiste fournit une cohérence relâchée qui est adaptée aux environnements à partitions ou à grande échelle.

Les critères de classification choisis ne reflètent pas toutes les caractéristiques des protocoles de duplication et de cohérence mais réussissent à donner un aperçu de la complexité de leur mise en œuvre. En effet, la construction d'un protocole implique le positionnement de celui-ci par rapport à chacun de ces critères. C'est la principale raison pour laquelle les travaux existants choisissent des mécanismes de gestion fixes qui s'appliquent à des contextes spécifiques d'utilisation. Or, les différents objectifs de performances et les différentes caractéristiques des environnements d'exécution font apparaître différents besoins en termes de duplication et de cohérence qui posent la question de la *configurabilité* de leur gestion.

La section suivante présente une classification des travaux existants en considérant leurs techniques de configuration.

2.4 Configuration de la duplication et de la cohérence

Nous avons choisi de présenter les travaux s'intéressant à la configuration de la duplication et de la cohérence selon les critères suivants :

- *Niveau de mise en œuvre*. Comment les systèmes mettent-ils en place différents protocoles? Considèrent-ils les protocoles au niveau langage, au niveau système ou au niveau intergiciel?
- *Degré de transparence.* Quelle est l'implication des applications dans la gestion de la duplication et de la cohérence? Cette gestion est-elle transparente ou fait-elle partie des traitements applicatifs?
- *Moment de configuration*. À quel moment une application peut-elle configurer sa gestion de la duplication et de la cohérence? Le fait-elle pendant l'étape de développement, au moment du déploiement ou pendant l'exécution?

Étant donné que la gestion de la cohérence dépend grandement de la duplication préalablement effectuée, la plupart des systèmes fournissent des traitements qui gèrent la duplication et la cohérence de manière inséparable. Pour cette raison, dans la suite de cette section, nous parlerons de ces traitements en tant que protocoles de duplication et de cohérence.

2.4.1 Niveau de mise en œuvre

Plusieurs systèmes se sont intéressés aux mécanismes permettant la mise en place de différents protocoles. Leur niveau de mise en œuvre des protocoles peut se situer au niveau langage, au niveau système ou encore au niveau intergiciel.

Mise en œuvre au niveau langage

Les efforts de fournir des outils langage pour la mise en œuvre de protocoles de duplication et de cohérence sont rares. À notre connaissance, seul le projet Teapot [27] s'est intéressé à la définition d'un langage spécifique au domaine.

Teapot permet la programmation et la vérification de protocoles de cohérence dans un environnement à mémoire partagée. Il se focalise sur la gestion de caches dont les traitements sont

automatiquement générés. Ces derniers sont obtenus par compilation à partir de programmes basés sur des constructions de type événement-réaction et modélisant un système en tant que machine à états.

La mise en place des traitements de protocole au niveau langage et l'automatisation de nombreux traitements réduisent énormément la complexité du développement. Teapot permet d'éviter de nombreuses erreurs et ainsi diminue le temps de programmation et de débogage. Toutefois, les auteurs font remarquer que la facilité de développement est obtenue au prix de plusieurs restrictions. Ainsi, Teapot s'avère inadapté à des environnements relativement complexes qui manipulent de nombreux processus légers (multi-threading) et ne traite pas de manière satisfaisante les primitives bloquantes d'accès aux données.

Mise en œuvre au niveau système

La mise en œuvre de protocoles au niveau système est l'approche inverse à l'approche langage. En effet, elle privilégie les performances des protocoles et n'impose pas de restrictions sur leurs implémentations. Elle permet de définir les protocoles à un grain très fin ce qui est payé par une grande complexité de développement.

L'implantation de protocoles à base de primitives système est typiquement faite dans les mémoires partagées réparties (MPR), ainsi que dans les systèmes de processus coopérants. Dans les MPR [71], les primitives de programmation concernent les structures de synchronisation telles les sémaphores ou les barrières, l'adressage des entités partagées et l'envoi de messages. Des représentants sont Munin [19] ou encore Arias [36]. Munin fournit des primitives système de mise en place de plusieurs protocoles mais les utilise pour définir quatre protocoles qui correspondent à quatre modes d'usage d'entités partagées (cf. *Critères de correction (Pourquoi)*, p. 45). Arias fournit une couche générique qui définit une interface de communication entre des copies maîtres et des copies esclaves qui doit être implémentée par les différents protocoles.

Dans les systèmes de processus coopérants distribués, les primitives de bas niveau concernent la gestion de groupe et contrôlent l'ordre et la fiabilité lors de la diffusion de messages. Des classifications des types de diffusion sont proposées par Chocler et al. [31] et par Chanso et al. [28].

Mise en œuvre au niveau intergiciel

L'approche intergicielle essaie de réconcilier les objectifs de performances des protocoles, mis en avant par l'approche système, et les objectifs de facilité de mise en place, privilégiés par les solutions langage. Elle propose des modèles structurels qui définissent les protocoles en termes de modules et d'interfaces entre ces derniers. Le cadre fourni facilite la mise en place de protocoles tout en laissant la liberté d'implantation pour les objectifs de performances.

Les projets qui adoptent l'approche intergicielle portent, dans leur majorité, sur la duplication et la cohérence d'objets. Dans la suite, nous présentons les systèmes Subcontract [56], PassiveReplicator [47], BAST [46] et TACT [155]. Des projets marquants comme GARF [51] et Globe [135] qui définissent des modèles à objets étendus pour la gestion de la duplication et de la cohérence sont considérés dans la section sur les composants.

- Subcontract.

Subcontract [56] s'intéresse à la modification de la sémantique d'un appel à distance dans un système à objets. Se plaçant dans un modèle client-serveur, il modifie cette sémantique en rajoutant des traitements additionnels (*sous-contrats*) sur le chemin des invocations. Les sous-contrats font partie des piles d'invocation des deux côtés et sont utilisés pour introduire la gestion de la duplication et de la cohérence dans les interactions entre objets. Différentes implémentations de primitives d'emballage et de déballage, ainsi que d'envoi et de réception de requêtes, correspondent à différentes protocoles de duplication et de cohérence. Les protocoles proposés par Subcontract sont la gestion de cache et la tolérance aux fautes.

L'approche de Subcontract est utilisée dans de nombreux autres projets comme, par exemple, CASCADE [30] et IRL [92] qui gèrent la duplication et la cohérence d'objets CORBA. Pour enrichir les appels entre objets, ces projets fournissent une couche d'interception placée au-dessus du bus à messages CORBA. Inspiré par Bayou (cf. *Critères de correction (Pourquoi),* p. 45), CASCADE y intègre différents stratégies de gestion de cache qui correspondent à différents modes d'usage des objets. IRL, quant à lui, intègre dans la couche d'interception un mécanisme de groupe pour gérer la tolérance aux fautes.

- Passive Replicator

Passive Replicator [47] définit un patron de conception pour la duplication et la cohérence. Il définit trois niveaux dans la structure d'un protocole : générique, spécifique au protocole et applicatif. Le niveau générique correspond à des interfaces abstraites de duplication et de cohérence qui définissent des méthodes pour la décision de l'ensemble des copies de synchronisation, pour la mise en place de la technique de synchronisation, pour la technique de copie, etc. Le niveau spécifique implémente ces interfaces en utilisant des informations sémantiques fournies par le niveau applicatif.

- BAST

BAST [46] modélise sous forme d'objets les protocoles de duplication et de cohérence, ainsi que les traitements algorithmiques qui les composent. Il représente, par conséquent, les protocoles en tant que graphes d'objets qui ont des interfaces et des interactions bien définies. Cette structuration permet la compréhension du fonctionnement d'un protocole et facilite sa modification.
Les briques de base des protocoles de BAST sont les objets modélisant des solutions aux problèmes généraux de consensus et de détection de fautes. Les appliquant surtout dans le cadre des groupes de processus, BAST démontre que ces objets peuvent être utilisés dans d'autres contextes tels que les transactions.

- TACT

TACT [155] s'intéresse à définir un modèle générique de gestion de cohérence qui permettrait la mise en place de différents critères de correction en fonction des besoins applicatifs (cf. *Critères de correction (Pourquoi),* p. 45). Il définit des unités de cohérence, nommées *conits*[1], dont les contraintes de divergence sont exprimées en utilisant trois types de métriques. Notamment, les conits ne doivent pas dépasser une erreur numérique, une erreur d'ordre et une erreur temporelle. L'erreur numérique donne la différence entre le nombre total d'écritures traitées par un groupe de copies et le nombre d'écritures répercutées sur une copie. L'erreur d'ordre donne le nombre d'écritures qu'une copie peut traiter sans se synchroniser. Finalement, l'erreur temporelle définit le délai de temps maximum entre deux synchronisations.

1. conit = *consistency unit*.

TACT définit une interface simple qui permet la définition de ces trois métriques pour les différentes opérations de modification dans une application. Il a été utilisé pour spécifier les conditions de cohérence d'un forum à messages, d'une application de réservation de billets d'avion, ainsi que d'une application de gestion de qualité de service pour un ensemble choisi de clients.

Malgré leur diversité, les approches considérant la mise en place de protocoles au niveau intergiciel se rapprochent sur les points suivants :

- *Encapsulation des protocoles de duplication et de cohérence.* Les travaux mettant en place les protocoles au niveau intergiciel ont pour objectif de définir ces derniers indépendamment des applications. Cette approche permet la réutilisation des protocoles pour différentes applications. De plus, la construction de protocoles peut bénéficier des performances des primitives système de bas niveau sans imposer la complexité de développement aux applications.

- *Technique d'interception.* Les protocoles de duplication et de cohérence sont basés sur les techniques d'interception. En effet, l'interception permet de faire le lien entre les traitements génériques des protocoles et les traitements spécifiques des applications.

- *Compromis entre généralité et spécificité.* Les objectifs de généralité, qui permettent la réutilisation des protocoles, sont conflictuels avec les objectifs de performances, qui dépendent de la sémantique spécifique des applications. Tous les projets qui mettent en place des protocoles au niveau intergiciel optent pour un compromis entre ces deux objectifs. Des projets comme tels que TACT ou CASCADE permettent que ce compromis soit choisi par les applications.

2.4.2 Transparence de la duplication et de la cohérence

La duplication et la cohérence peuvent être entièrement gérées par le support d'exécution des applications (*transparence totale*), être entièrement à la charge des applications (*gestion applicative*) ou encore utiliser une solution intermédiaire qui fait coopérer les deux niveaux. Dans la suite, nous détaillons ces trois types de solutions.

Transparence totale.

La gestion totalement transparente de la duplication et de la cohérence est basée sur l'interception des communications applicatives et sur leur enrichissement par des traitements de protocole. Elle est mise en place dans les couches qui fournissent le support d'exécution des applications, c'est à dire les couches du niveau système ou du niveau intergiciel. Des exemples de projets qui garantissent une transparence totale sont GARF [51], Subcontract [56] ou encore CASCADE [30]. Ils fournissent tous des protocoles qui se placent au niveau de la couche de communication des applications et qui n'affectent pas leur code.

La transparence totale facilite grandement la construction d'applications. En effet, elle permet de choisir un protocole ou encore de changer un protocole déjà choisi sans imposer des changements aux applications. Toutefois, ceci est acquis au prix d'une limitation du nombre de protocoles proposés et d'une incapacité de prendre en compte la sémantique applicative.

Gestion applicative

La gestion purement applicative de la duplication et de la cohérence se situe à l'opposé de la transparence totale. Les applications ont la possibilité d'assurer l'optimalité des performances des protocoles en mettant en place des protocoles qui leur sont spécifiques. Cependant, étant donné que cette approche implique l'utilisation explicite de primitives de bas niveau, elle se caractérise par une grande complexité. Cette approche est illustrée par le système Arias [36] qui laisse à l'application le soin d'implémenter la synchronisation des copies. Cette complexité est réduite par Munin [19] qui se charge d'implémenter et de proposer aux applications un ensemble limité de protocoles.

Solutions intermédiaires

Le compromis entre les objectifs de facilité d'utilisation de différents protocoles et l'efficacité de ceux-ci est recherché dans beaucoup de solutions intermédiaires où les applications participent à la gestion de la duplication et de la cohérence. Cette approche est choisie par les projets TACT, CASCADE et Munin, présentées précédemment, ou encore par les projets IceCube [75] et Bayou [143]. CASCADE et Munin permettent aux applications de choisir leur protocole. TACT leur permet de le paramétrer. Finalement, IceCube et Bayou qui sont des systèmes à cohérence optimiste, permettent l'intégration de traitements de réconciliation spécifiques aux applications dans des protocoles génériques.

2.4.3 Moment de configuration

Dans les systèmes proposant plusieurs protocoles, a configuration d'une application afin qu'elle fasse usage d'un protocole particulier peut être faite à différents moments. Notamment, une application peut choisir le protocole qu'elle va utiliser pendant son étape de construction, elle peut avoir la possibilité de configurer le protocole au moment du déploiement ou encore elle peut agir sur la gestion de la duplication et de la cohérence pendant son exécution.

Choix de protocole lors de la construction des applications

Dans une grande majorité, les projets qui proposent plusieurs protocoles de duplication et de cohérence, les applications doivent choisir le protocole à utiliser pendant l'étape de leur construction. Ainsi, elle peuvent être implémentées de manière spécifique suivant la logique du protocole choisi et assurer un fonctionnement optimal.

Les représentants de cette approche sont typiquement les travaux sur les mémoires partagées réparties. Dans le cas de Munin [19], le choix d'un modèle d'accès à une variable partagée[1] est faite pendant la programmation des applications. Dans le cas de Teapot [27], la programmation des applications inclut la programmation des protocoles.

L'approche où le choix de protocole est effectué pendant la construction des applications suit la logique des systèmes ne proposant qu'un seul protocole. Dans ces systèmes, les applications sont modélisées et implantées en prenant en compte les détails de gestion de la duplication et de la cohérence. Un tel exemple est Rover [70] qui impose un modèle de programmation aux applications qui veulent bénéficier de sa gestion des déconnexions à base d'objets mobiles.

L'intégration des protocoles de duplication et de cohérence dans les applications pendant leur étape de construction a l'avantage de garantir une optimalité des performances. Toutefois,

---

1. cf. Garanties de cohérence vis à vis des clients des copies. Cohérence relâchée., p. 46

les applications sont construites en fonction des protocoles et tout changement de protocole implique un redéveloppement.

Configuration de protocole lors du déploiement

Paramétrer un protocole de duplication et de cohérence lors du déploiement permet de fournir de meilleures performances en prenant en compte la configuration initiale de l'application déployée, ainsi que les caractéristiques de son environnement d'exécution.

Les travaux qui s'intéressent à la configuration des protocoles pendant le déploiement définissent des modèles de gestion de la duplication et de la cohérence et offrent aux applications les interfaces qui permettent d'instancier ces modèles pour leurs besoins. Ainsi, dans le cas de Globe [135], de GARF [51] ou encore de Core [17], les applications paramètrent l'initialisation des objets pour prendre en compte un choix de protocole. Dans le cas de PAST [121] et de CORBA [101], les protocoles respectifs de gestion de fichiers à grande échelle et de tolérance aux fautes sont paramétrés par le nombre de copies voulues.

Les travaux qui permettent la configuration de la duplication et de la cohérence lors du déploiement rendent, en réalité, explicite le déploiement des protocoles correspondants. Toutefois, les règles de déploiement, la structuration et le fonctionnement de ces protocoles restent implicites et non modifiables.

Reconfiguration de protocole lors de l'exécution

L'approche de reconfiguration à l'exécution consiste à changer dynamiquement le protocole utilisé par une application. Un tel changement permet d'adapter la gestion de la duplication et de la cohérence en fonction des performances observées de l'application et de l'état courant de l'environnement d'exécution.

Les objectifs de reconfiguration dynamique des protocoles de duplication et de cohérence sont ambitieux. En effet, la reconfiguration d'une application sans duplication ni cohérence pose déjà de nombreux problèmes liés à la correction d'exécution. Reconfigurer la gestion de duplication et de cohérence doit considérer non seulement les contraintes liées à cette correction applicative mais aussi assurer une transition cohérente entre deux protocoles. Typiquement, lors d'une telle configuration il ne faut pas qu'il y ait des interactions applicatives entre parties gérées par l'ancien protocole et des parties déjà reconfigurées.

La complexité de la tâche de reconfiguration dynamique de la duplication et de la cohérence a été traité par les travaux existants de deux manières différentes : ils considèrent l'adaptation de protocoles spécifiques, ou alors s'intéressent à des schémas d'adaptation génériques.

Dans le cas des adaptations spécifiques, il s'agit de construire des protocoles de duplication et de cohérence de plus haut niveau qui spécifient des règles de transition entre quelques protocoles de base. Un exemple est le projet TACT [155] qui permet aux applications de changer dynamiquement leurs protocoles tout en restant dans le cadre du modèle de gestion général. Les règles de changement sont câblées dans ce dernier.

Dans le cas d'adaptations génériques, les travaux définissent des mécanismes génériques de reconfiguration qui ne sont pas ciblés sur mais peuvent être utilisés pour les besoins de reconfiguration des protocoles de duplication et de cohérence. DART [119], par exemple, met en place des stratégies d'adaptation en contrôlant les protocoles de duplication et de cohérence par des mécanismes de gestionnaires, d'événements et de réactions. Un autre exemple est DarX [128] qui adapte des protocoles de tolérance aux fautes en utilisant des groupes de copies [13][28][106] dont les principes de gestion peuvent être modifiés. Toutefois, même si ces

projets fournissent des mécanismes de reconfiguration, ils ne définissent pas d'outil pour le contrôle de la correction de ces reconfigurations. Ce contrôle est laissé à la charge des concepteurs de systèmes.

2.4.4 Synthèse

La configuration de la gestion de la duplication et de la cohérence est abordée dans plusieurs systèmes. Les approches diffèrent par leur niveau de mise en œuvre de différents protocoles, leur degré de transparence, ainsi que par leur moment de configuration.

- *Niveau de mise en œuvre*. Les mises en œuvre peuvent se situer au niveau langage, au niveau système ou encore au niveau intergiciel. La première approche facilite la construction de protocoles, la deuxième approche met en avant l'objectif d'efficacité et, finalement, l'approche intergicielle essaye de trouver un compromis entre ces deux objectifs.

- *Transparence de gestion.* La gestion de la duplication et de la cohérence peut être totalement transparente pour l'application, être complètement à sa charge ou alors se baser sur une approche intermédiaire. La première approche permet aux applications de facilement changer de protocole mais le choix est limité par un ensemble de protocoles proposés. La deuxième approche permet aux applications d'implémenter le protocole le mieux adapté mais au prix d'une grande complexité. Finalement, la troisième approche cherche un compromis entre ces deux extrémités.

- *Moment de configuration*. L'intégration des protocoles dans les applications peut être faite lors de la construction de ces dernières, lors de leur déploiement ou encore à l'exécution. La première approche est la plus simple à mettre en place mais implique une conception applicative spécifique aux protocoles ce qui rend difficile le changement de protocole. La deuxième approche réduit cette limitation et permet aux application de configurer leur duplication et cohérence au déploiement. Ainsi, les protocoles sont spécialisés en fonction de la configuration applicative et de l'environnement concret d'exécution. Finalement, la troisième approche va encore plus loin et permet l'adaptation dynamique des protocoles en fonction des variations pendant l'exécution des applications. Une configuration retardée implique des dépendances limitées entre applications et protocoles et par conséquent des changements de protocole plus simplement réalisés.

Les travaux sur la configuration de la duplication et de la cohérence sont motivées par les différents compromis recherchés entre les objectifs de performances, de souplesse et de facilité. Ces objectifs sont considérés du point de vue des systèmes qui proposent des protocoles ou du point de vue des applications qui utilisent ces derniers.

- *Performances, souplesse et facilité du point de vue des applications*. Dans le travaux ciblés sur les applications, les *performances* évaluent le fonctionnement des applications avec la gestion de la duplication et de la cohérence. La *souplesse* caractérise la capacité des applications à utiliser différents protocoles et la *facilité* concerne le processus de construction d'applications avec duplication et cohérence. Quant aux performances, elles sont optimales quand les protocoles sont implémentés de manière spécifique aux applications et sont donc opposés aux objectifs de facilité et de souplesse.
  Des protocoles de duplication et de cohérence qui réconcilient ces objectifs ont surtout été mis en place dans des domaines tels que la tolérance aux pannes ou la gestion de cache. La manque de maturité dans d'autres domaines comme ceux traitant du noma-

disme ou de la mise à l'échelle fait que leurs protocoles se focalisent surtout sur des considérations de performances. Par conséquent, les protocoles restent spécifiques aux domaines et les applications voulant en profiter sont obligées de migrer vers un support correspondant.

- *Performances, souplesse et facilité de point de vue des protocoles.* Dans les travaux ciblés sur les systèmes fournissant différents protocoles, les *performances* sont liées au fonctionnement de ces protocoles. La *souplesse* caractérise les capacités d'évolution des protocoles, ainsi que la possibilité d'utiliser ces protocoles pour les besoins de différentes applications Enfin, la *facilité* pose la question du processus de mise en place des protocoles.
  Après avoir considéré les travaux dans le domaine, la constatation qui peut être faite est que très peu de projets considèrent la facilité de construction de protocoles. Les structurations de protocoles proposées sont surtout faites en vue de l'utilisation des protocoles dans des applications et la construction elle-même se retrouve d'une complexité égale à celle des protocoles système [2]. En ce qui concerne l'objectif de réutilisation des protocoles, symétrique à celui de la souplesse des applications, il est traité par les solutions intergicielles qui définissent les interactions entre applications et protocoles. L'objectif d'évolution des protocoles est moins présent, même s'il devient d'actualité avec l'évolution rapide des environnements d'exécution et donc des besoins en duplication et en cohérence.

La gestion de la duplication et de la cohérence évolue d'une gestion système, axée sur les performances et sans contraintes de structuration, vers une gestion modulaire qui considère surtout les besoins de facilité et de souplesse des applications. Le fonctionnement des protocoles reste défini par des algorithmes de bas niveau d'une grande complexité et souffre du manque de gestion explicite d'aspects comme l'évolution, la réutilisation de parties de protocoles, le déploiement et le contrôle explicite à l'exécution.

Chapitre 3

Positionnement du travail de thèse

Les différents objectifs de performances et la diversité des environnements d'exécution ont fait apparaître le besoin de configurabilité des protocoles de gestion de la duplication et de la cohérence. Toutefois, les solutions proposées ne considèrent le problème que du point de vue de la souplesse des applications qui utilisent des protocoles. Ils se concentrent sur les possibilités d'utilisation de différents protocoles au sein d'une application en ignorant les aspects de facilité de construction et d'évolution de ces derniers. Leurs approches de configuration sont ad hoc et restent spécifiques aux domaines d'application tels que la tolérance aux fautes, la gestion de cache, la gestion d'usagers mobiles, etc.

Les solutions à composants fournissent une approche générique et uniforme de gestion des aspects non fonctionnels des logiciels. Encapsulant le code métier spécifique, ils permettent l'utilisation des mêmes solutions de gestion non fonctionnelle dans différents domaines d'application. Ils suivent le principe de séparation des aspects et ainsi permettent la configuration de la gestion non fonctionnelle sans modifications au niveau du code fonctionnel. Cependant, les travaux existants proposent des mécanismes de construction de services système et de configuration qui ne considèrent pas la duplication ni la cohérence.

Cette thèse se concentre sur le problème de gestion configurable de la duplication et de la cohérence dans les applications à base de composants. Ses objectifs sont les suivants :

- Généralité

La solution de gestion de la duplication et de la cohérence doit pouvoir adresser aussi bien les problèmes classiques de tolérance aux fautes et de gestion de cache, que les nouveaux besoins pour les environnements à usagers mobiles et à grande échelle. Elle doit pouvoir intégrer, de manière uniforme, les protocoles de duplication et de cohérence déjà existant. Elle doit également faciliter la construction de nouveaux protocoles pour répondre à la diversité grandissante des plates-formes et de leurs besoins.

Pour répondre à l'objectif de généralité, nous avons basé la gestion de la duplication et de la cohérence sur l'approche composant. Elle peut ainsi bénéficier du principe d'encapsulation et respecter une architecture qui est applicable pour différents besoins de disponibilité et dans différents domaines d'application tels que les fichiers, les bases de données ou les mémoires.

- Souplesse

La solution de duplication et de cohérence doit permettre aux applications d'utiliser différents protocoles de gestion. Le choix de protocole doit pouvoir être fait non seulement en fonc-

tion de la nature des applications mais aussi en fonction de leur environnement d'exécution. En d'autres termes, les modifications de configuration de la duplication et de la cohérence doivent pouvoir être faites non seulement entre applications mais également au sein d'une même application pour lui permettre de s'exécuter dans des contextes différents.

Pour permettre la souplesse de gestion de la duplication et de la cohérence, il est nécessaire de définir un modèle d'interaction clair entre le niveau applicatif et le niveau de gestion. Nous avons choisi de fournir ce modèle dans une couche intergicielle sur laquelle la gestion d'applications pourra s'appuyer. Pour que la gestion de la duplication et de la cohérence reflète les contextes d'exécution des applications, nous considérons l'approche où l'intégration des protocoles dans les applications se fera au déploiement.

- Non intrusion

La configuration d'un protocole de duplication et de cohérence ne doit pas induire de modifications au niveau du code fonctionnel de l'application qui utilise ce protocole. Les fonctionnalités applicatives, encapsulées dans des composants, doivent former une structure de base qui reste indépendante de la gestion de la duplication et de la cohérence. Elle doit rester inchangée quand l'application passe d'une configuration *sans* duplication ni cohérence à une configuration *avec* duplication et cohérence, ainsi que quand l'application passe d'une configuration de protocole à une autre.

Pour que la gestion de la duplication et de la cohérence ne perturbe pas les aspects fonctionnels des applications, nous avons décidé de suivre le principe de séparation d'aspects. Pour cela, nous exploiterons la structure à base de composants des applications considérées et placerons les traitements de duplication et de cohérence dans les conteneurs de composants.

- Facilité de construction

La solution de gestion de la duplication et de la cohérence doit fournir des outils qui facilitent la construction de nouveaux protocoles ainsi que l'évolution de protocoles existants. Ces deux opérations doivent pouvoir réutiliser des briques de protocoles pré-existant. Ces caractéristiques sont nécessaires pour une production rapide et efficace de solutions de duplication et de cohérence répondant à une panoplie sans cesse évoluant des plates-formes d'exécution.

Pour répondre à cet objectif, le processus de construction de protocoles de duplication et de cohérence doit suivre des règles d'organisation qui produisent des structures explicites permettant leur manipulation pour de futures évolutions. Nous avons choisi d'appliquer aux protocoles les principes de gestion de cycle de vie des logiciels à base de composants et de bénéficier de la notion d'architecture et des techniques de gestion de configurations.

- Réutilisation

La solution proposée doit permettre aux protocoles de duplication et de cohérence d'être utilisés au sein de différentes applications. L'intégration et la définition d'une configuration de protocole qui satisfait les besoins de ces applications doivent être faites avec un minimum de modifications des protocoles.

Pour permettre la réutilisation des protocoles, nous considérerons la gestion de la duplication et de la cohérence comme une application à base de composants. Cette application pourra être configurée pour s'exécuter dans différents contextes qui, dans ce cas, correspondront aux contextes d'utilisation applicatifs de la duplication et de la cohérence.

Par rapport aux travaux dans le domaine à composants, nos objectifs correspondent à une gestion de la duplication et de la cohérence en tant que partie de la configuration et de la gestion non fonctionnelle des applications. Le travail de thèse a pour but de respecter la logique compo-

sant et de considérer le cycle de vie complet des logiciels. Cependant, le point de vue adopté est celui de la duplication et de la cohérence et l'objectif est de définir la manière dont la gestion de ces deux aspects doit y être prise en compte.

Par rapport aux travaux dans le domaine de la duplication et de la cohérence, le travail de thèse veut compléter les approches existantes en se focalisant sur la réutilisation et la facilité d'intégration. Nous prêtons une attention particulière aux aspects de construction et d'évolution des protocoles de duplication et de cohérence et, en appliquant l'approche composant, nous avons pour objectif de considérer tout le cycle de vie de ces protocoles.

Chapitre 4

Proposition

Ce chapitre présente notre proposition pour la gestion de la duplication et de la cohérence dans les applications réparties à base de composants.

Dans sa première partie, il décrit la modélisation séparée des applications métier et des protocoles de duplication et de cohérence. Il introduit le modèle de composants utilisé pour la construction des applications métier et montre comment il peut être appliqué pour modéliser la gestion de la duplication et de la cohérence.

Dans sa deuxième partie, le chapitre présente la composition entre les applications métier et les protocoles de duplication et de cohérence. Après avoir décrit les relations entre entités métier et entités de protocole, il traite de l'instrumentation des applications métier pour les besoins de la duplication et de la cohérence et discute la spécialisation des protocoles aux besoins spécifiques des applications métier. Il termine par une description de l'architecture associée à l'exécution.

4.1 Aperçu global de la proposition

Pour modéliser la gestion de la duplication et de la cohérence, nous utilisons une architecture à deux niveaux (Figure 4-1.). Un premier niveau correspond aux applications métier, construites sans duplication, ni cohérence. Le deuxième niveau correspond aux protocoles de duplication et de cohérence, construits sans hypothèses sur les applications. Les deux niveaux sont composés afin de créer des applications métier bénéficiant d'une gestion de duplication et de cohérence.

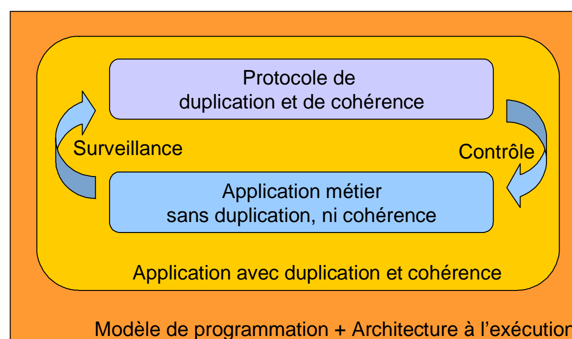


Figure 4-1. Vision globale de la gestion de la duplication et de la cohérence

Le modèle de gestion de la duplication et de la cohérence est inspiré du modèle général de l'administration et ses deux niveaux correspondent aux deux niveaux de ce modèle (cf. Chapitre 1, Section 1.1.4, p. 27). Les applications métier sans duplication, ni cohérence correspondent au niveau de base, alors que les protocoles de duplication et de cohérence correspondent au niveau de contrôle qui administre le niveau de base. La fonction d'administration, basée sur des relations de surveillance et de contrôle, est mise en place par la composition des deux niveaux. Ainsi, les solutions de duplication et de cohérence surveillent l'exécution des applications métier et les contrôlent pour garantir un degré de disponibilité et de correction satisfaisant.

L'originalité de notre proposition réside dans le fait que nous modélisons les protocoles de duplication et de cohérence de la même manière que les applications métier, c'est à dire en tant qu'applications à base de composants. Ils sont construits indépendamment des applications métier et traitent uniquement les aspects de gestion de copies. Les composants de protocole modélisent les composants métier dupliqués, leurs copies, ainsi que des gestionnaires de duplication et de cohérence. Leurs interactions mettent en place les opérations de création, de suppression et de synchronisation de copies.

Plus précisément, notre proposition porte sur les quatre points suivants :

- *Modélisation d'applications métier sans duplication, ni cohérence*. Nous proposons un modèle à composants pour la construction d'applications métier qui n'intègrent pas de gestion de duplication, ni de cohérence. Ce modèle est présenté dans la section 4.2.
- *Modélisation de protocoles de duplication et de cohérence*. La modélisation des protocoles en tant qu'applications à base de composants est faite en utilisant le modèle des applications métier. Ce modèle est spécialisé afin de permettre la définition de composants traitant uniquement des aspects de duplication et de cohérence. Le modèle des protocoles est discuté dans la section 4.3.
- *Composition entre applications métier et protocoles*. La composition entre les applications sans duplication, ni cohérence et les protocoles qui définissent les traitement de duplication et de cohérence est basée sur la logique des systèmes réflexifs (cf. Chapitre 1, Section 1.2.3, p. 34). Les protocoles définissent un niveau méta pour les applications qui, de leur côté, jouent le rôle d'un niveau de base. La composition est présentée dans la section 4.4.
- *Architecture à l'exécution*. Outre les modèles de programmation et de composition, notre proposition porte sur l'architecture à l'exécution des applications métier avec duplication et cohérence. Cette architecture est inspirée des architectures CCM [102] et EJB [136] et est décrite dans la section 4.5.

4.2 Modèle de programmation d'applications métier

Le modèle de composants choisi pour la programmation et la définition des architectures des applications métier est un modèle très général, inspiré des caractéristiques des modèles ODP [99] et CCM [102].

4.2.1 Composants et interfaces

Un composant encapsule du code et des données. Il peut avoir plusieurs interfaces qui sont ses points de communication avec le monde extérieur. Ces interfaces sont caractérisées par les

signatures des méthodes qu'elles définissent et par un sens de communication. Les interfaces *fournies* caractérisent les communications entrantes d'un composant, alors que les interfaces *requises* caractérisent ses communications sortantes.

Le type d'un composant est défini par l'ensemble de ses interfaces requises et fournies. Cet ensemble est *statiquement* défini ce qui signifie que le type d'un composant spécifie toutes les possibles interactions de ses instances à l'exécution.

Considérons l'exemple de l'interface graphique fournie à l'utilisateur dans une application "Hello World" (Figure 4-2.). Cette interface prend en compte des demandes d'affichage de chaînes de caractères et les transforme en requêtes vers un serveur traitant. En tant que composant, elle a des données comme la taille de l'affichage, le nom de l'utilisateur, etc. Ses interfaces comprennent une interface `GUI_itf` fournie aux utilisateurs et une interface requise `Print_itf` pour les interactions avec le serveur.

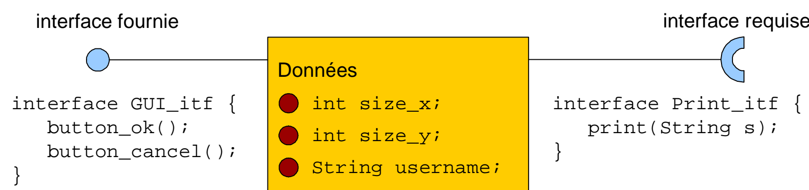


Figure 4-2. Exemple de composant dans notre modèle

4.2.2 Identités, références et noms

Les types de composants sont instanciés pour donner lieu à des entités concrètes manipulées dans les applications. Ces instances sont désignées à l'aide de plusieurs références qui peuvent être des références de composant ou des références d'interfaces.

Une référence de composant caractérise l'identité unique d'une instance de composant. Elle est utilisée pour l'administration et l'introspection de cette instance qui sont détaillées dans la suite. Entre autres, elle donne l'accès aux références d'interfaces.

Une référence d'interface caractérise un point de communication d'une instance de composant. Elle peut désigner aussi bien une interface fournie, qu'une interface requise et est utilisée lors des appels des instances.

Pour faciliter la manipulation des instances de composant et de leurs interfaces, en plus des noms "système" que sont leurs références, nous introduisons des noms symboliques. Ainsi, au sein d'un type de composant, les interfaces ont des noms grâce auxquels il est possible de récupérer leurs références. Quant aux instances de composants constituant une application, elles peuvent être désignées par des noms symboliques à l'aide d'un serveur de noms.

4.2.3 Assemblage et passage de références

La construction d'applications est basée sur l'assemblage d'instances de composant[1]. Cet assemblage est fait par interconnexion d'instances en utilisant leurs interfaces. L'interconnexion est possible si les instances sont compatibles en ce qui concerne leur nombre de

1. Dans la suite, nous abrégerons le terme "instance de composant" en "composant". Toutefois, dans les cas pouvant induire à la confusion, nous utiliserons les termes explicites "type de composant" et "instance de composant".

connexions autorisées, ainsi que les types et les modes de communications de leurs interfaces. Les notions de type d'interface, de mode de communication et de nombre de connexions sont détaillées dans ce qui suit.

Type d'interface

Le type d'une interface désigne l'ensemble des signatures de méthodes qu'elle définit. Pour que deux composants puissent être connectés, ils doivent respectivement disposer d'une interface requise et d'une interface fournie du même type. En d'autres termes, un des composants doit fournir une interface qui est requise par l'autre composant. La connexion entre ces deux interfaces signifie que tout appel sortant, effectué par l'interface requise du premier composant, aboutira en un appel entrant vers l'interface fournie du deuxième. Dans l'exemple de "Hello World" (Figure 4-2.), le composant pour l'interface requiert l'interface `Print_itf` et ne peut être connecté au serveur traitant que si ce dernier fournit `Print_itf` (Figure 4-3.a.).

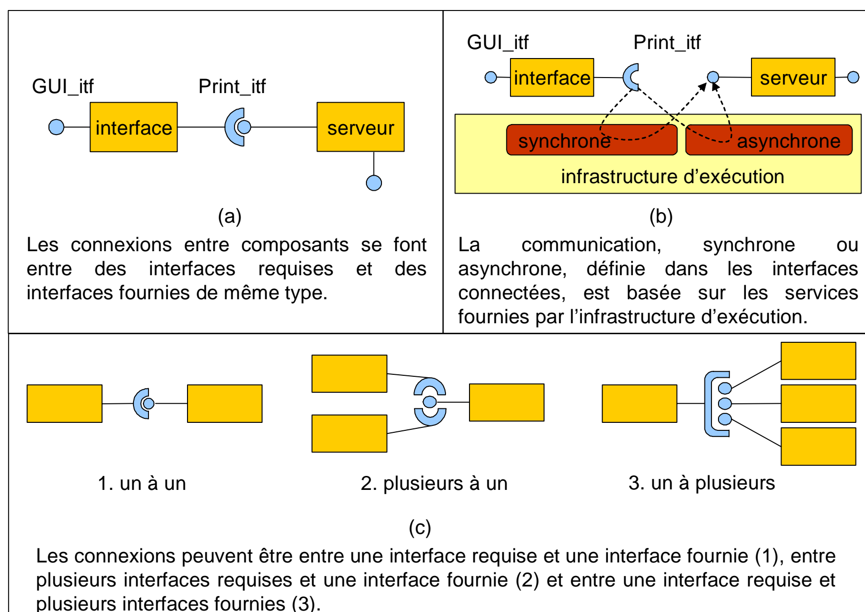


Figure 4-3. Interconnexion d'instances de composants

Mode de communication

La communication entre composants peut être synchrone ou asynchrone. Le modèle synchrone est le modèle client-serveur, alors que le modèle asynchrone est un modèle producteur-consommateur. La spécification des interfaces en tant que synchrones ou asynchrones est faite lors de la déclaration des types des composants.

- *Modèle synchrone*. Dans le modèle synchrone, la communication se fait entre une interface requise d'un composant, appelé *client*, et une interface fournie d'un autre composant, appelé *serveur*. La connexion de deux telles interfaces signifie que tous les appels d'un client à travers son interface requise seront traités par les services implémentes par le serveur.
- *Modèle asynchrone*. Le modèle asynchrone considéré est un modèle *push* dans lequel des composants *producteurs* produisent des événements et les transmettent à des composants *consommateurs*. Dans ce modèle, la connexion se fait entre une interface requi-

se de producteur d'événements et des interfaces fournies de consommateurs d'événements.

La gestion du synchronisme et l'acheminement des appels restent transparents au niveau applicatif. Les composants ne manipulent dans leur code que les références d'interfaces et leur communications reposent sur les services assurés par l'infrastructure d'exécution (Figure 4-3.b).

Nombre de connexions par interface

Dans une majorité de plates-formes, les connexions entre composants suivent le modèle client-serveur et sont de type "un à un" ou "plusieurs à un". Ils correspondent respectivement aux cas où un seul ou plusieurs clients font appel aux services d'un serveur. Dans notre modèle, en plus de ces deux possibilités, nous définissons également les connexions "un à plusieurs" (Figure 4-3.c). Dans ce cas, un composant peut contacter, par le biais d'une interface requise, plusieurs composants ayant l'interface fournie correspondant. Ce type de connexions se prête bien aux diffusions asynchrones telles le schéma publication-abonnement. Elles peuvent également être appliquées dans le cas synchrone pour, par exemple, contacter plusieurs composants et bénéficier de la réponse la plus rapide.

Connexions et références

Dans notre modèle, les composants disposent de deux interfaces prédéfinies : une interface d'introspection et une interface de connexion (Figure 4-4.). L'interface d'introspection sert à donner accès aux interfaces d'une instance et permet de faire la correspondance entre un nom d'interface et sa référence. L'interface de connexion définit les primitives qui sont utilisées pour toute opération d'établissement ou de destruction de connexion entre composants. Ces primitives manipulent les références d'interfaces obtenues via l'interface d'introspection.

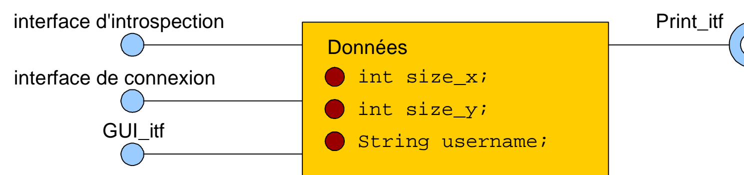


Figure 4-4. Interfaces prédéfinies d'introspection et de connexion

L'utilisation de primitives de connexion explicites différencie notre modèle de modèles comme RMI [139] où JavaPod [18] où l'établissement d'une connexion est automatiquement déclenché par la réception d'une référence. L'intérêt de telles primitives explicites est la connaissance à tout instant de l'état de connexion des composants.

Avant d'établir une connexion, les primitives de connexion vérifient la compatibilité des interfaces à connecter. La vérification porte sur le type, le mode de communication, ainsi que sur le nombre de connexions autorisées des interfaces. Par exemple, dans le cas d'une connexion client-serveur qui est "un à un", un client déjà connecté ne pourra pas se connecter à un autre serveur. Après une vérification réussie, la connexion est établie en enregistrant les références des interfaces interlocutrices au sein des composants. Lors d'une déconnexion, ces enregistrements sont effacés.

Les références de composant ou les références d'interfaces peuvent être passées en paramètre de méthode. Cependant, leur réception ne provoque pas une connexion automatique. Pour qu'une telle connexion soit établie, il faut utiliser les primitives dédiées mais, surtout, il

faut que le composant, par la définition statique de ces interfaces requises, ait déclaré son intention de connexion vers ce type d'interface. Encore une fois, les connexions des composants sont contrôlées et leur dépendances sont rendues explicites.

4.3 Protocoles de duplication et de cohérence

Un protocole de duplication et de cohérence est modélisé comme une application répartie à base de composants où sont représentés toutes les entités et toutes les interactions liées à la gestion de copies. Ces entités et interactions sont modélisées respectivement par les composants des protocoles et leurs interactions.

Les composants d'un protocole modélisent les composants métier dupliqués, leurs copies, ainsi que les composants métier qui font appel aux composants dupliqués. Ils modélisent également les gestionnaires de copies et les services utilisés par les copies.

La modélisation de différents types d'entités est faite en définissant différents types de composants de protocole. Ainsi, les composants métier dupliqués et leurs copies sont modélisés par des composants de protocole appelés *composant copies*. Les composants métier qui font des appels sur des composants dupliqués sont modélisés par des *composants clients*. Les gestionnaires de copies sont représentés par des *composants gestionnaires* et, finalement, les services utilisés par les copies sont modélisés par des *composants services*.

Les interactions entre les composants d'un protocole modélisent les opérations de gestion de copies dont la création, la destruction et la synchronisation de copies.

Pour définir les différents types de composants de protocole et les types de leurs interactions, nous utilisons le modèle à composants des applications métier. Nous modélisons les différents paramètres de gestion de la duplication et de la cohérence en reprenant les critères de classification présentés dans l'état de l'art (cf. Chapitre 2). Dans la suite, nous détaillons cette modélisation en considérant, dans l'ordre, la gestion de la duplication (Section 4.3.1), la gestion de la cohérence (Section 4.3.2) et leur configuration (Section 4.3.3).

4.3.1 Composants et duplication

D'après la description donnée dans l'état de l'art (cf. Chapitre 2, Section 2.3.1, p. 41), la gestion de la duplication est caractérisée par une unité de duplication (*Quoi?*), utilise une technique de copie (*Comment?*), choisit un moment de copie (*Quand?*) et décide des règles de placement des copies (*Où?*). Dans la suite, nous présentons notre proposition de modélisation de ces quatre points.

Unité de duplication (*Quoi*)

Au niveau des protocoles, l'unité de duplication est modélisée en tant que composant. Cela veut dire que toute entité métier dupliquée, ainsi que toute copie d'entité dupliquée, est représentée par un composant de protocole. Dans le cas des applications métier à base de composants, où l'unité de duplication est naturellement le composant, ceci se traduit par une réification au niveau des protocoles de tous les composants métier dupliqués et de toutes leurs copies. Les composants de protocole correspondants sont désignés sous le nom de *composants copies*.

Le choix de représenter l'unité de duplication en tant que composant a un impact direct sur la gestion du type de copie et sur les critères de sélection des entités effectivement copiées (cf. Chapitre 2, Unité de duplication (Quoi), p. 42).

- *Type des copies*. Dans l'état de l'art nous avons montré que de nombreuses solutions de duplication ne peuvent pas être réutilisées puisqu'elles sont définies de manière spécifique à leur domaine d'application (cf. Chapitre 2, Section 2.3.3, p. 50). Dans notre proposition, la modélisation des copies en termes de composants permet l'encapsulation des données spécifiques et de ce fait rend les protocoles indépendants des domaines d'application.
- *Sélection des entités à dupliquer*. Dans l'état de l'art nous avons vu que le choix des entités à copier peut être fait selon différents critères et qu'en général les algorithmes de duplication optent pour une solution figée. Dans notre proposition, les protocoles ne modélisent que des entités effectivement copiées et l'application de critères de sélection sur les composants métier est retardée. Elle n'est faite qu'au moment de l'intégration des protocoles dans les applications métier et peut être basée sur tout type de critères.

Technique de copie (*Comment*)

Dans l'état de l'art, nous avons considéré les techniques de copie en fonction de la structure et de l'état de l'entité copiée (cf. Chapitre 2, *Technique de copie (Comment)*, p. 43).

- *Structure de copie*. Nous ne dupliquons que des composants dont la structure est indivisible. Dans le cas où une application métier manipule des données de différents niveaux de granularité, nous supposons que ces données sont explicitement modélisées sous forme de composants dont les connexions représentent les relations de hiérarchie.
- *État de copie*. Dans notre modélisation nous avons décidé de nous focaliser sur la capture et la restauration d'états passifs et d'ignorer les états d'exécution courants. En effet, ainsi nous évitons la complexité de gestion liée non seulement à la capture des états d'exécution mais également à la gestion de la cohérence entre de tels états.

Pour permettre la capture et la restauration d'état d'un composant *copie* de protocole, nous définissons trois interfaces de composant : une interface de capture et de restauration, une interface d'accès à l'état et une interface de contrôle d'état stable[1].

- *Interface de capture et de restauration*. Cette interface fournit les primitives nécessaires pour la capture et la restauration de l'état d'un composant. Par défaut, ces primitives concernent l'état global d'un composant mais peuvent être surchargées pour ne considérer qu'un état partiel. Elles sont spécifiques aux types de composants puisqu'elles reposent sur des méthodes d'accès à leur état appartenant à une interface d'accès à l'état.
- *Interface d'accès à l'état*. Pour pouvoir capturer ou restaurer l'état d'un composant, son état doit être accessible. Nous choisissons de rendre cet état disponible non par des fonctions internes du support d'exécution mais par une interface explicite d'accès à l'état qui est fournie par les composants.
- *Interface de contrôle d'état stable*. L'interface de contrôle d'état stable est chargée de préserver la correction d'un composant lors des traitements de capture et de restauration. Elle a pour rôle d'assurer que le composant n'est pas modifié pendant ces traitements et de garantir la manipulation d'un état cohérent. En effet, si un composant est modifié pendant une capture de son état, la copie créée risque de ne pas refléter la modification en entier. Si un composant est modifié lors d'une restauration de son état, la

---

1. Ces interfaces sont définies au niveau des composants de protocole même si, après l'intégration de ces protocoles dans des applications, elles correspondront à des interfaces implémentées au niveau des composants métier.

modification risque d'être effectuée sur un état non cohérent.

Pour garantir la manipulation d'un état cohérent, l'interface fournit des primitives pour l'exclusion mutuelle entre les opérations de capture/restauration et les opérations de modification. Ces primitives concernent l'espionnage des composants pour connaître leur état de modification, le lancement de processus de capture/restauration et la mise en attente d'opérations de modification.

Moment de copie (*Quand*)

Dans l'état de l'art nous avons vu qu'il est possible d'appliquer différentes règles pour la création et la suppression de copies. Dans notre modélisation, nous permettons la mise en place de différentes stratégies en gérant la création et la suppression de copies sous forme de reconfigurations des architectures de composants des protocoles. Nous permettons la définition de différentes règles de reconfiguration qui définissent les actions nécessaires à la création ou à la suppression de composants copies.

Dans la suite nous détaillons les différents types de règles de reconfiguration que nous proposons pour la gestion de copies[1]. Nous définissons ces règles par rapport aux modèles de protocoles ou alors par rapport aux architectures obtenues après l'intégration de ces protocoles dans des applications métier. Les règles définies par rapport aux modèles sont génériques, alors que celles définies par rapport aux architectures instanciées sont spécifiques. Ces règles peuvent être des règles locales et porter sur un seul composant ou alors être des règles globales et porter sur plusieurs composants.

(i) Modèle et architecture instanciée d'un protocole

Le modèle d'un protocole de duplication et de cohérence définit ce dernier en termes de types de composants et de connexions possibles entre des instances de ces types. Le nombre exact d'instances n'est pas spécifié puisqu'il dépend de la manière dont ce modèle de protocole est instancié et intégré au sein d'une application métier. Nous désigneront les modèles de protocoles sous le nom d'architectures abstraites.

L'architecture instanciée d'un protocole est obtenue à partir du modèle du protocole après l'intégration d'un protocole au sein d'une application métier. Elle reflète la décision d'instanciation des différents types des composants de protocole et d'interconnexion de ces instances. Nous désignerons les architectures instanciées sous le nom d'architectures concrètes.

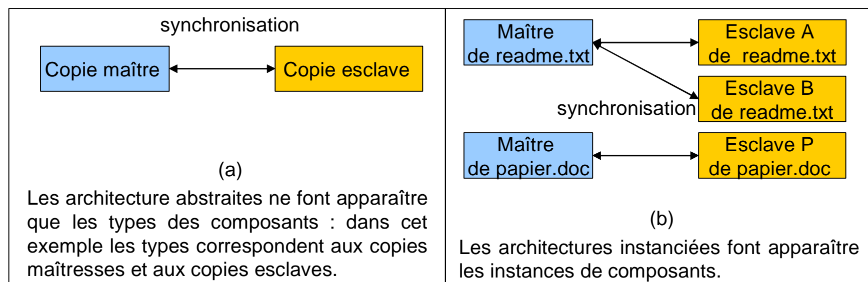


Figure 4-5. Architectures abstraites et instanciées

Pour illustrer les notions d'architectures abstraites et concrètes, considérons l'architecture maître-esclave (Figure 4-5.a). Cette architecture définit un modèle d'interaction entre deux

1. En effet, la synchronisation de copies peut également être vue comme une reconfiguration.

types de composants, notamment le type maître et le type esclave, et est donc une architecture abstraite. Elle peut être instanciée avec plusieurs maîtres et plusieurs esclaves par maître. Par exemple, l'instanciation peut être faite pour l'intégration du protocole dans une application de gestion de fichiers et donner lieu à un maître d'un fichier `readme.txt`, d'un maître d'un autre fichier `papier.doc`, de deux esclaves de `readme.txt` et d'un esclave de `papier.doc` (Figure 4-5.b).

(ii) Règles de reconfiguration

Nous distinguons quatre types de règles : les génériques, les spécifiques, les locales et les globales. Les deux premiers types sont relatifs aux architectures abstraites ou instanciées des protocoles, alors que les deux derniers sont liés au nombre de composants sur lesquelles elles portent.

Les règles génériques et spécifiques sont caractérisées de la manière suivante :

- *Règles génériques*. Dans le cas des architectures abstraites, les règles de reconfiguration sont génériques puisqu'elles ne peuvent être définies qu'en termes de types de composants et d'informations disponibles au niveau de ces types. Un exemple de règle générique qui s'applique au maître-esclave est la règle disant "Pour les composants de type maître, à la réception d'un événement de type E, créer copie esclave".
- *Règles spécifiques*. Dans le cas des architectures instanciées qui sont spécifiques à un assemblage entre un protocole et une application métier, les règles de reconfiguration peuvent prendre en compte la configuration des instances et des interconnexions qui est propre à l'assemblage en question. Un exemple de règle spécifique est la règle "Quand l'esclave B de `readme.txt` reçoit un appel d'invalidation M de la part du maître, supprimer cet esclave".

Les règles locales et globales sont caractérisées de la manière suivante :

- *Règles locales*. Les règles locales sont des règles de reconfiguration qui portent sur un seul composant. Elles sont définies à l'aide d'une *interface de reconfiguration* qui est fournie par les composants et qui permet d'enregistrer un traitement de réaction pour un événement (Figure 4-6.a). Elles peuvent être définies aussi bien au niveau des modèles des protocoles qu'au niveau de leurs architectures instanciées. En effet, les deux règles données en exemple dans les paragraphes précédents sont des règles locales puisqu'elles portent respectivement sur le composant de type maître et le composant esclave B. Dans le premier cas, l'interface de reconfiguration sera utilisée pour définir la règle locale au niveau de toutes les instances maîtres, alors que dans le deuxième cas, elle ne sera utilisée que pour le composant esclave B.
- *Règles globales*. Les règles locales ne peuvent porter que sur des informations disponibles au niveau d'un composant. Elles ne permettent pas de définir des réactions pour des événements reçus par d'autres composants. Cette limitation est levée par les règles globales qui permettent de définir des reconfigurations portant sur des assemblages de composants. Ces règles sont définies par des composants de protocole spécifiques qui sont des *gestionnaires de configuration*.

  Nous avons restreint les règles globales aux cas d'architectures instanciées des protocoles. En effet, nous avons voulu éviter une définition complexe de règles génériques portant sur un ensemble de composants et devant prévoir toutes les instanciations possibles de cet ensemble. Par conséquent, les règles globales que nous considérons sont uniquement des règles spécifiques.

  La définition des règles globales de reconfiguration est faite en utilisant l'interface

de reconfiguration au niveau du gestionnaire. Pour effectuer les traitements de reconfiguration sur les assemblages de composants, ce dernier référence toutes les instances de l'architecture instanciée à laquelle il est rattaché (Figure 4-6.b).

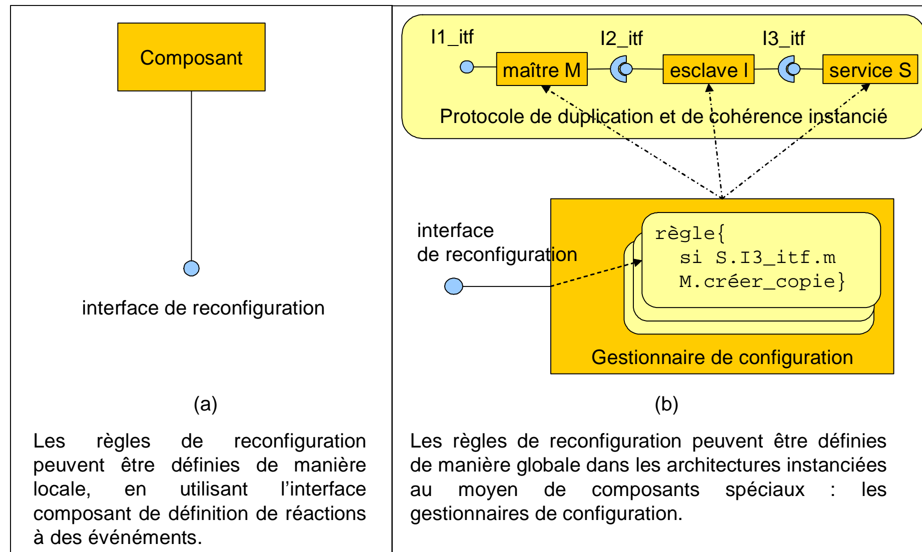


Figure 4-6. Règles de reconfiguration

Placement de copie (*Où*)

Comme pour les règles de création et de suppression de copies, les règles de placement peuvent être définies au niveau des architectures abstraites ou instanciées. Dans le premier cas, les emplacements ne sont pas directement désignés mais sont définis au moyen de leurs caractéristiques telles l'état du réseau ou la place mémoire. Dans le deuxième cas, les emplacements peuvent être désignés directement étant donné qu'il s'agit d'un protocole s'exécutant sur un ensemble d'emplacements connus. C'est ce deuxième cas que nous considérons dans notre proposition.

4.3.2 Composants et cohérence

Notre modélisation de la cohérence ne se place pas à un niveau aussi détaillé que celui de la classification des travaux autour de la cohérence présentée dans l'état de l'art (cf. Chapitre 2, Section 2.3.2, p. 45). En effet, notre objectif d'adaptabilité nous a conduit à proposer une structuration à plus gros grain où la cohérence est spécifiée en termes de perception d'événements relatifs à la cohérence, de traitements locaux de cohérence et de synchronisation entre copies. Ces traitements sont modélisés par des interfaces spécialisées qui ne sont pas figées mais varient en fonction des protocoles.

Avant de détailler la structuration proposée, nous introduisons les composants clients et les composants services qui sont des composants de protocole impliqués dans la gestion de la cohérence.

Composants clients et composants services

Les composants qui constituent un protocole de duplication et de cohérence incluent, en plus des composants copies et des gestionnaires de configuration, des composants clients et des composants services.

Les composants clients modélisent les composants métier qui ont des interactions avec des composants métier dupliqués. Étant donné que ces derniers sont modélisés par les composants copies d'un protocole, les interactions au niveau métier sont modélisées par des connexions entre les composants clients et les composants copies au niveau du protocole. La modélisation des clients et de leurs interactions avec les copies est nécessaire puisque ces interactions modifient les copies et par conséquent influencent leur relation de cohérence[1].

Pour illustrer ce point, considérons de nouveau l'exemple de l'application de gestion de fichiers dont il a été question précédemment (Figure 4-5.). Supposons que le fichier `readme.txt` est utilisé à travers un composant d'édition. Le fichier `readme.txt` est modélisé au niveau du protocole par un composant maître et par plusieurs composants esclaves qui sont des composants copies. Le composant d'édition est modélisé par un composant client. L'interaction entre le composant d'édition et le fichier est représentée par une connexion au niveau du protocole (Figure 4-7.).

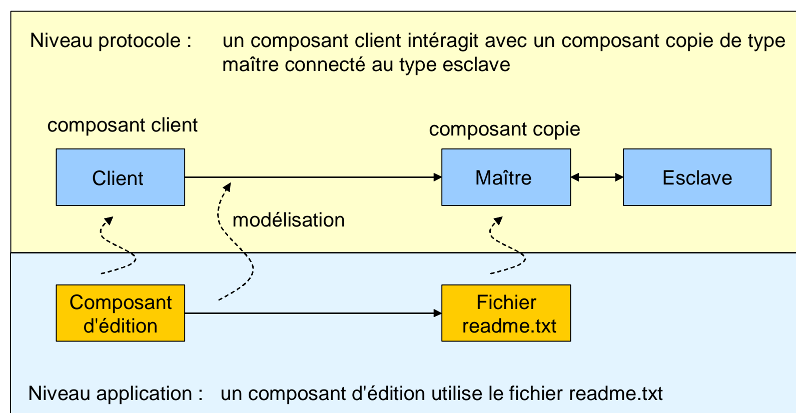


Figure 4-7. Composants clients

Les composants services fournissent des traitements génériques nécessaires à la gestion de la cohérence. Ils peuvent implémenter des traitements tels que la gestion d'ordre et la journalisation, ou espionner l'état du système d'exécution et fournir des informations comme la qualité des connexions réseau. Ils sont indépendants des spécificités des protocoles et peuvent donc être réutilisés.

Perception d'événements relatifs à la cohérence

Les événements de cohérence sont tous les événements au sein d'une application qui nécessitent une synchronisation entre copies. Ces événements recouvrent les modifications de composants copies par les composants clients, les pannes de copies, les créations de nouvelles copies, les demandes de synchronisation explicites, etc.

1. Il est possible que dans une application métier un composant dupliqué interagisse avec un autre composant dupliqué. Dans ce cas, le premier composant sera modélisé au niveau d'un protocole par un composant qui sera copie et client à la fois. En d'autres termes, les deux types de composants de protocole ne sont pas exclusifs.

La perception des événements de cohérence consiste à porter ces événements à la connaissance des composants qui fournissent les traitements de cohérence correspondant. Dans notre modélisation, ceci est fait en interconnectant les composants qui produisent de tels événements et les composants qui les traitent (Figure 4-8.).

- *Composants produisant des événements de cohérence*. Les événements relatifs à la cohérence peuvent être produits par les composants copies et les composants services. Les composants clients peuvent modifier l'état des copies et ainsi créer des situations nécessitant une synchronisation. Les composants services peuvent apporter la connaissance sur un événement de l'environnement d'exécution comme une dégradation des connexions réseau nécessitant la création et donc la synchronisation de copies.
- *Composants traitant les événements de cohérence*. Les composants qui peuvent traiter les événements de cohérence sont les composants copies, les composants clients et les composants gestionnaires. Leur traitements peuvent concerner tout type de reconfiguration liée à la création, à la destruction ou à la synchronisation de copies.
- *Interconnexions permettant la perception d'événements de cohérence*. Pour permettre la perception des événements de cohérence, les composants qui les produisent sont connectés aux composants qui les traitent. Ces interconnexions transmettent les événements de cohérence et relient, comme dans le modèle de composant métier, des interfaces fournies et des interfaces requises, appelées interfaces de perception.

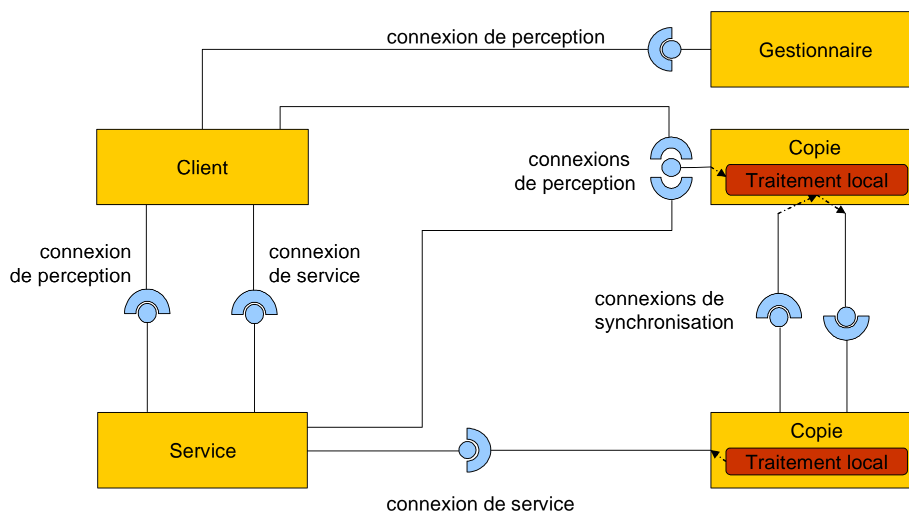


Figure 4-8. Modèle général de gestion de cohérence

Traitements locaux de cohérence

Les traitements locaux de cohérence sont tous les traitements relatifs à la gestion de la cohérence effectués sur une copie ou sur un client.

Sur une copie, les traitements locaux sont ceux qui assurent la correction de la copie et qui prennent les décisions d'ordre d'exécution (cf. Chapitre 2, Section 2.3.2, p. 45). Ce sont, par exemple, la détection de conflits entre opérations, l'enregistrement d'opérations dans un historique, etc.

Sur un client, les traitements locaux sont ceux qui rajoutent aux appels vers les copies l'information qui caractérise ces appels du point de vue de la cohérence. Par exemple, il peut s'agir de calculs d'estampilles qui indiquent l'ordre d'exécution d'opérations qui est demandé

par un client. Un tel ordre définit les critères de correction des copies du point de vue de ce dernier.

Les traitements locaux peuvent inclure, en plus de calculs spécifiques du protocole de cohérence, des traitements fournis par les composants services. Pour que les composants clients et les composants copies puissent utiliser les fonctions de ces composants services, ils ont besoin d'interconnexions correspondantes. Ces *interconnexions de service* (Figure 4-8.), permettent l'intégration de fonctionnalités génériques pour les besoins spécifiques du protocole de cohérence.

Synchronisation entre copies

La synchronisation entre copies implique des échanges entre elles et est donc basée sur des interconnexions de copies. Ces interconnections, appelées *interconnexions de synchronisation,* connectent des interfaces de cohérence spécifiques aux protocoles utilisés. Elles peuvent concerner des échanges spécifiques au protocole ou alors reposer entièrement sur les interfaces génériques des composants services.

4.3.3 Configuration des protocoles

Dans le chapitre sur la duplication et la cohérence, nous avons classifié les travaux ayant des objectifs de configuration selon trois critères : le niveau de mise en œuvre, la transparence de gestion et le moment d'intégration (cf. Chapitre 2, Section 2.4, p. 52). Dans la suite nous présentons la manière dont nous nous plaçons par rapport à ces critères dans notre proposition.

Niveau de mise en œuvre

Nous avons décidé de fournir un support pour la gestion de la duplication et de la cohérence au niveau intergiciel. En effet, comme il a été montré dans l'état de l'art, les solutions intergicielles ont plusieurs avantages. Elles permettent d'encapsuler les traitements de gestion de la duplication et de la cohérence et donc de les réutiliser dans différentes applications. Elles permettent, également, de choisir le compromis entre les objectifs de performances, liés à la spécificité des applications, et les objectifs de réutilisation qui sont des objectifs de généricité.

Transparence

Nous visons une séparation entre les traitements métier des applications et les traitements de gestion de la duplication et de la cohérence des protocoles. Par conséquent, nous nous orientons vers une approche de transparence qui s'applique aussi bien aux applications qu'aux protocoles. Toutefois, pour permettre l'intégration des protocoles dans les applications, nous introduisons des règles de composition qui reposent sur l'instrumentation des applications et sur la spécialisation des protocoles. Le premier point peut être vu comme une adaptation de la structure des applications pour permettre l'intégration des protocoles. Le deuxième point permet l'utilisation des traitements génériques des protocoles dans les cas spécifiques de différentes applications métier. Ces deux points sont détaillés dans la section sur la composition (Section 4.4).

Moment d'intégration

Notre choix, motivé par l'objectif de souplesse, s'est porté sur une intégration tardive lors du déploiement. En effet, le déploiement d'une application métier définit son architecture

initiale d'exécution et permet l'instanciation d'un protocole en fonction de cette architecture. De plus, le protocole peut être paramétré pour refléter l'environnement d'exécution qui est également caractérisé lors du déploiement.

4.3.4 Synthèse

Nous modélisons les protocoles de duplications et de cohérence en tant qu'applications à base de composants. Nous utilisons une spécialisation du modèle des applications métier où nous définissons quatre types de composants : les copies, les clients, les services et les gestionnaires.

- *Composants copies*. Ces composants modélisent les composants métier dupliqués, ainsi que leurs copies. L'utilisation du composant comme représentant de l'unité de duplication permet l'encapsulation de toutes les informations spécifiques et la définition de protocoles indépendants des domaines d'application.
- *Composants clients*. Les composants clients modélisent les composants métier qui ont des interactions avec des composants dupliqués. Les clients sont des composants qui déclenchent des reconfigurations de l'ensemble de copies. Ils peuvent déclencher des créations, des destructions ou des synchronisations de copies.
- *Composants services*. Les composants services fournissent des traitements génériques pouvant être réutilisés dans différentes protocoles. Ces traitements peuvent être liés à la cohérence ou alors servir d'espions de l'environnement d'exécution.
- *Composants gestionnaires*. Les composants gestionnaires contrôlent l'architecture des protocoles. Ils sont responsables des reconfigurations liées aux créations, aux suppressions et aux synchronisations de copies.

Les quatre types de composants sont interconnectés par des interfaces de reconfiguration, de perception, de synchronisation et de service. Ils sont également caractérisés par les interfaces prédéfinies de navigation et de connexion, définies dans le modèle principal (cf. Section 4.2.3). Ils disposent d'interfaces d'accès à l'état et de contrôle d'état stable.

- *Interface de reconfiguration*. L'interface de reconfiguration est utilisée pour définir des règles de reconfiguration. Elle sert à faire la correspondance entre événements significatifs et les réactions des protocoles pour ces événements.
- *Interface de perception*. L'interface de perception est utilisée pour transmettre les informations annonçant un besoin de reconfiguration.
- *Interface de synchronisation*. L'interface de synchronisation caractérise les échanges entre copies lors d'une synchronisation.
- *Interface de service*. L'interface de service caractérise les traitements fournis par un composant de service.
- *Interface d'accès à l'état*. L'interface d'accès à l'état est caractéristique aux composants copies et sert de base pour l'interface de capture et de restauration d'état, elle-même essentielle pour la création et la synchronisation de copies.
- *Interface de contrôle d'état stable*. L'interface de contrôle d'état stable accompagne l'interface d'accès à l'état et garantit la correction du processus de capture et de restauration d'état.

Un récapitulatif des différentes interfaces et de leur répartition sur les quatre types de composants est donné sur la Figure 4-9.

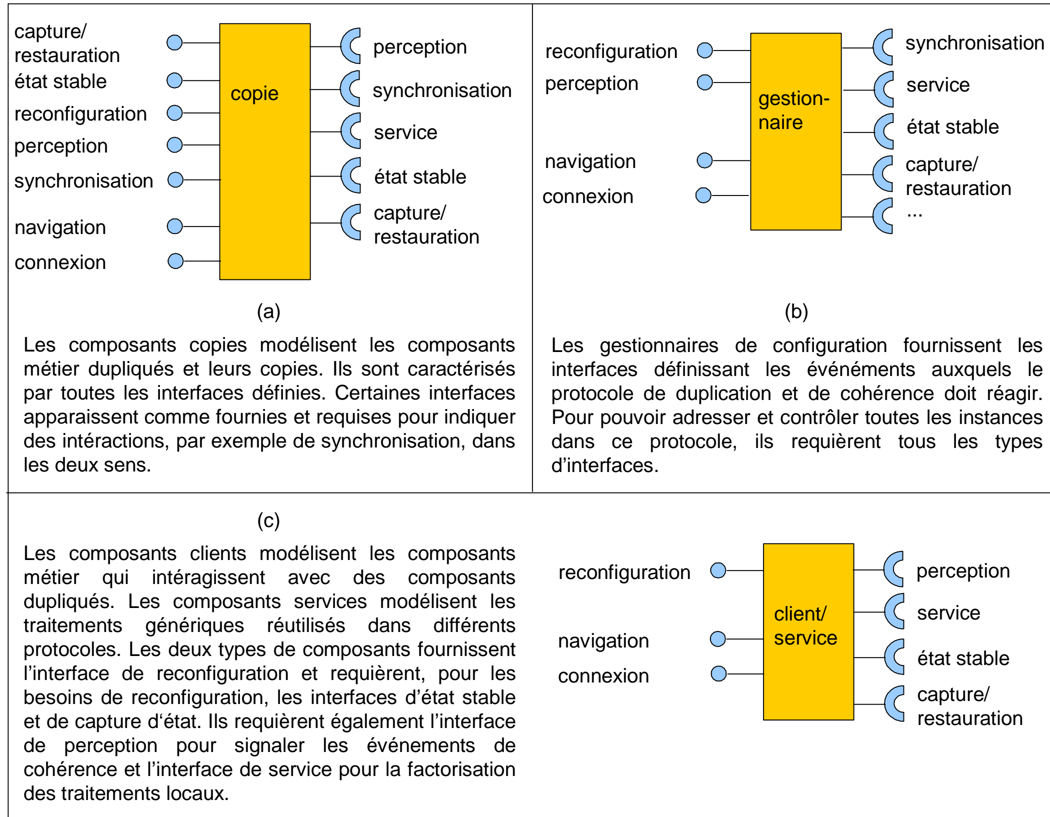


Figure 4-9. Les quatre types de composants de protocole et leurs interfaces

4.4 Composition entre applications métier et protocoles

Nous définissons la composition entre les applications métier, construites sans duplication ni cohérence, et les protocoles, construits sans considérations applicatives, en suivant la logique des systèmes réflexifs. Un protocole de duplication et de cohérence est défini en tant que méta-niveau pour une application métier et ses opérations d'administration sont définies par les relations de réification et de réflexion (cf. Chapitre 1, Section 1.2.3, p. 34).

Les relations entre le niveau de protocole et le niveau applicatif sont présentées dans la première partie de cette section (Section 4.4.1). La section 4.4.2 discute les adaptations qui doivent être faites au niveau de l'application métier pour permettre l'intégration d'un protocole de duplication et de cohérence. Finalement, la section 4.4.3 décrit les possibilités de spécialisation des protocoles en fonction des besoins des applications métier.

4.4.1 Relations entre application métier et protocole

Les architectures des applications métier, ainsi que des protocoles de duplication et de cohérence, sont définies en termes de composants, d'interfaces et d'interconnexions. Dans le cas des

applications métier, nous parlons d'*entités métier*, alors que dans le cas des protocoles, nous parlons d'*entités de protocole*. Nous définissons les relations entre ces entités au niveau des architectures, au niveau des composants, ainsi qu'au niveau des interfaces.

Relations entre architectures

L'intégration de protocoles au sein des applications métier pose le problème d'intégration de plusieurs protocoles au sein d'une même application métier. Ce cas se présente, par exemple, dans le DNS [16] qui fait simultanément usage d'un protocole de cache et d'un protocole de tolérance aux fautes. En termes de relations entre niveau méta et niveau de base, cette situation se traduit par des liens entre *une* architecture métier et *plusieurs* architectures de protocole. Or, si les architectures de protocole sont modélisées de manière indépendante et contrôlent les mêmes entités métier, elles risquent d'introduire des problèmes d'incohérence.

Pour éviter ce problème, nous considérons uniquement les relations entre *une* architecture de protocole et *une* architecture métier. En d'autres termes, sont traités uniquement les cas d'intégration d'un seul protocole au sein d'une application métier. Ce choix n'empêche pas l'intégration de plusieurs protocoles non concurrents. En effet, dans ce cas, les relations entre architecture métier et architectures de protocoles peuvent être établies chacune à son tour et de manière indépendante, la garantie d'indépendance étant de la responsabilité de l'intégrateur de protocoles. Dans le cas de protocoles concurrents, il sera de toute manière nécessaire de considérer les interférences au niveau méta dont la modélisation produira une nouvelle architecture de protocole à intégrer à l'architecture métier.

Relations entre composants

Étant donné que les composants services et les gestionnaires de configuration ne fournissent que des traitements de duplication et de cohérence, des quatre types de composants de protocole, seulement les composants clients et les composants copies réifient des composants métier. Le contrôle exercé par un type de composant de protocole peut porter sur un ou plusieurs composants métier. Le contrôle exercé sur un composant métier peut provenir d'un ou de plusieurs types de composants de protocole. Ces deux points sont discutés dans ce qui suit.

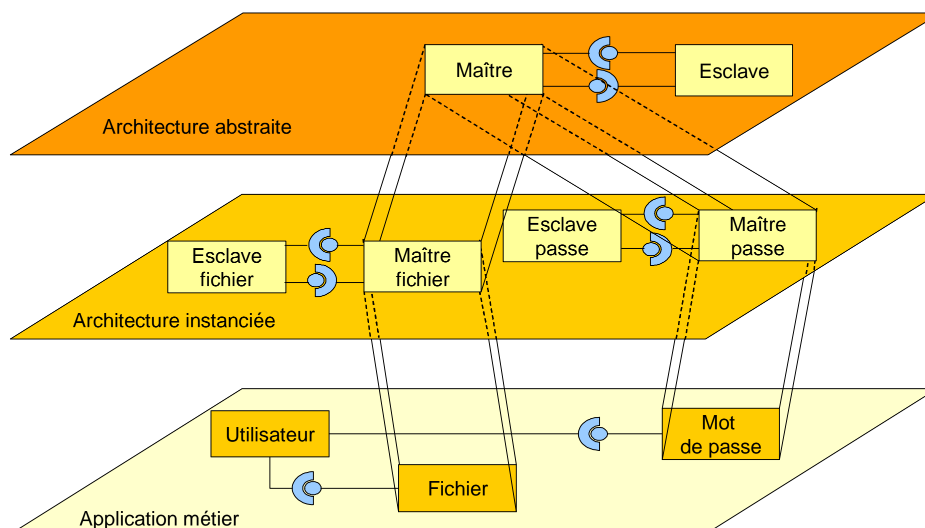


Figure 4-10. Correspondance de composants : de protocole à métier

(i) Composants métier contrôlés par un composant de protocole

Un protocole de duplication et de cohérence est défini en tant qu'architecture abstraite à base de types des composants (cf. Section 4.3.1). Lors de l'intégration d'un protocole au sein d'une application métier, les types de composants de protocole peuvent être instanciés plusieurs fois et donner lieu à des instances de protocole qui contrôlent plusieurs instances métier. Par conséquent, un type de composant de protocole peut définir les traitements de contrôle de plusieurs types de composants métier.

Prenons l'exemple de l'architecture abstraite maître-esclave et de son instanciation dans une application de gestion de fichiers et de mots de passe. La relation maître-esclave peut caractériser non seulement les copies de fichiers, mais également les copies des mots de passe. Le type maître sera instancié de manière multiple afin de contrôler les maîtres fichiers, ainsi que les maîtres des mots de passe (Figure 4-10.).

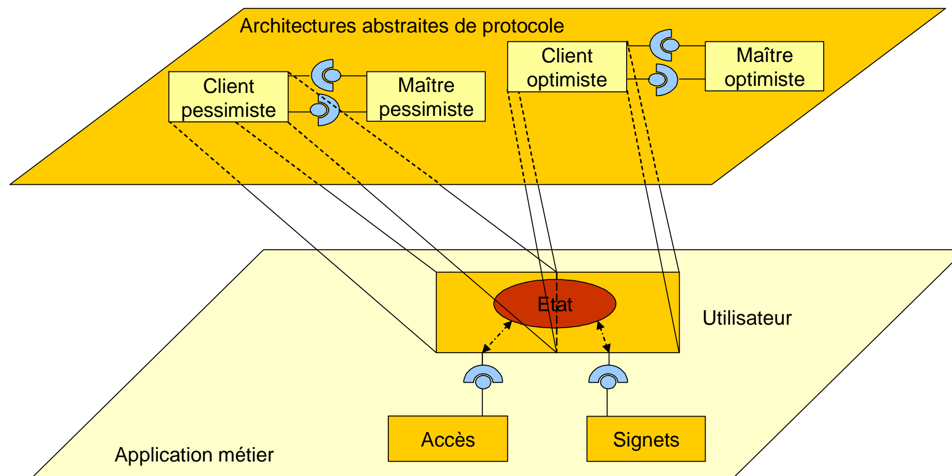


Figure 4-11. Correspondance entre composants : de métier à protocole

(ii) Composants de protocole contrôlant un composant métier

Si la correspondance entre un type de composant de protocole et plusieurs types de composants métier semble naturelle, la correspondance inverse, entre un type de composant métier et plusieurs types de composants de protocole, pose plus de problèmes. Comme dans le cas des architectures, cette correspondance introduit des problèmes de concurrence relatifs à l'état des composants.

Pour éviter le problème de manipulation incohérente des instances de composants métier, nous adoptons la solution appliquée dans le cas des architectures, c'est à dire nous ne considérons que les relations "un à un". Toutefois, nous permettons le rattachement de plusieurs composants de protocole dans la cas de manipulations non conflictuelles. La situation est équivalente à plusieurs correspondances "un à un" où les composants de protocoles manipulent une partie bien définie de l'état des composants métier et où leur contrôle est limité à un ensemble des interfaces métier.

Pour illustrer ce point, considérons l'exemple d'une application où un utilisateur consulte deux bases de données : une qui contient des informations critiques et une qui gère des informations personnelles (Figure 4-11.). Les informations critiques peuvent être les droits d'accès à des dossiers confidentiels, alors que les informations personnelles peuvent être des signets de

navigation. Le protocole de cohérence dans le premier cas est un protocole pessimiste, alors que celui dans le deuxième cas est un protocole optimiste.

Les deux protocoles font apparaître deux types de composants clients. Ces deux types de composants doivent contrôler le même composant métier utilisateur puisqu'il est impliqué dans les deux protocoles. Or, le comportement de l'utilisateur est défini non seulement par son état mais aussi par les interactions qu'il a avec d'autres composants. Les deux protocoles peuvent donc gérer ce même utilisateur s'ils ne manipulent pas les mêmes variables d'état et s'ils ne contrôlent pas les mêmes interactions.

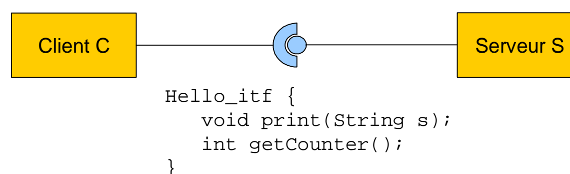
Relations entre interfaces

Les applications métier sont construites sans duplication ni cohérence et sont caractérisées par des interfaces métier. Les protocoles sont, quant à eux, caractérisés par des interfaces liées uniquement à la gestion de la duplication et de la cohérence. Ces interfaces comptent des interfaces de perception, de reconfiguration, de gestion d'état, de synchronisation et de service. Les interfaces de reconfiguration, de service, de synchronisation et de contrôle d'état stable restent exclusivement liées aux traitements de duplication et de cohérence. Ce sont les interfaces de perception et d'accès à l'état qui réifient les interfaces métier.

- *Interfaces d'accès à l'état*. Les interfaces d'accès à l'état réifient des interfaces qui donnent accès à l'état encapsulé des composants et qui de ce fait permettent les traitement de duplication et de synchronisation. Ils font partie, par conséquent, de l'instrumentation des applications métier pour les besoins de gestion des protocoles.
- *Interfaces de perception*. Les interfaces de perception permettent d'interpréter les interactions métier et de décider si ces interactions ont une incidence sur la gestion du protocole. Elles caractérisent les interactions métier en tant que des opérations sans importance pour la duplication et la cohérence ou en tant que des opérations qui nécessitent des reconfigurations de copies. En faisant cela, elles permettent la détection des événements qui doivent être traités par le protocole.

  Considérons l'exemple de l'application classique “Hello World”, constituée de deux instances de composant : un client et un serveur (Figure 4-12.). Le serveur fournit une interface `Hello_itf` qui permet au client d'afficher des chaînes de caractères. Il tient compte du nombre d'appels du client et lui permet de consulter ce nombre. Le client peut donc effectuer deux opérations sur le compteur du serveur : il peut le modifier, en appelant la méthode d'affichage `print(String s)`, ou alors le consulter, en appelant la méthode `getCounter()`.

  Imaginons que l'application “Hello World” est gérée par un protocole de gestion de cache portant sur le serveur. Pour ce protocole, une consultation du compteur ne modifie pas l'état de la copie qui est en cache et est donc sans importance pour le protocole. Au contraire, une modification du compteur nécessite une synchronisation de la copie de cache avec le serveur initial et est, par conséquent, perçue et traitée par le protocole.




```
Hello_itf {
   void print(String s);
   int getCounter();
}
```

Figure 4-12. L'application “Hello World”

4.4.2 Instrumentation de l'application métier

Pour permettre l'intégration des protocoles au sein des applications métier, ces dernières doivent être instrumentées. Elles doivent permettre aux protocoles de les observer et de les administrer en rendant leur fonctionnement explicite et en fournissant des mécanismes pour leur contrôle. L'instrumentation qu'elles doivent subir dépend, d'une part, du type de gestion effectuée par les protocoles et, d'autre part, de la partie de leur fonctionnement révélée par leur modèle de programmation.

Dans notre proposition, la gestion effectuée par les protocoles est spécifiée par leurs interfaces. Quant au modèle de programmation des applications métier, il révèle leur architecture en termes d'interconnexions et de types. Par conséquent, les mécanismes de surveillance et de contrôle sont les suivants :

- *Surveillance*. La surveillance du fonctionnement des applications par les protocoles est faite à l'aide des interfaces de perception et ne nécessite pas d'instrumentation additionnelle. En effet, les événements devant être observés sont les interactions métier et les variations de l'environnement d'exécution qui nécessitent des reconfigurations de copies. Les informations sur les interactions métier sont fournies par les interfaces métier, alors que les informations concernant les environnements sont fournis par les composants services. Ces informations sont transmises au niveau du protocole grâce aux interfaces de perception.
- *Contrôle*. Le contrôle exercé par un protocole sur une application métier s'exprime en termes de reconfigurations de copies qui peuvent concerner la création, le placement, l'interconnexion, la suppression, ainsi que la synchronisation de copies.

Étant donné les relations entre les protocoles et les applications métier, les reconfigurations effectuées par un protocole se traduisent par des opérations dans l'application métier. En effet, la création et la suppression d'une copie au niveau du protocole se traduit par la création et la suppression d'un composant métier. L'interconnexion d'une copie aux autres copies, aux clients ou aux services se traduit par l'établissement de connexions dans l'architecture métier. Finalement, la synchronisation entre copies concerne des composants métier et utilise l'accès aux états de ceux-ci.

Les opérations de reconfiguration étant reflétées par des opérations au niveau métier, les fonctionnalités de contrôle demandées par les protocoles sont les suivantes :

- *Création et suppression d'instances de composant métier* : ces opérations sont nécessaires aux protocoles pour la création et la suppression de copies. Alors que la création d'instances est une opération de base dans tout modèle à composants, la suppression d'instances est souvent laissée à la charge du ramasse-miettes de l'infrastructure d'exécution. Dans notre proposition, la suppression d'instances est définie comme une opération explicite, au même niveau que l'opération de création. Les deux opérations sont traitées par l'environnement d'exécution de composants, discuté dans la Section 4.5.
- *Gestion d'interconnexions entre composants métier* : cette gestion est nécessaire aux protocoles pour la gestion de groupes de copies. En effet, lors de la création de copies, les protocoles ont besoin d'établir des connexions de synchronisation, de reconfiguration, etc. Lors de la destruction de copies, ils doivent également détruire ces interconnexions. La gestion d'interconnexions est assurée par l'interface de connexion qui fait partie du modèle à composants (Section 4.2.3). Cette fonction de contrôle ne demande, par conséquent, aucune instrumentation supplémentaire de l'application métier.

- *Accès à l'état des composants métier.* L'accès à l'état des composants métier est nécessaire aux protocoles pour les captures et les restaurations d'état qui sont à la base de la création et de la synchronisation de copies. Or, selon la logique du paradigme composant, l'état des composants est encapsulé et non directement accessible. Pour permettre donc le contrôle de duplication et de cohérence, l'application métier doit être instrumentée pour fournir des primitives d'accès à l'état. Dans cette proposition, l'instrumentation consiste en l'implémentation, par les composants métier, d'une interface spécifique d'accès à leur état qui est laissée à la charge du programmeur de composants.
- *Contrôle de l'état stable des composants métier.* La manipulation d'état nécessite des garanties en ce qui concerne la cohérence de cet état (cf. Section 4.3.1). Par conséquent, les composants métier doivent fournir, en plus des interfaces d'accès à l'état, une interface de contrôle d'état stable. Dans notre proposition, cette interface est fournie au niveau de la structure exécutable d'un composant qui est détaillée dans la section 4.5.
- *Placement d'instances de composants métier.* Le placement d'instances est fourni comme une opération de base de l'environnement d'exécution des composants. Elle est également présentée dans la section 4.5.

4.4.3 Spécialisation des protocoles de duplication et de cohérence

Alors que l'instrumentation de l'application métier peut être vue comme l'adaptation de l'architecture métier pour permettre l'intégration de la gestion de la duplication et de la cohérence, la spécialisation des protocoles de duplication et de cohérence considère le problème inverse. Elle concerne l'adaptation des traitements génériques de gestion de la duplication et de cohérence, aux besoins spécifiques des applications métier. Ce cas se présente, par exemple, dans les cas de la cohérence optimiste où la prise en compte de la sémantique applicative facilite la détection et le réconciliation de conflits [75][143].

La première spécialisation des protocoles vis à vis des applications métier est faite lors de l'intégration des protocoles et de l'établissement des relations entre le niveau applicatif et le niveau de protocole. Les liaisons entre composants métier et composants de protocole, entre interfaces métier et interfaces de protocole, etc. définissent le degré minimal de spécialisation des protocoles aux applications métier (Figure 4-14.).

Les protocoles peuvent avoir un degré de spécialisation plus grand s'ils intègrent des traitements applicatifs dans le fonctionnement du protocole. L'approche proposée est la définition de protocoles de duplication et de cohérence en tant qu'applications avec du code “à trous”. Le code fourni correspond aux traitements génériques qui ne changent pas en fonction des applications métier, alors que les “trous” correspondent aux traitements spécifiques à ces applications. Ces traitements spécifiques sont référencés par la partie générique mais sont fournis par les applications métier. Leur intégration au niveau des protocoles est faite au moment de l'intégration de ce dernier au sein de l'application métier.

Pour illustrer ce point, considérons l'exemple d'un protocole de gestion optimiste des accès dans un système de fichiers. La spécialisation du protocole peut concerner la gestion des conflits d'accès. Les traitements peuvent être faits en fonction des types des fichiers et différer selon qu'il s'agit de fichiers de texte ou de feuilles de calcul (Figure 4-13.).

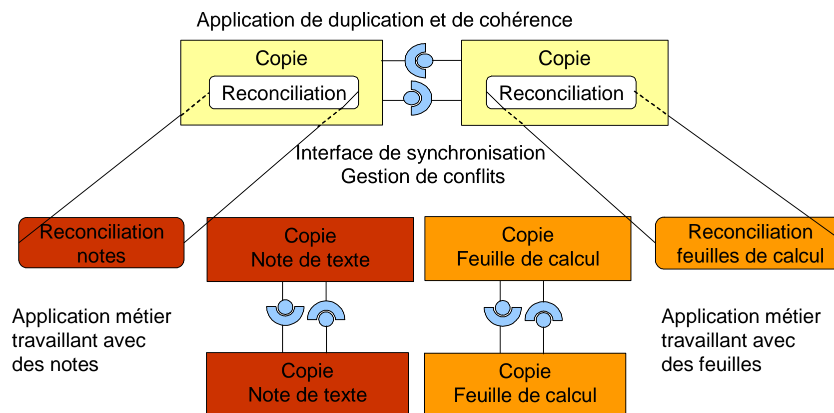


Figure 4-13. Spécialisation par des traitements spécifiques aux applications métier

L'approche de spécialisation peut être poussée à l'extrême avec des protocoles qui sont dédiés aux application métier. Dans ce cas, l'approche de modélisation des protocoles de duplication et de cohérence à base de composants est appliquée pour construire un protocole particulier. Ceci montre que, même si les objectifs initiaux de notre proposition visent la définition générique de protocoles et leur réutilisation dans différentes applications, elle peut être utilisée pour la construction de solutions spécifiques.

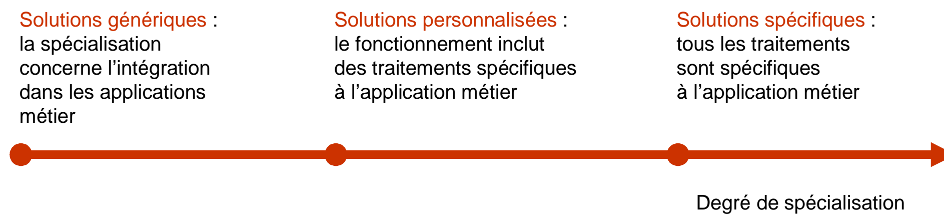


Figure 4-14. Degré de spécialisation des solutions de duplication et de cohérence

4.5 Architecture à l'exécution

Cette section présente l'architecture de la plate-forme d'exécution que nous proposons pour les applications métier et les protocoles. La section décrit l'architecture générale pour l'exécution des composants et explicite ensuite la place des structures de contrôle pour la gestion de la duplication et de la cohérence.

4.5.1 Architecture générale

L'architecture générale que nous proposons est inspirée des plates-formes à composants existantes tels que EJB ou CCM. Ses éléments sont les serveurs, les conteneurs, les talons et les squelettes. Son objectif principal est la séparation des aspects métier des aspects de gestion de la duplication et de la cohérence.

Les éléments de notre architecture sont détaillés dans ce qui suit.

Conteneurs

Comme dans tous les modèles à composants qui font usage de la notion, les conteneurs encapsulent les composants métier et gèrent leurs propriétés non fonctionnelles.

Dans notre proposition, un conteneur se charge d'une seule instance de composant métier. Cette instance métier a son état et ses fonctionnalités encapsulées dans un objet qui implémente ses interfaces fournies, ainsi que son interface d'accès à l'état. Cette dernière fait partie de l'instrumentation de l'application métier en vue de la gestion de la duplication et de la cohérence. La gestion d'un conteneur par instance permet d'attacher différents protocoles aux différentes instances d'un même type de composant.

Le conteneur fournit toutes les opérations concernant le déploiement et le contrôle à l'exécution d'une instance. Notamment, il fournit l'opération de création d'instance, ainsi que les interfaces prédéfinies d'introspection et de connexion (cf. Section 4.2.3). En effet, la création d'une instance de composant inclut la création d'un conteneur associé qui se charge des informations de type et de l'état d'exécution de cette instance. C'est le conteneur qui donne des renseignements sur l'instance (introspection) et qui contrôle son exécution (gestion de connexions).

Le conteneur fournit également tous les traitements relatifs à la gestion de la duplication et de la cohérence d'une instance. Il suit la logique du paradigme composant où les conteneurs se chargent des propriétés non fonctionnelles et fournit toutes les interfaces de protocole, notamment les interfaces de reconfiguration, de synchronisation, de contrôle d'état stable, etc.(cf. Section 4.4.2).

La Figure 4-15. donne la vision globale des conteneurs.

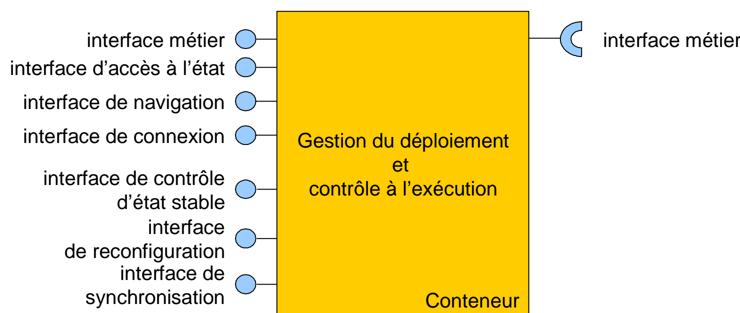


Figure 4-15. Conteneurs

Talons et squelettes

Les talons et les squelettes représentent les points de communication des composants. Un talon représente une interface requise d'un composant, alors qu'un squelette représente une interface fournie.

Les talons et les squelettes font partie du conteneur d'un composant. Ils sont le passage obligé pour toute communication entre le composant et son environnement. Ils interceptent, par conséquent, toutes les interactions entre composants.

L'identification des talons et des squelettes est faite à l'aide des références d'interfaces (cf. Section 4.2.2.) qui sont créées et gérées par le conteneur. Ce sont ces références qui sont manipulées par les interfaces d'introspection et de connexion.

Comme dans les plates-formes réparties classiques, les talons et les squelettes assurent la communication entre composants : ils effectuent des opérations d'emballage et de déballage de

paramètres et utilisent des services de transport. Toutefois, par leur nature d'interception, ils permettent de dévier le chemin standard des communications entre composants et d'y intégrer des traitements additionnels de duplication et de cohérence. La Figure 4-16. donne la vision des conteneurs avec les conteneurs et les talons.

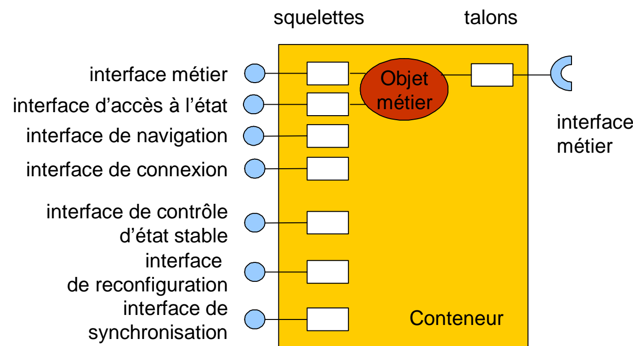


Figure 4-16. Talons, squelettes et conteneurs

Serveurs

Comme dans le cas d'EJB ou de CCM, un serveur offre un environnement d'exécution pour les conteneurs et fournit des services système. Il offre également des fonctions d'administration, notamment il assure le déploiement de composants et la surveillance de l'activité du serveur. Cette dernière est en effet surveillée par des composants service, présentés précédemment (cf. Section 4.3.2).

4.5.2 Gestion de la duplication et de la cohérence

Dans la section 4.3, nous avons défini l'architecture d'un protocole de duplication et de cohérence. Dans la section 4.4, nous avons détaillé les relations entre cette architecture et l'architecture de l'application métier. En introduisant les conteneurs, nous avons montré comment peut être exercé le contrôle concernant la duplication et la cohérence sur l'application métier. Le dernier point à définir est la forme exécutable des traitements de duplication et de cohérence.

La forme exécutable d'un protocole de duplication et de cohérence correspond à son intégration dans l'architecture exécutable de l'application métier. Cette intégration est faite de différentes manières selon qu'il s'agit de composants gestionnaires, services, clients ou copies.

- *Composants copies et composants clients*. Les traitements des composants clients et des composants copies sont intégrés dans les conteneurs des composants métier qu'ils modélisent. Les conteneurs se retrouvent, par conséquent, à gérer[1] les interfaces de duplication et de cohérence des composants de protocole, notamment celles de perception, de synchronisation, de reconfiguration et de service. Ces interfaces se rajoutent aux interfaces déjà gérées par les conteneurs qui incluent les interfaces d'introspection et d'interconnexion, définies pour les assemblages de composants métier (cf. Section 4.2.3) et l'interface de contrôle d'état stable, introduite pour l'instrumentation des ap-

1. Les interfaces gérées par les conteneurs apparaissent comme des interface fournies ou requises des composants.

plications métier (cf. Section 4.4.2). Toutes ces interfaces sont représentées, comme les interfaces métier, par des talons et des squelettes.

- *Composants services et gestionnaires*. Contrairement aux composants clients et aux composants copies, les composants services et les composants gestionnaires ne sont pas inclus dans les conteneurs des composants métier mais sont rajoutés au niveau de l'architecture de l'application. Ce sont des composants indépendants, connectés aux autres composants de protocole.

La vision globale de l'architecture obtenue après l'intégration d'un protocole au sein d'une application métier est donnée à la Figure 4-17. Elle fait apparaître trois types de composants. Les composants du premier type sont les composants métier qui restent non modifiés par la duplication et de la cohérence. Sur la figure, ils sont appelés composants métier "purs". Le deuxième type de composants sont ceux qui, de point de vue du protocole, sont des composants clients ou des composants copies. Ces composants intègrent des traitements de duplication et de cohérence au niveau de leurs conteneurs. Le troisième type de composants sont les composants services et les gestionnaires qui sont des composants chargés uniquement des traitements de protocole.

Les connexions entre composants sont, d'une part, les connexions métier (CM sur la figure) et, d'autre part, les connexions définies dans le protocole. Il s'agit des connexions de service (CS), de synchronisation (CSY), des connexions de configuration pour les traitements des gestionnaires (CC) et des connexions de perception (CP).

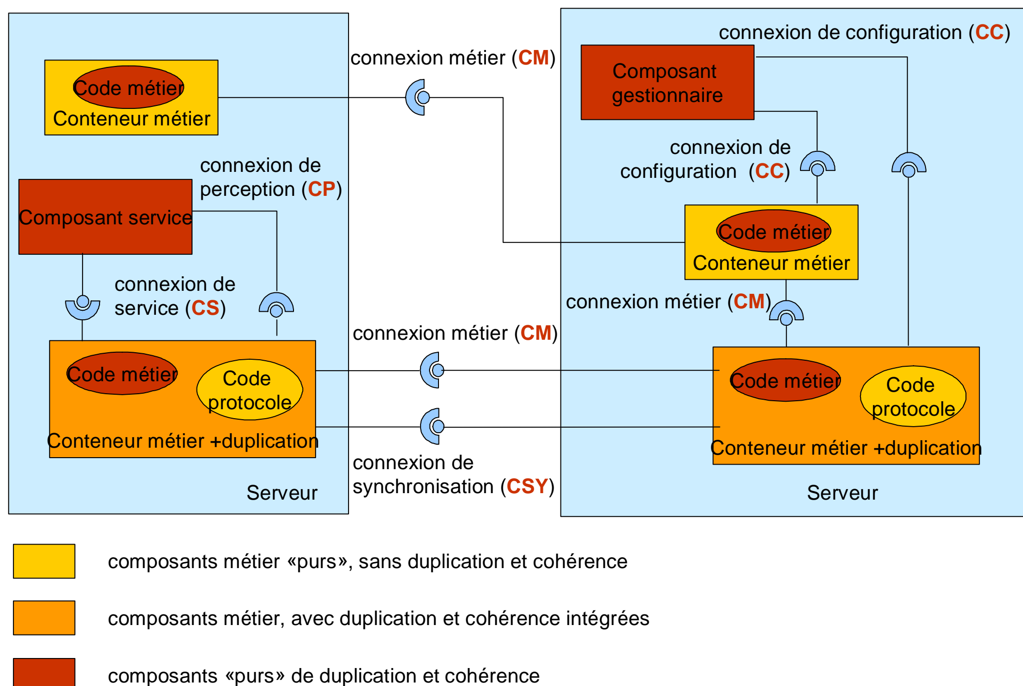


Figure 4-17. Architecture métier enrichie par la gestion de la duplication et de cohérence

4.6 Synthèse de proposition

La gestion de la duplication et de la cohérence proposée repose sur un modèle de programmation des applications métier et des protocoles, un modèle de composition entre applications métier et protocoles et une architecture à l'exécution.

- *Modèle de programmation.* Nous proposons un modèle de programmation à base de composants qui permet la construction d'applications métier sans considérer les aspects de duplication ni cohérence. Ce modèle est spécialisé et appliqué pour la construction de protocoles de duplication et de cohérence qui sont ainsi modélisés en tant qu'applications à base de composants. Ces protocoles sont programmés de manière séparée sans considérer les aspects métier des applications.
  La définition séparée est visée pour permettre l'intégration de différents protocoles au sein d'une même application, sans modification de celle-ci. Elle permet également la réutilisation d'un même protocole dans des applications différentes. La modélisation à base de composants est choisie pour faciliter cette intégration mais aussi pour permettre l'évolution et la maintenance des protocoles.

- *Modèle de composition.* Le modèle de composition définit l'intégration des protocoles de duplication et de cohérence dans les applications. Il suit la logique des systèmes réflexifs et traite les protocoles comme un méta niveau de contrôle pour les applications métier. Ce méta-niveau se charge d'observer les applications et de les contrôler en effectuant des traitements de duplication et de cohérence. Le modèle de composition établit des relations entre les entités des deux niveaux, impose une instrumentation au niveau métier pour permettre la gestion de la duplication et de la cohérence et permet une spécialisation de cette gestion en vue des besoins spécifiques des applications.

- *Architecture à l'exécution.* L'architecture à l'exécution définit l'instanciation des applications métier et des protocoles de duplication et de cohérence. Inspirée par les architectures à trois niveaux (serveurs-conteneurs-squelettes) des plates-formes EJB et CCM, elle gère les aspects de duplication et de cohérence au niveau des conteneurs et fournit des primitives permettant leur configuration pendant la phase de déploiement.

DEUXIÈME PARTIE

MISE EN ŒUVRE

Cette partie décrit le prototype FAR que nous avons implémenté en Java afin de valider l'approche proposée dans le chapitre précédent. Elle détaille la procédure de construction d'applications et de protocoles, décrit les structures de composants utilisées et présente les mécanismes d'intégration appliqués. Le chapitre décrit également les implémentations CCM [102] et EJB [136] que nous avons effectuées afin de démontrer l'applicabilité des principes proposés dans des environnement standards.

L'organisation de la partie est la suivante. Le chapitre 5 présente l'organisation générale du prototype FAR avant de se concentrer sur les mécanismes de construction et de gestion d'applications métier sans duplication ni cohérence. Le chapitre 6 présente les mécanismes de construction et d'intégration des protocoles de duplication et de cohérence modélisés à base de composants. Les chapitres 7 et 8 analysent l'applicabilité des principes de FAR dans les environnements CCM et EJB. Le chapitre 9 décrit les expérimentations que nous avons menées sur FAR et discute les résultats obtenus.

Chapitre 5

FAR et les applications sans duplication ni cohérence

Ce chapitre présente la gestion d'applications sans duplication ni cohérence dans FAR. Elle présente les étapes de description et de programmation de composants (Section 5.3), décrit la structure exécutable des composants (Section 5.4) et présente leur environnement d'exécution (Section 5.5). Les fonctionnalités fournies sont illustrées au fur et à mesure à l'aide d'un exemple.

5.1 Motivations du prototype FAR

Nous avons validé notre proposition, présentée au chapitre précédent, en implémentant un prototype, nommé FAR[1], dans l'environnement Java. Nous avons voulu pouvoir définir librement nos modèles de programmation, de composition et d'exécution sans être contraints par des modèles et des infrastructures imposés tels que CCM ou EJB. Nous avons donc implémenté dans FAR uniquement les mécanismes nécessaires à la gestion de la duplication et de la cohérence dans les applications à base de composants.

Nous avons choisi d'implémenter FAR en Java pour les raisons suivantes :

- *Contraintes minimales*. Java est un langage et un outil de programmation qui n'impose pas de contraintes en ce qui concerne nos modèles de programmation, de composition et d'exécution. Les seules contraintes qu'il impose sont celles relatives à la programmation orientée objet.
- *Facilité de prototypage*. Avec son approche objet et son typage fort, Java offre un outil de programmation qui facilite la construction de prototypes. L'environnement Java fournit en plus de nombreuses classes prédéfinies encapsulant des fonctionnalités utiles comme l'introspection des objets, la communication à distance, etc.
- *Large utilisation sur différentes plates-formes*. La dernière caractéristique intéressante de Java, celle qui est à la base de sa popularité, est sa machine virtuelle permettant l'exécution de programmes Java sur des plates-formes hétérogènes. En effet, la construction de FAR en Java nous a permis de faire abstraction des systèmes d'exploitation et des capacités des plates-formes. Nous avons pu l'utiliser aussi bien sur des machines

1. FAR est une abréviation pour *Framework for Adaptable Replication*.

de bureau que sur des assistants personnels[1].

L'implémentation que nous recherchons avec FAR ne veut pas dire que nous ignorons les plates-formes existantes et leur complexité. Nous considérons notamment les plates-formes CCM et EJB et explorons l'applicabilité de notre proposition en confrontant et en adaptant les concepts de FAR à ces plates-formes. Les réalisations qui en résultent sont discutées dans les chapitres suivants.

5.2 Organisation générale de FAR

FAR est composé d'un ensemble de classes prédéfinies et d'outils de génération de code qui permettent *la construction* d'applications métier et de protocoles de duplication et de cohérence, *l'intégration* des protocoles au sein de ces applications métier et *l'exécution* de l'ensemble qui en résulte. Plus particulièrement, ces classes et outils permettent la programmation de composants, la définition de leur structure exécutable, leur assemblage, leur déploiement et leur exécution.

Le schéma général d'utilisation de FAR est donné à la Figure 5-1. Il montre le cycle de vie des applications métier sans duplication ni cohérence, ainsi que leur cycle de vie lors de l'intégration d'un protocole de duplication et de cohérence. Dans le cas sans duplication ni cohérence, une application passe par des étapes de description déclarative de ses types de composants, de génération automatique de squelettes d'implémentation de composants, de programmation, de déploiement et d'exécution. Dans le cas avec duplication et cohérence, les applications métier sont configurées lors du déploiement par un protocole et s'exécutent de la manière correspondante. Pour que l'intégration soit possible, le protocole doit préalablement être conçu en passant par les mêmes étapes de description, de génération et de programmation. Il doit ensuite être spécialisé aux besoins de l'application en prenant en compte une description de la configuration de duplication et de cohérence et en passant par une étape de génération de composants de protocole qui sont rendus spécifiques aux types de composants applicatifs.

La construction et l'exécution des applications métier reposent sur des classes prédéfinies qui implémentent les notions de composant métier, de conteneur, de talon, de squelette et de serveur. Les classes respectives sont `Component`, `Container`, `CommPoint_Impl` et `ServerCtxt`. La classe `Component` est destinée aux programmeurs de composants. Les classes `Container` et `CommPoint_Impl` sont utilisées pour l'assemblage de composants et pour l'instrumentation des applications métier en vue d'une gestion de duplication et de cohérence. La classe `ServerCtxt` définit l'environnement d'exécution des composants. Ces classes sont montrées à la Figure 5-1. à côté des étapes pendant lesquelles elles sont utilisées.

La construction et l'exécution de protocoles de duplication et de cohérence reposent sur deux groupes de classes qui sont respectivement des classes introduites pour la construction des applications métier et des classes spécifiques à la gestion de la duplication et de la cohérence. Dans le premier groupe se trouvent les classes `Container` et `ServerCtxt` qui sont augmentées pour pouvoir assurer la gestion de la duplication et de la cohérence au sein des applications métier. Le deuxième groupe est constitué de classes utilisées pour la programmation des composants de protocole et pour la définition de leurs talons et squelettes. Il s'agit de `ProtocolComponent`, `ManagerComponent`, `ServiceComponent` et `RC_CommPoint_Impl`.

1. Des détails sur ce point sont donnés dans le Chapitre 9.

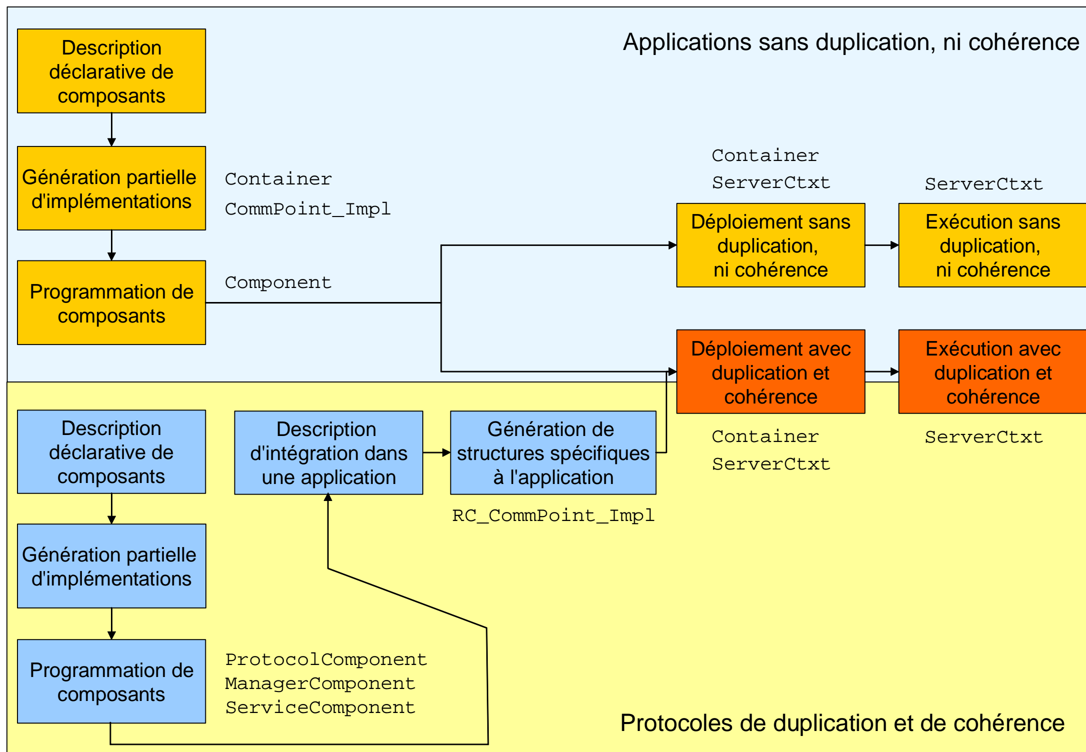


Figure 5-1. Schéma général d'utilisation de FAR

Dans la suite de ce chapitre nous présentons les classes et les fonctionnalités de FAR qui assurent la gestion des applications sans duplication ni cohérence. La gestion des protocoles de duplication et de cohérence est discutée dans le Chapitre 6.

Le chapitre est organisé comme suit. Dans la section 5.3 nous présentons l'étape de construction des applications : nous parlons de la description déclarative de leurs composants et de leur programmation. Dans la section 5.4 nous détaillons la structure exécutable des composants qui est constituée d'un conteneur, d'une implémentation métier et de talons et de squelettes. Finalement, dans la section 5.5, nous présentons les étapes de déploiement et d'exécution des applications.

5.3 Description et programmation de composants

Tout type de composant métier dans FAR est défini en tant que sous-classe de la classe `Component`. Le code de cette sous-classe correspond à l'implémentation métier du composant et est complété par un programmeur de composants. Ce code est partiellement généré à partir d'une description déclarative du type du composant.

Dans la suite, nous présentons, dans l'ordre, les déclarations des composants, la classe `Component` et les classes d'implémentation des composants.

5.3.1 Déclaration de composants

Dans FAR, les déclarations des types de composants sont données sous forme de descriptions XML. Nous avons choisi XML à cause des outils disponibles d'analyse de son format.

La description XML d'un type de composant (Figure 5-2.) spécifie ce dernier en termes d'interfaces fournies et requises, de communications synchrones ou asynchrones, de nombre de connexions autorisées (cf. Chapitre 4, Section 4.2.3, p. 67), etc. Elle utilise les tags XML suivants :

- Tag `component_type` (ligne 2). Ce tag indique le nom du type de composant.
- Tags `provided` et `required` (lignes 3 et 12). Ces tags définissent respectivement les ensembles d'interfaces fournies et requises pour le type de composant. Chaque interface est définie par le nom de son type (`interface_type`), le nom symbolique du point de communication correspondant (`name`), le mode de communication (`mode`) et le nombre de connexions autorisées (`connect`).
- Tag `name` (ligne 6). Ce tag donne le nom symbolique d'un point de communication. Ce nom peut être utilisé pour récupérer la référence de ce dernier (cf. Chapitre 4, Section 4.2.2, p. 67).
- Tag `mode` (ligne 7). Ce tag dit si le point de communication communique de manière synchrone ou asynchrone. Ses valeurs respectives sont "`SYNC`" et "`ASYNC`".
- Tag `connect` (ligne 8). Ce tag donne les connexions autorisées pour un point de communication. Il peut ne pas imposer de restrictions sur le nombre de connexions (valeur `N)` ou alors définir une limite numérique (p.ex. valeur "`4`").
- Tag `select` (ligne 18). Le tag `select` n'est applicable que dans le cas de connexions synchrones où un point de communication sortant est connecté à plusieurs points de communication entrants. Il permet d'indiquer un critère de sélection sur les résultats obtenus. La valeur de ce champ donne un nom de méthode du composant chargée de mettre en œuvre cette sélection.

```
1    <component>
2       <component_type> ... </component_type >
3          <provided>
4               <interface>
5                       <interface_type>...</interface_type>
6                       <name>...</name>
7                       <mode>...</mode>
8                       <connect>...</connect>
9               </interface>
10              ...
11         </provided>
12         <required>
13              <interface>
14                      <interface_type>...</interface_type>
15                      <name>...</name>
16                      <mode>...</mode>
17                      <connect>...</connect>
18                      <select>...</select>
19              </interface>
20              ...
21      </required>
22   </component>
```

Figure 5-2. Description XML d'un type de composant

En guise d'exemple, considérons l'application "Hello World" (Figure 5-3.). Dans cette application il y a deux instances de composants, notamment un client `C` et un serveur `S`. Elles sont connectées à l'aide de l'interface `Hello_itf` qui permet au client de demander au serveur l'affichage d'une chaîne de caractères. L'interface `Hello_itf` est donc requise pour le client et fournie pour le serveur.

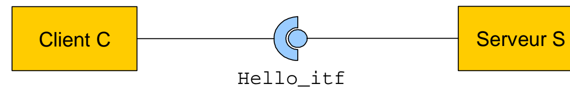


Figure 5-3. L’application “Hello World”

Considérons maintenant les déclarations des types du client et du serveur qui sont respectivement `Client_Comp` et `Server_Comp` (Figure 5-4.).

La déclaration du type du client `Client_Comp` (Figure 5-4.a) ne contient qu’une section `required` puisque le composant ne dispose que d’une interface requise. Comme indiqué par le tag `interface_type` cette interface est `Hello_itf`. Le point de communication pour cette interface est nommé `serverHello` (ligne 6), il ne peut se connecter qu’à un seul composant (ligne 8) et sa communication après une telle connexion est synchrone (ligne 7).

De manière symétrique, le type `Server_Comp` (Figure 5-4.b) ne définit qu’une interface fournie `Hello_itf`. Cette interface est représentée par un point de communication nommé `forClientHello` auquel peuvent se connecter plusieurs composants (ligne 8). Pour une instance de `Server_Comp`, le nom `forClientHello` peut être utilisé pour récupérer la référence de ce point de communication et pour établir une connexion vers lui[1].

```
1     <component>
2         <component_type> Client_Comp </component_type >
3             <required>
4                 <interface>
5                     <interface_type> Hello_itf </interface_type>
6                         <name> serverHello </name>
7                         <mode> SYNC </mode>
8                         <connect> 1 </connect>
9                 </interface>
10            </required>
11    </component>
```

(a)

```
1     <component>
2         <component_type> Server_Comp </component_type >
3             <provided>
4                 <interface>
5                     <interface_type> Hello_itf </interface_type>
6                         <name> forClientHello </name>
7                         <mode> SYNC </mode>
8                         <connect> N </connect>
9                 </interface>
10            </provided>
11    </component>
```

(b)

Figure 5-4. Déclarations XML pour les composants de l’application “Hello World”

5.3.2 La classe `Component`

Le rôle de la classe `Component` est d’assurer le lien entre l’implémentation métier d’un composant et sa structure en termes de conteneur, de talons et de squelettes (cf. Chapitre 4, Section 4.5.1, p. 85). Pour cela, la classe définit un traitement générique de création d’instances

1. L’établissement de connexions entre instances de composants à l’aide des références de leurs points de communication est discuté dans la suite.

qui inclut, en plus de la création des instances des sous-classes de `Component`, la création de conteneurs pour ces instances. La création des conteneurs est accompagnée par la création des talons et des squelettes pour les interfaces requises et fournies des instances.

Considérons de nouveau l'application "Hello World". Les types du client et du serveur, `Client_Comp` et `Server_Comp`, sont en effet définis en tant que sous-classes de `Component`. Ils sont instanciés en appelant le traitement générique défini dans la classe `Component`. Pour l'instance du client, ce traitement consiste en la création d'une instance `C` de `Client_Comp`, d'un conteneur pour `C` et d'un talon pour l'interface requise `Hello_itf`. De la même manière, pour le serveur, ce traitement crée une instance `S` de `Server_Comp`, un conteneur pour `S` et un squelette pour l'interface fournie `Hello_itf`.

5.3.3 Implémentation métier d'un composant

Le code métier d'un composant est fourni par une classe Java définie en tant que sous-classe de la classe `Component`. Cette sous-classe est partiellement générée à partir de la description XML du type du composant. Le processus de génération automatique suit des règles pour la mise en place des interfaces fournies et requises que nous détaillons dans ce qui suit.

Gestion des interfaces fournies

La classe Java fournit l'implémentation métier du composant et contient donc le code des services spécifiés par les interfaces fournies de ce dernier. Les interfaces fournies sont par conséquent déclarées dans la clause `implements` de cette classe Java.

```
interface Hello_itf {
   void print(String s) throws Exception;
}
class Server_Comp extends Component implements Hello_itf {
   public void print(String s) throws Exception {System.out.println(s);}
}
```

Figure 5-5. Implémentation métier du serveur de l'application "Hello World"

Pour illustrer ce point, considérons le serveur dans l'application "Hello World". Il n'a qu'une seule interface fournie qui est l'interface `Hello_itf` et qui définit une primitive `print(String s)` pour l'affichage de chaînes de caractères. L'implémentation métier du serveur `S` est donnée par la classe `Server_Comp` qui fournit le traitement pour l'interface fournie `Hello_itf` et qui donc l'implémente (Figure 5-5.).

Étant donné qu'une classe Java ne peut implémenter plusieurs fois une même interface, nos composants ne peuvent pas avoir deux interfaces fournies du même type. Dans l'exemple "Hello World" ceci correspond à l'impossibilité d'avoir deux points d'entrée de type `Hello_itf` pour le serveur `S`. Cette restriction nous a semblé mineure puisqu'elle peut facilement être levée en considérant des implémentations métier constitués de plusieurs classes Java[1].

Gestion des interfaces requises

Les interfaces requises caractérisent les services auxquels les composants font appel pour leur fonctionnement. Par conséquent, ces interfaces doivent pouvoir être désignées explicite-

1. Comme, par exemple, dans l'implémentation des composants dans la plate-forme OpenCCM [104].

ment dans le code des composants. Pour cela, nous introduisons dans FAR des attributs qui représentent ces interfaces. Ces attributs sont définis dans les classes d'implémentation métier des composants et sont utilisés de manière standard. Ainsi, les interactions, même distribuées, entre composants restent transparentes pour leur code.

Pour illustrer ce point, considérons le client dans l'application "Hello World". Il a une interface requise, l'interface `Hello_itf`, qu'il utilise pour afficher des chaînes de caractères. Dans son implémentation métier (Figure 5-6.), cette interface est représentée par un attribut `serverHello`. C'est cet attribut qui est utilisé chaque fois que le client veut appeler la fonction `print(String s)`. Son utilisation cache au client l'identité du composant appelé, ainsi que la nature de la communication avec ce dernier.

```
class Client_Comp extends Component {
   Hello_itf serverHello;
   public void run() {
      try {
         serverHello.print("Hello!");
      } catch Exception {...}
   }
}
```

Figure 5-6. Implémentation métier du client de l'application "Hello World"

Génération automatique de la classe d'implémentation

La classe d'implémentation métier d'un composant est partiellement générée à partir de la description XML de son type. La génération est basée sur l'outil Velocity [156], un outil en utilisation libre d'Apache. Velocity évite la mise en place d'une analyse syntaxique de la description XML et permet l'accès direct aux valeurs des tags.

La génération de la classe d'implémentation à partir de la description XML (Figure 5-4.) applique les principes de gestion des interfaces requises et fournies et est triviale. La classe générée est nommée d'après la valeur du tag `component_type`, elle implémente les interfaces spécifiées dans la section `provided` et contient un ensemble d'attributs représentant les interfaces requises de la section `required`. La génération reste partielle puisqu'elle ne crée qu'un squelette de classe dans lequel les implémentations des méthodes sont à rajouter par le programmeur de composants.

5.4 Structure de composant

Un composant est constitué d'une implémentation métier, fournie dans une sous-classe de `Component`, d'un conteneur, ainsi que d'un ensemble de talons et de squelettes (cf. Chapitre 4, Section 4.5.1, p. 85). Dans la suite, nous détaillons, tout d'abord, les talons et les squelettes, décrivons ensuite les conteneurs et terminons par la procédure de génération de toutes ces structures.

5.4.1 Talons et squelettes

Les talons et les squelettes d'un composant sont ses points de communication. Les talons sont les points de communication pour les interfaces requises, alors que les squelettes sont ceux pour les interfaces fournies. Les talons et les squelettes sont chargés des appels entre compo-

sants et leurs références sont utilisées pour l'interconnexion de composants. Ils sont implémentés en tant que sous-classes de la classe CommPoint_Impl. Dans la suite, nous présentons la classe CommPoint_Impl avant d'aborder le mécanisme d'appel et les références.

La classe CommPoint_Impl

La classe CommPoint_Impl caractérise un point de communication. Spécifiée par l'interface CommPoint_itf, elle fournit deux groupes de méthodes. Le premier groupe inclut des méthodes d'introspection sur le type d'un point de communication. Le deuxième groupe concerne les méthodes de gestion des connexions d'un point de communication.

(i) Méthodes d'introspection du type d'un point de communication

Les méthodes fournissant des informations sur le type d'un point de communication sont les suivantes :

- String getInterface() : fournit le nom de l'interface, fournie ou requise, que ce point de communication représente.
- String getName() : fournit le nom symbolique du point de communication comme défini dans la description XML du type du composant.
- boolean isSync() : retourne vrai si le mode de communication à utiliser pour ce point de communication est le mode synchrone.
- int getConnectLimit() : fournit le nombre limite de connexions autorisées pour ce point de communication. Cette méthode est utilisée lors des vérifications effectuées par le conteneur lors des demandes de connexion.
- ContainerRef getContainer() : retourne la référence du conteneur dont ce point de communication fait partie. La classe ContainerRef est présentée dans la suite.

La classe CommPoint_Impl ne fournit pas d'implémentation générique pour les quatre premières méthodes. Les implémentations sont générées lors de la génération des classes de talons et de squelettes spécifiques à un type de composant donné. Cette génération est basée sur les informations spécifiées dans la description XML de ce type.

```
class Client_Comp_Hello_itf_Stub  extends CommPoint_Impl
                                  implements Hello_itf {
   public String getInterface() {return "Hello_itf";)
   public String getName()      {return "serverHello";)
   public boolean isSync()      {return true;}
   public int getConnectLimit() {return 1;}
   ...
}
```

```
class Server_Comp_Hello_itf_Skel  extends CommPoint_Impl
                                  implements Hello_itf {
   public String getInterface() {return "Hello_itf";)
   public String getName()      {return "forClientHello";)
   public boolean isSync()      {return true;}
   public int getConnectLimit() {return NO_LIMIT;}
   //NO_LIMIT définit le maximum de connexions pour les cas sans restrictions
   ...
}
```

Figure 5-7. Classes spécifiques pour les points de communication dans “Hello World”

Considérons les points de communication pour les composants dans l'application "Hello World" (Figure 5-3.). Ces points de communication comprennent un talon pour l'interface requise du client et un squelette pour l'interface fournie du serveur. Les deux classes qui les implémentent sont respectivement `Client_Comp_Hello_itf_Stub` et `Server_Comp_Hello_itf_Skel` (Figure 5-7.).

La classe `Client_Comp_Hello_itf_Stub` est la classe spécifique du talon pour l'interface `Hello_itf` du type `Client_Comp`. L'implémentation des méthodes d'introspection est générée en récupérant les informations de la déclaration XML de ce type (Figure 5-4.a). De la même manière, la classe `Server_Comp_Hello_itf_Skel` est la classe spécifique du squelette pour l'interface `Hello_itf` du type `Server_Comp` et est basée sur la description XML de ce dernier.

(ii) Méthodes de gestion des connexions d'un point de communication

Les méthodes de connexion fournies par la classe `CommPoint_Impl` sont les suivantes :

- `int isConnected()` : cette méthode vérifie si le composant est connecté, par ce point de communication, à d'autres composants et rend le nombre de connexions existantes.
- `void addConnection(Connection ct)` : cette méthode est utilisée lors de l'établissement d'une connexion entre ce point de communication et un point de communication d'un autre composant. Elle est appelée par le conteneur est met à jour la liste de connexions pour ce point de communication. Le paramètre de type `Connection` fournit les références des entités connectées (conteneur et point de communication).
- `void removeConnection(Connection ct)` : cette méthode fournit la fonction inverse de la méthode précédente : elle détruit la connexion entre ce point de communication et celui spécifié par le paramètre `ct`.
- `Vector getConnections()` : cette méthode retourne la liste des connexions pour un point de communication.

Dans l'application "Hello World", avant la connexion des deux instances, l'appel à la méthode `isConnected()` retournera faux aussi bien pour le talon du client, que pour le squelette du serveur. Pour établir la connexion entre ces instances, les conteneurs du client et du serveur vont appeler `addConnection()`. Les paramètres enregistrés sur le talon du client identifieront le squelette du serveur et vice versa. Après la connexion, la méthode `isConnected()` retournera vrai et la méthode `getConnections()` retournera un vecteur contenant l'identification de la connexion entre les deux instances.

Traitement des appels

Les fonctionnalités principales des talons et des squelettes concernent les appels entre composants. Ils fournissent des traitements classiques d'emballage et de déballage de paramètres, ainsi que des opérations d'acheminement d'appels entre composants. Pour l'acheminement, ils utilisent des services d'envoi et de réception de messages entre serveurs fournis par l'environnement d'exécution[1].

Dans la suite, nous décrivons les traitements des talons et des squelettes en considérant le cas synchrone, le cas asynchrone se différenciant uniquement par l'absence de retour de

1. Ces primitives sont présentées dans la section sur l'environnement d'exécution, Section 5.5.

résultat. Nous discutons également le cas des appels au sein d'un composant qui ne font pas usage des talons ni des squelettes.

(i) Fonctionnement des talons

Les invocations *synchrones*, côté appelant, se déroulent de la manière suivante :

- *Emballage*. L'emballage de l'appel consiste en la préparation d'un message incluant un identificateur de la méthode à appeler et les paramètres de cette méthode. Le message créé est un objet qui utilise le mécanisme de *sérialisation* Java [140] afin de permettre un éventuel passage sur le réseau. Le message n'est pas transporté sur le réseau et n'est donc pas sérialisé et désérialisé dans le cas d'appels entre composants s'exécutant dans le même serveur[1].
- *Appel et attente de résultat*. L'envoi du message qui représente l'appel résulte en un appel bloquant[2] géré par les primitives d'envoi de messages entre serveurs. Dans le cas d'une connexion "un-à-un", un seul message est envoyé et l'appel est bloqué jusqu'à réception de réponse pour ce message. Dans le cas d'une connexion "un-à-plusieurs", le talon envoie plusieurs messages en parallèle et la terminaison de l'appel dépend du critère de sélection spécifié dans le descripteur XML. Le comportement par défaut attend la terminaison de tous les appels parallèles et retourne un tableau de valeurs. Dans le cas où un critère est spécifié, l'appel est bloqué jusqu'à réception de la première réponse vérifiant ce critère et les autres résultats sont ignorés.
- *Déballage de résultat*. Le déballage est l'opération inverse de l'emballage : il récupère les valeurs typées de retour qui sont encapsulées dans un message reçu à l'aide des services d'échange de messages entre serveurs.

```
class Client_Comp_Hello_itf_Stub  extends CommPoint_Impl
                                  implements Hello_itf {...
   public print(String s) throws Exception {
        //préparation du message correspondant à l'appel
        Message msg = new MethodCallMessage(1,{s});
        //délégation à la méthode invoke
        invoke(msg);
   }
   Message invoke(Message msg) {
      //vérification que le composant est bien connecté à un composant qui fournit Hello_itf
      if (!isConnected()) throw NotConnectedException();
      //récupérer la référence du destinataire
      TransportRef destRef = connections.elementAt(1).getTransportRef();
      //envoyer le message. Utiliser le service d'envoi de messages de l'environnement d'exécution
      //l'envoi vérifie la localité du composant destinataire
      ServerCtxt.msgSend(destRef, msg, true);
   }
}
```

Figure 5-8. Traitement des appels dans un talon. Cas spécifique de "Hello World"

Les fonctionnalités énumérées ne sont pas implémentées par la classe `CommPoint_Impl` mais sont générées de manière spécifique aux types des composants. En effet, la génération de la classe d'implémentation d'un composant est accompagnée par une génération des classes des

1. Ce comportement est assuré au niveau du service d'envoi de message des serveurs, présenté dans la suite.
2. Dans le cas asynchrone, l'appel est non bloquant.

talons correspondants. Ces talons se chargent de l'emballage et du déballage des messages spécifiques aux appels des composants[1]. Pour les appels, ils utilisent une méthode `invoke` qui se charge de l'interaction avec les primitives d'envoi de messages de l'environnement d'exécution.

Dans le cas du client de l'application "Hello World", les traitements relatifs à la gestion des invocations sont donnés sur la Figure 5-8. L'implémentation de la méthode `print(String s)` de l'interface `Hello_itf` se charge de vérifier si le client est bien connecté à un composant qui fournit l'interface `Hello_itf`. Si la connexion existe, il récupère la référence du destinataire et vérifie si le composant s'exécute dans l'environnement local. Si c'est le cas, il récupère la référence du squelette correspondant et effectue un appel Java. Si le composant est distant, il prépare un message et l'envoie en utilisant la primitive `msgSend` fournie par l'environnement d'exécution. Lors de l'envoi, il spécifie que l'appel est synchrone (paramètre `true`).

(ii) Fonctionnement des squelettes

Coté squelette, les appels synchrones arrivent soit sous la forme d'appels locaux Java, soit sous la forme d'appels distants traités par une méthode spécialisée `dispatch`. Dans le deuxième cas, les traitements de cette méthode incluent le déballage des paramètres, l'identification de la méthode à appeler, l'appel de cette méthode sur le code métier, l'emballage du résultat et son envoi à l'appelant. Comme dans les méthodes des talons, les traitements d'emballage et de déballage sont spécifiques aux types des données manipulées ce qui évite les invocations génériques et coûteuses de la réflexion Java.

Pour illustrer le fonctionnement des squelettes, considérons le cas du squelette du serveur dans l'application "Hello World". Il reçoit les appels de `print(String s)` avec sa méthode `dispatch` qui appelle le composant serveur. Après le traitement de `print`, `dispatch` envoie un message au talon appelant qui attend la terminaison de l'appel. Ceci est fait par le biais du service d'envoi de messages de l'environnement d'exécution.

```
class Server_Comp_Hello_itf_Skel  extends CommPoint_Impl
                                  implements Hello_itf {
   ...
   Message dispatch(Message msg) throws Exception {
      Message result = null;
      //récupérer l'identificateur de méthode
      int method = msg.getMethod();
      //récupérer les arguments de méthode
      Object[] args = msg.getMethodArgs();
      //test sur l'identité de la méthode à appeler
      switch (method) {
         //appel de print
         case 1 :
            //appel du composant, pas de résultat à emballer
            component.print((String)args[0]);
            break;
         ...
      }
      return result;
}}
```

Figure 5-9. Traitements des appels dans un squelette. Cas spécifique de "Hello World"

1. Soulignons que sous emballage/déballage nous parlons de la manipulation de messages qui n'implique pas la sérialisation/désérialisation.

(iii)Les appels internes

Les appels internes sont les appels qu'un composant effectue sur ses propres méthodes. Contrairement au cas d'appel entre composants, ces appels ne passent pas par les talons et les squelettes mais sont de simples appels Java.

Nous avons choisi de ne pas faire passer les appels internes par les talons et les squelettes du composant pour deux raisons. D'une part nous n'avons pas voulu modifier la sémantique du mot-clé `this` de Java afin de désigner, à la place de la classe Java d'implémentation métier du composant, la classe du conteneur de ce dernier. D'autre part, nous avons voulu éviter la dégradation des performances due aux indirections additionnelles.

L'utilisation d'appels Java standard pour les appels internes a deux conséquences. D'une part, les appels internes ne sont pas interceptés au niveau des talons ni des squelettes. D'autre part, la gestion d'appels asynchrones et la création de processus distincts (*threads*) qu'elle nécessite sont laissées à la charge du programmeur du composants.

Références de talons et de squelettes

Les références des talons et des squelettes correspondent aux références d'interfaces, introduites dans le Chapitre 4, Section 4.2.2. Dans notre implémentation, ces références sont composées d'une référence de conteneur et d'un identificateur de point de communication. Les références de conteneur sont définies par la classe `ContainerRef`, alors que les identificateurs des points de communications sont les noms symboliques déclarés dans le type de composant. Ainsi, la référence du talon du client dans "Hello World" est un couple `<référence du conteneur du client, "serverHello">`. La référence du serveur de "Hello World" est un couple `<référence du conteneur du serveur, "forClientHello">`.

5.4.2 Conteneur

Le conteneur est la structure qui transforme une implémentation métier en composant. Comme dans la majorité des plates-formes (cf. Chapitre 1, Section 1.1.2, p. 24), son rôle est de contrôler le fonctionnement de cette implémentation et de lui fournir tous les aspects *non fonctionnels* dont elle peut avoir besoin.

Les conteneurs dans FAR peuvent avoir deux formes principales : une forme minimale qui correspond à des composants métier sans duplication ni cohérence et une forme augmentée qui sert à la gestion de la duplication et de la cohérence. Dans cette section nous considérons la première forme, alors que la forme pour la gestion de la duplication et de la cohérence est présentée dans le chapitre suivant. (cf. Chapitre 6, Section 6.3.4, p. 134).

Dans la suite, nous détaillons les structures internes des conteneurs, le type de leurs références, l'interface générique qu'ils implémentent et leur rôle dans les interconnexions de composants.

Structures internes d'un conteneur

Le conteneur est une structure qui englobe toutes les autres structures constituant une instance de composant. Il est représenté par la classe `Container` qui dispose des attributs suivants :

- `Component component` : cet attribut référence l'objet d'implémentation métier du composant.

- `Hashtable skeletons` : cet attribut est une table de hachage des squelettes du composant. Elle fait correspondre le nom d'un squelette à l'objet qui le représente. Le nom est celui qui a été défini dans la description XML du type du composant, alors que l'objet est une instance d'une classe de squelette spécifique[1]. La table est utilisée pour l'introspection et la connexion du composant.
- `Hashtable stubs` : cet attribut est une table de hachage des talons du composant. Sa fonction est identique à la fonction de la table des squelettes.
- `ContainerRef ref` : cet attribut est la référence du conteneur, présentée dans la suite.

La génération de la classe de conteneur pour un composant consiste en la génération d'une sous-classe de `Container` dont le constructeur est spécifique au type du composant. Ce constructeur crée les instances des talons et des squelettes à partir des leurs classes générées et initialise les tables de hachage. Étant donné que les interfaces fournies et requises d'un composant sont définies statiquement (cf. Chapitre 4, Section 4.2.1, p. 66), l'opération d'initialisation d'un conteneur définit une fois pour toutes ses structures internes.

Considérons l'exemple du conteneur du client de l'application "Hello World" (Figure 5-10.). Son constructeur ne crée qu'un seul talon, instance de la classe `Client_Comp_Hello_itf_Stub` présentée précédemment. Il initialise la table des talons et enregistre le talon sous son nom de `serverHello`. Comme le type `Client_Comp` du client n'a pas de squelette, le constructeur n'en crée aucun.

```
class Client_Comp_Container extends Container {
   public Client_Comp_Container() {
      super();
      //création du talon
      Client_Comp_Hello_itf_Stub stub = new Client_Comp_Hello_itf_Stub(this);
      //initialisation de la table des talons, pas de création de squelettes
      stubs.put("serverHello", stub);
}}
```

Figure 5-10. Conteneur de client dans "Hello World"

Référence de conteneur

La référence de conteneur, instance de la classe `ContainerRef`, est la référence globale de composant, introduite au Chapitre 4, Section 4.2.2. Elle est unique et créée lors de la création d'une instance de composant. Dans FAR, elle est composée de l'identificateur de l'environnement d'exécution du composant et d'un numéro. La première information identifie de manière unique l'environnement d'exécution, alors que le numéro identifie de manière unique le conteneur au sein de cet environnement. Le numéro est un simple entier incrémenté à chaque fois qu'un conteneur est créé.

La référence de conteneur n'est pas seulement un moyen d'identification d'une instance de composant mais aussi un moyen de communication. Les informations qu'elle détient sur l'environnement d'exécution sont utilisées pour acheminer les appels et pour rendre transparente la répartition. En effet, la référence de conteneur permet de s'adresser de manière indifférente à une instance locale ou distante.

La référence de conteneur est le seul moyen pour utiliser les services fournis par un conteneur dont, par exemple, l'accès aux références des talons et des squelettes d'un composant.

1. Il s'agit des sous-classes de `PointComm_Impl` générées à partir des déclarations des types de composants.

Interface de conteneur

La classe `Container` fournit des méthodes d'introspection et de gestion de connexions qui permettent aux conteneurs de gérer les assemblages de composants.

Les méthodes d'introspection sont :

- `PointComm_itf getStub_fromName(String name)` : cette méthode retourne le talon qui a été nommé `name` dans la description XML du composant. En local, le résultat c'est la référence de l'objet Java qui est retournée. À distance, c'est un objet sérialisé qui permet de désigner le talon qui est retourné. Si on considère l'exemple du conteneur du client dans "Hello World", l'appel à `getStub_fromName("serverHello")` retournera la référence du talon du client.
- `PointComm_itf getSkel_fromName(String name)` : cette méthode retourne le squelette qui a été nommé `name` dans la description XML du composant. Elle a le même fonctionnement que la méthode précédente. Dans le cas du serveur de "Hello World", l'appel à `getSkel_fromName("forClientHello")` retournera la référence du squelette du serveur.
- `Enumeration getStub_fromType(String itf_name)` : cette méthode est identique à la première méthode mais fait la recherche sur un type de talon. Étant donné qu'un composant peut avoir plusieurs interfaces requises du même type, cette méthode retourne une liste. Dans le cas de "Hello World" la recherche aboutit si elle est faite sur le type `Hello_itf`.
- `PointComm_itf getSkel_fromType(String itf_name)` : retourne le squelette de type `itf_name`[1].
- `Vector getStubs()` : retourne la liste des talons.
- `Vector getSkels()` : retourne la liste des squelettes.
- `Vector getConnections()` : retourne la liste des connexions du composant. Pour cela, le conteneur utilise les informations stockées dans les instances de `PointComm_Impl` correspondant aux talons et squelettes.

Les méthodes de gestion de connexions sont :

- `void connect(PointComm_itf required, PointComm_itf provided)` : cette méthode établit une connexion entre un talon `required` et un squelette `provided`. Son traitement consiste à vérifier la compatibilité des deux points de communication et à mettre à jour leurs structures internes. La vérification inclut la vérification du type, du mode de communication et du nombre de connexions autorisées. La mise à jour des structures internes du talon et du squelette concerne la mise à jour de leurs listes de connexions. Les deux traitements reposent sur les méthodes fournies par `PointComm_Impl` (cf. La classe `CommPoint_Impl`, p. 100).
- `void disconnect(PointComm_itf required, PointComm_itf provided)` : cette méthode a l'action inverse de la méthode précédente. Elle détruit une connexion établie entre deux composants tout en mettant à jour les structures internes du talon et du squelette concernés.

---

1. Rappelons que les composants dans FAR ne peuvent pas avoir plusieurs interfaces fournies du même type (cf. Section 5.3.3).

Connexions de composants à l'aide des conteneurs

Les interfaces d'introspection et de connexion des conteneurs sont essentielles lors de l'établissement des connexions entre composants. Dans la suite, nous décrivons les processus de connexion dans les cas synchrone et asynchrone.

Pour illustrer l'utilisation des primitives de connexion dans le cas synchrone, considérons de nouveau l'application “Hello World”. La connexion à établir est entre le talon du client et le squelette du serveur. Les étapes de connexion sont les suivantes :

- *Récupération des références du talon et du squelette*. La connexion ne peut être établie si les deux points de communication ne sont pas identifiés. Pour ce faire, il faut récupérer leurs références à l'aide des interfaces d'introspection. Ainsi, la référence du talon est obtenue en appelant `getStub_fromName("serverHello")` sur le conteneur du client, alors que la référence du squelette est obtenue en appelant `getStub_fromName("forClientHello")` sur le conteneur du serveur.
- *Appel de la primitive* `connect`. La processus de connexion commence par un appel à la primitive `connect` sur le client. Les paramètres passés à cette primitive sont les deux références récupérées dans l'étape précédente.
- *Vérification de la possibilité de connexion*. La connexion réussit si le client n'est pas déjà connecté à un autre composant. En effet, le talon du client et le squelette du serveur sont compatibles en ce qui concerne leur type d'interface (`Hello_itf`) et leur mode de communication (synchrone). Le seul problème peut venir de la restriction sur le nombre de connexions du client qui ne peut être supérieur à un.
- *Mise à jour du talon et du squelette*. Si la connexion réussit, le nouvel état est enregistré en appelant `addConnection` sur le talon et le squelette. Cette méthode est définie par leur classe mère `PointComm_Impl`.

Dans le cas asynchrone, l'interface qui est utilisée pour la connexion est requise pour les producteurs d'événements et fournie pour les consommateurs (cf. Chapitre 4, Section 4.2.3, p. 67). Étant donné que la connexion de deux composants se fait dans le sens interface requise - interface fournie, la connexion devrait être demandée par les producteurs. Or, il est plus logique que ce soient les consommateurs qui se connectent à un producteur. Pour permettre ce fonctionnement, l'interface asynchrone est accompagnée d'une interface spécifique de connexion qui est générée automatiquement. Cette interface complémentaire est fournie pour les producteurs et requise pour le consommateurs.

Considérons l'exemple d'un producteur et d'un consommateur (Figure 5-11.). Le producteur émet des événements de type E, définis par l'interface `EventE_itf`. Il requiert donc cette interface, alors que pour le consommateur elle est fournie. Si le talon et le squelette respectifs sont `stubE` et `skeletonE`, l'opération `connect(stubE, skeletonE)` assure que l'événement `E` soit transmis du producteur vers le consommateur. Pour que le consommateur puisse s'abonner à cet événement, il requiert une interface `EventEConnection`_itf qui est fournie par le producteur. Ainsi, la connexion du consommateur au producteur par l'interface `EventEConnection`_itf se traduit par une connexion inverse pour `EventE_itf`.

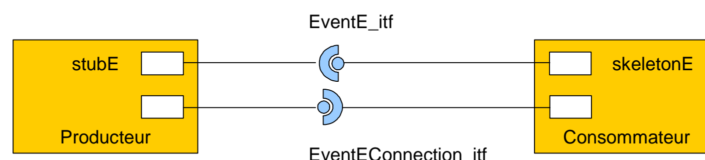


Figure 5-11. Schéma de connexion asynchrone

5.5 Environnement d'exécution

Dans notre implémentation, un environnement d'exécution de composants est une machine virtuelle Java [66]. Cet environnement d'exécution est défini par un serveur d'exécution (cf. Chapitre 4, Section 4.5, p. 85), mis en place par une classe `ServerCtxt`. Ce serveur a trois fonctions : le recensement des composants s'exécutant sur le serveur, le nommage des composants et la mise à disposition d'un service de transport de messages entre serveurs. Nous détaillons ces trois fonctions dans ce qui suit.

5.5.1 Gestion des composants locaux

La classe `ServerCtxt` joue le rôle de serveur de noms pour les composants locaux. Elle dispose d'une table de hachage contenant les références de tous les composants s'exécutant dans le serveur. Cette information est utilisée pour l'acheminement des appels vers les bons composants destinataires et pour des optimisations dans le cas d'appels entre composants colocalisés.

La mise à jour de la table de hachage est faite en utilisant les méthodes suivantes :

- `static ContainerRef create(String classname, String[] args)` : cette méthode est la méthode de création d'instance d'un composant. Elle se charge, en plus de la création de l'instance de la classe métier du composant, de la création de son conteneur, de la création de la référence unique pour ce conteneur et de son enregistrement dans la table de hachage. La référence unique que nous avons implémentée dans FAR est basée sur la numérotation incrémentale des conteneurs créés au sein d'un serveur. Elle est donc un couple de la forme `<adresse du serveur local, compteur de type entier>`.
- `static void destroy(ContainerRef comp)` : cette méthode détruit une instance de composant. Le traitement détruit les structures de l'instance et efface les informations la concernant de la table de hachage[1].

La consultation de la table de hachage est faite en utilisant la méthode suivante :

- `static Container getContainer(int cn)` : retourne le conteneur de numéro `cn`.

5.5.2 Nommage de composants

La classe `ServerCtxt` fournit un service de nommage qui permet l'enregistrement d'instances de composants sous des noms symboliques. Il est accessible via les méthodes suivantes :

- `static void register(ContainerRef ref, String name)` : permet d'enregistrer un composant identifié par `ref` sous le nom `name`.
- `static ContainetRef lookup(String name)` : permet d'obtenir la référence d'un composant nommé `name`.

Le service de nommage est lui-même un composant et les communications avec lui utilisent les mêmes mécanismes que les invocations entre composants métier.

1. La libération de la mémoire n'étant pas directement accessible en Java, cette destruction se contente à déréférencer les structures. Le ramasse-miettes, contrôlé par la machine virtuelle, se charge de la libération de tous les objets non référencés.

5.5.3 Service de transport de messages

Le service d'envoi et de réception de messages est le service utilisé par les talons et les squelettes pour l'acheminement d'appels entre composants. Il est spécifié par une interface contenant les trois méthodes suivantes :

- `static TransportRef getTransportRef()` : retourne la référence du serveur local.
- `static Message msgSend(TransportRef dest, Message msg, boolean sync)` : cette méthode envoie le message `msg` au serveur destinataire identifié par `dest`. Le message peut représenter un appel synchrone ou un événement asynchrone ce qui est reflété dans le paramètre `sync`. Quand `sync` est vrai, l'appel est bloquant et retourne la réponse du serveur destinataire. Sinon, l'appel est non bloquant et retourne `null`.
- `static Message msgRecv(Message invc, boolean sync)` : traite un message `msg` en provenance d'un serveur distant. `sync` indique si l'émetteur attend une réponse.

Dans FAR nous avons implémenté ce service en utilisant les *sockets* TCP/IP[1] ainsi que le mécanisme de sérialisation Java. Plus précisément, les méthodes précédentes sont implantées de la manière suivante :

- `TransportRef getTransportRef()` : retourne l'adresse du serveur local, constituée d'une adresse IP et d'un numéro de port TCP.
- `Message msgSend(TransportRef dest, Message msg, boolean sync)` : dans le cas de composant s'exécutant dans le même serveur, retrouve la référence du squelette correspondant et appelle directement `dispatch`. Dans le cas de composant distant, encapsule msg et sync dans une objet qui est sérialisé et envoyé, via un *socket* TCP, vers le serveur `dest`. Attend et retourne la réponse dans le cas d'invocation synchrone.
- `Message msgRecv(Message invc, boolean sync)` : reçoit un message par un socket, décode le message, retrouve le composant destinataire en utilisant la table de hachage des composants et lui transmet l'appel. Si nécessaire, emballe et renvoie une réponse, toujours via les *sockets* TCP.

5.5.4 Déploiement

Nous avons présenté la manière de décrire, d'implémenter, d'interconnecter et de faire communiquer les composants. Il nous reste à voir comment une application est définie en tant qu'un assemblage de composants et comment son architecture est déployée.

Dans FAR, nous nous sommes inspiré des travaux sur OpenCCM [104]. Le processus du déploiement est décrit dans des programmes de déploiement qui contrôlent les étapes d'identification des serveurs pour les instances de composant, de création des instances et de leur interconnexion.

- *Identification des serveurs*. L'identification des serveurs est faite en utilisant le service de nommage. Tout serveur est enregistré auprès de ce service sous un nom symbolique qui fait correspondre à ce nom une référence de déploiement. Cette référence permet

1. Java fournit des classes prédéfinies pour la manipulation des adresses IP et des sockets TCP/IP [66].

de s'adresser à distance à un serveur et de lui donner des directives de déploiement dont la création et l'interconnexion d'instances.

- *Création des instances*. La création d'instances est faite en utilisant la méthode `create` de la classe `ServerCtxt`. Après la création d'une instance, le programme de déploiement dispose de sa référence globale qu'il utilise pour la récupération des références des points de communication. Le programme de déploiement est également celui qui enregistre les instances de composant dans le service de nommage.
- *Interconnexion des instance*s. Pour l'interconnexion d'instances, le programme de déploiement utilise les méthodes d'introspection et de gestion de connexions fournies par la classe `Container`. Il appelle les méthodes d'introspection sur les références globales des instances afin d'obtenir les références des talons et des squelettes à connecter. Il les connecte avec les méthodes de connexion.

Pour illustrer ce processus, considérons un programme de déploiement possible de l'application "Hello World" (Figure 5-12.). Le programme retrouve les références de déploiement de deux serveurs d'exécution et y crée respectivement le client et le serveur. Il récupère les références du talon du client et du squelette du serveur en utilisant les méthodes d'introspection des conteneurs des deux instances et les utilise pour interconnecter ces dernières.

```
public deployApplication() {
        //récupérer les références de déploiement de deux serveurs
        DeploymentRef server1 = ServerCtxt.lookup("Server1");
        DeploymentRef server2 = ServerCtxt.lookup("Server2");
        //créer une instance de Client_Comp sur server1. client désigne le conteneur de l'instance créée.
        client = server1.create("Client_Comp", null);
        //créer une instance de Server_Comp sur server2
        server = server2.create("Server_Comp", null);
        //récupérer la référence du talon du client
        PointComm_itf helloClient =  client.getStub_fromName("serverHello");
        //récupérer le référence du squelette du serveur
        PointComm_itf helloServer = server.getSkel_fromName("forClientHello");
        //connecter les deux instances
        client.connect(helloClient, helloServer);
}
```

Figure 5-12. Déploiement de l'application "Hello World"

Chapitre 6

Les protocoles de duplication et de cohérence dans FAR

Après avoir décrit notre mise en œuvre pour des applications sans duplication ni cohérence, nous considérons, dans ce chapitre, les protocoles de duplication et de cohérence dans FAR. Après une description de l'instrumentation des applications métier pour permettre l'intégration de protocoles de duplication et de cohérence (Section 6.1), nous présentons la manière dont ces protocoles sont programmés (Section 6.2) et intégrés dans les applications métier (Section 6.3). Les fonctionnalités fournies sont illustrées au fur et à mesure à l'aide d'un exemple.

6.1 Instrumentation des applications métier

Pour permettre la composition entre une application métier, construite sans duplication ni cohérence, et un protocole qui fournit des traitements de duplication et de cohérence, l'application métier a besoin d'être instrumentée (cf. Chapitre 2, Section 4.4.2, p. 83). Cette instrumentation consiste à permettre l'accès à l'état des composants[1] et à intégrer des mécanismes de contrôle d'état stable. L'accès à l'état est nécessaire pour la création et la synchronisation de copies de composants, alors que le contrôle d'état stable garantit la manipulation d'états cohérents (cf. Chapitre 2, Section 4.3.1, p.71). L'accès à l'état est discuté dans la section 6.1.1, alors que les mécanismes de contrôle d'état stable sont considérés dans la section 6.1.2.

6.1.1 Accès à l'état

Dans notre implémentation, l'accès à l'état d'un composant peut être fait de deux manières : à travers une interface d'accès à l'état ou en utilisant le mécanisme de sérialisation de Java [140]. Le choix de la méthode d'accès est laissé au programmeur de composants qui peut également décider d'intégrer les deux mécanismes. Dans la suite, nous présentons ces deux mécanismes avec leurs avantages et les inconvénients.

Interface d'accès à l'état

Dans le cas des interfaces d'accès à l'état, les programmeurs de composants modélisent l'état d'un composants en termes d'attributs accessibles à travers des méthodes de consultation (méthodes *get*) et de mise à jour (méthodes *set*). Les attributs manipulés peuvent correspondre

1. Sous le terme d'accès à l'état nous regroupons la capture et la restauration d'état.

à des attributs de la classe Java qui fournit l'implémentation métier du composant ou cacher des traitements plus complexes. En effet, de tels traitements sont nécessaires afin d'éviter le partage par référence résultant des appels Java classiques et afin d'assurer la *copie* de l'état du composant. Par exemple, si l'objet consulté s'avère être un graphe d'objets, l'interface doit être capable de fournir toutes les informations nécessaires à la copie de ce graphe.

Le comportement global de l'interface d'accès à l'état doit permettre :

- pour un composant A, de type T, avec une interface d'accès I
- à un moment où le composant A n'est pas en cours de modification
- de créer une nouvelle instance B du même type T
- de capturer l'état de A en utilisant l'interface I
- de mettre à jour l'état de B avec l'état capturé de A en utilisant I, tout en s'assurant que B détient bien une copie d'état et ne partage rien avec A
- d'obtenir un composant B dont l'état est cohérent et dont le fonctionnement vérifie les spécifications pour le type T

L'utilisation d'une interface d'accès à l'état complique la tâche du programmeur puisqu'elle impose une connaissance parfaite de la structure du composant ainsi que du fonctionnement du langage Java. Toutefois, l'approche a l'avantage de permettre la mise en place de captures et de restaurations d'états partiels qui peuvent être utilisées dans des solutions de cohérence qui prennent en compte la sémantique des applications (cf. Chapitre 2, *Techniques de synchronisation (Comment)*, p. 48).

Sérialisation Java

Nous considérons la sérialisation, le mécanisme de capture et de restauration d'état de Java, puisqu'elle traite tous les problèmes soulevés dans le point précédent. Elle automatise la création de copies d'objets tout en traitant de manière cohérente les références et les graphes. Elle représente l'état d'un objet sous forme binaire et est appliquée pour les objets qui implémentent l'interface Java prédéfinie `Serializable`. Par défaut, la sérialisation capture l'état complet des objets mais peut être configurée pour ignorer certains des attributs.

La sérialisation Java a les avantages et les inconvénients inverses à ceux de l'approche avec interface d'accès à l'état. En effet, elle simplifie les traitements de capture et de restauration mais ne permet pas de traitement partiel, ni sémantique.

Exemple

Prenons l'exemple de l'application "Hello World" présentée dans le chapitre précédent (cf. Chapitre 5, Section 5.3.1, p. 95). Considérons le cas où le serveur de l'application dispose d'une table de hachage pour stocker les différentes chaînes de caractères affichées à la demande du client. L'état du serveur, dans ce cas, est cette table qui fait correspondre, à chaque chaîne, le nombre d'appels client pour cette chaîne. L'implémentation `Server_Comp` du serveur avec état est donnée à la Figure 6-1.

Pour pouvoir dupliquer le serveur afin d'intégrer un protocole, par exemple, de tolérance aux fautes, le serveur doit être instrumenté afin de permettre l'accès à son état. Comme il a été expliqué précédemment, ceci peut être fait en utilisant la sérialisation Java ou en introduisant une interface de méthodes d'accès à l'état.

```
class Server_Comp extends Component implements Hello_itf {
   Hashtable strings;
   public void print(String s) throws Exception {
      System.out.println(s);
      //récupérer le nombre d'appels pour la chaîne s
      Integer nb_appels = (Integer)strings.get(s);
      if (nb_appels == null)                  //si la chaîne n'a jamais été affichée
         strings.put(s, new Integer(1));      //l'enregistrer avec nombre d'appels égal à 1
      else                                    //sinon augmenter le nombre d'appels de 1
         strings.put(s, new Integer(nb_appels.intValue()+1);
}}
```

Figure 6-1. Serveur de l'application "Hello World". Implémentation avec état.

(i) Accès à l'état du serveur de "Hello World" en utilisant la sérialisation Java

Étant donné que l'état du serveur est une table de hachage contenant des chaînes de caractères et des objets de type `Integer`, l'introduction de la sérialisation Java est triviale. En effet, la table de hachage, les chaînes et les `Integer` sont déjà des objets sérialisables. Il ne reste plus qu'à rajouter l'interface `Serializable` au niveau de la classe `Server_Comp` (Figure 6-2.). Il est désormais possible de capturer ou de restaurer l'état d'une instance de `Server_Comp` en utilisant les méthodes standards de la sérialisation Java `writeObject` et `readObject`.

```
class Server_Comp extends Component implements Hello_itf, Serializable {
   Hashtable strings;
   public void print(String s) throws Exception {...}
}
```

Figure 6-2. Accès à l'état du serveur de "Hello World" par la sérialisation Java

(ii) Accès à l'état du serveur de "Hello World" en utilisant une interface d'accès

L'état du serveur peut être rendu accessible à travers deux méthodes de consultation et de modification de ce dernier. Cette technique est intéressante puisqu'il est possible de ne copier qu'une partie de la table de hachage, comme, par exemple, la partie contenant les chaînes commençant par "Hello". Si on appelle `getState` et `setState` ces deux méthodes, l'implémentation de `Server_Comp` devient celle montrée à la Figure 6-3.

6.1.2 Contrôle d'état stable

L'accès à l'état est indispensable mais non suffisant pour la création de copies. En effet, il est nécessaire de s'assurer que l'état manipulé est un état cohérent. En d'autres termes, l'état copié ne doit pas être en cours de modification par des calculs applicatifs pendant les traitements de capture et de restauration. Or, ni l'utilisation d'une interface d'accès à l'état, ni la sérialisation Java, ne garantissent l'exclusion mutuelle entre les traitements de capture et de restauration et les traitements de calcul applicatif. Par exemple, dans les deux implémentations du serveur de l'application "Hello World" montrées précédemment, rien n'empêche que la table de hachage soit modifiée pendant le traitement de copie (`writeObject` dans le cas de la sérialisation Java et `getState` dans le cas de l'interface d'accès). En effet, pour que les traitements de capture et de restauration soient exécutés en mode exclusif, l'interface d'accès à l'état, ainsi que l'interface de sérialisation doivent être implémentées avec des primitives explicites de synchronisation.

```
interface Accessors {
   Hashtable getState(String s);
   void setState(Hashtable h);
}

class Server_Comp extends Component implements Hello_itf, Accessors {
   Hashtable strings;
   public void print(String s) throws Exception {...}

   public Hashtable getState(String s) {
      Hashtable result = new Hashtable();
      //pour toutes les chaînes enregistrées dans la table
      for (Enumeration e = string.keys() ; e.hasMoreElements() ;) {
         String key = (String)e.nextElement();
         //si la chaîne commence par la chaîne s passée en paramètre
         if (key.startsWith(s))
            //enregistrer la chaîne dans la table de copie en copiant l'objet Integer du nombre d'appels
            result.put(key, strings.get(key).clone());
      }
      return result;
   }
   public void setState(Hashtable h) {strings = h;}
}
```

Figure 6-3. Accès à l'état du serveur de "Hello World" par des méthodes d'accès

Dans la suite de cette section, nous analysons quels sont les mécanismes nécessaires pour la mise en place d'un contrôle d'état stable et présentons notre réalisation de ces mécanismes.

Mécanismes pour le contrôle d'état stable

Pour assurer que l'état d'un composant n'est pas modifié pendant des traitements de capture et de restauration, il est nécessaire de pouvoir contrôler les modifications de cet état. Plus particulièrement, il faut connaître les traitements qui modifient l'état du composant, intercepter les modifications en cours d'exécution et contrôler ces exécutions.

- *Identification des traitements applicatifs modifiant l'état d'un composant* : pour assurer l'exclusion mutuelle entre les traitements de capture et de restauration et les traitements applicatifs qui modifient l'état d'un composant, il est nécessaire de pouvoir distinguer ces derniers parmi l'ensemble de tous les traitements applicatifs.
- *Interception des modifications en cours d'exécution* : pour connaître les modifications que subit un composant à un moment donné, il est nécessaire d'intercepter les débuts et les fins des exécutions des traitements qui modifient son état. Ces derniers font partie de l'ensemble identifié dans le point précédent.
- *Contrôle des traitements applicatifs modifiant l'état d'un composant* : un traitement de capture ou de restauration ne peut être lancé en présence de modifications en cours et ne peut s'exécuter si les modifications applicatives ne sont pas bloquées. Par conséquent, lors d'une intention de capture ou de restauration, il est nécessaire de pouvoir bloquer les nouveaux appels aux traitements de modification tout en autorisant la terminaison des modifications en cours. Les mécanismes d'attente et de blocage de traitements de modification reposent sur l'interception du point précédent.

Dans la suite, nous présentons notre approche de gestion de chacun de ces points.

Identification des traitements applicatifs modifiant l'état d'un composant

L'état d'un composant n'est modifié que par des méthodes de son implémentation métier. La définition de ces méthodes peut faire usage des patrons de nommage, être basée sur la synchronisation Java ou encore être faite dans un descripteur séparé.

- *Patrons de nommage*. L'utilisation de patrons de nommage consiste à nommer toutes les méthodes qui modifient l'état d'un composant de manière reconnaissable. Par exemple, un tel patron peut exiger que les noms de toutes ces méthodes commencent par le mot *set*. Nous avons rejeté cette approche à cause des contraintes de programmation qu'elle implique pour les programmeurs de composants.
- *Synchronisation Java.* Le mécanisme de synchronisation de Java permet de définir, en utilisant le mot-clé `synchronized`, les méthodes qui doivent s'exécuter en exclusion mutuelle. Toutefois, dans le cas général, parmi les méthodes `synchronized` se trouvent non seulement les méthodes de modification, mais également les méthodes de consultation. En plus, cette approche offre une synchronisation simpliste qui n'est pas adaptée à tout type de problème[1].
- *Descripteur séparé*. La troisième approche, celle que nous avons retenue, consiste à lister les méthodes qui modifient l'état d'un composant dans un descripteur externe fourni par le programmeur de composant. Les méthodes concernées peuvent inclure non seulement des méthodes spécifiées dans les interfaces du composant mais également des méthodes internes. En effet, ces dernières doivent être listées dans les cas où elles sont exécutées à l'initiative du composant et non suite à un appel externe qui est intercepté par le conteneur du composant.

  Pour le format du descripteur, nous avons choisi XML à cause de la disponibilité d'outils d'analyse (cf. Section 5.3.1, p. 95).

Dans l'exemple de l'application "Hello World", le descripteur XML pour le type de composant `Server_Comp` spécifie que les méthodes qui modifient l'état des instances de ce type se résument à la méthode `print`. Le descripteur est montré à la Figure 6-4.

```
<modification_methods>
     <component_type> Server_Comp </component_type>
     <method> print(String) </method>
</modification_methods>
```

Figure 6-4. Description des méthodes modifiant l'état du serveur dans "Hello World"

Interception des modifications en cours

Pour identifier les modifications qu'un composant subit à un moment donné, il est nécessaire d'intercepter les débuts et les fins d'exécution des méthodes définies dans le descripteur XML présenté. Par exemple, pour savoir si le serveur de "Hello World" est en train d'être modifié, il faudrait intercepter les débuts et les fins d'exécution de la méthode `print`.

---

1. Considérons le problème de partage de ressources lecteurs-rédacteur. Une ressource peut être lue en concurrence par plusieurs lecteurs mais est éditée de manière exclusive par un rédacteur. Si on modélise la ressource par une classe Java avec deux méthodes de lecture et d'écriture qui sont spécifiées comme étant `synchronized`, l'exclusion va être assurée non seulement pour un rédacteur, mais également entre deux lecteurs et ne permettra pas la lecture concurrente.

L'interception des débuts ou des fins d'une méthode de modification est signalée au conteneur du composant modifié à l'aide des deux méthodes de la classe `Container` suivantes :

- `void beginMethod(int methodId, String threadId)` : cette méthode se charge d'ajouter dans une liste de la classe `Container` l'information d'une nouvelle modification en cours. Cette exécution est identifiée par l'identificateur `methodId` de la méthode concernée et par l'identificateur `threadId` du processus chargé de son exécution. La méthode est également utilisée pour bloquer la modification demandée si des traitements de capture ou de restauration d'état sont en cours d'exécution.
- `void endMethod(int methodId, String threadId)` : cette méthode enlève de la liste des modifications en cours, l'exécution identifiée par `methodId` et `threadId`.

L'interception des débuts et des fins d'exécution des méthodes de modification est faite de deux manières. Dans les cas où les appels des méthodes et les retours de résultat sont gérés par les squelettes, l'interception est assurée par ces derniers. Dans les cas où il s'agit d'appels internes ou d'appels asynchrones sans retour de résultat, l'interception est intégrée dans le code compilé des composants.

- *Interception des méthodes de modification dans les squelettes*. Les squelettes interceptent les appels de méthodes provenant d'autres composants et donc interceptent les débuts des exécutions de ces méthodes. Ils interceptent également les fins des méthodes *synchrones* puisqu'ils s'occupent de renvoyer un résultat. Les squelettes peuvent, par conséquent, enregistrer au niveau des conteneurs, les débuts et les fins des exécutions correspondant en appelant respectivement `beginMethod` et `endMethod`. Par exemple, dans le cas de “Hello World”, les modifications en cours sont interceptées au niveau du squelette du serveur qui peut témoigner du début et de la fin des traitements de la méthode `print`.
- *Interception des méthodes de modification dans le code compilé des méthodes*. Les fins d'exécution des méthodes asynchrones[1] et les appels de méthodes internes ne sont pas interceptés par les structures des conteneurs. En effet, dans le cas de méthodes asynchrones appelées depuis l'extérieur, les squelettes ne font que lancer l'exécution et n'attendent pas de terminaison. Dans le cas d'appels internes, les composants s'appellent eux-mêmes et leurs appels ne passent ni par les squelettes ni par les talons (cf. Chapitre 5, Section 5.4.1, p. 99).

  Pour assurer les appels vers `beginMethod` et `endMethod`, nous avons choisi de modifier le code des méthodes à intercepter. Cette modification n'est pas imposée au programmeur de composants mais est faite après leur compilation et uniquement dans le cas où l'instrumentation de l'application métier devient nécessaire c'est à dire avant l'intégration d'un protocole de duplication et de cohérence.

  Étant donné que nos composants sont programmés en Java et que leur code compilé est sous la forme de *bytecode*[2], nous avons utilisé BCEL [11], un outil de manipulation de *bytecode*. En nous basant sur l'interface de programmation de BCEL, nous avons facilement inséré les instructions d'appel de `beginMethod` en début de méthode et les instructions d'appel de `endMethod` avant les instructions `return` de fin de méthode.

---

1. Il s'agit des méthodes qui traitent la réception d'un événement asynchrone.
2. Le bytecode est le code exécutable des programmes Java pour la machine virtuelle Java. Le lecteur peut se référer à la spécification de cette machine virtuelle [66] pour plus de détails.

Contrôle des traitements applicatifs

Nous définissons es traitements spécifiques pour l'exclusion mutuelle entre les traitements applicatifs qui modifient l'état d'un composant et les traitements de capture et de restauration. Ces traitements ont pour objectif de verrouille les composants lors de leurs captures et restaurations. Ils sont fournis aux protocoles de duplication et de cohérence qui peuvent baser leur traitements de gestion la dessus ou alors préférer d'implémenter leur propre mécanisme de synchronisation. Dans le Chapitre 9, nous décrivons un protocole qui fait usage du mécanisme de synchronisation fourni ainsi qu'un protocole qui définit sa gestion de verrous spécifique. Nous reviendrons sur ce point dans la section 6.2.2 qui parle des reconfigurations des applications liées à la duplication et à la cohérence.

Les traitements que nous fournissons doivent être exécutés avant et après les traitements de capture ou de restauration et sont implémentés par les méthodes de la classe `Container` suivantes :

- `void beginCaptureRestore()` : cette méthode garantit les conditions d'exclusion mutuelle pour les traitements de capture ou de restauration. Elle est appelée, de manière bloquante, par les entités voulant capturer ou restaurer l'état d'un composant. Elle se charge de bloquer les éventuels appels arrivants sur les méthodes de modification du composant et attend la terminaison des modifications en cours. Le blocage des appels se fait au niveau de la méthode beginMethod du conteneur. Les invocations sont mises dans une file d'attente qui est traitée à la fin des traitements de capture ou de restauration. L'attente de terminaison des exécutions en cours se fait en utilisant les informations dans la liste de ces exécutions. Quand cette liste devient vide, le conteneur est prévenu et termine la méthode beginCaptureRestore().
- `void endCaptureRestore()` : cette méthode signale la fin d'un traitement de capture ou de restauration et débloque la file d'attente des invocations sur le composant.

Si l'on applique ces traitements dans le cadre de la duplication du serveur de l'application "Hello World", la copie de ce dernier commence par l'appel de `beginCaptureRestore()`. Cette méthode bloque les nouveaux appels à `print`, attend la terminaison de tous les `print` en cours et finalement duplique le serveur. Quand la duplication est terminée, l'appel à `endCaptureRestore()` débloque les `print` en attente.

6.1.3 Synthèse

Pour pouvoir intégrer un protocole de duplication et de cohérence dans une application construite sans duplication, ni cohérence, les composants de cette dernière doivent supporter la duplication. Pour les rendre duplicables, nous les instrumentons en introduisant des primitives d'accès à l'état, ainsi que des primitives de contrôle d'état stable.

- *Primitives d'accès à l'état.* L'état des composants peut être rendu accessible de deux manières. La première repose sur l'utilisation de la sérialisation Java et bénéficie de l'automatisation des traitements de capture et de restauration d'état. La deuxième est basée sur l'implémentation d'une interface d'accès à l'état spécifique qui permet l'adaptation des primitives de capture et de restaurations aux besoins des applications.

- *Contrôle d'état stable.* Pour garantir la cohérence de l'état des composants manipulés, nous fournissons des primitives qui permettent l'exécution des primitives de capture et de restauration en mode exclusif. Ces primitives reposent sur la définition des méthodes de modification dans un descripteur XML et sur l'interception des débuts et des fins d'exécution de ces méthodes. Les primitives permettent aux méthodes de capture et de

restauration de bloquer les appels aux méthodes de modification d'un composant, d'attendre la terminaison des modifications en cours, d'effectuer leur travail et, finalement, de débloquer les appels bloqués.

6.2 Composants de protocole

Comme dans le cas des composants métier, les composants de protocole passent par une étape de description déclarative qui suit les mêmes règles que la description des composants métier. La programmation des composants de protocole diffère selon qu'il s'agit de composants clients, copies, gestionnaires ou services (cf. Chapitre 4, Section 4.3, p.70). Ces quatre types de composants sont détaillés dans ce qui suit.

6.2.1 Composants services

Les composants services sont des composants qui encapsulent des services système et sont utilisés sans modification dans les protocoles de duplication et de cohérence. Ils sont des instances de `ServiceComponent` et leur seule contrainte porte sur leur constructeur. Ils doivent, en effet, définir un nom unique pour le service et l'enregistrer auprès de l'environnement local d'exécution. Ce nom doit permettre la récupération de la référence du composant service qui est nécessaire au déploiement des composants de protocole. La désignation de services par des noms prédéfinis est une approche utilisée, par exemple, dans l'environnement CORBA.

Considérons l'exemple d'un composant de type `Timer` qui définit un service d'horloge au niveau d'un serveur d'exécution. C'est un composant qui, à la demande de différents autres composants qui fournissent un paramètre `X`, émet des événements tous les `X` secondes. L'initialisation du service peut être faite de la manière suivante, définie dans le constructeur de la classe `Timer` :

```
class ServiceComponent extends Component {
      ...
      public ServiceComponent(String s) {
         //création du conteneur, des talons, des squelettes...
         super();
         //enregistrement de la référénce de conteneur en utilisant le service de nommage
         ServerCtxt.register(s, this.myContainer);
      }
}
class Timer extends ServiceComponent implements Timer_itf {
      public Timer() {
         super("ServiceTimer");
         ... //initialisation spécifique
      }
      //implémentation
      ...
}
```

Figure 6-5. Implémentation d'un composant de service

6.2.2 Composants gestionnaires de configuration

Un gestionnaire de configuration est responsable de l'établissement de l'architecture initiale d'une application métier, ainsi que de l'intégration d'un protocole de duplication et de cohérence au sein de cette application. Il est en charge des reconfigurations concernant la création et la destruction de copies et est capable de donner des informations sur l'architecture de l'application en cours d'exécution.

Dans la suite, nous décrivons les fonctions de déploiement et d'introspection qu'un gestionnaire propose, ainsi que la structuration de ses traitements de reconfiguration. Nous discutons également du problème de cohérence lors d'une reconfiguration et présentons la solution que nous proposons.

Déploiement et introspection de l'architecture d'une application

Dans FAR, la notion de gestionnaire est représentée par la classe `ManagerComponent`. Cette classe propose les méthodes de déploiement et d'introspection suivantes :

- `void deployApplication()` : cette méthode se charge du déploiement d'une application métier. Elle fournit les traitements d'un programme de déploiement (cf. Chapitre 5, Section 5.5.4, p. 109) tout en incluant les traitements d'intégration d'un protocole de duplication et de cohérence, présentés dans la suite. Pendant ce déploiement, les références de toutes les instances créées sont enregistrées au niveau du gestionnaire pour les besoins d'introspection. La méthode n'est pas implémentée de manière générique mais est fournie au niveau des sous-classes de la classe `ManagerComponent` qui sont spécifiques aux applications métier. Son implémentation est à la charge de l'intégrateur de protocoles.
- `Enumeration getComponents()` : cette méthode fournit la liste des instances de composant qui constituent l'application sous le contrôle de ce gestionnaire. L'énumération contient les références des conteneurs qui peuvent être utilisées pour appeler les méthodes d'introspection de ces derniers (cf. Chapitre 5, Section 5.4.2, p. 104).
- `ContainerRef getComponent(String name)` : retourne la référence d'une instance nommée `name`.

Si on considère un gestionnaire de configuration pour l'application "Hello World", sa méthode `deployApplication()` contiendra les instructions de déploiement des deux instances du client et du serveur (cf. Chapitre 5, Figure 5-12., p. 110). Sa méthode `getComponents()` retournera une liste avec les références des deux instances. L'appel à `getComponent("C")` retournera la référence du client et l'appel à `getComponent("S")` retournera la référence du serveur.

Traitements de reconfiguration d'un gestionnaire

Dans notre proposition, nous avons défini les règles de reconfiguration, liées à la création et à la suppression de copies, comme étant de la forme événement-réaction. Nous avons introduit les notions de règles globales qui concernent plusieurs composants et de règles locales qui ne s'appliquent qu'à un seul composant (cf. Chapitre 4, *Moment de copie (Quand)*, p. 72). Dans FAR, les règles globales sont définies au niveau des gestionnaires de configuration, alors que les règles locales sont définies au niveau des composants qu'elles concernent[1]. Étant donné que les règles des deux types sont structurées de la même manière, nous ne détaillons que les règles

1. Toutefois, rien n'empêche de définir des règles locales au niveau d'un gestionnaire.

au niveau des gestionnaires. Dans la suite, nous présentons la forme que prennent ces règles de configuration et considérons la mise en place de règles génériques et spécifiques.

(i) Forme des règles de reconfiguration

Dans FAR, les événements et leurs réactions sont représentés par des méthodes dans des interfaces fournies et par leurs implémentations. Les composants qui produisent des événements appellent ces méthodes et déclenchent ainsi les traitements de réaction. Considérons, par exemple, un gestionnaire dont l'implémentation schématique est donnée à la Figure 6-6. Il traite un seul événement nommé `tooLong` représenté par la méthode du même nom de l'interface fournie `FT_itf`. Le gestionnaire reçoit un événement de ce type chaque fois que la méthode `tooLong()` est appelée et traite l'événement en exécutant cette méthode.

```
interface FT_itf {
   void tooLong();
}
class FT_Manager extends ManagerComponent implements FT_itf { //interface fournie
      ...
      public void tooLong() {...}  //réaction pour l'événement
}
```

Figure 6-6. Implémentation schématique d'un gestionnaire

Le gestionnaire `FT_Manager` peut être utilisé pour assurer la tolérance aux pannes d'un composant (Figure 6-7.)[1]. Dans ce protocole, il gère une copie primaire (composant `Primaire`) et deux copies secondaires (composants `Secondaire`) de ce dernier. Il traite les événements `tooLong` produits par un composant client qui fait appel à l'interface fournie `Service_itf` du composant dupliqué. Le gestionnaire interprète ces événements comme une alerte de panne puisqu'ils sont émis quand le client ne reçoit pas de réponse au bout d'un délai prédéfini[2]. Le traitement du gestionnaire consiste à vérifier l'état du composant primaire et, si l'état de panne est confirmé, de rediriger le client vers un des composants secondaires.

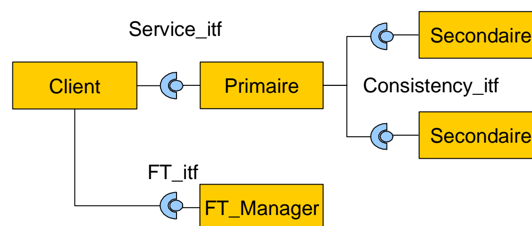


Figure 6-7. Modélisation d'une gestion de tolérance aux fautes

(ii) Mise en place de règles génériques

Nous avons défini les règles génériques comme étant des règles qui ne manipulent que des informations reliées aux architectures des protocoles c'est à dire leurs interconnexions et les types de leurs composants (cf. Chapitre 4, *Moment de copie (Quand)*, p. 72). Étant donné que ces modèles de protocoles sont instanciés lors de l'intégration de ces derniers dans des applica-

1. Le gestionnaire de type FT_Manager a également d'autres connexions au client et aux serveurs. Elles ne sont pas montrées sur la figure pour plus de lisibilité et seront introduites dans la suite.
2. La mesure du délai peut typiquement être faite en utilisant le composant service `Timer`.

tions métier, dans FAR, leurs règles génériques se traduisent par des règles qui portent sur leurs architectures instanciées. En effet, nous identifions, dans les architectures qui résultent de l'intégration des protocoles, les configurations qui correspondent aux configurations considérées dans les règles génériques et faisons porter ces règles sur les configurations identifiées.

Pour illustrer ce point, considérons l'application du protocole de tolérance aux pannes dans le cas du serveur de l'application "Hello World". L'intégration du protocole produit une architecture instanciée qui comprend, en plus des deux instances du client et du serveur, une instance de gestionnaire `FT_Manager` et deux instances copies du serveur. La règle générique pour `tooLong` est représentée par une règle qui traite des événements `tooLong` produits par le client de "Hello World". (Figure 6-8.).

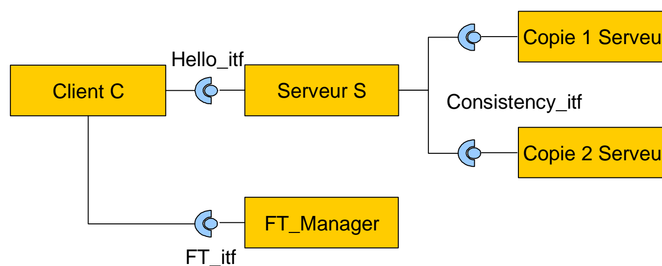


Figure 6-8. L'application "Hello World" avec le protocole de tolérance aux fautes

(iii)Mise en place de règles spécifiques

Les règles spécifiques sont des règles qui portent sur l'architecture qui résulte de l'intégration d'un protocole de duplication et de cohérence au sein d'une application métier. Elles permettent la spécialisation de la gestion de la duplication et de la cohérence en fonction de la situation considérée.

```
interface Spécifique_itf {
   void tooManyHellos();
}
class FT_Manager extends ManagerComponent implements FT_itf, Spécifique_itf{
                                                     //générique, spécifique
     ...
     public void tooLong() {...}
     public void tooManyHellos() {...}
}
```

Figure 6-9. Code de gestionnaire avec la règle spécifique `tooManyHellos`

Pour illustrer ce point, considérons la situation où l'application "Hello World" utilise le protocole de tolérance aux pannes mais veut le spécialiser en définissant une règle portant sur la gestion des chaînes de caractères commençant par "Hello". La règle consiste à déplacer ce type de chaînes dans un autre serveur si le serveur estime qu'il en contient un nombre trop important. Pour prendre cette règle en compte, un événement spécifique est défini et l'implémentation du gestionnaire est modifiée (Figure 6-9.). Ces modifications sont également répercutées au niveau de l'architecture qui résulte de l'intégration du protocole de tolérance aux pannes dans de l'application "Hello World" (Figure 6-10.). Dans cette architecture, les composants de protocole sont désormais accompagnés par un composant représentant le serveur de décharge des chaînes commençant par "Hello".

Le problème de cohérence globale

Dans la section consacrée à l'instrumentation des applications métier (Section 6.1), nous avons présenté le problème de cohérence lors des traitements de capture et de restauration d'état. Nous avons défini un composant comme étant cohérent lorsque son état n'est pas manipulé en parallèle par des méthodes applicatives et par des traitements de capture ou de restauration. La cohérence est *locale* puisqu'elle ne considère qu'un seul composant.

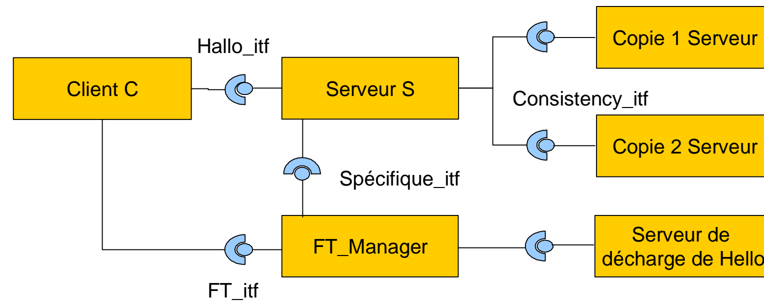


Figure 6-10. Architecture de “Hello World” avec la règle spécifique `tooManyHellos`

La notion de cohérence d'une application peut être définie de manière identique. Une application reste cohérente si on garantit que son état n'est pas manipulé de manière concurrente par des traitements applicatifs et des traitements de reconfiguration. La notion d'état est ici plus large et englobe les états des composants qui constituent l'application, ainsi que l'état des exécutions réparties en cours. La cohérence est *globale* puisqu'elle considère l'ensemble des composants.

Considérons les problèmes de cohérence globale qui peuvent survenir lors des actions de reconfiguration d'un gestionnaire, notamment lors de la création et de la suppression de copie, ainsi que lors de la synchronisation entre copies.

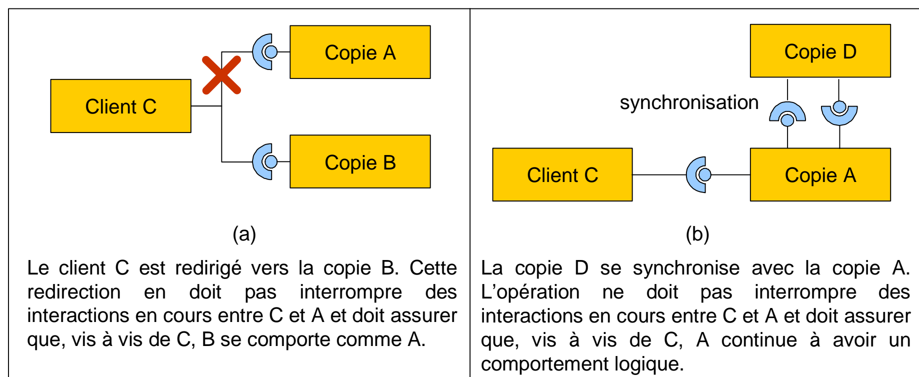


Figure 6-11. Situations mettant en danger la cohérence globale

- *Création et suppression de copie.* La création de copie est suivie de son interconnexion à d'autres composants. Dans le cas où la copie n'est connectée qu'à des composants copies, ses interactions sont uniquement des synchronisations, considérées dans ce qui suit. Dans le cas où la copie est connectée à des composants clients, le gestionnaire doit assurer leur cohérence d'exécution. En effet, la création de copie n'est qu'une partie d'une opération de reconfiguration plus complexe qui implique la redirection des clients vers cette nouvelle copie. Si les clients ont des interactions en cours, il ne faut pas qu'el-

les soient perturbées par cette redirection. Considérons l'exemple d'un client C qui demande un service auprès d'une copie A (Figure 6-11.a). C ne doit pas être redirigé avant la réception de la réponse de A. En plus, si C est redirigé vers la copie B, celle-ci doit se comporter comme la copie A ce qui implique qu'elle doit être synchronisée avec A.

- *Synchronisation entre copies*. La synchronisation entre copies est une opération qui met à jour les états et donc les comportements des copies. Or, ces copies sont impliquées également dans des interactions applicatives. Leurs synchronisations ne doivent pas perturber des interactions métier en cours et doivent préserver leur correction. Si on considère le même exemple du client C connecté à la copie A (Figure 6-11.b), la synchronisation de A avec une autre copie D ne doit pas fausser le traitement d'une requête de C. En plus, cette synchronisation ne doit pas modifier le comportement de A de manière incompréhensible du point de vue de C : elle doit lui fournir des garanties de cohérence (cf. Chapitre 2, *Critères de correction (Pourquoi)*, p. 45).

Mise en place des reconfigurations

Pour préserver la cohérence globale de l'application, nous fournissons un mécanisme qui suit le même principe que dans le cas de la cohérence locale : nous assurons que les traitements applicatifs et les traitements de reconfiguration s'exécutent en exclusion mutuelle. Nous ne gérons pas de manière explicite les garanties de cohérence : elles sont laissées à la charge des protocoles de duplication et de cohérence.

Comme dans le cas de la cohérence locale, l'exclusion mutuelle entre traitements de reconfiguration et traitements applicatifs qui modifient l'état de l'application est basée sur l'identification, l'interception et le contrôle de ces derniers.

(i) Identification des traitements applicatifs modifiant l'état d'une application

La solution que nous proposons pour l'identification des traitements applicatifs est inspirée des systèmes transactionnels. Dans ces systèmes, les traitements ne devant pas être perturbés par d'autres traitements s'exécutent en tant que *transactions*. Ces transactions définissent des suites d'opérations qui, dans le cas d'une terminaison réussie, se caractérisent par quatre propriétés :

- *Atomicité* : cette propriété garantit que toutes les opérations de la suite sont exécutées.
- *Cohérence* : cette propriété garantit que l'état global de l'application, après l'exécution de ces opérations, est cohérent.
- *Isolation* : cette propriété garantit que les résultats intermédiaires des opérations sont invisibles aux autres transactions.
- *Durabilité* : enfin, cette propriété garantit que le résultat de l'exécution des opérations est permanent, même après défaillance.

La mise en place de ces propriétés peut varier mais, de manière générale, les transactions utilisent une gestion de verrous et transmettent l'information de détention de ces verrous en utilisant des *contextes transactionnels*.

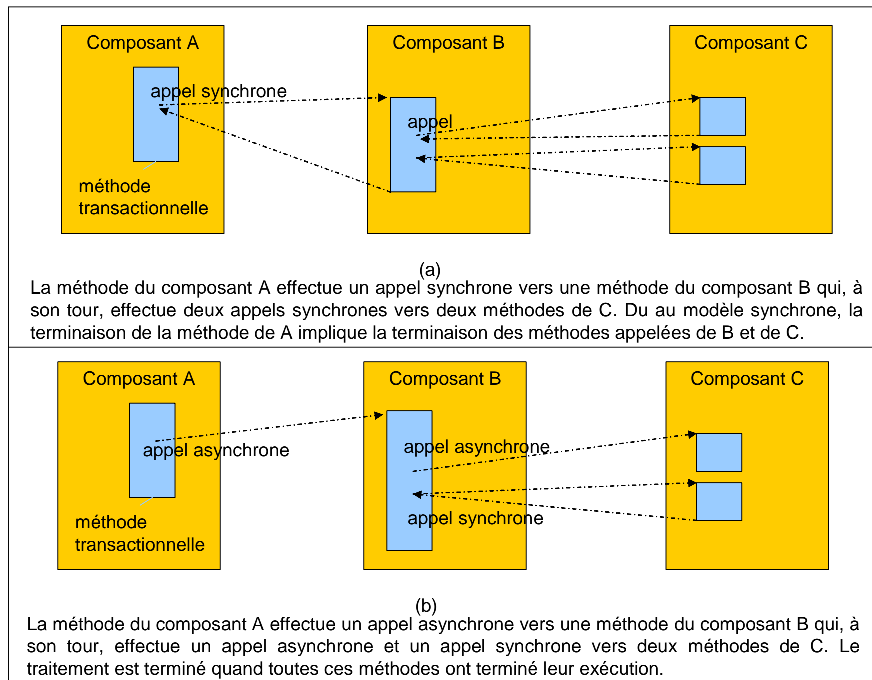
Notre solution simplifie le schéma général des transactions. Étant donné que nous nous intéressons surtout à l'exclusion mutuelle entre les traitements applicatifs et les traitements de reconfiguration, nous identifions ces traitements à des transactions caractérisées uniquement par la propriété d'isolation. Nous parlerons donc, dans le suite, de transactions applicatives et de transactions de reconfiguration.

Les transactions dans FAR, qu'il s'agisse de transactions applicatives ou de transactions de reconfiguration, sont représentées par des méthodes de composants. En effet, nous supposons que ce sont les opérations constituant le corps de ces méthodes qui définissent les suites des opérations réparties qui constituent les transactions. Les débuts et les fins des exécutions de ces méthodes sont respectivement considérés en tant que débuts et fins de transaction. Nous appelons ces méthodes des *méthodes transactionnelles*.

Pour connaître l'ensemble des méthodes transactionnelles définies au sein d'un composant, nous utilisons, comme dans le cas des méthodes de modification d'un composant, un descripteur XML. Nous garantissons l'exclusion mutuelle entre ces méthodes et les méthodes de reconfiguration du gestionnaire. L'exclusion porte sur l'ensemble des composants impliqués dans ces méthodes.

(ii) Interception des transactions applicatives en cours

L'interception des transactions applicatives en cours suit le principe introduit dans le cas de la capture et de restauration d'état : elle considéré les débuts et les fins des méthodes transactionnelles. Toutefois, cette approche ne marche bien que dans le cas synchrone. En effet, si les appels faits dans le corps d'une méthode sont synchrones, la fin de cette méthode garantit bien que tous les appels effectués sont également terminés (Figure 6-12.a). Cependant, si certains des appels sont asynchrones, leurs traitements peuvent toujours être en cours d'exécution lors de la terminaison de la méthode (Figure 6-12.b).



(a)
La méthode du composant A effectue un appel synchrone vers une méthode du composant B qui, à son tour, effectue deux appels synchrones vers deux méthodes de C. Du au modèle synchrone, la terminaison de la méthode de A implique la terminaison des méthodes appelées de B et de C.

(b)
La méthode du composant A effectue un appel asynchrone vers une méthode du composant B qui, à son tour, effectue un appel asynchrone et un appel synchrone vers deux méthodes de C. Le traitement est terminé quand toutes ces méthodes ont terminé leur exécution.

Figure 6-12. Terminaison de traitement applicatif

Pour traiter le cas asynchrone, nous proposons un mécanisme inspiré de la solution proposée dans [40]. Nous supposons que les invocations asynchrones faites par une méthode transactionnelle, ainsi que leurs propagations éventuelles, sont connues en avance. Le traitement initié est considéré comme terminé quand toutes ses propagations se sont terminées. La terminaison est validée par le gestionnaire qui, après la fin de la méthode transactionnelle, attend les terminaisons des toutes les propagations. Comme dans le cas de la synchronisation sur un composant, ce

mécanisme n'est pas imposé aux protocoles qui peuvent préférer d'implémenter une gestion spécifique dans le cas des appels asynchrones ou tout simplement les ignorer.

L'interception des débuts et des fins des méthodes transactionnelles est faite à l'aide des deux méthodes de la classe `ManagerComponent` suivantes :

- `GlobalMethodId beginGlobalMethod(int MethodId)` : cette méthode est appelée au commencement d'une méthode transactionnelle `MethodId`. L'appel est effectué par la méthode `beginMethod` du conteneur du composant initiant le traitement. Le résultat est un identificateur unique composé du type de la méthode et d'un numéro. Cet identificateur est propagé lors des appels effectués par la méthode transactionnelle.
- `void endGlobalMethod(GlobalMethodId id, ContainerRef c)` : cette méthode signale au gestionnaire que le composant `c` a terminé l'exécution issue d'un appel du traitement initié par la méthode transactionnelle identifiée par `id`.

Les méthodes locales `beginMethod` et `endMethod` sont celles qui appellent les méthodes globales `beginGlobalMethod` et `endGlobalMethod`. Dans les cas où tous les traitements de modification, même ceux qui ne portent que sur un composant, sont modélisées en tant que des méthodes transactionnelles, les appels à `beginMethod` et à `endMethod` marquent les débuts et les fins de traitements transactionnels au niveau d'un composant. Ces méthodes servent à établir une correspondance entre les exécutions locales et les méthodes transactionnelles ce qui est indispensable pour la propagation des contextes transactionnels. En effet, lors d'un appel depuis une méthode, il faut être capable de savoir si l'exécution locale fait partie d'un traitement transactionnel afin de transmettre cette information au composant appelé.

(iii)Contrôle des transactions applicatives

Les méthodes de `ManagerComponent` qui sont utilisée pour contrôler les traitements applicatifs et pour assurer l'exclusion mutuelle avec les traitements de reconfiguration sont :

- `void beginManagerMethod(Enumeration components)` : marque le début d'une méthode de reconfiguration portant sur l'ensemble de composants `components`. Cette méthode assure le mode exclusif pour la reconfiguration en attendant la terminaison des exécutions en cours des composants et en bloquant les nouvelles exécutions. L'attente des terminaisons et le blocage des exécutions repose sur les mêmes mécanismes que dans le cas de la cohérence locale.
- `void endManagerMethod(Enumeration components)` : marque la fin d'une méthode de reconfiguration et débloque les exécutions en attente sur les composants.

Les méthodes globales de début et de fin de traitement de reconfiguration reposent entièrement sur les méthodes qui assurent la synchronisation au niveau composant. En effet, `beginManagerMethod` utilise `beginCaptureRestore` pour signaler aux composants à reconfigurer son intention à commencer une reconfiguration. La méthode `beginCaptureRestore` se charge à bloquer les appels des méthodes qui sont faits en dehors des contextes des méthodes transactionnelles dont le gestionnaire attend la terminaison. De même, `endManagerMethod` utilise `endCaptureRestore` pour débloquer les appels précédemment bloqués.

L'algorithme qui assure l'exclusion mutuelle entre les traitements métier et les traitements de reconfiguration est le suivant :

- *Lancement de méthode transactionnelle*. Quand un composant lance une méthode transactionnelle, son conteneur contacte le gestionnaire, obtient une identification pour la méthode et enregistre l'exécution en cours.
- *Propagation de méthode transactionnelle*. Quand la méthode transactionnelle appelle

des méthodes d'autres composants, le conteneur intercepte ces appels et les transforme pour rajouter comme paramètre l'identification de la méthode transactionnelle. La transformation de l'appel est effectuée par des talons spécifiques, introduits pour la gestion de la duplication et de la cohérence et discutés dans la suite.

- *Réception d'appel de méthode transactionnelle*. La réception des invocations augmentées par des contextes transactionnels est traitée par des squelettes spécifiques qui, comme les talons mentionnés juste avant, sont introduits pour la gestion de la duplication et de la cohérence. Ces squelettes enregistrent l'information d'une exécution transactionnelle en cours, enregistrent le composant appelé auprès du gestionnaire pour pouvoir lui annoncer la fin du traitement et passent la main aux squelettes applicatifs. Si l'appel local appelle d'autres méthodes sur d'autres composants, la procédure est répétée et le contexte d'exécution globale est propagé.
- *Lancement d'une méthode de reconfiguration*. Quand le gestionnaire lance une méthode de reconfiguration, il teste pour tous les composants impliqués dans ce traitement s'ils ne sont pas en cours de modification. Si ce n'est pas le cas, il bloque les demandes de modification issues de méthodes locales ou de propagations de méthodes transactionnelles et attend la terminaison des traitements en cours.
- *Attente de terminaison des traitements en cours.* La terminaison des traitements locaux à un composant est détectée comme nous l'avons décrit dans la Section 6.1. La détection de la terminaison des traitements répartis *synchrones* est faite au niveau des méthodes transactionnelles. La terminaison des traitements incluant des invocations *asynchrones* est faite au niveau du gestionnaire qui est mis au courant de chaque terminaison de traitement local correspondant à une propagation asynchrone.
- *Terminaison de méthode de reconfiguration.* À la fin d'un traitement de configuration, le gestionnaire débloque les invocations en attente au niveau des composants configurés.

(iv) Exemple

Considérons l'exemple de l'application "Hello World", dans le cas où le serveur S émet l'événement `tooManyHellos`. Le traitement de cet événement est assuré par le gestionnaire de configuration de la manière suivante :

- *Lancement d'une méthode de reconfiguration*. Le gestionnaire commence l'exécution de la méthode `tooManyHellos()` qui est la réaction pour l'événement émis par le serveur. Étant donné que c'est une méthode de reconfiguration, c'est également une méthode transactionnelle. Par conséquent, au début de cette méthode est appelée la méthode `beginManagerMethod` avec en paramètre la liste des composants sur laquelle cette méthode opère, dans ce cas l'instance du serveur.
- *Blocage des méthodes applicatives modifiant l'état*. La seule méthode qui modifie l'état du serveur qui est le seul composant concerné par la reconfiguration est la méthode `print`. Par conséquent, pour permettre l'exécution du traitement de reconfiguration et éviter une attente infinie, la méthode `beginManagerMethod` bloque les appels de `print` sur le serveur. Ceci est fait en utilisant la méthode `beginCaptureRestore` définie au niveau du conteneur.
- *Attente des méthodes applicatives en cours d'exécution*. Avant de lancer la suite de la méthode `tooManyHellos()`, le gestionnaire attend la fin des exécutions de `print` en cours.
- *Reconfiguration*. Le gestionnaire reconfigure l'application : il crée un nouveau serveur,

il duplique la partie de la table de hachage qui concerne les chaînes commençant par “Hello” et connecte le serveur S à ce serveur de “décharge” (Figure 6-10.).

- *Retour à un fonctionnement normal*. La fin de la reconfiguration est marquée par l’appel de `endManagerMethod` qui débloque les appels de `print` sur le serveur.

(v) Remarques

Nous proposons une gestion des reconfigurations qui est analogue à la synchronisation proposée au niveau composant. D’ailleurs, elle repose entièrement sur les primitives définies au niveau des conteneurs des composants. Les méthodes `beginGlobalMethod` et `endGlobalMethod`, définies au niveau global, utilisent respectivement les méthodes `beginMethod` et `endMethod`, définies au niveau local. De même, les méthodes `beginManagerMethod` et `endManagerMethod` correspondent et utilisent les méthodes `beginCaptureRestore` et `endCaptureRestore`.

Les mécanismes proposés permettent de spécifier des méthodes qui définissent des traitements applicatifs qui ne doivent pas être perturbés par les traitements de reconfiguration, qui intercepte les débuts et les fins d’exécution de ces méthodes et qui met en place un algorithme d’exclusion mutuelle. Toutefois, comme il été déjà dit précédemment, ces mécanismes ne sont pas imposés aux protocoles qui peuvent choisir de mettre en place leur propre gestion. Par exemple, nous présentons dans le Chapitre 9, un protocole de gestion de cache qui ne fait pas usage des traitements proposés. Il implémente sa propre gestion de verrous et ne fait appel ni aux primitives de synchronisation au niveau composant, ni aux primitives de reconfiguration au niveau de l’application.

6.2.3 Composants copies et clients

Les composants copies et les composants clients sont les composants qui implémentent la logique de fonctionnement d’un protocole de duplication et de cohérence. Les composants copies représentent les composants métier dupliqués, ainsi que leurs copies, alors que les composants clients modélisent les composants qui interagissent avec ces copies. Leurs types sont des sous-classes de la classe `ProtocolComponent` spécialisés afin d’implémenter des interfaces spécifiques aux protocoles. Dans la suite, nous présentons la classe `ProtocolComponent` et détaillons les règles de programmation de ces composants de protocole.

La classe `ProtocolComponent`

La classe `ProtocolComponent` dispose des attributs et des méthodes suivants :

- `ManagerComponent manager` : cet attribut référence le composant gestionnaire. La connaissance du gestionnaire est nécessaire pour pouvoir lui transmettre les événements déclenchant des traitements de reconfiguration. Cet attribut est positionné lors du déploiement d’une application métier et de l’intégration du protocole de duplication et de cohérence.
- `Component component` : cet attribut référence le composant métier qui est géré par ce composant de protocole.
- `ContainerRef getService(String name)` : cette méthode permet de récupérer la référence d’un composant service auprès de l’environnement local d’exécution.
- `void deploy(String[] args)` : cette méthode n’est pas implémentée de manière générique mais existe dans chaque composant de protocole. Elle définit une partie du

programme de déploiement du protocole de duplication et de cohérence. Étant donné que certaines des connexions entre composants de protocoles sont établies de manière obligatoire, cette méthode factorise les traitements concernés et évite aux intégrateurs de protocoles de les réécrire. Par exemple, dans le protocole de tolérance aux pannes, si le composant de type `Client` émet les événements `tooLong` en utilisant des délais mesurés à l'aide d'un composant service `Timer`, sa méthode `deploy()` va récupérer la référence de ce `Timer` en utilisant `getService` et établira une connexion entre les deux composants.

Programmation des composants clients et copies

Les méthodes des composants clients et des composants copies peuvent être divisées en deux groupes. Les méthodes du premier groupe concernent les traitements liés uniquement à la gestion de la duplication et de la cohérence et sont programmées sans suivre de règles particulières. Les méthodes du deuxième groupe sont responsables de rajouter des traitements de duplication et de cohérence pour des interactions métier et sont composées de trois éléments : un prétraitement, un appel du traitement métier et un post-traitement.

Pour illustrer ces points, prenons encore l'exemple du protocole de tolérance aux pannes (Figure 6-8.). Dans ce protocole, les traitements liés uniquement à la gestion de la duplication et de la cohérence sont les traitements de mise à jour des deux copies secondaires à partir de la copie primaire. Les traitements liés aux interactions métier sont ceux qui spécifient que tout appel au serveur de la part du client doit être entouré par des appels au composant de service `Timer`. L'appel au serveur doit être précédé par un traitement du `Timer` qui déclenche la mesure du délai d'attente et qui définit donc un prétraitement pour cet appel. L'appel doit être suivi par un traitement qui annule l'alerte de délai dépassé qui est donc un post-traitement de l'appel. Ces traitements font le lien entre les opérations métier, représentées par la notion abstraite d'"operation d'appel du serveur" et les traitements de tolérance aux pannes.

```
class Client extends ProtocolComponent {
    Timer_itf timer;                        //référence du composant service Timer

... CallServer(...) {                       //opération métier abstraite
      pre_CallServer  (...);                //pré-traitement
      call_CallServer (...);                //traitement métier effectif
      post_CallServer (...);                //post-traitement
}
... pre_CallServer(...) {                   //implémentation du pré-traitement
      timer.startDelay(...);                //début de mesure du délai d'attente
    }
... post_CallServer(...) {...               //implémentation du post-traitement
      timer.stopAlert(...);                 //prévenir l'alerte en cas de délai dépassé
    }
... abstract call_CallServer (...) {}       //traitement métier non implémenté
}
```

Figure 6-13. Implémentation de méthodes dans un composant de protocole

La structure des méthodes de protocole, liées aux traitements métier, est donnée à la Figure 6-13. Pour les opérations métier, considérées du point de vue du protocole comme étant du type `CallServer`, la méthode de protocole contient un prétraitement (`pre_CallServer`), l'appel effectif aux opérations métier (`call_CallServer`) et un post-traitement (`post_CallServer`). Les pré et post traitements sont implémentés par le composant de protocole, alors que le traitement métier n'est spécifié que lors de l'intégration du protocole au sein d'une application métier.

6.2.4 Composants de protocole : synthèse.

Les composants de protocole suivent toutes les règles de description et de programmation des composants métier. Leurs particularités relatives à la gestion de la duplication et de la cohérence sont définies dans des classes spécifiques aux quatre types de composants.

- *Composants services*. Les composants services sont des composants qui encapsulent des services système et sont utilisés sans modification dans les protocoles de gestion de duplication et de cohérence. Ils sont implémentés en tant que sous-classes de la classe `ServiceComponent` qui définit un traitement générique d'enregistrement des composants services auprès de l'environnement d'exécution.
- *Composants gestionnaires de configuration*. Les composants gestionnaires sont responsables des traitements de reconfiguration liés à la gestion de copies. Leurs traitements de reconfiguration sont définis de manière générique, en termes de modèles de protocoles, ou de manière spécifique relative à une instanciation de protocole. Les gestionnaires sont instances de la classe `ManagerComponent` qui fournit des primitives de gestion de la cohérence globale d'une application. Ces primitives assurent une exclusion mutuelle entre les traitements globaux de modification de l'état de l'application et les traitements de reconfiguration définis dans les méthodes du gestionnaire.
- *Composants clients et copies*. Les composants clients identifient les composants interagissent avec des copies, elles-mêmes identifiées par les composants copies. Ils sont définis en tant que sous-classes de la classe `ProtocolComponent` qui distingue deux types de traitements : les traitements liés uniquement à la gestion de la duplication et de la cohérence et les traitements reliant les calculs applicatifs à cette gestion. Les traitements du premier type sont programmés de manière standard, alors que les traitements du deuxième type sont structurés en termes de pré et de post traitements.

6.3 Intégration dans les applications métier

L'intégration des protocoles de duplication et de cohérence au sein des applications métier est faite en trois étapes :

- *Établissement des relations entre interfaces de protocole et interfaces métier* : cette étape définit les interactions métier qui ont une incidence sur la gestion du protocole.
- *Établissement des relations entre composants de protocole et composants métier* : cette étape définit les rôles des composants métier du point de vue du protocole.
- *Intégration de protocole* : cette étape génère et intègre les structures de protocole spécifiques à l'application métier.

Ces trois étapes sont détaillées dans ce qui suit.

6.3.1 Relations entre interfaces

Lors de l'intégration d'un protocole au sein d'une application métier, les opérations applicatives sont caractérisées vis à vis de leur rôle dans la gestion de la duplication et de la cohérence. Au niveau d'un protocole, une partie seulement des opérations métier apparaissent en tant qu'événements significatifs qui déclenchent des synchronisations, des créations ou des

suppressions de copies. Les autres opérations métier n'ont pas d'incidence sur la gestion du protocole et sont ignorées. Dans la terminologie des systèmes réflexifs (cf. Chapitre 1, Section 1.2.3, p. 34), ces deux groupes d'opérations correspondent respectivement aux opérations réifiées et non réifiées au niveau méta du protocole.

Dans FAR, les opérations à réifier sont définies en spécifiant une correspondance entre les opérations métier et les opérations de protocole. Cette correspondance est donnée dans des descripteurs XML et utilise les définitions des interfaces des opérations.

Les descripteurs XML sont fournis au niveau des composants métier (Figure 6-14.) et font correspondre une opération métier à une seule opération de protocole[1]. L'opération de protocole est identifiée par son nom (ligne 7) et par le nom de l'interface qui la spécifie (ligne 6).

```
1    <component_mapping> <component_type> ... </component_type >
2       <provided> <interface>
3          <interface_type>...</interface_type>
4             <operation_mapping>
5                <operation>...</operation>
6                <protocol_interface>...</protocol_interface>
7                <protocol_operation>...</protocol_operation>
8             </operation_mapping>
9            ...
10      </provided>
11      <required>
12          ...                //même structure que pour provided
13      </required>
14   </component_mapping>
```

Figure 6-14. Descripteur de correspondance entre interfaces métier et interfaces de protocole

Si l'on prend l'exemple de l'application "Hello World" et du protocole de tolérance aux pannes, la correspondance à établir pour le client sera entre la méthode `print` et la méthode `CallServer` (Figure 6-15.).

```
1    <component_mapping> <component_type> Client_Comp </component_type >
2       <required><interface>
3          <interface_type> Hello_itf /interface_type>
4             <operation_mapping>
5                <operation> print </operation>
6                <protocol_interface> Service_itf </protocol_interface>
7                <protocol_operation> CallServer </protocol_operation>
8             </operation_mapping>
9       </interface></required>
10   </component_mapping>
```

Figure 6-15. Correspondance d'interfaces pour "Hello World"

6.3.2 Relations entre composants

Les relations entre composants sont établies afin de définir les rôles des composants métier du point de vue de la duplication et de la cohérence. Étant donné qu'ils peuvent être des compo-

1. Si une opération métier doit susciter plusieurs actions au niveau protocole, ces actions sont supposées séquencées et définies dans une seule méthode de protocole. C'est cette méthode qui réifie l'opération métier.

sants dupliqués ou des composants faisant appel à des composants dupliqués, les composants métier sont perçus au niveau du protocole en tant que des composants copies et des composants clients (cf. Section 6.2.3).

L'établissement de la correspondance entre un composant de protocole et un composant métier est faite en utilisant une méthode particulière, intégrée au niveau du conteneur de composant. Cette méthode fait partie de la classe `Container` (cf. Chapitre 5, Section 5.4.2, p. 104) et a les deux signatures suivantes :

```
- void attach(String protocolComp,
              Enumeration provided,
              Enumeration required)
  throw AlreadyConfiguredException;
- void attach(String protocolComp)
  throw AlreadyConfiguredException;
```

Appelée sur le conteneur d'un composant métier, cette méthode spécifie que le comportement de ce dernier sera géré par un composant de protocole de type `protocolComp`[1]. La première forme d'`attach` spécifie que le contrôle de `protocolComp` couvre uniquement les interactions métier issues des interfaces données en paramètre (`provided`, `required`). Dans sa deuxième forme, la méthode spécifie que la totalité des interactions seront contrôlées par `protocolComp`. Si un composant est déjà configuré comme étant géré par un composant de protocole et que la configuration demandée est conflictuelle[2], la méthode génère une exception.

Si l'on considère toujours l'exemple de “Hello World” et du protocole de tolérance aux pannes, pour dire que le serveur doit être dupliqué et rendu tolérant aux fautes, il faut appeler, sur le conteneur du serveur, la méthode `attach` avec comme paramètre le type `Primaire`.

Le schéma général du contrôle qu'un composant de protocole exerce sur un composant métier est le suivant (Figure 6-16.). En utilisant les talons et les squelettes métier du composant, le composant de protocole intercepte les appels issus des interfaces dont il est responsable. En se basant sur le descripteur XML, il interprète les différentes opérations métier interceptées et exécute les méthodes de protocole correspondantes. Ces dernières s'occupent de tous les traitements nécessaires pour le contrôle de la duplication et de la cohérence. Ce sont des méthodes qui définissent des pré et des post traitements, d'après la logique introduite précédemment (Section 6.2.3).

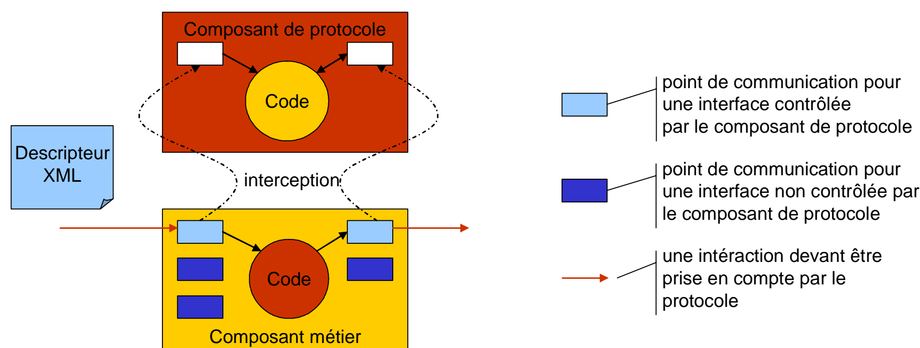


Figure 6-16. Contrôle d'un composant métier par un composant de protocole

1. Dans la suite nous dirons que le composant de protocole est *attaché* au composant métier.
2. C'est à dire porte sur des interfaces déjà contrôlées par un composant de protocole.

6.3.3 Génération des structures de protocole

Les composants de protocoles qui définissent uniquement des traitements de gestion de la duplication et de la cohérence, notamment les composants services et les gestionnaires, n'ont pas besoin de classes spécifiques d'implémentation. Ils peuvent directement être instanciés à partir des leurs classes d'implémentation.

Les composants clients et copies ne peuvent pas être directement instanciés puisqu'ils sont définis en tant que classes génériques. Ils spécifient des pré et des post traitements pour des traitements métier qui ne sont pas implémentés par les protocoles (cf. Section 6.2.3). Pour être instanciés, ils ont besoin de classes spécifiques d'implémentation où les appels abstraits des traitements métier sont remplacés par des appels concrets vers des composants métier.

Pour illustrer ce point, considérons le type `Client` du protocole de tolérance aux pannes. Ce type ne peut être directement instancié puisque la méthode `call_CallServer` n'est pas implémentée (cf. Figure 6-13., p. 128). Il pourra être instancié après l'intégration du protocole dans l'application "Hello World" où `call_CallServer` sera replacé par l'appel à `print`.

Dans la suite, nous présentons la procédure de génération automatique des classes spécifiques d'implémentation des composants clients et copies. Nous discutons également la génération des talons et des squelettes correspondants.

Classes spécifiques d'implémentation des composants copies ou clients

Pour considérer en détail la procédure de génération des classes d'implémentations spécifiques, prenons une opération de protocole `Op`, définie dans un composant de protocole `A_ProtocolComp` (Figure 6-17.a). Comme indiqué dans le descripteur XML d'un composant applicatif `A_AppliComp`, l'opération `Op` réifie plusieurs opérations métier `m1`, `m2`,..., `mN` (Figure 6-17.b).

(a)
```
class A_ProtocolComp
extendsProtocolComponent{

... Op(...) {          //opération réifiant des opérations métier
        pre_Op(...);
        call_Op(...);
        post_Op(...);
}
... pre_Op(...) {...}  //pré-traitement de Op
... post_Op(...) {...} //post-traitement de Op
... abstract call_Op(...) {}  //à remplacer par appel métier
}
```

(b)
```
<component_mapping>
<component_type> A_AppliComp </component_type>
<provided>
   ...
   <operation> m1 </operation>
   <protocol_operation> Op </protocol_operation>
   ...
   <operation> m2 </operation>
   <protocol_operation> Op </protocol_operation>
   ...
<required>
   <operation> mN </operation>
   <protocol_operation> Op </protocol_operation>
</component_mapping>
```

(c)
```
class A_ProtocolComp_Gen
extends ProtocolComponent {

... Op_m1(...) {
       pre_Op(...);
       component.Op_m1(...);
       post_Op(...);
}

... Op_m2(...) {
       pre_Op(...);
       component.Op_m2(...);
       post_Op(...);
}

...
... Op_mN(...) {
       pre_Op(...);
       appel de Op_mN(...)
       à distance;
       post_Op(...);
}

... pre_Op(...) {...}
... post_Op(...) {...}
}
```

Figure 6-17. Génération d'une classe de protocole spécifique

La classe spécifique d'implémentation de A_ProtocolComp pour l'intégration dans un composant métier de type A_AppliComp est nommée A_ProtocolComp_Gen (Figure 6-17.c). Dans cette classe, la méthode Op est remplacée par un ensemble de méthodes Op_m1, Op_m2,..., Op_mN qui ont les caractéristiques suivantes :

- *Réification*. Les méthodes Op_m1, Op_m2,..., Op_mN réifient respectivement les méthodes métier m1, m2,..., mN.
- *Signatures de méthodes*. Les méthodes Op_m1, Op_m2,..., Op_mN recopient la signature d'Op en ce qui concerne les types des paramètres mais rajoutent respectivement la liste des paramètres de m1, m2,..., mN. Ces derniers sont utilisés pour les appels des méthodes métier.
- *Remplacement des appels de la méthode abstraite*. Les méthodes Op_m1, Op_m2, ..., Op_mN remplacent l'appel de la méthode abstraite call_Op par les appels respectifs vers les méthodes métier m1, m2, ..., mN. Elles gardent les appels des pré et post traitements pre_Op et post_Op.
- *Traitement des appels métier*. Dans le cas d'appels métier rentrants, les appels sont délégués au composant métier auquel le composant de protocole est attaché. Ce composant métier est accessible via l'attribut component de la classe ProtocolComponent. Dans le cas d'appels sortants, ils sont effectués en utilisant la référence du composant distant, connue au niveau du conteneur du composant métier.

Classes spécifiques de talons et de squelettes des composants copies ou clients

Le changement des méthodes dans l'implémentation d'un composant de protocole a une répercussion sur la génération des talons et des squelettes pour ce composant. En effet, ces talons et squelettes ne peuvent plus implémenter des interfaces génériques contenant des opérations comme Op mais doivent prendre en compte les signatures des nouvelles méthodes Op_m1, Op_m2, ..., Op_mN.

Pour prendre en compte les signatures spécifiques des méthodes d'un composant de protocole, nous générons des classes spécifiques de talons et de squelettes pour chaque correspondance entre interface métier et interfaces de protocole. En effet, quand un composant métier est attaché à un composant de protocole, nous créons un talon ou squelette de protocole pour chaque talon ou squelette métier qui correspond à une interface métier réifiée. Les talons et les squelettes de protocole implémentent les interfaces de protocole qui représentent ces interfaces réifiées. Il s'agit des interfaces spécifiques obtenues à base des correspondances entre interfaces de protocole et interface métier.

Pour illustrer ce point, considérons encore l'exemple de la Figure 6-17. Supposons que le composant métier A_AppliComp, dont les correspondances d'interfaces sont données par la Figure 6-17.b, dispose d'une interface métier fournie MFournie_itf et d'une interface requise MRequise_itf (Figure 6-18.a). MFournie_itf spécifie trois méthodes m1, m2 et m3 qui prennent en paramètre params_m1, params_m2 et params_m3 et qui rendent result_m1, result_m2 et result_m3. MRequise_itf spécifie deux méthodes mN et mM dont les paramètres et les résultats sont respectivement params_mN, params_mM, result_mN et result_mM. D'après le descripteur XML (Figure 6-17.b), les opérations m1 et m2 de MFournie_itf, ainsi que mN de MRequise_itf sont réifiées par l'opération de protocole Op.

(a)	`interface MFournie_itf {`
	`   result_m1 m1(params_m1);`
	`   result_m2 m2(params_m2);`
	`   result_m3 m3(params_m3);}`
	`}`
		`interface MRequise_itf {`
		`   result_mN mN(params_mN);`
		`   result_mM mM(params_mM);`
		`}`

(b)	`interface Protocol_MFournie_itf {`
	`   result_Op_m1 Op_m1(params_protocole,`
	`                     params_m1);`
	`   result_Op_m2 Op_m2(params_protocole,`
	`                     params_m2);`
	`}`
		`interface Protocol_MRequise_itf {`
		`   result_Op_mN Op_mN(params_protocole,`
		`                     params_mN);`
		`}`

Figure 6-18. Interfaces métier et les interface de protocole correspondant

Comme nous l'avons vu précédemment, l'implémentation spécifique de protocole transforme `Op` en `Op_m1`, `Op_m2` et `Op_mN` avec des signatures augmentées. La vision de protocole de l'interface fournie `MFournie_itf` est donc une interface `Protocol_MFournie_itf` qui définit `Op_m1` et `Op_m2`. La vision de protocole de l'interface requise `MRequise_itf` est une interface `Protocol_MRequise_itf` qui définit `Op_mN` (Figure 6-18.b). Le talon métier qui représente `MFournie_itf` sera accompagné par un talon de protocole qui représente `Protocol_MFournie_itf`, alors que le squelette qui représente `MRequise_itf` sera accompagné d'un squelette représentant `Protocol_MRequise_itf`.

Les changements de signatures lors du passage de `m1`, `m2`.. `mN` à `Op_m1`, `Op_m2`, ..., `Op_mN` sont faits afin d'intégrer le contexte d'exécution relatif à la gestion de la duplication et de la cohérence. Ce contexte est présent au niveau des paramètres des méthodes, ainsi qu'au niveau des résultats. En effet, il permet aux composants de protocole d'échanger des informations de duplication et de cohérence en relation avec une méthode métier traitée.

Les classes pour les talons et les squelettes de protocole sont des sous-classes de la classe prédéfinie `RC_CommPoint_Impl`. La fonction de cette classe est identique à celle de `CommPoint_Impl` utilisée pour les points de communication métier (cf. Chapitre 5, Section 5.4.1, p. 99). La seule différence est l'acheminement des appels vers les implémentations de composants de protocole et non vers des implémentations de composants métier.

6.3.4 Intégration des composants de protocole

Comme nous l'avons spécifié dans notre proposition, les composants services et les composants gestionnaires sont intégrés au niveau de l'architecture de l'application métier (cf. Chapitre 2, Section 4.5.2, p. 87). Les composants copies et les composants clients ne sont pas instanciés en tant que composants indépendants mais sont intégrés dans la structure des composants métier auxquels ils sont attachés.

Intégration des composants clients et copies

L'intégration des composants clients et copies consiste en l'intégration de leurs implémentations, de leurs talons et de leurs squelettes dans les conteneurs des composants métier. Elle est faite en appelant l'opération `attach` pendant le déploiement et utilise les classes spécifiques d'implémentation des composants de protocole. Ces dernières peuvent être générées avant le déploiement ou être générées à la volée.

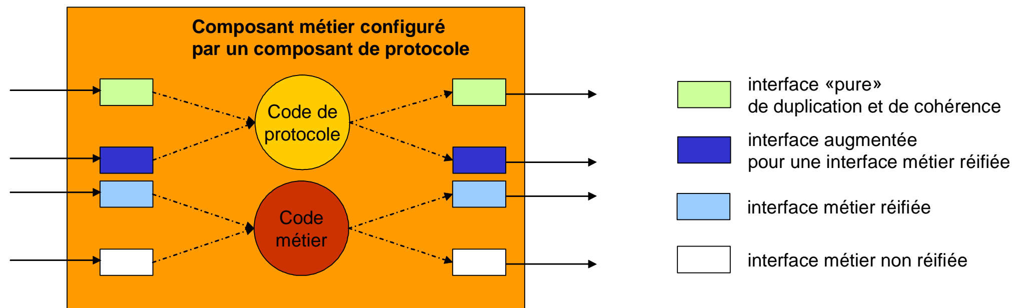


Figure 6-19. Intégration de composant de protocole au niveau du conteneur métier

- *Intégration des implémentations de protocole*. L'intégration de l'implémentation des composants de protocole est faite en rajoutant, au niveau du conteneur métier, un attribut complémentaire `ProtocolComponent pcomp`. Ainsi, le conteneur métier contient deux objets Java : l'un correspondant à l'implémentation métier du composant et l'autre correspondant à l'implémentation de composant de protocole (Figure 6-19.).
- *Intégration des talons et des squelettes de protocole*. L'intégration des talons et des squelettes de protocole se fait en utilisant les deux méthodes de la classe `Container` `createStub(String className, Object args[])` et `createSkel(String className, Object args[])`. Ces méthodes instancient des sous-classes de `RC_CommPoint_Impl` ayant pour nom `className`, initialisent avec `args` les talons ou les squelettes qui en résultent et les enregistrent au niveau du conteneur.

La structure de composant métier qui résulte de cette intégration de composant de protocole est montrée de manière schématique à la Figure 6-19.

Dans le cas de la configuration du serveur de "Hello World" pour les besoins de tolérance aux pannes, l'instanciation de cette structure est la suivante (Figure 6-20.). Le conteneur contient l'implémentation du serveur (une instance de `Server_Comp)` et l'objet d'implémentation du composant du protocole (instance de sous-classe de `Primaire`). Il dispose du squelette pour `Hallo_itf` qui est accompagné par un squelette généré pour le composant de protocole. Ce squelette implémente l'interface `Protocol_Helo_itf` qui reflète la réification de `Hello_itf` par `Service_itf` du type `Primaire`. Le conteneur dispose également d'un talon de protocole pour les communications avec les deux copies secondaires.

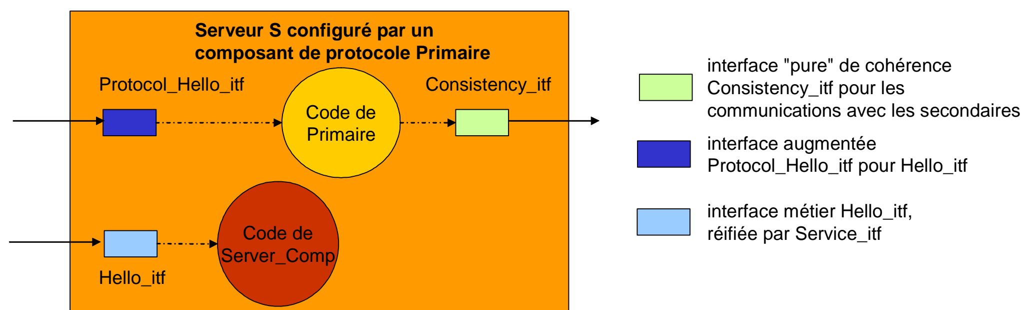
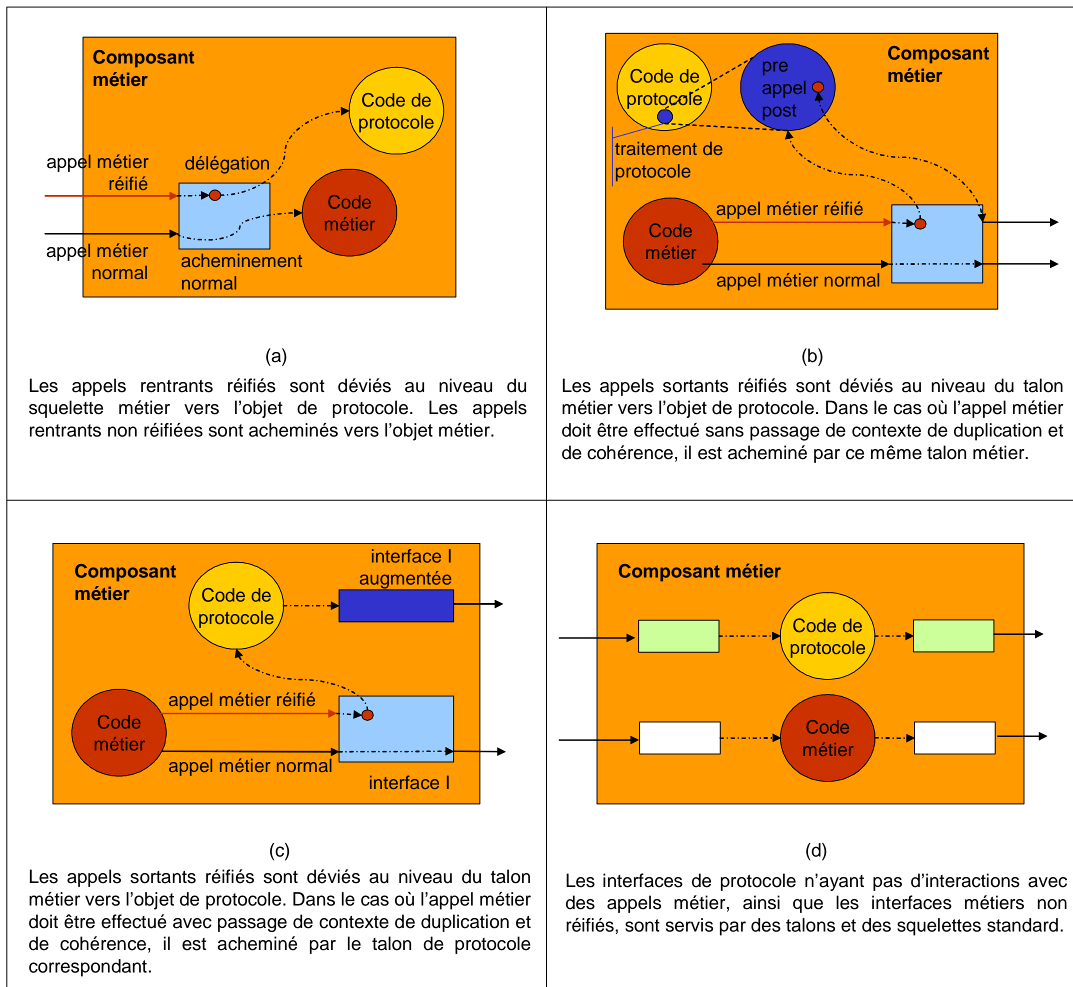


Figure 6-20. Intégration de composant de protocole au niveau du serveur de "Hello World"

Coordination entre interaction métier et interactions de protocole

Pour inclure les traitements de protocole avant et après les traitements métier réifiés, il est nécessaire d'intercepter ces appels métier et de les déléguer au niveau du protocole. Or, les talons et les squelettes métier permettent uniquement l'interception de ces appels mais n'incluent pas de "crochets" pour leur déviation.

Pour intégrer la déviation d'appels au niveau des talons et des squelettes, nous générons des classes de talons et de squelettes modifiées. Dans ces classes, l'appel métier est remplacé par un appel de délégation vers le traitement correspondant de protocole.



(a)
Les appels rentrants réifiés sont déviés au niveau du squelette métier vers l'objet de protocole. Les appels rentrants non réifiées sont acheminés vers l'objet métier.

(b)
Les appels sortants réifiés sont déviés au niveau du talon métier vers l'objet de protocole. Dans le cas où l'appel métier doit être effectué sans passage de contexte de duplication et de cohérence, il est acheminé par ce même talon métier.

(c)
Les appels sortants réifiés sont déviés au niveau du talon métier vers l'objet de protocole. Dans le cas où l'appel métier doit être effectué avec passage de contexte de duplication et de cohérence, il est acheminé par le talon de protocole correspondant.

(d)
Les interfaces de protocole n'ayant pas d'interactions avec des appels métier, ainsi que les interfaces métiers non réifiés, sont servis par des talons et des squelettes standard.

Figure 6-21. Acheminement des interactions métier et de protocole

Dans le cas des squelettes (Figure 6-21.a), l'appel de délégation est directement dirigé vers l'objet d'implémentation de protocole, accessible à l'aide du conteneur. Ce traitement exécute les pré et post traitements nécessaires tout en se chargeant de l'appel sur le composant métier.

Dans le cas des talons, le traitement de protocole concernant un appel métier peut se dérouler de deux manières différentes. La première possibilité concerne les cas où les pré et les post traitements du protocole n'influencent pas le déroulement de l'appel métier (Figure 6-21.b). Dans ce cas, ce dernier est entouré des pré et post traitements de protocole mais est acheminé de manière standard par le talon métier. La deuxième possibilité concerne les cas où il est nécessaire de transmettre un contexte de duplication et de cohérence lors du traitement de l'appel

métier (Figure 6-21.c). Dans ce cas, l'appel métier est encapsulé dans une interaction plus complexe, acheminée par les talons de protocole qui disposent des méthodes à signatures augmentées.

Les interactions métier qui ne sont pas significatives de point de vue de la duplication et de la cohérence sont gérées de manière standard par des talons et des squelettes métier non modifiées (Figure 6-21.d). C'est le cas également des interactions de protocole qui ne sont pas liées aux interactions métier.

Chapitre 7

Mise en œuvre dans CCM

Afin de valider l'applicabilité des principes de FAR, nous les avons confrontés à l'environnement standard CCM [102]. Dans cette section, nous présentons brièvement les principes et le contexte de notre mise en œuvre de la gestion de la duplication et de la cohérence dans CCM. Nous approfondissons la description fournie dans l'état de l'art (cf. Chapitre 1, Section 1.2.2, p. 31) et discutons de la gestion des applications métier et des protocoles de duplication et de cohérence. Nous comparons FAR et CCM, expliquons la mise en place des différents mécanismes de gestion et résumons les principales difficultés de cette réalisation.

7.1 Applications CCM sans duplication et cohérence

La spécification CCM ne traite pas les aspects de duplication et de cohérence. Les applications CCM standard sont, par conséquent, des applications métier sans duplication ni cohérence. CCM définit ces applications en termes de leur modèle de composants, de leurs règles de programmation, ainsi que de leur environnement d'exécution.

Dans la suite, nous présentons une analyse comparative des modèles de CCM et de FAR. Nous considérons le modèle des composants CCM (Section 7.1.1), leur programmation (Section 7.1.2), ainsi que leur structure à base de conteneurs (Section 7.1.3). Nous terminons par leur environnement d'exécution et la logique de déploiement (Sections 7.1.4).

7.1.1 Modèle de composants

Le modèle de composants de CCM est très proche du modèle de FAR (cf. Chapitre 5, Section 5.3, p. 95). En effet, FAR a été défini après étude de CCM et EJB et beaucoup de ces caractéristiques ont été inspirées des principes de ces deux environnements.

Comme dans FAR, CCM définit les types de composants en termes de leurs interfaces fournies et requises (*portes d'entrée et de sortie*). Les interfaces des composants ont un type, peuvent être utilisées pour des communications synchrones ou asynchrones et sont caractérisées par un nombre de connexions autorisées.

Comme dans FAR, la connexion entre deux composants CCM est une opération explicite qui établit une correspondance entre une porte d'entrée et une porte de sortie compatibles. La connexion de portes est une étape obligatoire sans laquelle la communication entre composants n'est pas possible. Elle est basée sur des interfaces prédéfinies d'introspection et de connexion qui utilisent des références de composant et des noms symboliques des portes de composant.

Nous pouvons conclure que le modèle de composants de CCM fournit les caractéristiques nécessaires à la gestion de la duplication et de la cohérence. En définissant les composants en termes de leur types, des services qu'ils fournissent et de leurs dépendances d'autres composants, il fournit un vue architecturale des composants. Il permet la manipulation de cette vue architecturale par des primitives d'introspection et de contrôle explicite de connexions.

7.1.2 Programmation de composants

Comme dans FAR, la programmation de composants CCM passe par une étape déclarative et suit des règles de programmation. CCM définit les types de composants en utilisant le langage de déclaration d'interfaces (IDL) de CORBA [101], génère des squelettes d'implémentations partielles et représente les interfaces requises et fournies de manière explicite dans ces squelettes.

Les différences entre CCM et FAR résident dans le processus d'implémentation des composants et sont les suivantes :

- *Description déclarative d'implémentations de composants*. CCM introduit une étape supplémentaire dans le processus d'implémentation des composants qui est la description déclarative des la structuration des composants en termes de traitements fonctionnels et non fonctionnels (cf. Chapitre 1, Section 1.1.1, p. 23). Faite dans un langage nommé CIDL[1], pour un type de composant, cette description spécifie la classe d'implémentation métier d'un composant, l'entité en charge de la persistance (composant ou conteneur) et donne une description de l'état persistant du composant[2].
- *Implémentations gérant des propriétés non fonctionnelles*. Dans FAR, les composants sont programmés sans d'autres préoccupations que les fonctionnalités métier. Dans CCM, les composants peuvent choisir de gérer eux-mêmes leur persistance et leurs transactions et par conséquent peuvent avoir des implémentations bien complexes. Ceci introduit le problème des interférences entre la gestion non fonctionnelle qui fait partie du code des composants métier et la gestion de la duplication et de la cohérence. Ce problème est discuté dans la suite.
- *Définition d'état persistant*. Comme il a été déjà mentionné, les composants persistants peuvent utiliser leur description CIDL afin de spécifier leur état persistant. Cette caractéristique est intéressante lorsque l'on considère l'instrumentation des applications métier pour la gestion de la duplication et de la cohérence (cf. Chapitre 6, Section 6.1, p. 111). La description de l'état persistant est un pas en avant par rapport aux composants FAR qui ne donnent, a priori, aucun accès à leur état.
- *Langage de programmation*. CCM est défini comme une couche composant au-dessus de l'environnement CORBA, connu pour ses objectifs d'interopérabilité entre systèmes. Les composants CCM peuvent, par conséquent, être implémentés dans tous les langages supportés par CORBA.

La comparaison montre que les principes de programmation de composants FAR et de composants CCM sont très proches mais que CCM introduit une complexité additionnelle due à la gestion des transactions, de la persistance et de la sécurité. Cette complexité ne peut que se répercuter sur la gestion de la duplication et de la cohérence.

1. Rappelons que CIDL est une abréviation de *Component Implementation Definition Language*.
2. La gestion de la sécurité et des transactions est spécifiée lors du déploiement.

7.1.3 Structure de composant

La structure d'un composant CCM est très semblable à celle de FAR : elle est composée d'un conteneur qui, utilisant le principe d'interception, gère les propriétés non fonctionnelles du composant. Toutefois, elle diffère de la structure d'un composant de FAR sur les deux points suivants :

- *Conteneur CCM global.* Contrairement à FAR, où un conteneur est rattaché à une instance de composant, un conteneur CCM est responsable de toutes les instances d'un type de composant.
- *Objets d'interception non spécifiés*. Même si le principe de fonctionnement d'un conteneur CCM est basé sur l'interception, les objets qui implémentent effectivement cette interception ne sont pas spécifiés et sont laissés à l'implémentation. Ceci est différent de FAR où tout point de communication d'une instance est représenté par un objet distinct à l'exécution.

Les difficultés que ces différences posent pour la gestion de la duplication et de la cohérence dans CCM sont au nombre de deux. D'une part, nous ne disposons pas d'une structure de conteneur qui permet la configuration de la duplication et de la cohérence *par instance*. D'autre part, si nous définissons la gestion de la duplication et de la cohérence en termes des objets d'interception spécifiques à une implémentation de CCM, nous risquons de nous retrouver avec une mise en œuvre non applicable dans d'autres cas d'implémentations de CCM. La gestion serait dépendante de l'implémentation et non définie dans le cadre de la spécification.

Pour résoudre ces difficultés et retrouver la structure de conteneur définie dans FAR, nous proposons d'étendre les implémentations des composants CCM. En effet, nous intégrons ces implémentations dans des structures plus complexes qui correspondent aux structures de conteneur de FAR. Dans le cas d'un langage orienté objet[1], cela consiste à faire hériter les classes d'implémentation des composants de classes intermédiaires. Ces dernières définissent des objets explicites pour les portes des instances et fournissent ainsi les crochets pour la gestion de la duplication et de la cohérence. La structure qui en résulte est montrée sur la Figure 7-1.[2]

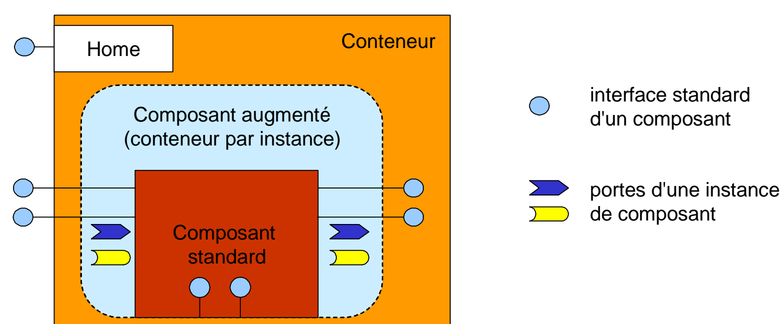


Figure 7-1. Structure augmenté d'un composant CCM

Pour arriver à la structure augmentée de composant, nous rajoutons une étape au processus de génération partielle des implémentations des composants. Nous reprenons la classe d'implé-

1. En principe, CCM peut être utilisé non seulement avec des langages orientés-objet mais également avec des langages procéduraux.
2. La structure standard de CCM est montrée dans l'état de l'art sur les composants, Chapitre 1, Section 1.2.2, p. 31

mentation d'un composant générée à partir de sa description CIDL et générons une sous-classe qui définit des attributs pour les portes du composant.

- *Attribut pour interface fournie*. Les attributs pour les interfaces fournies contiennent la référence de la porte d'entrée correspondant. Ces attributs changent de valeur selon que le composant est configuré avec un protocole de duplication et de cohérence ou non. Dans le cas sans duplication ni cohérence, ils référencent les objets qui implémentent les interfaces, alors que dans la cas avec duplication et cohérence, ils référencent des objets d'indirection qui permettent l'appel des structures de protocole.
- *Attribut pour interface requise*. Comme les attributs pour les interfaces fournies, les attributs pour les interfaces requises permettent une communication directe avec un composant distant ou une communication qui passe par un objet d'indirection. Pour assurer le passage par l'indirection, nous modifions les règles de programmation et demandons l'utilisation explicite des attributs et non des primitives CCM retournant les références des composants distants[1].

7.1.4 Environnement d'exécution et déploiement

Comme dans FAR, l'environnement d'exécution de CCM définit un serveur d'exécution qui donne accès à des services de communication et de nommage.

- *Service de communication* : la communication entre composants CCM est assurée par le bus à messages CORBA. Ce dernier s'occupe de l'acheminement des messages, ainsi que de leur emballage et déballage.
- *Service de nommage* : grâce aux services prédéfinis de CORBA, les composants CCM ont accès à un service de nommage qui permet de faire correspondre un nom symbolique à toute partie référençable d'un composant.

Comme FAR, CCM prête une attention explicite à la phase de déploiement où sont définis, à l'aide de descripteurs XML, les assemblages de composants et leur configuration non fonctionnelle. Ces descripteurs fournissent le moyen de décrire la configuration de la duplication et de la cohérence lors du déploiement des applications CCM.

7.2 Duplication et cohérence dans CCM

Après avoir décrit le modèle des applications CCM, nous nous concentrons dans cette section sur la mise en œuvre des protocoles de duplication et de cohérence. En suivant le plan de présentation des protocoles dans FAR, nous considérons l'instrumentation des applications CCM, la construction des protocoles et les techniques d'intégration de ces protocoles au sein des applications.

---

1. Cette stratégie permet de réutiliser sans modifier les implémentations de CCM. En effet, vu la spécification de CCM et surtout des primitives d'introspection des composants, pour éviter aux programmeurs la contrainte d'utilisation des attributs des interfaces requises, il sera nécessaire de changer les implémentations des ces primitives d'introspection.

7.2.1 Instrumentation des applications CCM

Dans la partie consacrée à FAR, nous avons discuté de l'instrumentation nécessaire aux applications métier afin d'intégrer une gestion de duplication et de cohérence. Plus précisément, nous avons identifié les besoins suivants :

- *Accès à l'état d'un composant*. Les applications métier doivent donner accès aux états des composants qui doivent être dupliqués. Cet accès est nécessaire aux opérations de capture et de restauration impliquées lors des créations et des synchronisations de copies.
- *Contrôle d'état stable d'un composant.* L'accès à l'état d'un composant doit être accompagné par des primitives permettant la capture et la restauration d'états cohérents. En effet, les applications métier doivent éviter les interférences entre les modifications applicatives des états des composants et les opérations de capture et de restauration.
- *Reconfiguration cohérente d'une application*. La gestion de la duplication et de la cohérence au sein d'une application s'exprime en termes de reconfigurations de cette dernière. Les applications métier doivent être instrumentées de manière à ce que les reconfigurations liées à la duplication et à la cohérence puissent être effectuées sans perturber, ni fausser le fonctionnement applicatif.

Nous discutons chacun de ces points dans ce qui suit.

Accès à l'état d'un composant

La sérialisation étant spécifique au langage Java, la seule possibilité pour donner accès à l'état d'un composant CCM est de lui attribuer une interface d'accès à l'état.

La définition d'une interface d'accès à l'état peut être faite de deux manières selon qu'il s'agit de composants CCM persistants ou non persistants. Dans le cas de composants persistants, l'état persistant est défini dans les descriptions CIDL[1] et l'interface d'accès à cet état est automatiquement générée. Dans le cas de composants non persistants, cette interface reste à charge du développeur de composants.

La définition classique d'interfaces d'accès à l'état ne suffit pas pour les opérations de capture et de restauration. En effet, définies en tant qu'interfaces CORBA, ces interfaces ne copient pas des valeurs mais manipulent des références. Pour effectivement manipuler des copies d'état, les interfaces doivent s'assurer d'un passage de paramètres par valeur et non par référence Pour cela elles doivent travailler sur des définitions des états des composants en termes des types valeurs (*valuetype*) de CORBA[2].

Contrôle d'état stable d'un composant

Le contrôle d'état stable des composants consiste à exécuter de manière exclusive les opérations de capture et de restauration d'état. Cette exclusion est basée sur l'identification des opérations applicatives qui modifient l'état des composants, sur l'interception des modifica-

1. La déclaration est faite en utilisant PSDL : le Persistent State Definition Language. Le langage est un sur-ensemble de l'IDL de CORBA et est défini dans la spécification du service de persistance [103].
2. Le *valuetype* de CORBA a été défini pour permettre le passage de paramètres par valeur [101].

tions en cours et sur le blocage de nouvelles demandes de modification (cf. Chapitre 6, Section 6.1, p. 111).

La mise en place dans CCM de ces trois types d'opérations est faite de la même manière que dans FAR. Les seules différences concernent la modification du code des composants CCM pour l'interception des appels internes où les techniques de manipulation de code dépendent du langage de programmation utilisé.

Reconfiguration cohérente d'une application

Une reconfiguration cohérente d'une application est obtenue par exclusion mutuelle entre les traitements de reconfiguration et les traitements de l'application. Cette exclusion est basée sur la connaissance des traitements applicatifs qui modifient l'état de l'application, sur l'interception des traitements de ce type en cours et sur l'interdiction de lancement de tels traitements après une demande de reconfiguration (cf. Chapitre 6, Section 6.2.2, p. 119).

- *Identification des traitements applicatifs*. Nous supposons que les traitements d'une application CCM qui ne doivent pas être perturbés par des traitements de reconfiguration sont les traitements modélisés en tant que *transactions*.
- *Interception des traitements applicatifs en cours*. Pour intercepter les traitements en cours, il est nécessaire d'intercepter les appels au service de transactions. Dans le cas de transactions gérées par le composant lui-même, nous laissons l'insertion des appels d'interception à la charge du développeur. Dans le cas de transactions gérées par le conteneur, nous sommes obligés de modifier les objets d'interposition du conteneur qui sont générés automatiquement et qui assurent les appels au moniteur transactionnel. Les modifications peuvent être faites en reprenant le code généré de ces derniers ou en modifiant le processus de leur génération.
- *Blocage de traitements applicatifs*. L'exclusion mutuelle entre les traitements de reconfiguration et les calculs applicatifs ne peut pas être assurée par une modélisation des reconfigurations en tant que transactions. En effet, dans ce cas, la politique d'isolation dépendra de l'implémentation du service de transactions et l'algorithme de reconfiguration, introduit dans FAR ne sera pas garanti. Pour assurer la priorité des traitements de reconfiguration il est donc nécessaire d'implémenter un mécanisme spécifique où les traitements de reconfiguration bloquent le lancement de nouvelles transactions et attendent la terminaison des transactions en cours. Pour le blocage des transactions, une nouvelle modification des objets d'interposition du conteneur CCM s'impose : ils doivent vérifier, avant de lancer des transactions, l'existence de traitements de reconfiguration en attente.

7.2.2 Construction de protocoles dans CCM

La construction de protocoles de duplication et de cohérence est faite en préservant la logique selon laquelle les composants de protocole sont programmés en suivant le même modèle que les composants métier. Nous décrivons donc les composants de protocole avec le langage IDL de CORBA et les programmons en tant que composants CCM. Comme dans FAR, les composants services et les composants gestionnaires sont programmés de manière définitive, alors que les composants clients et les composants copies servent de modèle aux composants spécifiques générés lors d'une intégration dans une application métier (cf. Chapitre 6, Section 6.3, p. 129).

Les différences entre CCM et FAR en ce qui concerne la programmation de composants de protocole sont les suivantes :

- *Programmation de la procédure de déploiement*. Dans FAR nous définissons la procédure de déploiement en tant qu'une méthode `deployApplication()` dans les gestionnaires de configuration. Dans CCM, nous suivons la logique de déploiement guidé par des descripteurs XML et fournissons des descripteurs additionnels décrivant la configuration d'une application métier avec un protocole de duplication et de cohérence. Ces descripteurs additionnels sont pris en compte lors du déploiement grâce à des modifications des outils de déploiement des implémentations CCM.
- *Composants de protocole persistants, transactionnels ou sécurisés*. En raison du modèle de CCM, les composants de protocole peuvent avoir des implémentations plus complexes que celles dans FAR. En effet, ils peuvent bénéficier de la gestion des transactions, de la persistance ou de la sécurité dans CCM. Ceci a deux conséquences majeures sur l'intégration des protocoles. D'une part, il faut prendre en compte le fait que les composants de protocole et les composants métier auxquels ils doivent être intégrés peuvent avoir des conteneurs non compatibles[1]. D'autre part, l'intégration des composants de protocole doit gérer la coordination entre les transactions au niveau du protocole et les transactions métier. Ce dernier point est discuté immédiatement dans ce qui suit, alors que le problème de conteneurs incompatibles est détaillé dans la section dédiée à l'intégration.
- *Protocole de cohérence, transactions de protocole et transactions applicatives.* Les transactions sont un moyen de gestion de la cohérence des applications métier. Par conséquent, elles ne doivent pas être ignorées par les protocoles de cohérence qui accompagnent l'intégration de la duplication dans les applications métier. Pour éviter d'éventuelles incohérences, nous proposons de modéliser explicitement les transactions au niveau du protocole. En d'autres termes, les interférences entre transactions métier et cohérence de copies sont à gérer par le protocole.

7.2.3 Intégration des protocoles dans CCM

Comme dans FAR, l'intégration des protocoles de duplication et de cohérence dans CCM est faite lors du déploiement. Les composants services et les composants gestionnaires sont instanciés et connectés de manière standard, alors que les composants clients et les composants copies passent par une étape de génération de structures spécifiques avant d'être intégrés dans des composants métier (cf. Chapitre 6, Section 6.3.4, p. 134).

L'intégration des composants clients et copies dans des composants métier dépend de leur utilisation des services non fonctionnels de CCM. Les composants de protocole peuvent être des *composants simples* qui n'utilisent pas les services de CCM, ou alors des *composants augmentés* qui font appel aux services de transactions, de persistance ou de sécurité.

- *Intégration de composants simples*. L'intégration de composants qui ne bénéficient pas des services non fonctionnels de CCM est faite selon le modèle utilisé dans FAR. Les classes spécifiques pour les objets d'implémentation et les objets d'indirection sont générés et intégrés directement dans les composants métier. L'intégration est faite à l'aide

1. Typiquement, il s'agit des cas où les composants de protocole sont programmés en tant que composants persistants, alors que les composants métier ne le sont pas. Si les composants de protocole sont intégrés au niveau des conteneurs métier, ils ne pourront pas faire appel au service de persistance.

de la structure augmentée de ces composants que nous avons introduite précédemment (cf. Section 7.1.3). La structure qui en résulte est montrée à la Figure 7-2.a.

- *Intégration de composants augmentés*. Lors de l'intégration de composants de protocole qui bénéficient des services non fonctionnels de CCM, nous gardons une structure à deux niveaux. Nous préservons les conteneurs des composants de protocole, générés par le processus standard de CCM, et n'intégrons pas ces derniers directement dans la structure des composants métier. Les composants métier ont toujours accès aux fonctions de duplication et de cohérence par leurs objets d'indirection mais cette fois-ci ils ne référencent plus des objets se trouvant dans leur conteneurs mais des objets appartenant à un autre composant. Cette structure est montrée à la Figure 7-2.

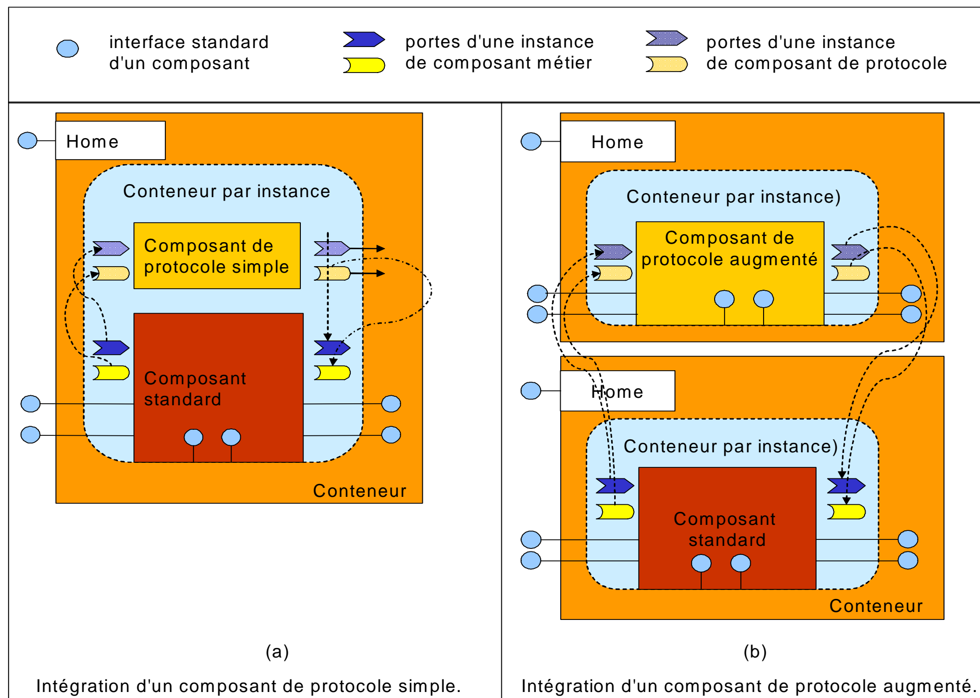


Figure 7-2. Intégration des composants de protocole dans CCM

7.3 Mise en œuvre dans OpenCCM

Nous avons réalisé les principes que nous venons de présenter dans la plate-forme OpenCCM [104]. Dans cette section nous détaillons le contexte de notre réalisation sans décrire les détails d'implémentation. Nous donnons en annexe une description détaillée de l'exemple considérant la construction et l'intégration de l'application "Hello World" et du protocole de tolérance aux fautes.

La plate-forme OpenCCM a été initialement développée dans le cadre du projet RNRT CESURE [26] par l'équipe GOAL de l'Université de Lille. Notre implémentation sur OpenCCM a été faite en tant que contribution à CESURE[1].

L'implémentation que nous avons faite est basée sur la version 0.2 de OpenCCM, développée avant l'acceptation de la spécification de CCM par l'OMG[2]. Cette version est une implé-

1. La contribution au projet CESURE est décrite dans le chapitre dédié aux expérimentations.
2. Cette version date de décembre 2001. CCM est devenu un standard officiel en juin 2002.

mentation partielle de CCM qui n'intègre pas les transactions, la persistance ou la sécurité ce qui a beaucoup facilité notre expérimentation. Depuis, la plate-forme est devenue membre de l'initiative pour l'intergiciel en source libre ObjectWeb [98] et sa base de code a évolué. Toutefois, la gestion des transactions qui est l'aspect le plus problématique lors de l'intégration de la duplication et de la cohérence n'est toujours pas finalisée[1].

OpenCCM fournit un support CCM seulement pour des composants programmés dans le langage Java ce qui nous a permis d'utiliser les techniques de manipulation de code de FAR[2].

7.4 Synthèse

Dans cette section nous avons décrit les principes de mise en œuvre de notre proposition de gestion de la duplication et de la cohérence dans l'environnement CCM.

Nous avons réussi l'intégration de la duplication et de la cohérence dans CCM grâce aux caractéristiques suivantes :

- *Définition d'architecture applicative lors du déploiement.* CCM définit l'architecture initiale d'une application lors du processus de déploiement. Cette information est indispensable à la configuration avec un protocole de duplication et de cohérence qui se fait en établissant des relations entre l'architecture de ce dernier et l'architecture de l'application métier.

- *Introspection et contrôle de connexions.* CCM permet de connaître et de contrôler les connexions existant des instances de composant. Ces fonctionnalités sont indispensables à la gestion des reconfigurations liées à la duplication et à la cohérence.

- *Définition d'état persistant*. CCM permet la description déclarative de l'état des composants persistants et génère automatiquement les primitives d'accès à cet état. Cette caractéristique facilite l'instrumentation des applications métier où l'introduction de l'accès à l'état est une opération lourde qui demande la modification du code métier.

Les caractéristiques qui rendent difficile l'intégration de la duplication et de la cohérence dans CCM sont les suivantes :

- *Gestion des transactions*. Les transactions sont une caractéristique inhérente des applications CCM. Or, la gestion de transactions est en réalité une gestion de cohérence applicative et ne peut être ignorée lors de l'intégration de la duplication de composants et des protocoles de cohérence qui l'accompagnent. Nous proposons une approche qui évite d'éventuelles incohérences en modélisant explicitement, au niveau des protocoles, les transactions et leurs interférences avec les traitements de cohérence de copies. D'autre part, nous avons implémenté la gestion de la duplication et de la cohérence sur la plate-forme OpenCCM qui ne fournit pas de gestion de transactions, de sécurité ou de persistance. Cette caractéristique a annulé le problème de coordination entre transactions applicatives et protocoles de cohérence de copies.
- *Langage de programmation*. CCM est un environnement de composants défini au-dessus de CORBA et, par conséquent, respecte la possibilité d'implémentation dans différents langages. Les techniques d'instrumentation des applications métier, surtout en ce

1. Il s'agit de la fin de 2002.
2. En réalité, l'implémentation sur OpenCCM précède celle de FAR. Toutefois, la logique de mise en place des processus de génération et de modification de code est la même.

qui concerne l'interception des appels, sont dépendantes de l'implémentation. Dans OpenCCM, notre travail a été facilité du fait que les composants sont programmés en Java comme dans FAR.

- *Définition de conteneur*. CCM définit le rôle d'un conteneur et spécifie ses interfaces vis à vis des composants clients, du composant contenu et des services CORBA. Toutefois, sa structuration en termes d'objets d'interception responsables de la gestion des transactions, de la sécurité et de la persistance est laissée à l'implémentation. Cette caractéristique rend dépendantes de l'implémentation l'intégration de nos traitements d'interception pour la duplication et à la cohérence, ainsi que les optimisations obtenues par réduction des objets d'interception et de leurs redirections.

Chapitre 8

Mise en œuvre dans EJB

Ce chapitre présente notre mise en œuvre de la gestion de la duplication et de la cohérence dans l'environnement EJB [136]. Comme dans le chapitre sur CCM, nous considérons, tout d'abord, les applications standard des EJB qui sont des applications sans duplication ni cohérence et discutons ensuite les techniques d'intégration des protocoles de duplication et de cohérence. Nous identifions les différences entre EJB et notre proposition et discutons les principes de leur rapprochement.

8.1 Les applications EJB sans duplication et cohérence

Dans cette section nous présentons une analyse comparative d'EJB et de FAR. Nous approfondissons la présentation d'EJB fournie dans l'état de l'art (cf. Chapitre 1, Section 1.2.1, p. 28) et analysons le modèle de composants des applications EJB, le modèle de programmation des composants EJB et la structure du conteneur EJB. Nous terminons par le processus de déploiement et l'environnement d'exécution EJB.

8.1.1 Modèle de composants

Contrairement au modèle de FAR, le modèle de composants EJB adopte une approche beaucoup plus programmatique qu'architecturale. Fortement lié au modèle de programmation de Java, il laisse cachés dans le code de nombreux détails de la structuration des applications.

Les différences entre EJB et FAR sont les suivantes :

- *Modèle de composants et modèle d'application*. Contrairement à FAR, EJB ne modélise en termes de beans que la partie serveur d'une application. Les clients qui s'adressent aux serveurs EJB n'ont pas l'obligation d'être des beans mais peuvent être des programmes Java arbitraires. Dans la suite nous désignerons sous le nom de *client* toute entité s'adressant à des beans.
- *Types de composants*. Contrairement à FAR, les composants EJB ne sont caractérisés que par *une* interface fournie, appelée *interface de composant*. Les interfaces requises ne sont pas modélisées et les dépendances entre clients et beans restent cachés dans le code. Des indications sur les dépendances entre beans peuvent être fournies au moment du déploiement mais elles concernent uniquement le type des dépendances et non le nombre de références utilisées. Par exemple, le descripteur de déploiement d'un com-

posant EJB qui modélise un achat en ligne peut spécifier une dépendance de beans qui modélisent des produits. La dépendance ne précisera ni le nombre de ces “produits”, ni leur identité.

- *Interfaces de composants*. Dans FAR, un composant peut avoir plusieurs interfaces fournies et plusieurs interfaces requises impliquées dans des communications synchrones ou asynchrones. Dans EJB, les beans entités et sessions peuvent recevoir uniquement des appels synchrones, alors que les beans à messages traitent uniquement des appels asynchrones. Quant aux appels sortants, ils peuvent adresser des interfaces synchrones ou asynchrones d’un nombre illimité de beans. Nous détaillons ce point dans le paragraphe sur la connexion entre composants.
- *Références et noms de composants.* Étant donné que les composants EJB ne sont caractérisés que par leur interface fournie, EJB définit des références de composant qui permettent de s’adresser à cette interface. Pour les beans entité et session, cette référence identifie une instance de composant, alors que pour les beans à messages cette référence identifie un objet de destination de messages. Les instances des beans à messages ne sont pas directement référencées.

  La spécification EJB n’introduit pas de notion de nom pour les instances de composants mais tout ce qui peut être référencé peut avoir un nom symbolique à l’aide du service de nommage JNDI[1] de Java.
- *Connexion de composants*. Contrairement à FAR et à CCM, EJB ne définit pas de primitives explicites de connexion de beans. Les clients peuvent appeler les beans en utilisant directement leurs références qu’ils peuvent obtenir à l’aide du service de nommage JNDI, en appelant des méthodes de recherche d’instances de beans[2] ou par passage de paramètres. Étant donné que les clients n’ont pas besoin d’une étape explicite de connexion, nous désignerons l’ensemble de références détenues par un client sous le nom de *dépendances* et non de connexions. L’ensemble de ces dépendances peut varier dynamiquement en fonction de l’exécution et des critères de recherche de références.
- *Introspection*. Contrairement à CCM et à FAR, EJB ne permet pas l’introspection de l’architecture d’une application. Les dépendances entre clients et beans étant établies de manière implicite par le passage de références, il n’est pas possible de les connaître pendant l’exécution. De plus, vu que la manipulation de références de beans est accompagnée par la création cachée d’instances, il n’est pas, non plus, possible de connaître les instances qui composent une application à un moment donné.

Pour introduire la gestion de la duplication et de la cohérence dans les applications EJB, tout en suivant les principes de notre proposition, nous avons besoin de transformer les clients qui ne sont pas des beans en composants et de gérer de manière explicite les dépendances entre composants. En ce qui concerne les clients non beans, nous devons augmenter leur structure afin de pouvoir intercepter et configurer leurs interactions avec les beans du côté serveur. En ce qui concerne les dépendances, nous devons les gérer de manière explicite afin de pouvoir intégrer et effectuer les traitements de duplication et de cohérence. En effet, la connaissance de ces dépendances est à la base de la définition des relations entre architectures lors de la configuration des applications avec des protocoles de duplication et de cohérence (cf. Chapitre 6, Section

1. Java Naming Domain Interface (JNDI) est la spécification Java d’un service de noms [68].
2. Les méthodes de recherche d’instances sont fournies par les maisons des composants persistants. La recherche est faite sur les champs de l’état d’un tel composant.

6.3, p. 129). Elle est également nécessaire aux reconfigurations applicatives dirigées par ces protocoles.

La manière dont nous traitons les points énoncés ci-dessus est décrite dans ce qui suit.

8.1.2 Programmation de composants

Les composants EJB n'ont pas de description déclarative comme la description IDL de CCM ou la description XML de FAR. Ils sont directement implémentés par les développeurs de composants en tant que classes Java qui implémentent les interfaces de composant[1].

Pour rendre explicites les dépendances de beans, nous suivons la logique de FAR et les représentons par des attributs dans les classes d'implémentation. Ces attributs, que nous appelons des *attributs de dépendance*, caractérisent les types des dépendances, sont spécifiés dans une description déclarative et sont manipulés à l'aide de primitives spécifiques. Le types des dépendances, leur déclaration et manipulation est détaillée dans ce qui suit.

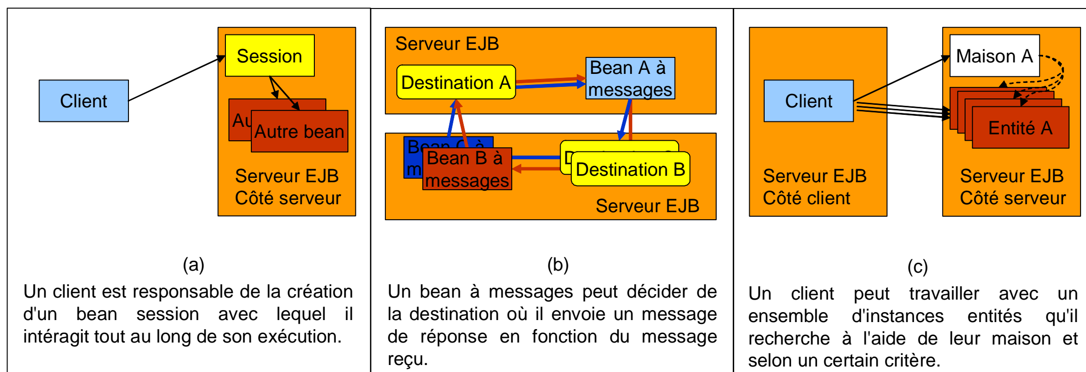


Figure 8-1. Types de dépendances entre beans

Type de dépendance

Nous identifions plusieurs types de dépendances en fonction de leur variabilité et du nombre de beans qu'elle concernent (Figure 8-1.). En effet, les dépendances d'un client peuvent concerner un ensemble d'instances bien identifiées (*dépendances statiques*) ou alors être dynamiquement recalculées (*dépendances dynamiques*). Une dépendance peut être liée à une instance spécifique ou à un ensemble d'instances.

Le cas typique de dépendance statique d'une instance spécifique est le cas d'un client qui crée et qui utilise pendant toute son exécution une instance de bean session[2] (Figure 8-1.a).

Le cas d'un client qui appelle une instance dont l'identité change dynamiquement est illustré par les beans qui renvoient un message de réponse en s'adressant à la destination spécifiée dans l'entête d'un message reçu (Figure 8-1.b).

---

1. En réalité, une classe de bean implémente les méthodes de l'interface du bean mais n'a pas l'obligation de déclarer l'implémentation de cette interface (clause `implements` de Java). Ceci est du au fait que les instances de bean ne sont appelées que par les objets d'interposition du conteneur qui leur sont rattachés et qui connaissent explicitement leur type. Ce sont ces objets d'interposition qui implémentent l'interface du bean et qui fournissent aux clients la vue du service offert par le bean.
2. Les cas de figure énumérés font partie de la spécification EJB.

Le cas d'une dépendance qui concerne un ensemble dynamique d'instances est illustré par un client qui recherche et ensuite travaille sur des instances de beans entités vérifiant un certain critère (Figure 8-1.c).

Description déclarative de dépendances

Nous définissons une description déclarative des dépendances de beans. Plus précisément, pour tout type de bean ou de client non bean, nous caractérisons ses dépendances en leur donnant un nom, en disant si elles concernent une ou plusieurs instances et en spécifiant si elles sont calculées dynamiquement. Toutes ces informations sont définies dans un descripteur XML[1].

Manipulation explicite de dépendances.

Quand un client appelle des beans, il utilise des variables qui contiennent leurs références. Ces variables sont soit directement les variables des paramètres de méthodes, soit des variables internes affectées avec des valeurs obtenues à l'aide du service de nommage ou d'une maison d'instances.

Pour éviter l'affectation et l'utilisation cachée de ces variables, nous imposons l'utilisation des attributs de dépendance. Les appels de beans ne doivent être faits qu'à travers ces attributs qui sont modifiés à l'aide de primitives spécifiques.

La modification d'attributs explicites et non de variables cachées permet l'introspection des dépendances des beans. L'utilisation de primitives d'affectation spécifiques permet l'insertion de code de contrôle qui est nécessaire lors de la configuration de la duplication et de la cohérence.

8.1.3 Structure de composant

La structure d'une instance de bean à l'exécution est composée d'un objet d'interposition, appartenant au conteneur[2], ainsi que d'un objet d'implémentation. L'objet d'interposition (*EJBObject*) intercepte tous les appels rentrants et effectue les traitements nécessaires à la gestion des transactions, de la persistance et de la sécurité.

La différence par rapport à FAR est l'absence d'objets d'interception pour les appels sortants. Pour permettre l'intégration de protocoles de duplication et de cohérence qui ont besoin de cette interception, nous définissons des objets d'indirection dans l'implémentation des composants. Pour cela, nous générons partiellement les classes d'implémentation des beans et y définissons les attributs de dépendances de manière à ce qu'ils puissent référencer des beans ou des objets d'interception (Figure 8-2.). Ces classes sont générées en utilisant les informations données dans le descripteur des dépendances du bean.

Nous modifions également la structure des clients Java qui ne sont pas des beans. Nous supposons que l'implémentation de ces clients est fournie dans une classe Java que nous transformons pour rajouter les attributs de dépendance et les objets d'indirection correspondant. En réalité, nous modifions les clients Java non beans en utilisant les structures définies dans FAR.

---

1. Un exemple d'un tel descripteur est donné dans l'Annexe B.
2. Rappelons qu'un conteneur EJB peut contenir les instances de plusieurs types de beans.

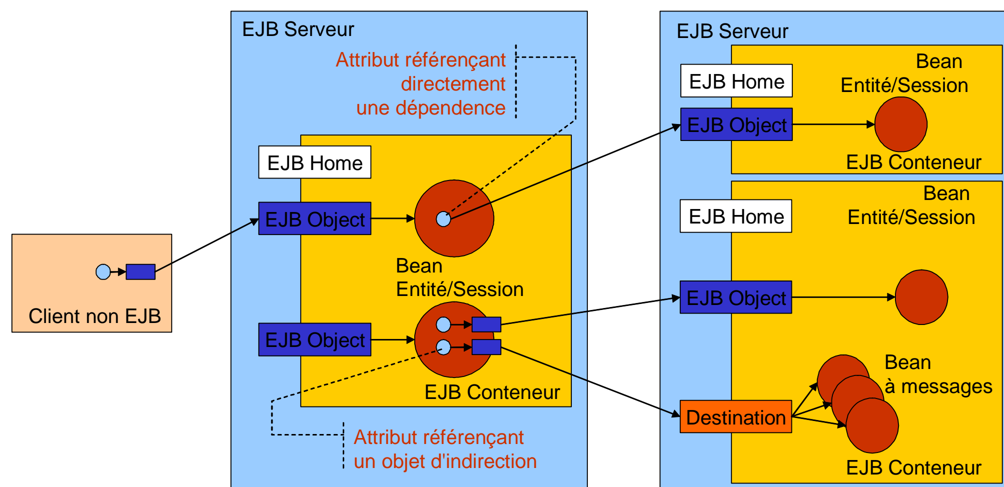


Figure 8-2. Structure modifiée d'un client de bean

8.1.4 Environnement d'exécution et déploiement

L'environnement d'exécution défini par EJB est entièrement basé sur Java. Il offre un service de nommage qui est basé sur JNDI et fournit des services de communication synchrone et asynchrone. Le service de communication synchrone est le mécanisme d'appel à distance de Java (Java RMI[1]) qui impose que les types passés en paramètre soit sérialisables. Le services de communication asynchrone est le service d'échange de messages JMS[2].

Le déploiement dans EJB est guidé par des descripteurs XML qui donnent des informations sur la base logicielle à installer sur les différents serveurs, sur la configuration des composants en termes de transactions, persistance et sécurité, ainsi que sur les dépendances des beans. Ces informations sont utilisées pour la génération des structures de conteneurs nécessaires à l'exécution des instances de beans.

La différence principale d'EJB par rapport à FAR est l'absence de définition de l'architecture initiale d'une application déployée. EJB spécifie un ensemble de types de beans et définit éventuellement des dépendances entre eux mais ne décrit pas la création et l'interconnexion d'instances. Or, ces informations d'architecture nous sont nécessaires pour pouvoir rattacher un protocole de duplication et de cohérence à une application métier.

Pour les besoins de la duplication et de la cohérence, nous introduisons dans EJB un descripteur XML additionnel qui décrit l'architecture initiale de l'application. Toutefois, ce descripteur spécifie plus un modèle d'application qu'une architecture applicative concrète. En effet, comme le fonctionnement des beans EJB est basé sur le calcul dynamique des références de leurs dépendances, il n'est pas possible, au moment de déploiement, d'énumérer et d'identifier exactement toutes les instances d'une application. Pour cette raison, nous raisonnons sur les types de ces instances et leurs éventuelles dépendances pendant l'exécution. Nous créons des descriptions qui sont du même type que les descriptions des protocoles de duplication et de

1. RMI pour Remote Method Call.
2. JMS pour Java Message Service. La version 2.0 de la spécification définit la communication asynchrone uniquement sur JMS. La version 2.1 introduit la possibilité d'utiliser d'autres intergiciels à messages mais reste vague.

cohérence dans FAR où les architectures ne sont instanciées que lors de l'intégration dans une application.

8.2 Les protocoles de duplication et de cohérence dans EJB

Pour décrire la construction et l'intégration des protocoles de duplication et de cohérence dans EJB, nous détaillons l'instrumentation des applications EJB, la mise en place des protocoles et leur intégration.

8.2.1 Instrumentation des applications EJB

Comme dans CCM, l'instrumentation des applications EJB consiste en la mise en place de primitives d'accès à l'état des beans, de contrôle d'état stable et de gestion de la cohérence globale de l'application (cf. Chapitre 7, Section 7.2, p. 142).

Accès à l'état

Les composants EJB sont programmés en Java et par conséquent peuvent donner accès à leur état par une interface d'accès à l'état ou en utilisant la sérialisation Java. Toutefois, la capture et la restauration d'état doivent prendre en compte les spécificités des trois types de beans : les sessions, les entités et les beans à messages.

- *Capture et restauration de beans sessions*. La capture et la restauration des beans sessions sont identiques à celles des composants dans FAR. Les deux opérations sont faites en utilisant une interface d'accès à l'état ou la sérialisation Java sur des instances de beans session identifiées par des références Java.
- *Capture et restauration de beans à messages*. La capture et la restauration d'instances de beans à messages ne peut être effectuée pour la simple raison que ces instances ne sont pas référençables. En effet, seul le conteneur connaît les instances des beans à messages et se charge de faire le lien entre elles et l'objet de destination de messages, adressé par les clients. Ne pouvant pas travailler directement sur les instances, nous assurons la copie d'un bean à messages par la création d'un nouvel objet de destination desservi par des nouvelles instances. En nous basant sur l'hypothèse que l'état des instances à copier est composé de références de beans et d'informations d'environnement, nous initialisons les nouvelles instances indirectement en leur faisant consulter les variables d'environnement concernées et en les obligeant à traiter des messages particuliers contenant les références de beans à copier.
- *Capture et restauration de beans entités*. Les beans entités peuvent avoir un état composé des états d'autres beans entités. En effet, vu qu'ils représentent des données, ils sont structurés pour également représenter les relations entre ces données. Par exemple, une donnée de type `Famille`, définie comme un couple de données de type `Personne`, sera représentée par une instance de bean entité qui référencera deux autres instances entités. L'état de l'instance de `Famille` sera composé des états des instances de `Personne`. En d'autres termes, son état ne sera capture que si l'on capture les états des instances référencées. Par conséquent, pour capturer ou restaurer l'état d'une instance de bean entité, nous considérons tout le graphe d'entités référencées. L'accès à ce graphe peut être assuré, dans le cas d'une persistance gérée par le conteneur, par une interface générée automatiquement.

Contrôle d'état stable

Le contrôle d'état stable doit garantir que l'état capturé ou restauré n'est pas rendu incohérent par des opérations applicatives concurrentes. Les mécanismes qui permettent la mise en place d'un tel contrôle dépendent du type de bean considéré.

- *Beans sessions et à messages*. Dans le cas des beans sessions et des beans à messages, la documentation EJB spécifie que les appels aux méthodes des beans sont sérialisés. Les conteneurs garantissent l'exécution de ces méthodes une par une et de ce fait garantissent l'exécution en mode exclusif des méthodes de capture et de restauration.
- *Beans entités*. Les beans entités peuvent être manipulés de manière concurrente en utilisant les transactions. Or, nous considérons que les transactions modélisent les traitements applicatifs qui ne doivent pas être perturbés par les traitements de reconfiguration liés à la duplication et à la cohérence. Par conséquent, en assurant que les traitements de reconfiguration s'exécutent de manière exclusive par rapport aux transactions applicatives, nous garantissons l'exécution exclusive des méthodes de capture et de restauration dans le cas des beans entités.

Gestion de cohérence globale

Comme dans CCM, nous considérons que les calculs modifiant l'état global d'une application sont les traitements modélisés en tant que transactions. Par conséquent, pour assurer la cohérence globale d'une application nous assurons l'exclusion mutuelle entre les traitements de reconfiguration, liés à la duplication et à la cohérence, et les transactions de cette application.

Pour assurer l'exclusion mutuelle avec les transactions d'une application, nous avons besoin d'intercepter et de contrôler le lancement de ces transactions. L'interception est nécessaire pour savoir si il est possible de lancer des traitements de reconfiguration, alors que le contrôle sur le lancement évite une trop longue attente pour ces traitements.

- *Interception des transactions applicatives*. Dans le cas où les transactions sont gérées par le bean lui-même (*Bean-Managed Transactions*), nous laissons au programmeur le soin d'appeler les méthodes d'interception. Dans le cas de transactions gérées par le conteneur, nous modifions l'objet d'interception (*EJBObject*) qui se charge des appels au moniteur transactionnel. Les modifications consistent à insérer des appels vers des méthodes de la classe que nous fournissons en tant que squelette d'implémentation d'un bean. Ces méthodes se chargent d'appeler le gestionnaire de configuration et de lui signaler les débuts et les fins de transactions.
- *Contrôle des transactions applicatives*. Comme dans CCM, la gestion des transactions sous-jacente ne garantit pas le suivi de l'algorithme de reconfiguration que nous avons défini dans FAR. Pour cette raison, nous introduisons notre propre mécanisme d'isolation entre traitements de reconfiguration et les transactions de l'applications. À cette fin, nous modifions encore l'objet d'interception EJBObject et rajoutons, avant chaque appel de début de transaction, une vérification de droit de lancement auprès du gestionnaire de configuration.

8.2.2 Construction de protocoles dans EJB

Les protocoles de duplication et de cohérence sont programmés en utilisant le modèle EJB, enrichi avec les structures de composants augmentés et les descriptions déclaratives présentées

dans ce qui a précédé. Les problèmes concernant leur construction sont identiques à ceux rencontrés dans l'environnement CCM :

- *Prise en compte des transactions applicatives*. Nous modélisons explicitement les transactions au niveau des protocoles afin d'éviter des incohérences lors de l'intégration de ces protocoles et de leur gestion de la cohérence au sein des applications EJB.
- *Composants de protocole avec propriétés non fonctionnelles*. Les composants de protocoles peuvent bénéficier de la gestion des transactions, de la persistance et de la sécurité dans EJB et se retrouver avec des structures de conteneur qui leur sont propres. Pour assurer la liaison entre composants de protocole et composants métier, nous imposons que les composants de protocole soient accessibles de manière locale. En d'autres termes, ils doivent implémenter non l'interface prédéfinie `EJBObject` qui est utilisée pour accéder les beans à distance, mais l'interface `EJBLocalObject` qui est utilisée pour adresser des beans colocalisés[1].
- *Déploiement*. Pour décrire la manière dont un protocole de duplication et de cohérence doit s'intégrer dans une application EJB, nous fournissons un descripteur XML qui suit les règles définies dans FAR et qui définit les correspondances entre les entités (composants et interfaces) d'une applications métier et d'un protocole. Toutefois, vu que les deux architectures sont décrites en tant que modèles, les correspondances ne concernent pas des instances métier spécifiques mais des types.

Chacun de ces deux points est repris dans la section qui suit.

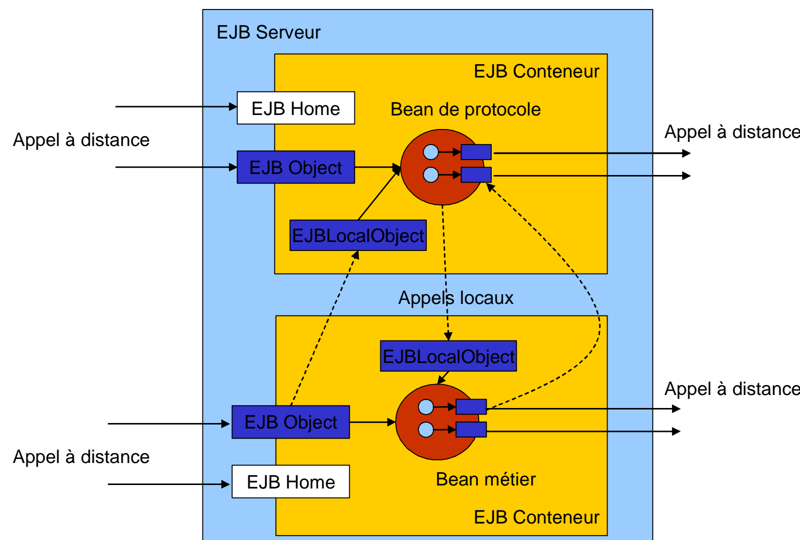


Figure 8-3. Intégration d'un composant de protocole client ou copie dans EJB

8.2.3 Intégration des protocoles dans EJB

L'intégration des protocoles de duplication et de cohérence dans EJB est faite au déploiement en suivant les principes d'intégration de FAR. Les composants gestionnaires et les services sont instanciés et "connectés" de manière standard, alors que les composants clients et les copies

1. L'interface `EJBLocalObject` est utilisée pour faire communiquer des beans s'exécutant dans la même machine virtuelle Java.

passent par une étape de génération spécifique aux types des composants métier. Toutefois, l'intégration dans l'environnement EJB a les particularités suivantes :

- *Beans configurés en clients ou en copies*. Contrairement à FAR, où les composants clients et les composants copies sont fusionnés avec les composants métier, dans EJB nous gardons une structure à deux niveaux. En effet, nous pouvons assurer une communication locale entre les composants métier et les composants de protocole sans que cela exige une colocalisation dans un même conteneur. La communication locale est possible grâce à l'objet d'interposition `EJBLocalObject` introduit par la spécification EJB dans un souci d'efficacité. La structure qui résulte de l'intégration d'un composant de protocole client ou copie est montrée à la Figure 8-3.
- *Configuration dynamique d'instances de bean*. Dans le cas où un bean référence des instances dont les identités varient pendant l'exécution, la configuration en duplication et cohérence ne peut porter que sur les types de ces instances. Le descripteur XML définit, par conséquent, une même configuration pour toutes ces instances. Toutefois, il est possible que ces instances soient différenciées selon des critères applicatifs et qu'elles nécessitent des configurations différentes. Dans ce cas, le protocole doit être spécialisé pour mieux répondre à la logique applicative et le descripteur de déploiement spécifie les différentes configurations en précisant des méthode applicatives de sélection d'instances. Ainsi, lors de l'obtention d'une référence d'instance qui n'est pas configurée, la référence est testée en utilisant les méthodes de sélection d'instances. Dans le cas où cette référence vérifié un des critères de sélection, elle est configurée avec le composant de protocole correspondant[1].

8.3 Mise en œuvre dans JOnAS

Après l'étude de la spécification EJB, nous avons partiellement expérimenté les principes énoncés ci-dessus dans la plate-forme JOnAS [69] qui est fournie en source libre par ObjectWeb [98]. Les détails de cette implémentation sont donnés dans l'annexe B où ils sont illustrés à l'aide de l'exemple de l'application “Hello World” et du protocole de tolérance aux pannes, présentés dans le Chapitre 6.

8.4 Synthèse

Nous avons présenté les principes de notre proposition de mise en œuvre de la gestion de la duplication et de la cohérence dans l'environnement EJB. Les principales difficultés rencontrées lors de l'intégration de la duplication et de la cohérence sont dues aux caractéristiques EJB suivantes :

- *Modèle d'application*. Le modèle EJB ne s'applique qu'à la partie serveur des applications. Or, la gestion de la duplication et de la cohérence a besoin de structures augmentées aussi bien du côté serveur, que du côté client. Dans JOnAS, nous avons traité les structures de bean mais, en ce qui concerne les clients non beans, nous nous sommes restreints aux programmes Java standard. Nous avons ainsi ignoré d'autres types de clients comme, par exemple, les clients WEB

1. Nous supposons que la situation où une même instance est référencée par plusieurs beans et nécessite en conséquence plusieurs configurations est traitée au niveau du protocole.

- *Manque de vue architecturale*. Le modèle EJB a une approche programmatique à la construction d'applications : il ne rend pas explicites les instances qui participent à une application et cache leurs dépendances. Or, la vision architecturale des applications est à la base de notre gestion de la duplication et de la cohérence. Elle permet la définition des traitements de reconfiguration dans les protocoles de duplication et de cohérence et est utilisée au moment de l'intégration de ces protocoles au sein des applications métier. Nous avons introduit la vision architecturale dans les applications EJB en enrichissant le modèle et en définissant explicitement les dépendances de beans. Dans JOnAS, ceci se traduit par des modifications des objets d'interposition des beans et par la définition de classes additionnelles fournissant des conteneurs pour les instances de bean.

- *Gestion de transactions*. Les applications EJB peuvent faire usage des transactions pour assurer un fonctionnement correct à l'exécution. Pour éviter d'éventuelles incohérences du à l'intégration d'un protocole de duplication et de cohérence, nous adoptons la même solution que dans CCM. Nous modélisons explicitement les transactions et leurs interférences avec les traitements de gestion de cohérence de copies.

- *Partage d'état*. Dans EJB, les composants et en particulier les beans entités peuvent partager leur état ce qui complique la gestion des captures et des restaurations d'état. Nous avons proposé une solution qui utilise l'information de dépendances de beans et qui effectue une capture ou restauration d'état en considérant tout le graphe d'instances concernées.

Les caractéristiques qui ont facilité notre expérience sur JOnAS sont les suivantes :

- *Structure de conteneur*. Contrairement à CCM, la spécification EJB définit clairement la structuration du conteneur en termes d'objets d'interposition. Cette information, ainsi que l'organisation des objets d'interposition JOnAS en termes de pré et de post traitements, ont facilité l'identification des points d'indirection nécessaires à l'intégration des traitements de protocole.

- *Langage Java*. La programmation des beans en Java a permis la réutilisation depuis FAR des techniques de manipulation de code et des classes implémentant le modèle de composant.

- *Accès à l'état*. EJB facilite l'instrumentation des composants métier entités dont la persistance est gérée par le conteneur. En effet, l'interface d'accès à l'état dans ce cas est générée automatiquement.

Chapitre 9

Expérimentations

Dans les chapitres précédents, nous avons présenté notre prototype FAR pour la gestion de la duplication et de la cohérence et avons montré la portabilité de ses principes dans les environnements standards EJB et CCM. Dans ce chapitre, nous présentons les expérimentations que nous avons menées sur FAR afin de valider l'approche vis à vis des objectifs énoncés en début de cette thèse. Plus précisément, nous présentons les applications et les protocoles sur lesquels nous avons travaillé et discutons les points suivants :

- *Généralité*. Nous montrons que notre approche permet la modélisation de protocoles issus de différents domaines d'application. Après avoir utilisé un exemple de protocole de tolérance aux pannes tout au long des chapitres précédents, nous détaillons ici la modélisation d'un protocole de déconnexion, ainsi que d'un protocole de cohérence à l'entrée.
- *Souplesse*. Nous montrons qu'une même application peut être configurée pour utiliser différents protocoles. Nous utilisons une application de gestion de listes de courses et montrons qu'elle peut intégrer aussi bien le protocole de déconnexion que le protocole de cohérence à l'entrée.
- *Non-intrusion*. Nous montrons qu'il est possible d'intégrer un protocole de duplication et de cohérence au sein d'une application sans modification de son code métier. Nous discutons le conflit entre l'objectif de non-intrusion, qui implique la mise en place de protocoles génériques, et l'objectif de performances, qui nécessite des configurations spécifiques.
- *Réutilisation*. Nous montrons qu'un même protocole peut être utilisé dans le cadre de différentes applications métier. Nous décrivons la configuration de l'application de gestion de listes de courses et d'une application de gestion de réservations avec le même protocole de déconnexion.
- *Coût de FAR*. Après avoir démontré que FAR répond aux objectifs annoncés, nous discutons le coût qu'il introduit dans les applications en les organisant en termes de composants et de structures de contrôle pour la duplication et la cohérence.

9.1 Applications métier

Cette section présente les trois applications métier sur lesquelles ont porté les expérimentations de configuration de la duplication et de la cohérence. Pour chaque application nous décrivons le contexte de son utilisation, son fonctionnement, ainsi que sa modélisation en termes de composants.

9.1.1 L'application de gestion de réservations AGENDA

L'application AGENDA est utilisée à l'institut INRIA Rhône-Alpes pour la réservation de salles de réunion et de différents types de matériel.

Fonctionnement de l'AGENDA

L'application de réservation est accessible via une page web sur laquelle un utilisateur peut demander la création, la suppression ou la modification d'une réservation. La requête utilisateur est envoyée à un serveur de traitement des réservations qui se charge de vérifier la correction de la manipulation demandée et d'enregistrer le résultat dans une base de données.

Modélisation de l'AGENDA

Nous avons récupéré le code source de l'application AGENDA et l'avons structuré en utilisant le modèle de FAR. Nous avons transformé en composants l'applet et la servlet Java initialement utilisées et avons réalisé l'architecture montrée sur la Figure 9-1. Dans cette architecture, la servlet est représentée par un composant de type `ServerAgenda`, alors que l'applet est représentée par deux composants dont les types sont `GUI` et `ClientAgenda`.

- Composant `GUI`. Le composant `GUI` modélise l'interface graphique qui interagit avec l'utilisateur. Ce composant a une interface requise `GUI_itf` qui définit les opérations qui reflètent les actions de l'utilisateur sur l'interface graphique.
- Composant `ClientAgenda`. Le composant `ClientAgenda` traduit les actions utilisateur en requêtes vers le serveur. Il fournit l'interface `GUI_itf` et requiert l'interface `Reservation_itf`. L'interface `Reservation_itf` définit les opérations de création, de suppression et de modification de réservations.
- Composant `ServerAgenda`. Le composant `ServerAgenda` traite les requêtes de réservation et par conséquent fournit l'interface `Reservation_itf`. Vu que dans FAR nous ne considérons pas la persistance, les informations de réservation ne sont pas stockées dans une base de donnée mais dans l'état interne de `ServerAgenda`.

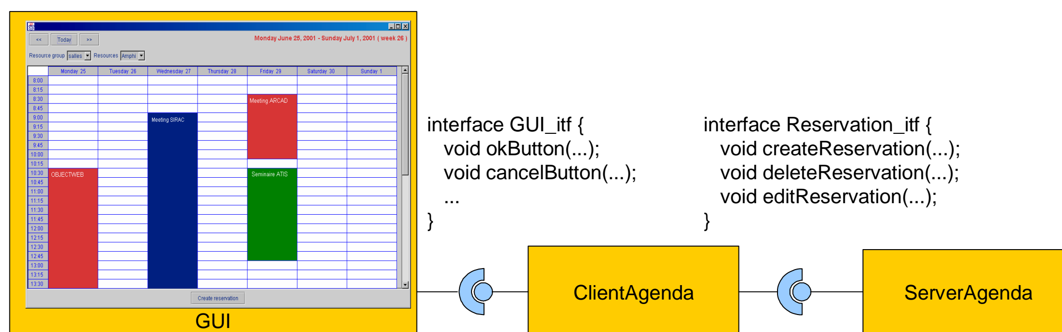


Figure 9-1. L'application AGENDA

9.1.2 L'application de gestion de listes de courses SHOPPINGLIST

L'application SHOPPINGLIST a été considérée dans le cadre d'une coopération avec le laboratoire Hewlett-Packard de Grenoble portant sur l'introduction de la duplication et de la cohérence configurables dans les systèmes de gestion de profils. Avant de détailler le fonctionnement et la modélisation de l'application, nous présentons brièvement le contexte de gestion de profils.

Contexte

Un profil est défini comme une collection d'informations sur l'environnement de travail d'un utilisateur. Ces informations peuvent concerner des paramètres de configuration de l'environnement, des préférences utilisateur, des droits d'accès à des outils et à des informations, etc.

La notion de profil est devenue importante avec le développement de l'informatique mobile où les utilisateurs se déplacent tout en demandant l'accès à leur environnement habituel de travail. En effet, le profil permet à l'environnement, dans lequel un usager mobile se trouve à un instant donné, de fournir les outils de travail demandés. Les informations qu'il contient peuvent être plus ou moins critiques et peuvent être consultées par différentes applications.

Pour assurer la disponibilité des informations de profil lors des déplacements d'un usager, le profil doit être dupliqué. La duplication doit être faite dans différents environnements d'exécution et doit permettre la consultation du profil par différentes applications. La gestion de profils doit, par conséquent, être configurable en termes de la duplication et de la cohérence et s'inscrit directement dans la problématique de cette thèse.

Fonctionnement de la SHOPPINGLIST

L'application SHOPPINGLIST est un exemple de gestion de profils dans lequel un profil est utilisé pour stocker des informations sur des achats de produits. Plus précisément, un profil contient des coordonnées des magasins, les caractéristiques des produits qui y sont vendus, ainsi que des informations sur les derniers achats d'un utilisateur.

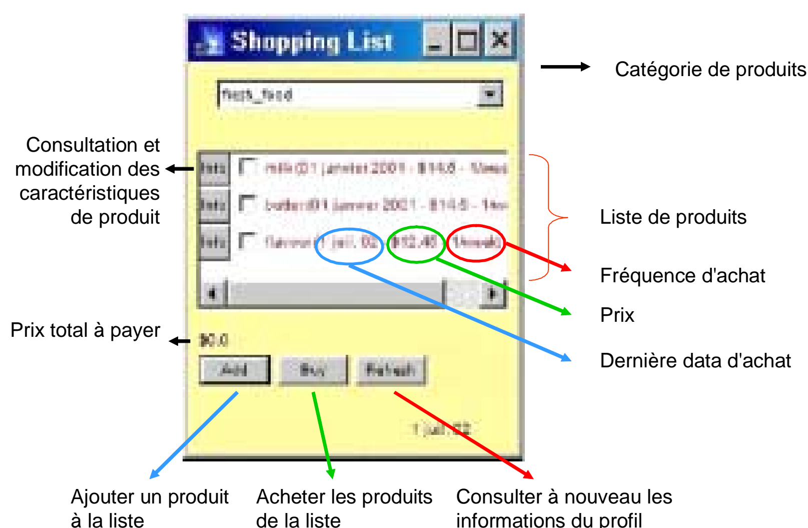


Figure 9-2. L'interface graphique de la SHOPPINGLIST

Pour utiliser la SHOPPINGLIST un utilisateur dispose d'une interface graphique qui lui permet de consulter les produits disponibles et d'effectuer des achats après d'un des magasins de vente. Les produits sont organisés en catégories et sont caractérisés par leur prix, par une fréquence d'achat, ainsi que par leur dernière date d'achat (Figure 9-2.).

La gestion de profils se fait à l'aide de structures indépendantes de l'application. Un profil est structuré en arbre dans lequel les nœuds représentent des informations. L'interface qui est utilisée pour la manipulation de cet arbre est donc une interface de création, de suppression et de modification de nœuds. Toute action de l'utilisateur sur l'interface graphique est traduite en une opération sur les nœuds de l'arbre du profil.

Modélisation de la SHOPPINGLIST

La version de l'application SHOPPINGLIST qui nous a été fournie est basée sur le modèle RMI de Java. Nous l'avons transformée afin d'introduire une architecture à base de composants.

Comme dans l'application AGENDA, nous modélisons par deux composants, `SL_GUI` et `SL_Client`, l'interface graphique et les traitements de traduction des actions utilisateur en opérations sur le profil. Le composant `SL_GUI` a une interface requise `SL_itf` qui est fournie par le composant `SL_Client`. Ce dernier a une interface requise `Profile_itf` qui est fournie par le composant représentant le gestionnaire de profils `ProfileServer` (Figure 9-3.a).

La modélisation du gestionnaire de profils peut être faite de différentes manières en fonction de la granularité des structures que nous voulons manipuler et contrôler explicitement. Une première modélisation possible est celle où le gestionnaire de profils est représenté par un seul composant `ProfileServer` (Figure 9-3.a). Dans ce cas, c'est ce composant qui gère de manière cachée toutes les informations d'un profil. Une deuxième modélisation possible est celle qui rend explicite la décomposition d'un profil en nœuds d'information (Figure 9-3.b). Dans ce cas, tout nœud est représenté par un composant et la structure d'arbre du profil est assurée par des interconnexions explicites entre composants. D'autres modélisations sont possibles où les composants représentent différents sous-arbres dans l'arbre global de profil.

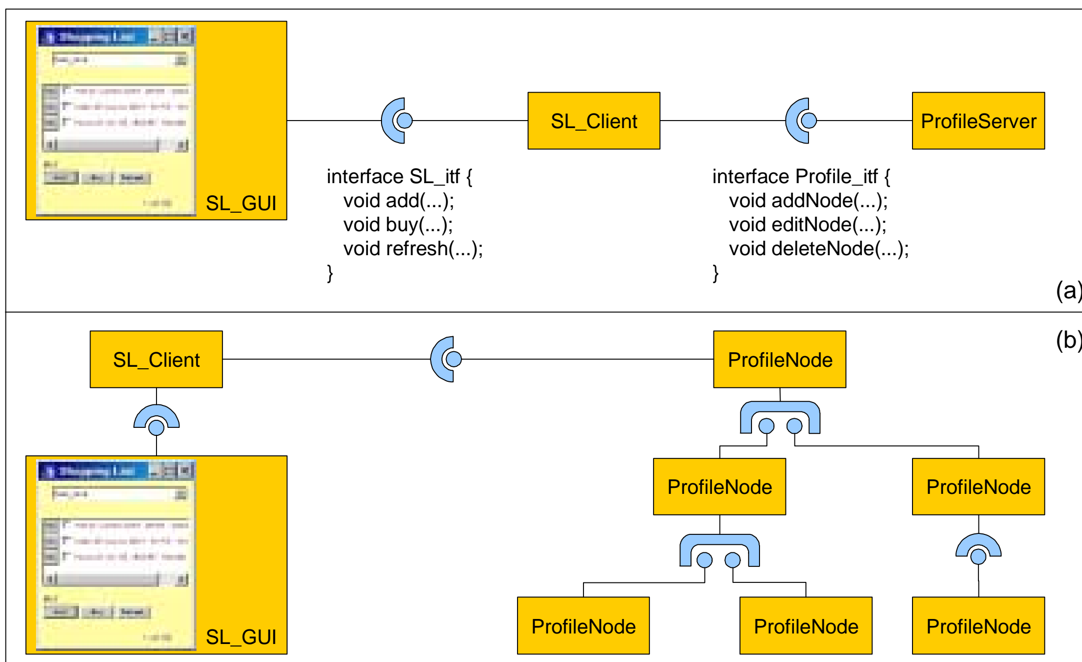


Figure 9-3. Modélisation de la SHOPPINGLIST

9.1.3 L'application de gestion de comptes bancaires BANKING

L'application de gestion de comptes bancaires a été choisie comme application de démonstration pour le projet RNRT CESURE. Avant de présenter l'application, nous introduisons brièvement les objectifs de CESURE, ainsi que notre contribution à ce projet.

Contexte

Le projet CESURE[1] traite du problème de mise à disposition de services à valeur ajoutée aux usagers mobiles du réseau. Plus précisément, CESURE fournit une infrastructure permettant la construction d'*applications de service* dont l'objectif est de fournir des services à valeur ajoutée à des usagers. Ces applications réparties assurent une qualité de service en définissant des traitements de configuration, de déploiement et de contrôle de l'accès aux services fournis.

Notre contribution au projet CESURE concerne la garantie de la QoS lors des problèmes de connexion réseau. Nous avons appliqué notre approche de gestion de la duplication et de la cohérence et avons fourni un service de gestion de déconnexions permettant le fonctionnement des applications de service en mode dégradé.

Fonctionnement de BANKING

L'application BANKING permet à un usager d'effectuer des opérations bancaires sur ses comptes. Développée dans le modèle CCM, elle est structurée avec les composants suivants :

- Composant `Account_UI`. Le composant `Account_UI` est l'interface graphique qui donne accès à l'utilisateur aux opérations de manipulation des comptes bancaires (Figure 9-4.). Ces opérations sont accessibles via les interfaces requises `Balance_itf`, `Transfer_itf` et `Account_itf`. `Balance_itf` est utilisée pour consulter l'état courant des comptes bancaires, `Transfer_itf` est utilisée pour effectuer des virements entre deux comptes et `Account_itf` sert à créditer ou à débiter un compte.

Figure 9-4. L'interface graphique de l'application BANKING

1. CESURE [26] est une abréviation de Configuration et Exécution de Services pour les Usagers mobiles des Réseaux Etendus.

- Composant `AccountService`. Le composant `AccountService` fournit les opérations de consultation d'une liste de comptes et de transfert entre deux comptes bancaires. Il fournit par conséquent les interfaces `Balance_itf` et `Transfer_itf` et requiert l'interface `Account_itf`. Dans la configuration de CESURE, ce composant travaille sur deux comptes bancaires et par conséquent a deux ports de sortie `Account_itf`.
- Composant `Bank`. Le composant `Bank` donne accès à un compte via l'interface `Account_itf` qui définit les opérations de crédit et de débit.

L'architecture de l'application BANKING est montrée à la Figure 9-5.

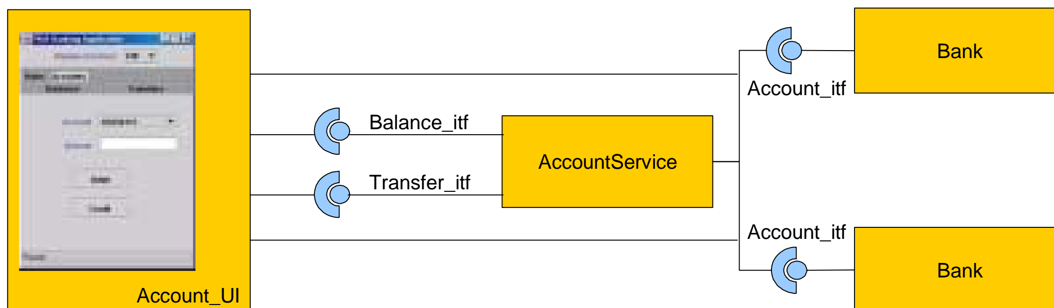


Figure 9-5. Architecture de BANKING.

Dans les expérimentations présentées plus loin dans ce chapitre, nous nous concentrons exclusivement sur les applications AGENDA et SHOPPINGLIST. En effet, nous n'avons intégré qu'un protocole de gestion de déconnexions dans l'application BANKING, alors que l'AGENDA et la SHOPPINGLIST ont été configurées avec plusieurs protocoles.

9.2 Protocoles de duplication et de cohérence

Les protocoles de duplication et de cohérence que nous présentons dans cette section ont été choisis de manière à démontrer la capacité de FAR de modéliser des protocoles venant de différents domaines d'application. Après l'exemple de protocole de tolérance aux pannes, traité dans les chapitres précédents (cf. Chapitre 6, Section 6.2.2, p. 119), nous décrivons ici un protocole de gestion de déconnexions et un protocole de cohérence à l'entrée. La gestion des déconnexions est caractéristique aux environnements à usagers mobiles qui ont des problèmes de connexion, alors que la cohérence à l'entrée est typiquement utilisée dans les mémoires partagées réparties.

9.2.1 Le protocole de gestion de déconnexions DISCONNECTP

Avant de décrire la modélisation à base de composants FAR du protocole de gestion de déconnexions DISCONNECTP, nous présentons brièvement les principes de son fonctionnement.

Fonctionnement

Le protocole de gestion de déconnexions permet à un usager mobile de travailler avec un service même si l'accès à ce service est perturbé par des problèmes de connexion. Les problèmes peuvent venir de la capacité du réseau, de la distance entre le point de connexion de l'usager et le réseau du service, du prix de la connexion, etc.

Le protocole définit deux états pour un usager mobile : connecté et déconnecté. L'usager travaille en mode connecté quand la connexion vers le service est assurée et l'interaction se déroule de manière standard. L'usager travaille en mode déconnecté quand le service n'est plus disponible. Le passage du mode connecté au mode déconnecté peut être déclenché par un outil de surveillance du trafic réseau ou alors par l'usager lui-même.

Le travail en mode déconnecté est assuré par la duplication du service dans un environnement accessible par l'usager. Après une déconnexion, l'usager est redirigé vers une copie du service avec laquelle il interagit de manière *provisoire*. Les opérations qu'il effectue sont enregistrées dans un journal et ne sont validées qu'après une reconnexion et synchronisation avec le service principal. La synchronisation consiste à rejouer les opérations effectuées en essayant de détecter et de résoudre les conflits avec les opérations traitées par les service principal pendant la déconnexion de l'usager. La notion de conflit, ainsi que les procédures de réconciliation ne sont pas définies par le protocole puisqu'elles dépendent de la nature du service copié.

Dans la suite, nous appellerons opération déconnectée (respectivement connectée) une opération effectuée par l'usager pendant son travail en mode déconnecté (respectivement connecté).

Modélisation

Le protocole est modélisé en utilisant cinq types de composants : `Gestionnaire`, `Usager`, `ServicePrincipal`, `ServiceCopie` et `Journal`. Ils sont interconnectés dans l'architecture montrée sur la Figure 9-6. et ont les caractéristiques suivantes.

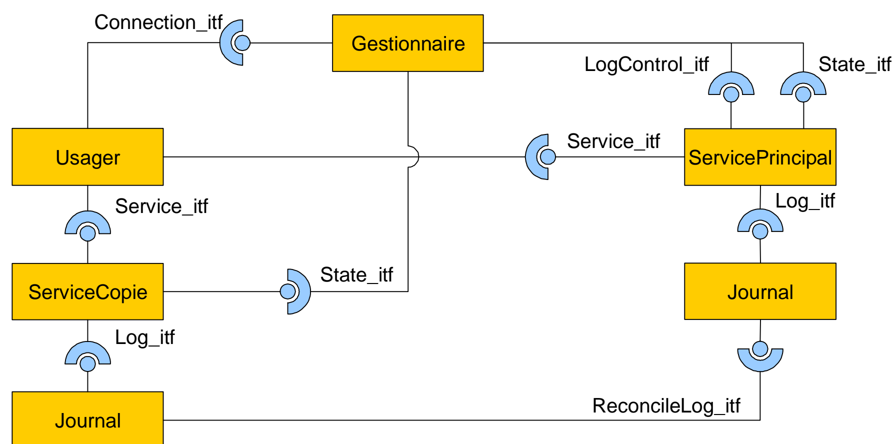


Figure 9-6. Modélisation du protocole DISCONNECTP

- Composant `Usager`.

Le composant `Usager` modélise le programme client d'un usager mobile. Il a deux interfaces requises : `Connection_itf` et `Service_itf`. L'interface `Connection_itf` définit les méthodes `connect` et `disconnect` qui signalent au gestionnaire que l'usager doit changer de mode de travail. Les méthodes peuvent être paramétrées afin de prendre en compte l'état spécifique ou les préférences de l'usager. Dans notre modélisation, les déconnexions sont volontaires et les appels à `disconnect` et à `connect` sont faits par l'usager.

L'interface `Service_itf` définit l'opération `ServiceOp` qui modélise les interactions entre l'usager et le service. Une seule opération suffit puisque toutes les interactions sont journalisées de la même manière par le protocole.

Le composant `Usager` est initialement connecté au composant `ServicePrincipal` ce qui correspond à un travail en mode connecté. Après déconnexion, l'usager est redirigé vers une copie du service principal qui est modélisée par un composant de type `ServiceCopie`.

- Composant `Gestionnaire`.

Le composant `Gestionnaire` est le gestionnaire de configurations. Il fournit les primitives nécessaires à la gestion de la cohérence d'une application métier et définit les traitements de reconfiguration lors des déconnexions et des reconnexions d'un usager (cf. Chapitre 6, Section 6.2.2, p. 119). Ces traitements sont déclenchés par la réception d'appels de `connect` et de `disconnect` de l'interface `Connection_itf`, fournie par le gestionnaire.

Lors de la réception d'un appel à `disconnect`, le gestionnaire procède de la manière suivante. Il attend tout d'abord que l'`Usager` et le `ServicePrincipal`[1] finissent leurs traitements en cours et bloque les traitements arrivants. Il capture ensuite l'état du `ServicePrincipal` avec lequel il restaure l'état d'une instance nouvellement créée de `ServiceCopie`. Pour cela, le gestionnaire est connecté à `ServicePrincipal` et à `ServiceCopie` via l'interface `State_itf`. Après avoir initialisé et rendu opérationnelle le composant `ServiceCopie`, le gestionnaire redirige l'`Usager` vers celui-ci et relance les interactions bloquées.

Lors de la réception d'un appel de `connect`, le gestionnaire assure de nouveau l'exclusion entre le traitement de reconfiguration et les traitements applicatifs et lance la synchronisation entre les journaux de la copie et du service principal. Après l'étape de réconciliation, il détruit la `ServiceCopie` et redirige l'usager vers le `ServicePrincipal`.

Le gestionnaire gère également le journal du `ServicePrincipal par l'`interface `LogControl_itf`. Il active le journal lors de la première déconnexion d'un client pour que les opérations connectées ne soient journalisées que lorsqu'il existe des usagers déconnectés. Il vide et désactive le journal lorsque le dernier usager déconnecté s'est reconnecté.

- Composant `ServicePrincipal`.

Le composant `ServicePrincipal` modélise le service avec lequel un usager mobile travaille en mode connecté. Il joue le rôle de copie maître et détient l'état de référence du service. Il est connecté à un composant `Journal` qui est créé lors de la création du service lui-même. La création du journal et l'établissement de la connexion sont spécifiées dans les traitements prédéfinis de déploiement de `ServicePrincipal` (cf. Chapitre 6, Section 6.2.3, p. 127). L'interface qu'il utilise pour communiquer avec le journal est une interface requise `Log_itf` qui définit la méthode `putOperation` d'enregistrement d'une opération dans le journal.

- Composant `ServiceCopie`.

Le composant `ServiceCopie` modélise la copie locale du service qui est utilisée en mode déconnecté par l'usager. Le journal auquel elle est connectée est créé et activé par les traitements prédéfinis de déploiement de `ServiceCopie`. Ces traitements s'occupent également de connecter le journal de `ServiceCopie` au journal de `ServicePrincipal`.

- Composant `Journal`.

Le composant `Journal` définit les traitements génériques de gestion d'un journal, mais repose sur des traitements spécifiques dont l'implémentation est laissée à l'intégrateur de protocole. Il fournit l'interface `Log_itf` qui définit l'opération `putOperation(Op_itf)` qui s'occupe de la journalisation d'une opération. Pour la réconciliation entre journaux, il fournit et requiert l'interface `ReconcileLog_itf` qui définit la méthode `reconcile`.

1. Il s'agit, bien évidemment, d'instances de ces composants.

Dans `putOperation` de `Log_itf`, l'opération à journaliser est représentée par une interface `Op_itf` qui doit être implémentée en fonction de la spécificité des applications métier. D'autres traitements qui dépendent de la logique métier sont également les traitements de détection de conflit `checkConflict(Op_itf, Op_itf)` et de réconciliation `resolveConflict(Op_itf,Op_itf)`. Ces deux opérations sont à la base de l'étape de réconciliation entre les journaux de `ServiceCopie` et de `ServicePrincipal`.

9.2.2 Le protocole de cohérence à l'entrée ECP

Comme pour le protocole DISCONNECTP, nous présentons brièvement la logique de fonctionnement d'ECP, avant de décrire son architecture à base de composants.

Fonctionnement

Le protocole ECP est un protocole de cache qui gère les accès à des entités partagées. Il permet des accès concurrents en lecture et impose un accès exclusif en écriture. Il assure la localité d'accès en copiant les entités partagées. Il garantit la cohérence à l'entrée en synchronisant les copies avant chaque accès. Plus précisément, il procède de la manière suivante.

- *Lancement du système.* Lors du lancement d'un système géré par ECP, les entités ne sont pas copiées. Ce sont des copies *maîtres*.
- *Premier accès à une entité.* Quand un client accède pour la première fois à une copie maître, il la copie dans son cache local. La copie qui en résulte est une *copie esclave* utilisée pour toute lecture et écriture suivante.
- *Accès en lecture.* Pour qu'un client puisse lire sa copie locale, il doit disposer d'un verrou correspondant qu'il demande auprès de l'entité maître. Si l'entité partagée est en cours d'écriture, la copie maître n'autorise pas les lectures avant la fin de l'écriture, la récupération du nouvel état de l'entité et l'invalidation du verrou d'écriture. Si l'entité partagée n'est pas en cours d'écriture, la copie maître attribue le droit de lecture sans délai.
- *Accès en écriture.* Comme pour les lectures, un client a besoin d'un verrou pour écrire sur sa copie locale. Ce verrou est obtenu auprès de la copie maître qui attend la fin des lectures en cours et qui invalide les verrous correspondants.

Modélisation

Dans le protocole ECP, nous modélisons sous forme de composants les entités maîtres, les entités esclaves, ainsi que les clients qui lisent ou écrivent des entités. L'architecture du protocole fait usage par conséquent d'un composant de type `Maître`, d'un composant de type `Esclave`, et d'un composant `de type Client` (Figure 9-7.).

Il n'y a pas de composant de gestion de configurations puisque la gestion de la cohérence est assurée par la gestion de verrous du protocole et les traitements de reconfiguration sont gérés localement par les composants. Le protocole de gestion de la cohérence à l'entrée est un exemple de protocole qui ne fait pas usage des mécanismes de FAR pour la garantie de la correction des traitements de reconfiguration liés à la duplication et à la cohérence.

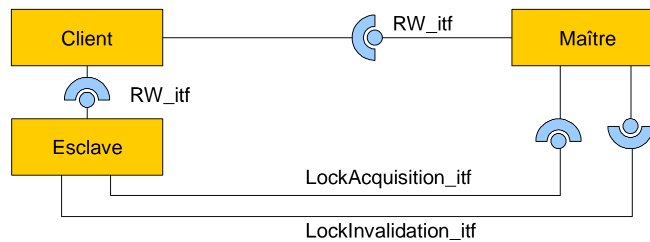


Figure 9-7. Modélisation du protocole ECP

- Composant `Client`. Le composant `Client` est celui qui effectue des écritures ou des lectures sur une entité partagée. Il utilise pour cela l'interface `RW_itf` qui définit respectivement les opérations `read` et `write`. Au lancement du système, il est connecté à une entité `Maître` mais au premier accès il déclenche un traitement de reconfiguration qui crée une entité locale et qui redirige le client vers elle. La tentative d'accès en lecture ou en écriture se traduit par une demande de verrou qui se charge de la synchronisation de l'entité locale avec l'entité maître. Cette demande est gérée par l'entité locale qui est modélisée par un composant de type `Esclave`.
- Composant `Maître`. Le composant `Maître` fournit l'interface `RW_itf` pour les clients qui accèdent à l'entité maître en tant qu'entité locale. Elle fournit également l'interface `LockAcquisition_itf` qui est utilisée par les entités esclaves lors des demandes de verrous. Elle est connectée à ces dernières par l'interface `LockInvalidation_itf` qui sert à invalider les verrous mis en cache. Ce point est expliqué dans ce qui suit.
- Composant `Esclave`. Le composant `Esclave` reçoit des appels `read` et `write` à travers son interface fournie `RW_itf` et se charge de l'obtention des verrous correspondant. Á la fin d'une opération, le verrou n'est pas explicitement libéré mais reste dans le cache. Ceci vise à éviter d'inutiles communications avec le maître dans les cas d'accès séquentiels sur la même entité du cache. Pour que le verrou soit réellement libéré, il doit être invalidé par la copie maître lors d'une demande de verrou en écriture par une autre copie esclave.

9.3 Intégration de DISCONNECTP dans SHOPPINGLIST. Non-intrusion.

Dans cette section nous discutons l'objectif de non-intrusion qui consiste à intégrer un protocole de duplication et de cohérence au sein d'une application métier sans modification de cette dernière. Nous évaluons l'approche de FAR en considérant l'intégration du protocole de gestion de déconnexions DISCONNECTP dans l'application de gestion de liste de courses SHOPPINGLIST. La modélisation de la SHOPPINGLIST pouvant être faite de différentes manières, nous traitons les deux modélisations présentées précédemment : tout d'abord, nous considérons le cas de serveur monolithique et présentons ensuite le cas de serveur de profil avec un composant par nœud.

9.3.1 SHOPPINGLIST avec serveur monolithique

Dans cette section nous considérons la modélisation de la SHOPPINGLIST dans laquelle le serveur de profils est représenté par un seul composant (Figure 9-3.a). Nous détaillons la procé-

dure d'intégration du protocole DISCONNECTP dans cette architecture et discutons les possibles améliorations de performances.

Procédure d'intégration

L'intégration du protocole DISCONNECTP dans la SHOPPINGLIST passe par l'instrumentation de la SHOPPINGLIST et par les étapes d'intégration définies dans FAR (cf. Chapitre 6, Section 6.3, p. 129). Il s'agit notamment de donner accès à l'état du serveur `ProfileServer` pour qu'il soit dupliqué, d'établir les relations entre les architectures de la SHOPPINGLIST et de DISCONNECTP et de générer les structures spécifiques de DISCONNECTP pour son intégration dans l'application.

(i) Instrumentation de la SHOPPINGLIST.

Pour permettre au serveur `ProfileServer` d'être dupliqué, nous lui faisons implémenter une interface d'accès à l'état. Les méthodes de cette interface permettent la capture et la restauration de la table de hachage qui contient les informations concernant les listes de courses.

(ii) Relations entre interfaces.

Cette étape est la première étape d'établissement de correspondances entre les entités de l'application et les entités du protocole. Elle établit des relations entre les interfaces des deux niveaux afin de spécifier les opérations métier qui doivent être contrôlées par le protocole. Dans le cas de la SHOPPINGLIST, cette étape spécifie que les opérations de mise à jour des listes de courses doivent être gérées pour supporter le mode déconnecté. Plus particulièrement elle dit que les opérations `editNode`, `addNode` et `removeNode` de l'interface `Profile_itf` de l'application sont réifiées en tant qu'opérations `ServiceOp` de l'interface `Service_itf` du protocole. En effet, les trois opérations applicatives sont celles qui sont utilisées pour les mises à jour d'un profil, alors que l'opération du protocole modélise les opérations gérées en mode déconnecté.

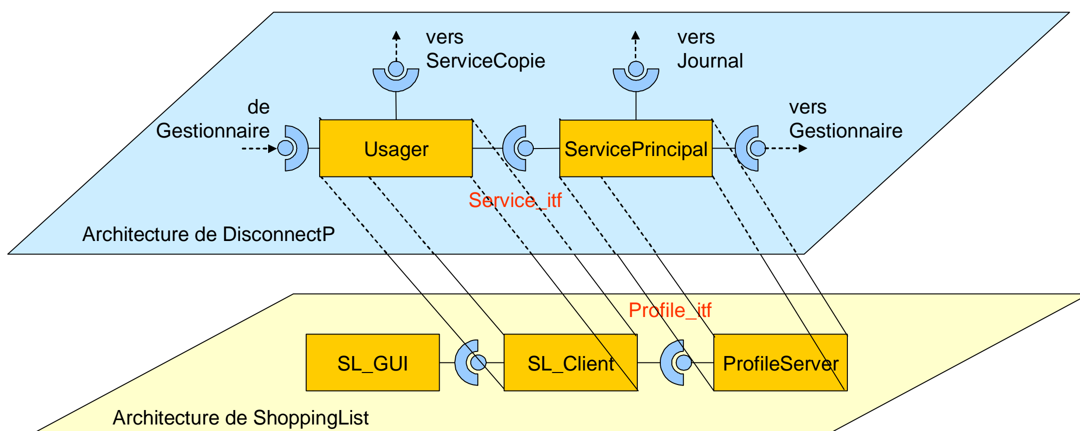


Figure 9-8. Relations entre les composants de la SHOPPINGLIST et de DISCONNECTP

(iii)Relations entre composants.

L'étape d'établissement de relations entre composants détermine les rôles des composants de l'application du point de vue du protocole. Dans le cas considéré, le type `SL_Client` de la SHOPPINGLIST est configuré avec le type `Usager` de DISCONNECTP pour permettre au client de l'application de travailler en mode déconnecté. Le type `ServicePrincipal` de DISCONNECTP

est attaché au service auquel le client doit accéder c'est à dire au serveur `ProfileServer` de la SHOPPINGLIST. La connexion et la déconnexion sont faites à la demande de l'usager qui appuie un bouton rajouté dans l'interface graphique SL_GUI.

(iv) Spécialisation de protocole.

Avant l'intégration effective du protocole DISCONNECTP au sein de l'application SHOPPINGLIST, il est nécessaire de générer des types de composants de protocole qui sont spécifiques à l'application. Plus précisément, il faut passer par les étapes suivantes :

- Génération de classes spécifiques d'implémentation pour les composants de protocole.

Le processus d'intégration génère des classes spécifiques pour les composants `ServicePrincipal`, `ServiceCopie` et `Usager` qui n'utilisent pas `Service_itf` mais `Profile_itf`. Lors de cette génération, les classes de `ServicePrincipal` et de `ServiceCopie` se retrouvent à implémenter leur interface fournie `State_itf` avec les primitives d'accès à l'état du composant `ProfileServer`. Après la génération des classes spécifiques des composants, le processus d'intégration s'occupe de la génération des classes des talons et des squelettes pour ces composants.

- Implémentation des traitements non fournis par le protocole.

Pour que le traitement de DISCONNECTP soit complet, il faut implémenter les primitives de réconciliation `checkConflict` et `resolveConflict` qui manipulent des opérations de type `Op_itf`. Étant donné que l'intégration est faite dans SHOPPINGLIST, il est nécessaire de fournir une implémentation de `Op_itf` qui permet la représentation des opérations métier de `Profile_itf`.

Dans la SHOPPINGLIST, les trois opérations `addNode`, `editNode` et `removeNode` modifient l'état du serveur `ProfileServer`. Si `checkConflict` prend comme unité de synchronisation l'état global de ce dernier, il détecterait systématiquement des conflits. Pour éviter cela, nous avons fait porter `checkConflict` sur les informations de plus petit grain que sont les nœuds de `ProfileServer`. Ainsi, c'est seulement dans le cas où deux opérations modifient le même nœud qu'elles sont considérées en conflit. L'implémentation la plus simple de `resolveConflict`, dans ce cas, est de rejeter toute opération déconnectée qui porte sur un nœud modifié en mode connecté.

Amélioration des performances côté client

Du point de vue du client de la SHOPPINGLIST, les améliorations qui peuvent être faites portent sur le volume des informations dupliquées et sur le nombre d'opérations rejetées (nombre de conflits).

- Volume d'informations à dupliquer.

Le volume d'informations copiées est important puisqu'il est lié au temps de copie et à la taille de stockage disponible du côté client. En effet, plus le serveur `ProfileServer` a de nœuds, plus la primitive de copie de la table de hachage qui contient ces nœuds prendra du temps. De même, la taille de l'espace de stockage est proportionnelle au nombre de nœuds. Or, il est probable que l'usager travaille sur un dispositif ayant des capacités limitées de stockage et qu'il n'utilise en déconnecté qu'une partie des informations copiées.

Pour réduire la taille de stockage utilisé, nous avons implémenté une version de la SHOPPINGLIST où l'interface d'accès à l'état de `ProfileServer` permet une capture partielle. Le para-

métrage est fait par l'usager qui, avant de se déconnecter, choisit la catégorie de produits qu'il veut manipuler. Nous avons expérimenté avec succès cette configuration de la SHOPPINGLIST dans un environnement où le client et le serveur s'exécutent respectivement sur un assistant personnel et sur une station de bureau (Figure 9-9.).

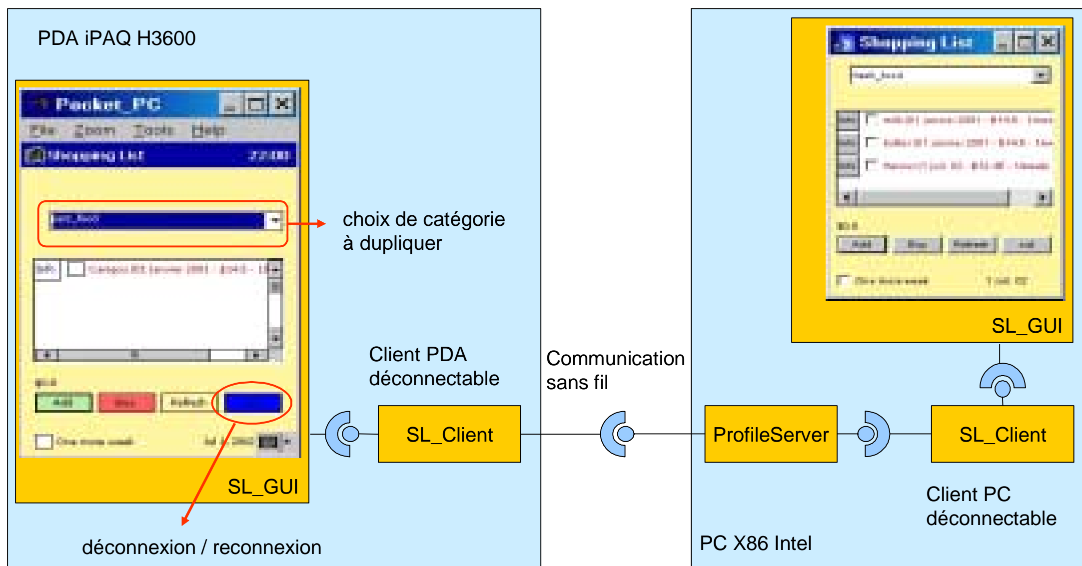


Figure 9-9. Gestion des déconnexions dans la SHOPPINGLIST.

- Nombre de conflits.

L'implémentation de `resolveConflict` influence grandement l'efficacité du travail en mode déconnecté. Dans le cas où cette méthode ne tolère aucun conflit, l'usager risque d'effectuer de nombreuses opérations qui seront ignorées après reconnexion. Par exemple, si `resolveConflict` rejette les opérations `editNode` déconnectées, les achats programmés par l'usager ne seront pas effectués. En effet, vu que les opérations `buy` se traduisent en des opérations `editNode`, elles seront ignorées.

Nous avons donc implémenté `resolveConflict` de manière à distinguer les opérations de `Profile_itf`. Nous avons mis en place une matrice de compatibilité qui spécifie les opérations en conflit et leurs traitements de réconciliation (Figure 9-10.). Ainsi, lorsqu'un nœud a été modifié en mode déconnecté et en mode connecté, il n'y a un conflit que si la modification porte sur un même attribut de ce nœud. Le traitement de réconciliation dit que si, par exemple, l'attribut en conflit est la dernière date d'achat, alors c'est la date la plus récente qui est validée.

Amélioration des performances côté serveur

Le problème de performances qu'a le serveur de la SHOPPINGLIST est lié à l'étape de réconciliation lors de la reconnexion d'un client déconnecté. En effet, vu que cette réconciliation fait partie d'une opération de reconfiguration qui modifie l'état du serveur, le serveur est bloqué et ne traite pas les appels arrivants. En d'autres termes, pendant l'étape de réconciliation, les clients connectés ne peuvent plus travailler même s'ils veulent manipuler des parties de l'état du serveur qui ne sont pas en cours de réconciliation. Le grain de réconciliation au niveau du serveur empêche, par conséquent, la concurrence entre clients. Nous avons donc implémenté une version où le serveur est modélisé en termes de plusieurs composants qui définissent un grain de réconciliation meilleur. Cette version est présentée dans ce qui suit.

Opération connectée Opération déconnectée	editNode	addNode	removeNode
editNode	- CONFLIT si sur même attribut - si fréquence d'achat, garder la plus grande - si date d'achat, garder la plus récente	- PAS DE CONFLIT - editNode a été précédé par un addNode, résolution ci-dessous	- PAS DE CONFLIT - editNode a été précédé par un addNode, résolution ci-dessous
addNode	- CONFLIT - addNode ignoré - addNode a été précédé par removeNode	- CONFLIT - garder un seul addNode - reconciliation identique à editNode / editNode	- PAS DE CONFLIT - exécuter addNode
removeNode	- CONFLIT - ignorer removeNode	- CONFLIT - ignorer removeNode	-PAS DE CONFLIT - exécuter removeNode

Figure 9-10. Matrice de réconciliation

9.3.2 SHOPPINGLIST avec serveur composé

Dans cette section nous considérons la modélisation du serveur de profils qui utilise un composant par nœud d'information (Figure 9-3.b). Avant de décrire l'intégration du protocole dans le cas d'un tel serveur composé, nous discutons du fonctionnement de la SHOPPINGLIST avec cette modélisation.

Fonctionnement de la SHOPPINGLIST avec un serveur composé

Dans l'architecture à serveur composé, le client reste connecté au nœud principal qu'est la racine de l'arbre de profil. Les différences par rapport à l'architecture à serveur monolithique concernent la connexion du client aux autres nœuds du profil, ainsi que l'interconnexion des nœuds au sein de l'arbre.

Dans notre implémentation nous avons voulu mettre en place un traitement rapide des opérations de l'interface `Profile_itf`. Pour cela, nous permettons la connexion directe entre le nœud racine du serveur et n'importe quel autre nœud dans l'arbre. De même, le client peut se connecter directement au nœud qu'il veut manipuler et ne pas être obligé de passer par la racine du serveur. Plus particulièrement, ces deux points sont gérés de la manière suivante.

Dans le serveur, les composants nœuds "fils" sont connectés aux composants nœuds "pères". Toutefois, pour éviter le parcours de l'arbre à chaque opération de `Profile_itf`, telle que `editNode`, nous gérons, au niveau de la racine, une structure qui contient les références de tous les nœuds de l'arbre. Ainsi, pour retrouver le nœud à manipuler, ce ne sont pas les connexions dans l'arbre qui sont parcourues mais cette structure qui est faite pour accélérer et faciliter le traitement. En effet, la recherche des références des nœuds est nécessaire pour permettre aux clients de se connecter directement au nœud qui les intéresse et leur éviter de déléguer le traitement au nœud racine du serveur.

Au niveau du client, nous gérons également une structure de cache qui contient les références des nœuds déjà manipulés par le client. Le client interprète toute action sur l'interface graphique en tant qu'opération sur un nœud identifié. Par exemple, si l'utilisateur clique sur le produit "Lait", le composant client cherchera à manipuler le nœud identifié par ce nom. Pour effectivement traiter l'opération demandée, le client vérifie si il ne détient pas déjà la référence de ce nœud et, si la référence est disponible localement, le client se connecte directement au nœud correspondant. Si la référence n'est pas connue, elle est demandée auprès du nœud racine du serveur.

Procédure d'intégration

La procédure d'intégration est quasiment identique à celle décrite dans le cas du serveur monolithique. La correspondance entre interfaces est toujours entre `Profile_itf` et `Service_itf`, le client de la SHOPPINGLIST est toujours configuré avec le composant `Usager` du protocole et le serveur est configuré avec `ServicePrincipal`.

Les différences sont les suivantes :

- *Configuration et instrumentation des composants nœuds*. L'intégration de DISCONNECTP implique la configuration de *tous* les composants nœuds du serveur par le composant `ServicePrincipal`. Cette configuration est faite au moment du déploiement en parcourant toute l'architecture de l'arbre. Étant donné que les nœuds doivent pouvoir être dupliqués, l'interface d'accès à l'état doit être définie au niveau du type de composant `ProfileNode` qui modélise un nœud.
- *Traitement de déconnexion*. Alors que dans la première version, le client se déconnecte d'un seul composant, dans le cas de serveur composé il référence plusieurs nœuds qui doivent être dupliqués. De plus, la déconnexion doit assurer que les nœuds copiés forment une structure d'arbre cohérente qui permet le traitement correct des opérations d'ajout et de suppression de nœuds. Même si cette vérification peut impliquer la prise en compte de nœuds non référencés par le client, l'ensemble de nœuds dupliqués est optimal puisqu'il contient l'arbre minimal recouvrant les nœuds utilisés par l'usager.

Discussion des performances

La modélisation du serveur comme un ensemble de composants influence les performances de la SHOPPINGLIST sans duplication, ni cohérence, ainsi que celles de l'assemblage de l'application avec le protocole DISCONNECTP.

De point de vue de l'application, les traitements dans le cas d'un serveur composé sont accélérés puisqu'ils ne demandent pas de parcours de l'arbre et ne sont pas séquencées par le nœud racine qui se charge de l'exécution de toutes les opérations. Toutefois, la structuration composée introduit un coût supplémentaire puisqu'elle transforme les opérations de `Profile_itf` en opérations de reconfiguration. En effet, toute opération d'un client est précédée par une opération de connexion au nœud correspondant. Toute opération de création d'un nœud est composée d'une opération de connexion au nœud père et d'un enregistrement au niveau du nœud racine. Enfin, toute opération de suppression de nœud détruit les connexions de ce dernier et l'efface au niveau du nœud racine. En résultat, le temps de traitement des opérations client est, dès le début de l'exécution de ce dernier, plus long que dans le cas de serveur monolithique et augmente avec le nombre de nœuds que le client référence. Dans les mesures que nous avons effectuées, nous avons relevé des cas où des appels sont dix fois plus lents dans la version avec serveur composé que dans la version avec serveur monolithique[1].

Si on ignore les performances dues à l'implémentation de FAR, la gestion de la duplication et de la cohérence est optimale. En effet, lors d'une déconnexion, le volume des informations copiées est minimal et ne considère que les informations manipulées par le client. Lors d'une reconnexion, la réconciliation ne bloque que les nœuds qui ont été manipulés pendant la déconnexion et permet l'utilisation concurrente du serveur.

9.4 Intégration d'ECP dans SHOPPINGLIST. Souplesse.

Dans cette section nous discutons l'objectif de souplesse qui consiste à permettre à une même application métier d'utiliser différents protocoles de duplication et de cohérence. Nous considérons l'application SHOPPINGLIST et montrons qu'elle peut être configurée avec un autre protocole que le protocole DISCONNECTP : le protocole ECP.

9.4.1 Modélisation de la SHOPPINGLIST

Nous avons vu dans la section précédente que la version modélisant le serveur de la SHOPPINGLIST comme un composant unique pose des problèmes de concurrence entre les clients. Les opérations de déconnexion et de réconciliation bloquent les modifications applicatives du serveur et de ce fait bloquent le fonctionnement des clients.

Si l'on considère l'intégration du protocole de gestion de la cohérence à l'entrée ECP dans SHOPPINGLIST, le problème de concurrence se pose pendant toute l'exécution et non seulement pendant certaines phases de fonctionnement. En effet, si l'on utilise la modélisation à composant monolithique, les clients seront obligés de demander des verrous sur l'état global du serveur même s'ils ne manipuleront qu'un sous-ensemble d'informations. Par conséquent, ils ne pourront pas modifier en parallèle différentes parties de l'état du serveur. Sachant que les opérations `editNode`, `addNode` et `removeNode` sont des opérations qui modifient le serveur, les clients seront amenés à sérialiser leurs accès ce qui annule les avantages d'un protocole comme ECP.

Pour ces raisons, nous considérons la version de la SHOPPINGLIST où tout nœud d'information du serveur est modélisé comme un composant.

9.4.2 Procédure d'intégration

Comme dans tous les cas d'intégration, la procédure compte cinq étapes. La première étape est l'instrumentation de l'application métier. La deuxième et la troisième étapes s'occupent de définir les correspondances entre les architectures métier et de protocole. La quatrième étape génère les composants de protocole spécifiques au fonctionnement métier et, finalement, la cinquième étape déploie l'assemblage de l'application métier et du protocole.

(i) Instrumentation de la SHOPPINGLIST

Étant donné que les entités qui seront mises en cache et donc dupliquées sont les nœuds d'information, le composant qui modélise un nœud doit être muni d'une interface d'accès à l'état.

1. Plus de détails sur les performances de FAR sont donnés dans la section sur le coût de FAR (Section 9.6)

(ii) Relations entre interfaces

Les relations entre interfaces sont établies entre l'interface `RW_itf` du protocole et l'interface `Profile_itf` de l'application afin de caractériser les opérations applicatives de `Profile_itf` en tant que des opérations de lecture ou d'écriture. Ainsi, les opérations `editNode`, `addNode` et `removeNode` qui sont des opérations qui modifient l'état du serveur sont modélisées en tant qu'opérations d'écriture (opération `write` de `RW_itf`). L'opération `currentNode` qui sert à choisir le nœud à manipuler et à le visualiser au niveau de l'interface graphique est une opération de lecture représentée par l'opération `read` de `RW_itf`.

(iii)Relations entre composants

Le client `SL_Client` de la ShoppingList est représenté par le composant `Client` d'ECP puisqu'il accède en lecture ou en écriture aux nœuds du serveur. De leur côté, les nœuds du serveur sont représentés par le composant `Maître` d'ECP.

(iv)Composants ECP spécifiques

La dernière phase avant le déploiement de la ShoppingList et du protocole ECP est la génération de composants de protocole spécifiques aux types de l'application métier. Cette génération crée des composants `Client`, `Maître` et `Esclave` qui n'interagissent plus à travers une interface `RW_itf` mais qui utilisent l'interface `Profile_itf`. Ainsi, lors d'une opération `addNode`, modélisée en tant qu'opération d'écriture, le protocole ECP rajoutera un prétraitement d'acquisition de verrou en écriture et un post-traitement de libération de verrou. Lors d'une opération `currentNode`, le prétraitement demandera un verrou en lecture et le post traitement le libérera.

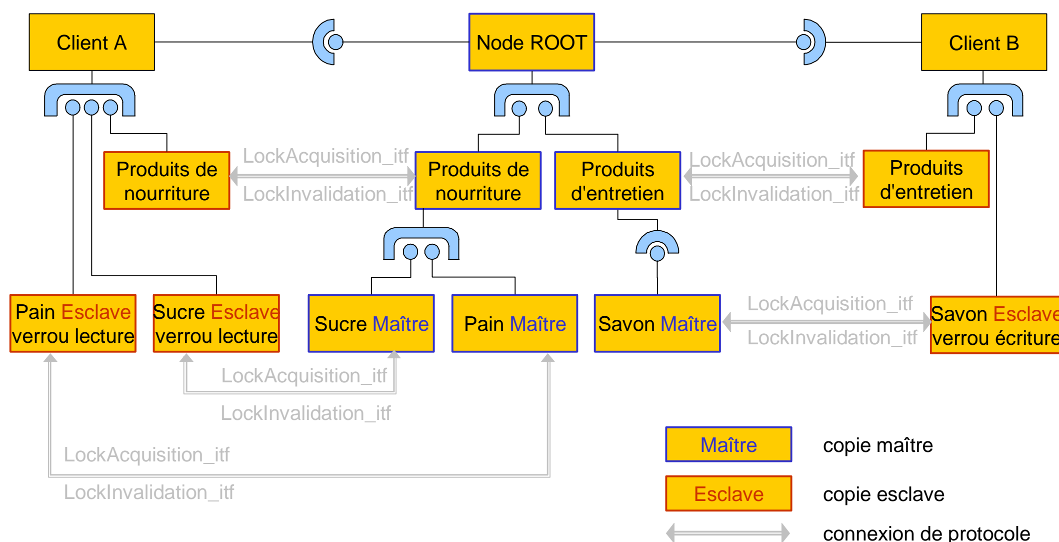


Figure 9-11. Architecture de la ShoppingList avec gestion de la cohérence à l'entrée

(v) Déploiement

L'intégration effective du protocole ECP dans l'application ShoppingList est faite au déploiement en attachant un composant `Client` spécifique au composant `SL_Client` et en attachant un composant `Maître` à chaque composant nœud du serveur. Lorsqu'un nœud est mis en cache, il est copié dans un composant de type `Esclave`.

La Figure 9-11. montre l'architecture que l'application SHOPPINGLIST, configurée avec le protocole ECP, peut avoir à un moment de son exécution. Dans cette situation, un client A détient des copies mises en cache, accessible en lecture, des produits de la catégorie "Produits de nourriture". Un autre client B, détient des copies des produits qu'il veut acheter (accès en écriture).

9.5 Intégration de DISCONNECTP dans AGENDA. Réutilisation.

Cette section traite de l'objectif de réutilisation qui consiste à pouvoir intégrer un même protocole de duplication et de cohérence au sein de différentes applications métier. Pour cela, elle considère le protocole de gestion de déconnexions DISCONNECTP et son intégration dans les applications SHOPPINGLIST et AGENDA (cf. Section 9.1.1). La section 9.3 ayant détaillé l'intégration de DISCONNECTP dans l'application SHOPPINGLIST, cette section se concentre sur l'intégration du protocole dans l'application de gestion de réservations AGENDA.

9.5.1 Procédure d'intégration

La procédure d'intégration est très semblable à celle discutée dans le cas de l'application SHOPPINGLIST. Les différences concernent surtout l'implémentation des opérations `checkConflict` et `resolveConflict` qui spécialisent le protocole DISCONNECTP pour le fonctionnement de l'AGENDA.

(i) Instrumentation de l'AGENDA

Le composant qui doit être accessible en mode déconnecté et qui doit donc être dupliqué est le composant serveur de l'AGENDA. Par conséquent, le composant `ServerAgenda` doit fournir une interface d'accès à l'état. Dans notre implémentation, nous avons opté pour une approche mixte faisant usage de la sérialisation Java et d'une interface d'accès sélectif. La sérialisation Java facilite la capture globale de l'état du serveur, alors que l'interface d'accès sélectif est utile lors de l'étape de réconciliation après reconnexion.

(ii) Relations entre interfaces

Les opérations qui doivent être enregistrées dans un journal pendant une déconnexion sont les opérations de l'interface `Reservation_itf`. Par conséquent, toutes les opérations de cette interface sont réifiées par l'opération `ServiceOp` de l'interface `Service_itf` du protocole.

(iii)Relations entre composants

Pour permettre que le client `ClientAgenda` puisse travailler en mode déconnecté et utiliser le serveur `ServerAgenda`, `ClientAgenda` est défini en tant qu'un `Usager`, alors que `ServerAgenda` est défini en tant qu'un `ServicePrincipal`.

La Figure 9-12. montre les relations d'architecture entre l'application AGENDA et le protocole DisconnectP. La ressemblance avec le cas de l'application SHOPPINGLIST, dans le cas d'un serveur monolithique, est évidente.

(iv) Spécialisation de DISCONNECTP

En plus de la génération de composants de protocole qui n'utilisent pas `Service_itf` mais `Reservation_itf`, DISCONNECTP a besoin d'implémentations spécifiques de l'interface `Op_itf` et des opérations `checkConflict` et `resolveConflict`.

Notre implémentation de `checkConflict` détecte les réservations portant sur le même objet (salle ou matériel) et dont les durées se chevauchent. Dans le cas de conflit, `resolve-Conflict` essaie d'effectuer une réservation équivalente et s'il ne réussit pas rejette la réservation effectuée en mode déconnecté.

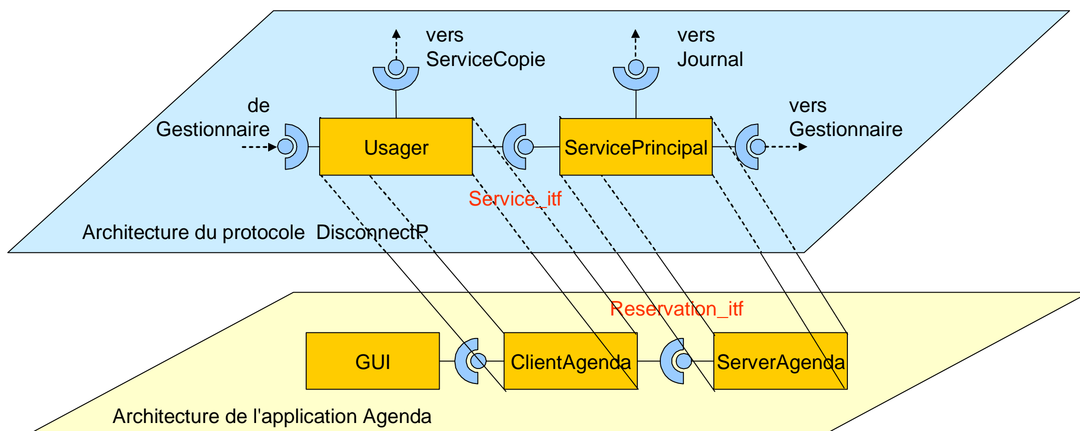


Figure 9-12. Relations d'architecture entre Agenda et DisconnectP

9.5.2 Performances

Comme dans le cas de la SHOPPINGLIST, les performances de l'assemblage dépendent de l'architecture de l'AGENDA. Avec un serveur modélisé en composant unique qui gère toutes les réservations, les étapes de déconnexion et de réconciliation posent les mêmes problèmes de taille de copie et de concurrence. Les performances peuvent être améliorées en modélisant le serveur de l'AGENDA comme un ensemble de composants qui permettent la copie partielle du serveur, ainsi que la gestion d'un grain plus fin de synchronisation.

9.6 Coût de FAR

FAR a été conçu avec des objectifs de souplesse et non de performances. Ses traitements sont implémentés de la manière la plus simple et sans optimisation. La structuration d'une application en termes de composants FAR, ainsi que sa configuration avec un protocole de duplication et de cohérence rajoute un coût en termes de temps d'exécution et de taille mémoire.

9.6.1 Configuration de test

Pour mesurer le coût de FAR nous avons considéré une application très simple constitué de deux instances A et B (Figure 9-13.). La première instance est de type `ComponentA` qui fournit une interface `Interface1_itf` et qui requiert une interface `Interface2_itf`. `ComponentA` ne définit pas d'attributs internes à part l'attribut représentant son interface requise. La deuxième instance est de type `ComponentB` qui fournit une seule interface `Interface2_itf`.

Les deux instances sont connectées à l'aide de l'interface Interface2_itf et communiquent de manière synchrone. Les deux interfaces, `Interface1_itf` et `Interface2_itf`, définissent respectivement les méthodes `m1_itf1()` et `m1_itf2()` qui ne rendent pas de résultat et qui ne prennent pas de paramètres.

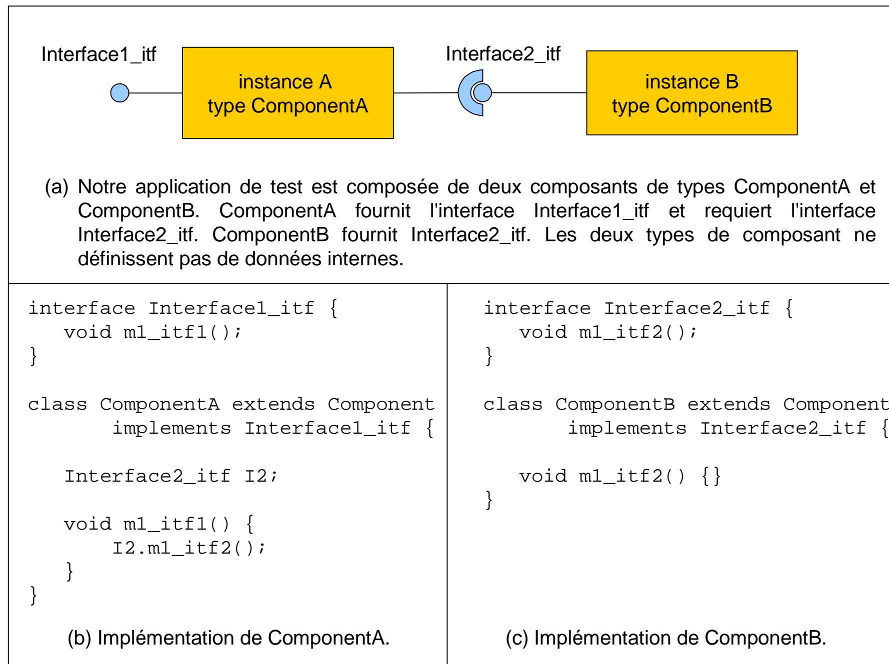


Figure 9-13. Application de test

Pour évaluer les performances de FAR en ce qui concerne le temps d'exécution, nous avons mesuré le temps d'un appel entre les deux instances A et B. Nous avons mesuré ce temps pour le cas de communication locale, où A et B s'exécutent dans un même serveur FAR (cf. Chapitre 5, Section 5.5.3, p. 109), ainsi que pour le cas de communication à distance où A et B s'exécutent dans deux serveurs lancés sur deux machines différentes.

Nous avons effectué le test pour un million d'appels et avons comparé FAR aux performances des appels Java standard et des appels Java à distance (Java RMI). Pour effectuer les mesures pour Java, nous avons implémenté deux versions de l'application de test. La première version définit deux classes A et B standard qui correspondent aux deux types de composants. La deuxième version définit B en tant qu'objet Java accessible à distance.

Les tests ont été effectués sur deux machines Athlon MP1800Hz, connectées par un réseau Fast Ethernet d'une bande passante de 100 mo/s. Les deux machines travaillent sous Linux et disposent de Java 1.4.1. Les mesures obtenues sont montrées à la Figure 9-14.

9.6.2 Coût des appels de FAR

Si on considère les mesures sur les appels dans FAR en local, dans le cas sans duplication, ni cohérence, nous constatons qu'ils sont environ cent cinquante fois plus lents que les appels Java standard. En effet, les appels Java sont optimisés dans les machines virtuelles Java, alors que les appels de FAR utilisent le mécanisme de FAR de communication à distance, optimisé pour le cas local. Par conséquent, la comparaison est plus significative entre FAR et Java RMI qui est le mécanisme de Java pour la communication à distance. Dans ce cas, FAR est plus effi-

cace dans le cas local (environ 150 fois plus rapide que Java RMI sur une machine) et plus lent dans le cas distant (4 fois plus lent que Java RMI). Ces résultats montrent que le coût de FAR dans les appels à distance est surtout du au mécanisme d'échange de messages entre serveurs qui n'est pas suffisamment optimisé.

Mesure	Temps	Commentaire
Java standard	7.10^{-9} s (7 ns)	
Java RMI "local" (A et B sur la même machine)	114.10^{-6} s (114 µs)	~ 10^5 fois plus lent que Java local
Java RMI (A et B sur deux machines)	215.10^{-6} s (215 µs)	~ 2 fois plus lent que RMI sur une machine
FAR local, sans protocole (A et B dans le même serveur)	$1{,}36.10^{-6}$ s (1,36 µs)	~ $1{,}5.10^2$ fois plus lent que Java local ~ $1{,}5.10^2$ fois plus rapide que Java RMI sur une machine
FAR distant, sans protocole (A et B sur deux machines)	830.10^{-6} s (830 µs)	~ 6.10^2 fois plus lent que FAR local ~ 4 fois plus lent que Java RMI
FAR local, avec protocole (A et B dans le même serveur)	$1{,}48.10^{-6}$ s (1,48 µs)	rajout de 9% par rapport à FAR local sans protocole
FAR distant, avec protocole (A et B sur deux machines)	840.10^{-6} s (840 µs)	rajout de 1% par rapport à FAR distant sans protocole

Figure 9-14. Coût d'un appel FAR

Nous avons également comparé les performances de FAR dans le cas d'appels applicatifs standard et d'appels interceptés par un protocole de duplication et de cohérence. Pour cela, nous avons défini deux composants de protocole, `ComponentA_Proto` et `ComponentB_Proto`, que nous avons respectivement attaché aux composants `ComponentA` et `ComponentB`. Ces deux composants ne rajoutent pas d'autres traitements à part l'interception des appels entre composants applicatifs. Les instances de protocole sont respectivement `AP` et `BP`.

La chaîne d'appel mesurée est la suivante (Figure 9-15.). Quand `A` appelle `B`, l'appel passe par le talon de `A` et est redirigé vers le talon de `AP`. Ce dernier appelle le squelette de `BP` qui appelle `BP`. Finalement, `BP` appelle `B`. Le retour prend le chemin inverse.

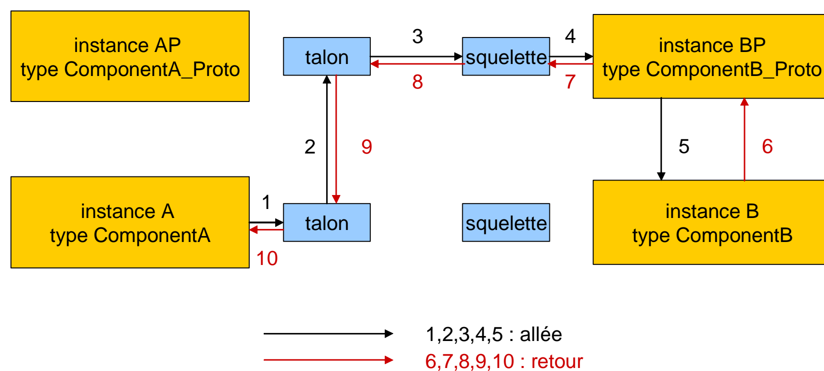


Figure 9-15. Chemin de l'appel de A à B dans le cas d'un protocole

Nos mesures montrent que, dans ce cas, le rajout de trois objets d'interception, notamment le talon de `BP`, le squelette de `BP` et `BP` lui-même, ralentit l'appel de 9% dans le cas local et seulement de 1% dans le cas distant. En d'autre termes, le coût d'indirection est perceptible dans le

cas local et nécessite des optimisations. Il est négligeable dans le cas distant comparé au coût de l'acheminement entre deux machines.

9.6.3 Taille des composants FAR

Étant donné que dans FAR nous introduisons une structure de conteneur qui contient de nombreuses données dont des tables de hachage pour les talons et les squelettes, nous avons voulu mesurer la taille d'un composant FAR. Pour cela, nous avons pris le composant de type `ComponentB` et l'avons modifié afin de pouvoir configurer son nombre d'interfaces fournies[1]. Ainsi, nous avons mesuré la taille d'une instance de `ComponentB` en variant son nombre d'interfaces de 1 à 20 ce qui correspond à une variation du nombre des squelettes correspondant. Nous avons également mesuré le cas où ComponentB et configuré avec un protocole qui rajoute un squelette de duplication et de cohérence par squelette applicatif. Les mesures sont montrées sur la Figure 9-16.

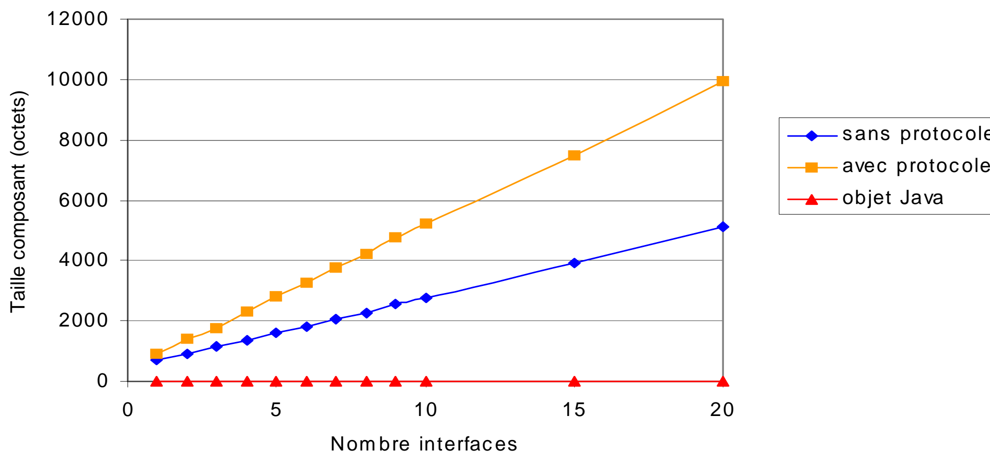


Figure 9-16. Taille d'un composant FAR

Ces mesures montrent que, alors que la taille de l'objet Java qui correspond au composant de type `ComponentA` et de 7 octets, la taille du composants sans duplication et cohérence et d'environ 700 octets. La situation est encore pire en rajoutant la gestion de la duplication et de la cohérence. Alors que dans le cas d'une interface, la taille du composant configuré avec le composant de protocole reste quasiment inchangée, dans le cas de 20 interfaces, cette taille est doublée.

L'amélioration des performances de FAR est une perspective de notre travail. Quelques pistes sont les techniques d'implémentation efficace des structures de composants et les techniques d'optimisation des chaînes d'appels à base d'objets d'interception. En effet, en ce qui concerne les structures efficaces de composants, FAR peut s'inspirer du modèle à composants proposé par l'ObjectWeb [98], Fractal [43], qui s'intéresse aux aspect d'administration. L'optimisation des chaînes d'objets d'interception peut utiliser, par exemple, les techniques d'injection de code qui transforment une séquence d'appels en un seul appel [55].

1. Les interfaces requises sont gérées par des structures qui sont symétriques à celles pour les interfaces fournies. En d'autres termes, la taille d'un objet dépend du nombre d'interfaces total et non du type de ces interfaces.

9.7 Synthèse

Dans ce chapitre nous avons montré que FAR répond aux objectifs que nous nous sommes fixés dans cette thèse. Nous avons évalué FAR par rapport aux points suivants :

- *Généralité*. Nous avons montré que FAR permet la mise en place de protocoles de duplication et de cohérence provenant de différents domaines d'application. Nous avons considéré un protocole de tolérance aux fautes, un protocole de gestion des déconnexions et un protocole de cohérence à l'entrée. Le protocole de tolérance aux fautes, détaillé en guise d'exemple dans les chapitres précédents, fait partie des protocoles indispensables garantissant la correction des calculs répartis. Le protocole de gestion de déconnexions est typiquement appliqué dans les environnements à usagers mobiles ou dans les environnements distribués à partitions. Le protocole de cohérence à l'entrée est utilisé dans les mémoires partagées réparties.

- *Non-intrusion*. En considérant l'exemple d'une application de gestion de courses et du protocole de gestion des déconnexions, nous avons montré que FAR permet l'intégration de la gestion de la duplication et de la cohérence dans les application métier sans modification du code de ces dernières. Toutefois, FAR exige l'instrumentation des applications métier avec des interfaces d'accès à l'état et impose un modèle de programmation. En outre, FAR ne garantit aucunement le fonctionnement optimal de l'assemblage obtenu après une intégration. En effet, les applications métier construites sans hypothèses sur la gestion de la duplication et de la cohérence se prêtent plus ou moins bien à un travail en mode dupliqué. Par conséquent, si l'assemblage d'une application et d'un protocole conçus indépendamment n'est pas satisfaisant, les applications métier doivent être modifiées afin de modéliser explicitement les besoins en duplication et en cohérence.

- *Souplesse*. Nous avons montré que FAR permet l'intégration de différents protocoles au sein d'une même application métier. Nous avons illustré ce point en configurant une application de gestion de listes de courses avec le protocole de gestion des déconnexions, ainsi qu'avec le protocole de cohérence à l'entrée. Pour chaque configuration, FAR se charge de la génération des composants de protocole spécifiques au fonctionnement de l'application métier, ainsi que de leur intégration. L'intégrateur de protocole fournit uniquement des descripteurs de correspondances entre les entités métier et les entités du protocole et définit la procédure de déploiement.

- *Réutilisation*. Nous avons montré que FAR permet la réutilisation des traitements de duplication et de cohérence au sein de différentes applications métier. Nous avons décrit le cas où le protocole de gestion des déconnexions est intégré dans une application de gestion de listes de courses, ainsi que dans une application de réservation. En outre, nous avons montré que FAR peut servir d'outil de prototypage et être utilisé pour intégrer un protocole dans différentes versions d'une même application métier. Ainsi, il peut aider à évaluer et à comparer les différentes configurations possibles de l'application métier et à choisir la configuration optimale.

- *Coût de FAR*. Nous avons constaté que la facilité de configuration et de réutilisation proposée par FAR est payée par des mécanismes de gestion qui alourdissent les applications. Comparé à Java, les appels entre composants sont lents et les composants consomment plus de mémoire. Par conséquent, si on veut utiliser FAR en dehors d'un contexte de prototypage, il est nécessaire de le réimplémenter de manière optimisée.

Conclusion

Objectifs de thèse

Cette thèse porte sur la gestion configurable de la duplication et de la cohérence dans les applications réparties à base de composants. Nos objectifs ont été de pouvoir configurer les applications avec différents protocoles de duplication et de cohérence, ainsi que de pouvoir réutiliser les protocoles au sein de différentes applications. La configuration des applications, ainsi que la réutilisation des protocoles devaient se faire sans modification de leur code.

Notre motivation principale pour travailler sur ce problème a été la diversification des plates-formes matérielles et logicielles, amenée par les avancées importantes des technologies et des réseaux. En effet, en permettant aux applications d'être configurées avec différents protocoles, il est possible d'exécuter ces applications dans différents environnements et de satisfaire différentes exigences de disponibilité. En permettant aux protocoles d'être réutilisés, le processus de construction de nouveaux protocoles pour les environnements émergents est facilité et accéléré. Enfin, la minimisation des modifications nécessaires à la configuration des applications et à la réutilisation des protocoles réduit les coûts de développement.

Approche proposée

Pour fournir une gestion configurable de la duplication et de la cohérence, nous nous sommes basés sur les principes de gestion de composants et sur les principes de la programmation par aspects.

Nous avons structuré les applications en termes de composants afin de bénéficier de la possibilité de configurer leur gestion des services système sans modifier leur code métier. En effet, la gestion séparée du code métier et du code non fonctionnel des composants permet que les applications soient programmées sans considérer les besoins en duplication et en cohérence.

L'originalité de notre approche consiste à également modéliser les protocoles de duplication et de cohérence en tant qu'applications à base de composants. Le composant est utilisé comme unité de duplication et de cohérence ce qui permet la programmation de protocoles indépendants des domaines d'application. Les protocoles sont programmés de manière séparée des applications et peuvent donc être réutilisés dans différentes applications. Leurs composants peuvent être réutilisés pour la construction de nouveaux protocoles.

Réalisation

Nous avons implémenté un prototype en Java, nommé FAR, qui met en œuvre les principes proposés. FAR définit un environnement de construction et de gestion d'applications métier (sans duplication ni cohérence) et de protocoles de duplication et de cohérence. Plus particuliè-

rement, il fournit un modèle de programmation pour les applications et les protocoles, un modèle pour leur composition, ainsi qu'un environnement d'exécution.

- *Modèle de programmation*. Nous avons défini un modèle à composants pour la programmation d'applications sans duplication ni cohérence. Ce modèle permet la description des types des composants et de leurs assemblages. Nous avons également défini un modèle de programmation pour les protocoles de duplication et de cohérence. Ces derniers sont modélisés à l'aide du modèle à composants utilisé pour les applications qui a été spécialisé pour refléter uniquement les aspects de gestion de la duplication et de la cohérence.
- *Modèle de composition*. Nous avons défini un processus d'intégration des protocoles de duplication et de cohérence au sein des applications. L'intégration est faite lors du déploiement des applications ce qui permet une configuration de protocole en fonction de l'environnement concret d'exécution. Elle suit la logique des systèmes réflexifs et est basée sur la définition de relations de correspondance entre les entités des applications et les entités des protocoles.
- *Environnement d'exécution*. Nous avons fourni un environnement d'exécution pour les applications et les protocoles. Inspiré par les plates-formes existantes, telles que CCM et EJB, il est structuré en termes de serveurs, de conteneurs et de composants. Les serveurs fournissent les service systèmes nécessaires à la bonne exécution des composants. Les composants sont des entités d'encapsulation et de réutilisation de fonctionnalités. Finalement, les conteneurs sont des encapsulateurs qui font le lien entre les fonctionnalités des composants et les services système des serveurs.

Évaluation

Nous avons évalué FAR de trois manières différentes. Nous avons évalué ses résultats par rapport aux objectifs de configuration et de réutilisation, nous avons exploré l'applicabilité des ses principes dans les environnements standards CCM et EJB et, finalement, nous avons évalué le coût des structures de FAR.

Objectifs de configuration et de réutilisation.

Afin d'évaluer les prestations de FAR par rapport à nos objectifs principaux, nous avons effectué plusieurs expérimentations avec différentes applications et différents protocoles. Nous avons travaillé avec un protocole de gestion de cache, un protocole de tolérance aux pannes, ainsi qu'un protocole de gestion de déconnexions. En ce qui concerne les applications, nous avons considéré une application de gestion de réservations, une de gestion de listes de courses et une de gestion de comptes bancaires.

Nous avons organisé nos expérimentations autour des quatre points suivants :

- *Généralité*. Nous avons montré que FAR permet la modélisation de protocoles venant de différents domaines en implantant les trois protocoles énumérés précédemment.
- *Souplesse*. Nous avons montré que FAR permet l'utilisation de différents protocoles au sein d'une même application.
- *Réutilisation*. Nous avons montré que FAR permet la réutilisation d'un même protocole au sein de différentes applications.

- *Non intrusion*. Nous avons montré que FOR propose un processus d'intégration d'un protocole au sein d'une application qui demande très peu de modifications. Toutefois, il ne garantit pas que l'assemblage qui en résulte est optimal en ce qui concerne la gestion de la duplication et de la cohérence de l'application.

Application dans des environnements standards

En plus des expérimentations ciblées sur FAR, nous avons travaillé avec les environnements CCM et EJB. Le but de ce travail a été d'analyser l'applicabilité des principes de FAR dans ces environnements. Nous avons comparé le modèle de gestion de FAR aux modèles de CCM et EJB et avons identifié les caractéristiques qui influencent l'intégration de la duplication et de la cohérence dans ces environnements.

- *CCM*. Nous avons constaté que le modèle de CCM facilite l'intégration de la duplication et de la cohérence à cause de sa gestion explicite de l'architecture des applications. Toutefois, il présente également deux grandes difficultés. D'une part, la spécification ne définit pas explicitement la structure des conteneurs des composants et de ce fait rend la définition du processus d'intégration des protocoles spécifique aux implémentations. D'autre part, les applications CCM peuvent faire usage des transactions qui sont fortement liées à la notion de correction d'exécution et qui donc interfèrent avec la gestion des protocoles de duplication et de cohérence.

- *EJB*. EJB a les caractéristiques contraires de CCM : il ne considère pas explicitement l'architecture des applications mais définit en détail la structure et l'organisation des traitements non fonctionnels des conteneurs des composants. Les applications EJB peuvent également faire usage des transactions.

Après l'analyse des spécifications CCM et EJB, nous avons proposé deux mises en œuvre de gestion de la duplication et de la cohérence pour ces deux environnements. Dans ces propositions nous avons tenté de préserver au maximum la logique des modèles CCM et EJB tout en les enrichissant afin de les rapprocher au modèle de FAR. En ce qui concerne le problème des transactions, nous avons proposé une modélisation explicite au niveau des protocoles. Nous avons partiellement implanté ces propositions dans OpenCCM et JOnAS qui sont deux plates-formes implémentant respectivement CCM et EJB et qui sont proposées en source libre par le consortium ObjectWeb.

En résumé, l'expérimentation avec EJB et CCM a eu deux résultats majeurs. D'une part, nous avons montré que l'application des principes de FAR est faisable mais qu'elle ne peut être faite sans prendre en compte les autres propriétés non fonctionnelles gérées par ces environnements. D'autre part, nous avons identifié les caractéristiques qui facilitent et celles qui rendent difficile la mise en place d'une gestion de la duplication et de la cohérence dans CCM et EJB.

Coût de FAR

FAR a été conçu avec des objectifs de configuration et de facilité de gestion et non de performances. Nous avons, par conséquent, choisi une stratégie d'implémentation où les mécanismes sont simples et sans optimisation. Ainsi, la structuration d'une application en composants a des répercussions aussi bien sur le temps des appels entre composants que sur la taille mémoire nécessaire à cette application. La dégradation des performances est encore plus grande dans le cas d'une gestion de protocole qui rajoute des composants et une déviation d'appels pour passer par ces composants. Toutefois, FAR peut être utilisé à des fins de prototypage en vue de la comparaison de différentes configurations entre applications et protocoles.

Perspectives

Ce travail est un des premiers efforts sur la gestion de la duplication et de la cohérence dans le domaine composant et de ce fait reste encore très incomplet. En effet, dans le réalisation de FAR nous avons ignoré l'objectif de performances et n'avons pas considéré les relations entre la gestion de la duplication et de la cohérence avec d'autres services systèmes tels que les transactions ou la persistance. Ces deux points définissent les deux grands axes d'études qui doivent compléter notre travail.

- *Amélioration de performances*. L'amélioration des performances des applications est un des objectif principaux de la duplication et de la cohérence. Améliorer les performances de FAR est, par conséquent, une condition nécessaire pour que l'outil passe d'un usage de prototypage à un usage plus généralisé. Pour une telle amélioration nous pouvons nous inspirer des travaux prêtant une attention particulière à l'implémentation efficace des composants, ainsi que des travaux portant sur l'optimisation de chaînes d'indirection pour les appels entre composants

- *Duplication, cohérence et autres services système*. En considérant les environnements EJB et CCM nous n'avons qu'effleuré le problème des relations entre la gestion de la duplication et de la cohérence et la gestion d'autres services système. Ce point fait partie du problème général de gestion de plusieurs propriétés non fonctionnelles au sein des conteneurs des composants. En effet, une bonne compréhension des relations entre les gestions des services système permettrait la génération de conteneurs efficaces et faciliterait la configuration spécifique des services système utilisés par les applications.

Bibliographie

[1] ACHARYA S., ZDONIK S. B. An Efficient Scheme for Dynamic Data Replication. CS-93-43, Department of Computer Science, Brown University, 1993.
Disponible sur : http://citeseer.nj.nec.com

[2] ADVE S., GHARACHORLOO K. Shared Memory Consistency Models: A Tutorial. WRL-95/7. Western Research Laboratory, DEC, 1995, p. 27.
Disponible sur : http://rsim.cs.uiuc.edu/~sadve/#Publications

[3] ALBITZ P., LIU C. DNS and BIND. O'Reilly & Associates, Sebastopol, CA, USA. ISBN 0-596-00158-4

[4] ALLEN R., GARLAN D. Formal Connectors. CMU Tech. Report CMU-CS-94-115, School of Computer Science, Carnegie Mellon University, Pittsburgh, PA 15213, USA, March 1994.

[5] AMADE E., BAKKER A., KUZ I. ET VERKAIK P. Globe Operations Guide : Installation and Usage. Department of Mathematics and Computer Science, Vrije University, 2002.
Disponible sur : http://www.cs.vu.nl/~steen/globe/

[6] BABAOGLU O., BARTOLI A., DINI G. Replicated File Management in Large-Scale Distributed Systems. In : Distributed Algorithms (WDAG8), LNCS 857, Springer Verlag, 1994, pp. 1-6.

[7] BAL H., BHOEDJANG R., HOFMAN R. ET AL., Performance Evaluation of the Orca Shared-Object System, ACM Trans. on Computer Systems, Vol. 16 No. 1, pp.1-40, Fevrier 1998.
Disponible sur : http://citeseer.nj.nec.com (consulté le 27.03.2003)

[8] BALASUBRAMANIAM S., PIERCE B.C. What is a File Synchronizer?. In : Fourth Annual ACM/IEEE International Conference on Mobile Computing and Networking (MobiCom '98). 1998.
Disponible sur : www.cis.upenn.edu/~bcpierce/papers

[9] BARATLOO A., CHUNG R.E., HUANG Y. et al. Filterfresh: Hot Replication of Java RMI Server Objects. In : Proceedings of the 4th Conference on Object-Oriented Technologies and Systems (COOTS). Santa Fe, New Mexico. 1998.

[10] BAUMANN J., HOHL F., STRABER M. et al. Mole - Concepts of Mobile agent System. In : WWW Journal, Special Issue on Applications and Techniques of Web Agents, Vol. 1, No. 3, 1998.
Disponible sur : http://www/mole.informatik.uni-stuttgart.de

[11] BCEL [en ligne]
Disponible sur : http://jakarta.apache.org/bcel/ (consulté le 27.03.2003)

[12] BEA Systems - BEA WebLogic Server [en ligne]
http://www.bea.com/products/weblogic/server (consulté le 27.03.2003)

[13] BIRMAN K. P. The Process Group Approach to Reliable Distributed Computing. Communications of the ACM, Vol. 36, No. 12, pp. 37-53, 1993.

[14] BIRMAN K. P. Building Secure and Reliable Network Applications. Prentice Hall, NJ, 1996.

[15] BOUCHENAK S. Mobilité et persistance des applications dans l'environnement Java. Thèse Informatique.
Grenoble : Institut National Polutechnique de Grenoble, 2001. 200 p.

[16] BRISCO T. DNS Support for Load Balancing, April 1995.
Network Working Group RFC 1794.

[17] BRUN-COTTAN G. Cohérence de données répliquées partagées par un groupe de processus coopérant à distance. Thèse Systèmes Informatiques. Paris : Université Pierre & Marie Curie - Paris VI, 1998, 157 p.

[18] BRUNETON E. Un support d'exécution pour l'adaptation des aspects non-fonctionnels des applications réparties. Thèse Informatique. Grenoble : Institut National Polytechnique de Grenoble, 2001. 132 p.

[19] CARTER J.B. Design of the Munin Distributed Shared Memory System. The Journal of Parallel and Distributed Computing, Vol. 11, No. 6, 1995, pp. 219-227.

[20] CARTER J.B., BENNETT J.K., ZWAENEPOEL W. Techniques for Reducing Consistency-Related Communication in Distributed Shared Memory Systems. ACM Transactions on Computer Systems, Vol. 13, No. 3, 1995, pp. 205-243.

[21] CARTER J., RANGANATHAN A., SUSARLA S. Khazana: An Infrastructure for Building Distributed Services. In : the Proceedings of the 18th Annual International Conference on Distributed Computing Systems, 1998, pp. 562-571.
Disponibble sur : http://www.cs.utah.edu/khazana/ (consulté le 27.03.2003)

[22] CASE J.D. , FEDOR M. , SCHOFFSTALL M.L. et al. Simple Network Management Protocol (SNMP). The Internet Engineering Task Force, May 1990. Request for Comments 1157.

[23] CECCHET E. Whoops! : a Clustered Web Cache for DSM Systems using Memory Mapped Networks. In : IEEE International Workshop on Web Caching Systems in Proceedings of ICDCS'02, Vienne, Autriche, 2002.

[24] CEDERQVIST P. et al. Version Management with CVS [en ligne]. 2001.
Disponible sur : http://www.cvshome.org/docs/manual (consulté le 27.03.2003)

[25] CERI ST., HOUTSMA M., KELLER A., et al. A Classification of Update Methods for Replicated Databases, STAN-CS-91-1932, Stanford University, 1991.

[26] CESURE (Configuration et Exécution de Service pour Usagers mobiles des Réseaux Etendus). Projet subventionné par le Réseau National de Recherche en Télécommunications (RNRT). Nov. 1999 - Nov. 2001.[en ligne]
Disponible sur : http://www.gemplus.com/smart/cesure/ (consulté le 27.03.2003)

[27] CHANDRA S., DAHLIN M., RICHARDS B. et al. Experience with a Language for Writing Coherence Protocols. In : Proceedings of the 1st USENIX Conference on Domain-Specific Languages, Santa Barbara, California, Oct. 1997.
Disponible sur : http://citeseer.nj.nec.com/ (consulté le 27.03.2003)

[28] CHANSO S.T., NEUFELD G.W., LIANG L. A Bibliography on Multicast and Group Communication. In : ACM Operating Systems Review, Vol. 23, No. 4, 1989, pp.20-25.

[29] CHEN S-W., PU C. A Structural Classification of Integrated Replica Control Mechanisms. CUCS-006-92. New York : Columbia University. 1992. 25 p.

[30] CHOCKLER , DOLEV D., FRIEDMAN R. et al. Implementing a Caching Service for Distributed CORBA Objects. In : Proceedings of Middleware'00. 2000. pp. 1-23.

[31] CHOCKLER G., KEIDAR I. et VITENBERG R. Group Communication Specifications: A Comprehensive Study. ACM Computing Surveys (CSUR), Vol. 33, No. 4, Dec. 2001.

[32] CORBA & CORBA Component Model (CCM) [en ligne]
Disponible sur : http://ditec.um.es/~dsevilla/ccm/ (consulté le 27.03.2003)

[33] DANIELS D., BOON DOO L., DOWNING A. et al. Oracle's Symmetric Replication Technology and Implications for Application Design. In : Proceedings of ACM SIGMOD International Conference on Management of Data, page 467, Minneapolis, MN, May 1994.

[34] DAVIDSON S., GARCIA-MOLINA H., SKEEN D. Consistency In a Partitioned Network: A Survey. ACM Computing Surveys. Vol. 17, No. 3, 1985, pp. 341 - 370

[35] DASGUPTA P., CHEN R.C. , MENON S. et al. The Design and Implementation of the Clouds Distributed Operating System, Computing Systems Journal, Vol. 3, No. 1, pp. 11-46, 1990.

[36] DÉCHAMBOUX P., HAGIMONT D., MOSSIÈRE J. et al. The Arias Distributed Shared Memory: An Overview, In : SOFSEM'96: Theory and Practice of Informatics, K. Jeffery, J. Kral, M. Bartosek (éditeurs), LNCS 1175, Springer, 1996.

[37] DÉDIEU O., PACITTI E. Optimistic Replication for Collaborative Applications on the Web. In : International Workshop on information Intégration on the Web - Technology and Application. Brazil.2001.

[38] DELP G., SETHI A. ET FARBER D. An Analysis of Memnet: An Experiment in High-Speed Shared-Memory Local Networking. In : ACM Symposium on Communications Architectures and Protocols, 1988.

[39] DEMERS A., GREENE D., HAUSER C. et al. Epidemic Algorithmes for Replicated Database Maintenance. In : 6th Symposium on Principles of Distributed Computing (PODC). Vancouver, Canada, 1987, pp. 1-12.

[40] DE PALMA N. Services d'administration d'applications réparties. Thèse Informatique, Système et Communication. Grenoble : Universite Joseph Fourier, 2001, 200 p.

[41] EJB-SIG :: Entreprise JavaBeans Special Interest Group [en ligne].
Disponible sur : http://www.ejbsig.com/ (consulté le 27.03.2003)

[42] Enterprise Java CORBA Component Model [en ligne]
Disponible sur : http://www.cpi.com/ejccm/ (consulté le 27.03.2003)

[43] Fractal Home Page [en ligne]
Disponible sur : http://www.objectweb.org/fractal/index.html (consulté le 27.03.2003)

[44] FRITZKE U. Les systèmes transactionnels répartis pour données dupliquées fondés sur la communication de groupes. Thèse Informatique. Rennes : Université de Rennes, 2001, 179 p.

[45] FROESE K.W., BUNT R.B. Cache Management for Mobile File Service. In : Computer Journal. Vol. 42, No.6. 1999. pp. 442-454.

[46] GARBINATO B., GUERRAOUI R. Flexible Protocol Composition in BAST. In : The 18th International Conference on Distributed Computing Systems (ICDCS), Mai, 1998 Amsterdam, Pays Bas.

[47] GONCALVES T., SILVA A.R. Passive Replicator: A Design Pattern for Object Replication. In : The 2nd European Conference on Pattern Languages of Programming (EuroPLoP '97). pp. 165-178. 1997.

[48] GRAY C., CHERITON D. Leases: An Efficient Fault-Tolerant Mechanism for Distributed File Cache Consistency. In : Proceedings of the 12th ACM Symposium on Operating System Principles (SOSP), December 1989. pp. 202-210.

[49] GRAY J., HELLAND P., P. E. O'NEIL, et al. The Dangers of Replication and a Solution. In : ACM SIGMOD International Conference on Management of Data, pp. 173-182, June 1996.
Disponible sur : http://citeseer.nj.nec.com/ (consulté le 27.03.2003)

[50] GRAY J., REUTER A. Transactional Processing: Concepts and Techniques. Morgan Kaufmann (Ed.), San Mateo, CA, USA. 1993.

[51] GUERRAOUI R., GARBINATO B., MAZOUNI K. GARF: A Tool for Programming Reliable Distributed Applications. IEEE Concurrency, Vol. 5, No. 4, 1997, pp. 32-39.

[52] GUERRAOUI R., SCHIPER A. Software-Based Replication for Fault-Tolerant Systems. IEEE Computer, Vol.30, No. 4, 1997, pp. 68-74.

[53] GUYENNET H., LAPAYRE J-C., TRÉHEL M. The Pilgrim: A New Consistency Protocol for Distributed Shared Memory. In : Proc. of IEEE 3rd Int'l Conf. on Algorithms & Architectures for Parallel Processing (ICA3PP'97), 1997.

[54] HADZILACOS V., TOUEG S. Fault-Tolerant Broadcast and Related Problems. In : Mullender S. Distributed systems. 2nd ed. Mullender, Sape. Ed. Wokingham, GB : Addison-Wesley, 1993. (ACM Press frontier series) 0-201-62427-3

[55] HAGIMONT D. ET DE PALMA N. Removing Indirection Objects for Non-Functional Properties. In : Proc. of PDPTA 2002, International Conference on Parallel and Distributed Processing Techniques and Applications, Las Vegas (USA), June 2002.

[56] HAMILTON G., MITCHELL G.J., POWELL M.L. Subcontract: A Flexible Base for Distributed Programming. In : ACM Symposium on Operating Systems Principles. 1993. pp. 69-79.
Disponible sur : http://citeseer.nj.nec.com (consulté le 27.03.2003)

[57] HAYTON R., ANSA TEAM. FlexiNet Architecture. In Eds. Citrix Systems (Cambridge) Limited. Architecture raport.1999.

[58] HOLDER O., BEN-SHAUL I..,GAZIT H. System Support for Dynamic Layout of Distributed Applications. In : 19th International Conference on Distributed Computing Systems (ICDCS'99), Austin, TX : USA, 1999, pp. 403-411.

[59] HOWARD, J.H. An Overview of the Andrew File System. In : USENIX Winter Technical Conference. Dallas, TX. 1988.

[60] HUSTON L.B., HONEYMAN P. Disconnected Operation for AFS. In : USENIX Mobile and Location-Independent Computing Symposium, Cambridge, MA, 1993.

[61] HUSTON L. B., HONEYMAN P. Partially Connected Operation. In : Symposium on Mobile and Location-Independent Computing. 1995. pp. 91-98

[62] iCMG :Software for CORBA Components [en ligne]
Disponible sur : http://www.icmgworld.com/ (consulté le 27.03.2003)

[63] IBM WebSphere - middleware, application server, e-business, infrastructure software [en ligne] Disponible sur : http://www.ibm.com/websphere (consulté le 27.03.2003)

[64] ISSARNY V., SARIDAKIS T. et ZARRAS A. A Survey of Architecture Description Languages, C3DS Deliverable A3.1, ESPRIT LTR Project N°24962, pages 8-28, 1998

[65] JAHANIAN F., FAKHOURI S. , RAJKUMAR R. Processor Group Membership Protocols: Specification, Design, and Implementation. In : Symposium on Reliable Distributed Systems, Princeton, NJ, 1993.

[66] Java(TM) SE Platform Documentation [en ligne]
Disponible sur : http://java.sun.com/docs/index.html (consulté le 27.03.2003)

[67] JBoss :: All your J2EE are belong to us. [en ligne]
Disponible sur : http://www.JBoss.org. (consulté le 27.03.2003)

[68] JNDI 1.2.1 Specification [en ligne]
Disponible sur : http://java.sun.com/products/jndi/1.2/javadoc/
(consulté le 27.03.2003)

[69] JOnAS Home Page [en ligne]
Disponible sur : http://www.objectweb.org/jonas/index.html (consulté le 27.03.2003)

[70] JOSEPH A.D., TAUBER J. A., KAASHOEK M. F. Mobile Computing with the Rover Toolkit. IEEE Transactions on Computers. Special Issue on Mobile Computing. Vol.

46, No. 3, pp.337-352. 1997.
Disponible sur : http://www.pdos.lcs.mit.edu/rover/ (consulté le 27.03.2003)

[71] JUDGE A., NIXON P., CAHILL V. et al. Overview of Distributed Shared Memory. TCD-CS-1998-24, Trinity College Dublin. 1998. p.44.
Disponible sur : http://www.cs.tcd.ie/publications/tech-reports/tr-index.98.html

[72] KARAMANOLIS C., MAGEE J. Client-Access Protocols for Replicated Services. In : IEEE Transactions on Software Engineering, Vol. 25, No.1, 1999.

[73] KELLY T. Thin-client Web Access Patterns: Measurements from a Cache-busting Proxy. In : Computer Communications, Vol.25, No.4, pp.357-366, 2002.
Disponible sur : http://citeseer.nj.nec.com (consulté le 27.03.2003)

[74] KEMME B., ALONSO G. A New Approach to Developing and Implementing Eager Database Replication Protocols. ACM TODS, Vol.25, No.3, pp.333-379, 2000.
Disponible sur : http://citeseer.nj.nec.com (consulté le 27.03.2003)

[75] KERMARREC A.M., ROWSTRON A., SHAPIRO M. et al. The IceCube Approach to the Reconciliation of Divergent Replicas. In : ACM Symposium on Principles of Distributed Computing (PODC), Newport, RI USA, 2001.

[76] KIM M., COX L. P., NOBLE B. D. Safety, Visibility, and Performance in a Wide-Area File System. In : USENIX Conference on File and Storage Technologies, Monterey, CA.2002.

[77] KICZALES G., HISDALE E., HUGUNIN J. et al. An overview of AspectJ. LNCS 2072, pp. 327-353. Springer-Verlag, 2001.

[78] KICZALES G., LAMPING J., MENDHEKAR A. et al. Aspect-Oriented Programming. In : European Conference on Object-Oriented Programming (ECOOP), LNCS 1241. Springer, 1997

[79] KLERER S.M. The OSI Management Architecture: an Overview. IEEE Network, Vol. 2, No. 2, 1998, pp. 20-29.

[80] KON F., COSTA F., BLAIR G. et al. The Case for Reflective Middleware. Communications of the ACM, Vol. 45 No. 6, pp.33-38, 2002. ACM Press.

[81] KOPETZ H., DAMM A., KOZA CH. et al. Distributed Fault-Tolerant Real-Time Systems: The MARS Approach. IEEE Micro. 1989. pp. 25-50.

[82] KRAMER J., MAGEE J., SLOMAN M., Constructing Distributed Systems in CONIC. In : IEEE Trans. Software Engineering, Vol.SE-15, No. 6, 1989, pp.663-675.

[83] KUENNING G. H., POPEK G. J. Automated Hoarding for Mobile Computers. In : Proc. of the Sixteenth ACM Symposium on Operating Systems Principles (SOSP), pp. 264-275. 1997.

[84] LAMPORT L. Time, Clocks and the Ordering of Events in a Distributed System. Communications of the ACM, Vol.21, No.7, 1978, pp.125-133.

[85] LAMPORT L. How to Make Multiprocessor Computer that Correctly Executes Multiprocess Programs. IEEE Transanctions Computers, Vol.C-28, no.9, p.690-1. 1979

[86] LEI H., DUCHAMP D. An Analytical Approach to File Prefetching. In : Proc. of the USENIX Annual Technical Conference. Anaheim CA, 1997.
Disponible sur : http://citeseer.nj.nec.com (consulté le 27.03.2003)

[87] LI Q, MOON B. Distributed Cooperative Apache Web Server. In : Proc. of the 10th World Wide Web Conference, Hong Kong, May 2001.

[88] LI Q., RUS D.. InterMezzo: File Synchronization with InterSync. In : Proc. of the 6th Annual ACM/IEEE International Conference on Mobile Computing (MOBICOM'00), Boston, MA. 2000.

[89] LISKOV B., GHEMAWAT S., GRUBER R. et al. Replication in the Harp File System. In : Proc. of the 13th ACM Symposium on Operating System Principles (SOSP), Pacific Grove, CA. 1991. pp. 226-238.

[90] Mark C. Little, Santosh K. Shrivastava, Integrating Group Communication with Transactions for Implementing Persistent Replicated Objects, LNCS, Vol. 1752, Springer-Verlag, pp. 238-253

[91] LUCKHAM D. C., VERA J. An Event-Based Architecture Definition Language, IEEE Trans. Software Engineering, Vol. 21, No. 9, 1995, pp.717-734

[92] MARCHETTI C., MECELLA M., VIRGILLITO A. et al. An Interoperable Replication Logic for CORBA Systems. In : Proc. of the 2nd International Symposium of Distributed Object Applications. Antwerp, Belgium. 2000.

[93] MEDVIDOVIC, N., TAYLOR, R. N. A Classification and Comparison Framework for Software Architecture Description Languages IEEE Transactions on Software Engineering, Vol. 26, pp. 70-93, 2000.

[94] MICROSOFT. ActiveSync Software. Microsoft 2002 [en ligne].
Disponible sur : www.microsoft.com/mobile

[95] Microsoft. .NET [en ligne]
Disponible sur : http://www.microsoft.com/net/ (consulté le 27.03.2003)

[96] MULLENDER S. Distributed systems. 2nd ed. Mullender, Sape. Ed. Wokingham, GB : Addison-Wesley, 1993. (ACM Press frontier series) 0-201-62427-3

[97] NEUMAN B. Scale in Distributed Systems. In : Readings in Distributed Computing Systems, IEEE Computer Society Press, 1994
Disponible sur : http://www.isi.edu/people/bcn/publications.html
(consulté le 27.03.2003)

[98] ObjectWeb Consortium. Home Page [en ligne]
Disponible sur : http://www.objectweb.org (consulté le 27.03.2003)

[99] ODP Reference Model : Architecture
ITU-T Recommandation X.903 | ISO/IEC International Standard 10746-3, 1995.
Disponible sur : http://www.dstc.edu.au/Research/Projects/ODP/standards.html

[100] OLIVA A., BUZATO L.E. An overview of MOLDS: a Meta-Object Library for Distributed Systems. Technical report IC-98-15, Universidade Estadual de Campinas, 1998.
Disponible sur : http://citeseer.nj.nec.com (consulté le 27.03.2003)

[101] OMG. CORBA™/IIOP™ Specification. formal/02-06-01. OMG. 2002
Disponible sur : http://www.omg.org/technology/documents/formal/corba_iiop.htm

[102] OMG. CORBA Components 3.0. formal/02-06-65. OMG. 2002.
Disponible sur : http://www.omg.org/technology/corba/corba3releaseinfo.htm

[103] OMG. Persistent State Service Specification 2.0. OMG 2002.[en ligne]
Disponible sur : http://www.omg.org/docs/formal/ (consulté le 27.03.2003)

[104] OpenCCM Home Page [en ligne]
Disponible sur : http://www.objectweb.org/openccm/index.html
(consulté le 27.03.2003)

[105] Oracle9iAS Containers for J2EE [en ligne]
Disponible sur : http://otn.oracle.com/tech/java/oc4j/content.html
(consulté le 27.03.2003)

[106] OSKIEWICZ E., EDWARDS N. A Model for Interface Groups. APM.1002.01. ANSA. 1994.

[107] PAULO J., ALMEIDA A., WEGDAM M. et al. Transparent Dynamic Reconfiguration for CORBA. In : Proc. of the 3rd International Symposium on Distributed Objects & Applications (DOA 2001), Italy, 2001.
Disponible sur : http://wwwhome.cs.utwente.nl/~alme/cvitae/ (consulté le 27.03.2003)

[108] PAWLAK R., SEINTURIER L., DUCHIEN L. et al. JAC : A Flexible Framework for AOP in Java. In : Proc. of the 3d International Conference on Metalevel Architectures and Separation of Crosscutting Concerns (Reflection'01), Kyoto, Japan. 2001.
Disponible sur : http://jac.aopsys.com/ (consulté le 27.03.2003)

[109] PETERSEN K., SPREITZER M. J., TERRY D. B. et al. Flexible Update Propagation for Weakly Consistent Replication. In : Proc. of the 16th ACM Symposium on Operating Systems Principles (SOSP-16), Saint Malo, France, 1997, pp. 288-301.
Disponible sur : http://www2.parc.com/csl/projects/bayou/ (consulté le 27.03.2003)

[110] PIERRE G., STEEN M., TANENBAUM A. Dynamically Selecting Optimal Distribution Strategies for Web Documents. IEEE Transactions on Computers, Vol. 51, No.6, 2002

[111] PODNAR I., HAUSWIRTH M., JAZAYERI M. Mobile Push: Delivering Content to Mobile Users. In : International Workshop on Distributed Event-Based Systems, 2002.
Disponible sur : http://www.opelix.org/index2.shtml (consulté le 27.03.2003)

[112] POWELL D. Distributed Fault Tolerance - Lessons Learnt from Delta-4.
IEEE Micro, Vol. 14, No. 1, pp.36-47, 1994.

[113] PLASIL F., BALEK D., JANECEK R., SOFA/DCUP: Architecture for Component Trading and Dynamic Updating, In : Proc. of the 4th Intertionnal Conference of Configurable Distributed Systems (ICCDS'98), 1998, pp. 43-51.

[114] RABINOVICH M., RABINOVICH I., RAJARAMAN R. et al.. A Dynamic Object Replication and Migration Protocol for an Internet Hosting Service. In : Proc. of the IEEE International Conference on Distributed Computing Systems, pages 101-113, May 1999.

[115] RADOSLAVOV P., GOVINDAN R., ESTRIN D. Topology-Informed Internet Replica Placement. Computer Communications, Vol. 25, No. 4, pp.384-392, 2002.
Disponible sur : http://citeseer.nj.nec.com (consulté le 27.03.2003)

[116] RAMSEY N., CSIRMAZ E. An Algebraic Approach to File Synchronization In 9th Int. Symp. on the Foundations of Softw. Eng. (FSE) (Austria, Sept. 2001).
Disponible sur : http://www.eecs.harvard.edu/~nr/pubs/ (consulté le 27.03.2003)

[117] RATNER D., POPEK G.J., REIHER P. Peer replication with Selective Control. CSD-960031. University of California : Los Angeles. 1996.

[118] RATNER D., REIHER,P., POPEK G.J. Roam: A Scalable Replication System for Mobile Computing. In : Workshop on Mobility In Databases and Distributed Systems @ DEXA'99, September 1999.

[119] RAVERDY P-G., LEA R. DART: A Distributed Adaptive Run-Time. In : Proc of Middleware'98, Lake District, England. 1998.

[120] RIPEANU M., FOSTER I. A Decentralized, Adaptive Replica Location Mechanism, IEEE 2002

[121] ROWSTRON A., DRUSCHEL P. Storage Management and Caching in PAST, a Large-Scale, Persistent Peer-to-Peer Storage Utility, 18th ACM SOSP'01, Lake Louise, Alberta : Canada. 2001.
Disponible sur : http://www.research.microsoft.com/~antr/PAST/
(consulté le 27.03.2003)

[122] SAITO Y., SHAPIRO M. Replication: Optimistic Approaches. HPL-2002-33. Palo Alto, CA, HP Labs. 2002. pp. 40.
Disponible sur : http://www.hpl.hp.com/personal/Yasushi_Saito/survey.pdf
(consulté le 27.03.2003)

[123] SATYANARAYANAN, M. Mobile Information Access. IEEE Personal Communications, Vol. 3, No. 1, 1996.
Disponible sur : http://www-2.cs.cmu.edu/afs/cs/project/coda/Web/docs-coda.html
(consulté le 27.03.2003)

[124] SATYANARAYANAN M., The Evolution of Coda. ACM Transactions on Computer Systems, Vol. 20, No. 2, Mai 2002. pp 85-124.

[125] SANDBERG R., GOLDBERG D., KLEIMAN S. et al. Design and Implementation of the Sun Network Filesystem. In Proc. of USENIX Association Summer Conference. Portland, OR. 1985. pp. 119-30.

[126] SCHNEIDER F. Implementing Fault-tolerant Services Using the State Machine Approach: A tutorial. ACM Computing Surveys, 22(4), December 1990.

[127] SCHROEDER M.D., BIRRELL A.D., NEEDHAM R.M. Experience with Grapevine: The Growth of a Distributed System. ACM Transactions on Computer Systems, 1984, p.3-23.

[128] SENS P. Contribution à l'intégration de la tolérance aux fautes dans les environnements répartis. Habilitation à diriger les recherches. 2000.

[129] SHAW M., DELINE R., KLEIN D. V. et al. Abstractions for Software Architecture and Tools to Support Them.
IEEE Transanctions on Software Engineering, Vol. 21, No. 6, 1995.
Disponible sur : http://citeseer.nj.nec.com (consulté le 27.03.2003)

[130] SHAW M., GARLAN D. Software Architecture: Perspectives on an Emerging Discipline; Upper Saddle River, NJ US : Prentice Hall, 1996. 0-13-182957-2

[131] SMITH B. Procedural reflexion in Programming Lanuages. Thèse Informatique. États Unis : MIT, 1982.

[132] SLOMAN M. Policy Driven Management for Distributed Systems. Journal of Network and System Management, Vol.2, No. 4, Plenum Press, 1994, pp. 333-360.

[133] SPENCER H., LAWRENCE D. Managing Usenet. O'Reilly & Associates, Sebastopol, CA, USA.

[134] SPREITZER M. J., THEIMER M. M., PETERSEN K. et al. Dealing with Server Corruption in Weakly Consistent, Replicated Data Systems. In : Proc. of the Third Annual ACM/ IEEE International Conference on Mobile Computing and Networking (MobiCom '97), Budapest, Hungary, 1997, pp. 234-240.
Disponible sur : http://www2.parc.com/csl/projects/bayou/ (consulté le 27.03.2003)

[135] STEEN M., HOMBURG P., TANENBAUM A.S. Globe: A Wide-Area Distributed System. IEEE Concurrency. 1999. pp. 70-78.
Disponible sur : http://www.cs.vu.nl/~steen/globe/ (consulté le 27.03.2003)

[136] SUN MICROSYSTEMS. Entreprise Java Beans Specification 2.0. [en ligne] Sun Mycrosystems. 2001.
Disponible sur : http://java.sun.com/products/ejb/docs.html (consulté le 27.03.2003)

[137] SUN MICROSYSTEMS. Java Message Service Specification 1.1 [en ligne]. Sun Mycrosystems. 2002.
Disponible sur : http://java.sun.com/products/jms/docs.html (consulté le 27.03.2003)

[138] SUN MICROSYSTEMS, Java Management Extensions Instrumentation and Agent Specification 1.0. [en ligne] Sun Mycrosystems. 2000.
Disponible sur : http://jcp.org/aboutJava/communityprocess/first/jsr003/
(consulté le 27.03.2003)

[139] SUN MICROSYSTEMS. Remote Method Invocation Specification. [en ligne] Sun Mycrosystems. 1997. Disponible sur :
http://java.sun.com/products/jdk/1.1/docs/guide/rmi/spec/rmiTOC.doc.html

[140] SUN MICROSYSTEMS. Object serialization. [en ligne] Sun Microsystems. 2001.
Disponible sur : http://java.sun.com/j2se/guide/serialization

[141] TANENBAUM A. Distributed operating systems. Upper Saddle River, NJ US : Prentice Hall, 1995. 0-13-219908-4

[142] TERRY D. B., DEMERS A.J., PETERSEN K. et al. Session Guarantees for Weakly Consistent Replicated Data In : Proc. International Conference on Parallel and Distributed Information Systems (PDIS), Austin, Texas, 1994, pp. 140-149.
Disponible sur : http://www2.parc.com/csl/projects/bayou/ (consulté le 27.03.2003)

[143] TERRY D., PETERSEN K., SPREITZER M. et al.The Case for Non-Transparent replication: examples from Bayou. IEEE Data Engineering, December 1998, pages 12-20
Disponible sur : http://www2.parc.com/csl/projects/bayou/ (consulté le 27.03.2003)

[144] TERRY D., THEIMER M., PETERSEN K. et al. Managing Update Conflicts in Bayou, a Weakly Connected Replicated Storage System. In : Proc. of the 15th Symposium on Operating Systems Principles (SOSP-15) , Cooper Mountain, Colorado, December 1995, pages 172-183
Disponible sur : http://www2.parc.com/csl/projects/bayou/ (consulté le 27.03.2003)

[145] THEIMER M., DEMERS A., PETERSEN K. et al. Dealing with Tentative Data Values in Disconnected Work Groups. In : Proc. of the Workshop on Mobile Computing Systems and Applications, Santa Cruz, CA, 1994, pp. 192-195.
Disponible sur : http://www2.parc.com/csl/projects/bayou/ (consulté le 27.03.2003)

[146] The MICO CORBA Component Project [en ligne]
Disponible sur : http://www.fpx.de/MicoCCM/ (consulté le 27.03.2003)

[147] VINGRALEK R., BREITBART Y., SAYAL M. et al. Web++: A System For Fast and Reliable Web Service. In : Proc. of the USENIX Annual Technical Conference, Sydney, Australia, 1999, pp. 171-184.
Disponible sur : http://www.usenix.org/publications/library/proceedings/usenix99/full_papers/vingralek/vingralek.pdf (consulté le 27.03.2003)

[148] WEBER S., NIXON P., TANGNEY B. A Flexible Framework for Consistency management in Object Oriented Distributed Shared memory. TCD-CS-1998-21.
Trinity College Dublin. 1998. p.31.
Disponible sur : http://www.cs.tcd.ie/publications/tech-reports/tr-index.98.html (consulté le 27.03.2003)

[149] WHITE B., WALKER M., HUMPHREY M. et al. LegionFS: A Secure and Scalable File System Supporting Cross-Domain High-Performance Applications. Supercomputing. Denver, Colorado. 2001.
Disponible sur : http://legion.virginia.edu/papers.html (consulté le 27.03.2003)

[150] WIESMANN M., PEDONE F., SCHIPER A., et al. Understanding Replication in Databases and Distributed Systems. In : Proc. of 20th International Conference on Distributed Computing Systems (ICDCS'2000), Taipei, Taiwan : R.O.C., 2000. pp. 264-274

[151] WIESMANN M., PEDONE F., SCHIPER A., et al. Database Replication Techniques: a Three Parameter Classification. In Proc. of the 19th IEEE Symposium on Reliable Distributed Systems (SRDS). 2000.

[152] WOO S., OHARA M., TORRIE E. et al. The SPLASH-2 Programs: Characterization and Methodological Considerations. In : Proc. of the 22nd Annual International Symposium on Computer Architecture. 1995.
Disponible sur : http://www-flash.stanford.edu/apps/SPLASH (consulté le 27.03.2003)

[153] WOOD M., MARZULLO K., Tools for Distributed Application Management, In : Proc of the Spring 1991 European Conference, Tromso, Norway, May 1991, pp 185-196.

[154] YOSHIKAWA C., CHUN B., EASTHAM P. et al. Using Smart Clients to Build Scalable Services. In : Proc. of the USENIX Annual Technical Conference, 1997, Anaheim, California, USA.

[155] YU H., VAHDAT A. Design and Evaluation of a Conit-based Continuous Consistency Model for Replicated Services. ACM Transactions on Computer Systems (TOCS). 2002.

[156] Velocity [en ligne]
Disponible sur : http://jakarta.apache.org/velocity/ (consulté le 27.03.2003)

Annexe A

Mise en œuvre dans OpenCCM

Dans cet annexe nous détaillons notre mise en œuvre de la duplication et de la cohérence dans la plate-forme OpenCCM. Nous illustrons les différentes étapes de gestion des applications métier et des protocoles en utilisant les exemples de l'application métier "Hello World" et du protocole de tolérance aux fautes.

1.1 Construction de l'application "Hello World"

La construction de l'application "Hello World" passe par les étapes de description déclarative, de programmation et de déploiement. Nous détaillons chacune de ces étapes dans ce qui suit.

1.1.1 Description IDL des composants de "Hello World"

La description déclarative de l'application se fait dans le langage IDL qui fait partie de la spécification CORBA. La description IDL des composants de l'application "Hello World" est donnée à la Figure A-1. Cette description contient la définition du client et du serveur en tant que deux types de composants `Client_Comp` et `Server_Comp`.

```
module HelloWorld {
   interface Hello_itf {void print(in string s);};
   component Client_Comp {
      uses Hello_itf serverHello; };        //interface requise, porte de sortie
   home ClientHome manages Client_Comp {}; //maison d'instances pour Client_Comp
   component Server_Comp {
      provides Hello_itf forClientHello;}; //interface fournie, porte d'entrée
   home ServerHome manages Server_Comp {}; //maison d'instances pour Server_Comp
};
```

Figure A-1. Description IDL des composants de "Hello World"

Nous retrouvons la configuration déjà introduite dans le Chapitre 5 : le composant `Client_Comp` requiert l'interface `Hello_itf` (clause `uses`), alors que le composants `Server_Comp` fournit cette interface (clause `provides`).

1.1.2 Programmation des composants

Rappelons que, afin d'introduire une structure de conteneur par instance de composant, nous proposons une augmentation de la structure des composants CCM standards (cf. Chapitre 7). Cette structure augmentée contient des attributs qui représentent explicitement les interfaces requises et fournies d'une instance. Ainsi, nous introduisons des objets d'interception qui permettent la déviation des appels applicatifs vers les traitements de protocole.

Pour arriver à la structure augmentée de composant, nous proposons de rajouter une étape au processus de génération traitant les descriptions CIDL des composants CCM. Pour illustrer ce point, considérons l'exemple d'un composant de type `Comp_Exemple` dont la déclaration IDL est donnée à la Figure A-2. Ce composant requiert une interface `A_itf` et fournit une interface `B_itf`.

```
component Comp_Exemple {
     provides A_itf a;    //interface fournie
     uses B_itf b;        //interface requise
}
```

Figure A-2. Description IDL d'un composant

Supposons que la description de l'implémentation de `Comp_Exemple` spécifie que la classe d'implémentation de `Comp_Exemple` est `Comp_Exemple_Impl`. Le squelette qui est généré pour cette classe implémente l'interface `A_itf`, contient les opérations de navigation permettant la récupération de la référence de la porte `A_itf` et définit également les opérations de gestion des connexions de la porte `B_itf` (Figure A-3.).

```
abstract public class Comp_Exemple_Impl implements A_itf, ...{
   abstract public A_itf provide_a();        //méthode retournant la référence de la porte A_itf a
   public void connect_b(B_itf cnct) {...}  //connexion de la porte B_itf b
   public B_itf disconnect_b() {...}         //déconnexion de la porte B_itf b
   public B_itf get_connection_b() {...}     //référence de la porte d'entrée adressée par B_itf b
   ...
}
```

Figure A-3. Classe générée d'implémentation d'un composant

Pour introduire la structure de conteneur de FAR, nous reprenons la classe générée `Comp_Exemple_Impl` et générons la classe `Comp_Exemple_Impl_Container` qui définit des attributs pour les portes du composant (Figure A-4.).

```
abstract public class Comp_Exemple_Impl_Container implements A_itf, ...{
     //attributs pour les portes
     A_itf a_attribute;
     B_itf b_attribute;
     //méthode retournant la référence de la porte A_itf a, implémentée
     public A_itf provide_a() {return a_attribute;}
     //méthode retournant la référence de l'objet qui implémente A_itf, à implémenter
     abstract public A_itf init_a();
     //méthode retournant la référence de l'objet connecté à B_itf b
     public B_itf init_b() {return get_connection_b();}
     //méthodes de A_itf, à implémenter sans utiliser get_connection_b().
     ...
}
```

Figure A-4. Classe augmentée d'implémentation d'un composant

Ces attributs, `a_attribute` et `b_attribute` respectivement pour la porte `A_itf a` et `B_itf b`, sont utilisés de la manière suivante :

- *Attribut pour interface fournie*. L'attribut `a_attribute` pour l'interfaces fournie `A_itf` contient la référence de la porte d'entrée correspondante. Cet attribut change de valeur selon que le composant est configuré avec un protocole de duplication et de cohérence ou non. Dans le cas sans duplication et cohérence, il référence l'objet qui implémente l'interface `A_itf` et est positionné dans une méthode `init_a()` fournie par le développeur. Dans la cas avec duplication et cohérence, il référence un objet d'indirection qui permet l'appel des structures de protocole.
- *Attribut pour interface requise*. L'introduction d'attributs explicites pour les interfaces requises d'un composant nécessite une modification dans les règles de programmation de composants. En effet, pour assurer le passage par un éventuel objet d'indirection pour la porte de sortie `B_itf`, le développeur du composant `Comp_Exemple` ne doit plus utiliser la méthode `get_connection_b()` pour appeler la porte connectée à `b` mais passer par `b_attribute`[1].

Ne gérant pas de propriétés non fonctionnelles de transaction, de persistance ou de sécurité, OpenCCM 0.2 ne fournit pas d'étape de description déclarative de l'implémentation des composants. Ses squelettes d'implémentation sont générés directement à partir de la description IDL des types de composants. Les classes générées pour l'application "Hello World" sont les classes `ClientCCM`, `ServerCCM`, `Client_Comp_Impl` et `Server_Comp_Impl`. Les deux premières classes définissent les opérations d'introspection et de gestion de connexions spécifiques aux deux types de composants. Les classes `Client_Comp_Impl` et `Server_Comp_Impl` sont les squelettes d'implémentation générés et étendent respectivement les deux classes précédentes.

```
//classes générées, contiennent les méthodes d'introspection et de contrôle de connexions
public abstract class ClientCCM ...{
   public void connect_serverHello(HelloWorld.Hello_itf ct)throws ... {...}
   public HelloWorld.Hello_itf disconnect_serverHello()throws ... {...}
   public HelloWorld.Hello_itf get_connection_serverHello() ... {...}
}
public abstract class ServerCCM ...{
    abstract public HelloWorld.Hello_itf  provide_forClientHello()  {...}
}
//squelettes générées par OpenCCM
public class Client_Comp_Impl extends ClientCCM {...
}
public class Server_Comp_Impl extends ServerCCM implements Hello_itf { ...
}
```

Figure A-5. Structures de composants générées par OpenCCM

Nous reprenons ces squelettes et fournissons aux développeurs des sous-classes qui introduisent les attributs explicites pour les interfaces des composants, ainsi que toutes les structures additionnelles pour la gestion de la duplication et de la cohérence. Les attributs explicites sont ceux présentés précédemment : ils permettent l'introduction d'objets d'indirection. Les structures et les opérations additionnelles sont celles définies dans la classe `Container` de FAR et concernent la référence de l'implémentation du composant de protocole, les méthodes d'inté-

1. Ceci est nécessaire si on veut garder le processus de génération fourni par la plate-forme d'implémentation.

gration d'objets d'indirection de protocole, etc. (cf. Chapitre 6, *Intégration des composants clients et copies*, p. 134).

Les squelettes augmentés sont montrés à la Figure A-6. La classe `Client_Comp_Impl` est étendue par une classe `Client_Comp_Protocol_Impl` qui introduit un attribut `serverHello` pour l'interface requise `Hello_itf`. Cette classe joue le rôle de conteneur pour les instances de `Client_Comp` et donc fournit également les structures définies dans FAR. Par exemple, elle fournit les méthodes `attach` pour la configuration des composants applicatifs avec des composants de protocole, elle contient un attribut qui référence l'objet d'implémentation du composant de protocole, etc.

En ce qui concerne la classe `Server_Comp_Impl`, elle est étendue de la même manière par une classe `Server_Comp_Protocol_Impl` qui implémente les méthodes `provide_forClientHello` et `init_forClientHello` pour la manipulation de l'attribut `forClientHello` représentant l'interface fournie `Hello_itf`. La première méthode est appelée au lancement du composant et initialise l'attribut avec la valeur de `this` pour indiquer que c'est l'objet d'implémentation du composant qui implémente l'interface `Hello_itf`. Dans les cas sans duplication et cohérence, c'est cet objet qui traite directement les appels. La deuxième méthode sert à récupérer l'objet vers lequel l'attribut `forClientHello` pointe, qu'il s'agisse de l'objet d'implémentation de `Hello_itf` ou d'un objet d'indirection.

```
//classes partiellement générées, héritent des classes générées par OpenCCM
public class Client_Comp_Protocol_Impl extends Client_Comp_Impl { ...
   //attribut pour l'interface requise serverHello,
   //instancié dans le cas de non duplication en utilisant get_connection_serverHello()
   Hello_itf serverHello;
   public void run ()   {
     serverHello.print("Hello!"); //appeler la fonction print du service, indirection possible
   }
   //attributs et méthodes générées pour la configuration avec un composant de protocole
   ProtocolComponent pcomp;
   public void attach(...) {...}
   ...
}
public class Server_Comp_Protocol_Impl extends Server_Comp_Impl {
    //attribut pour l'interface fournie,instancié par init_forClientHello, fourni par le développeur
    Hello_itf forClientHello;
    public Hello_itf provide_forClientHello() {return forClientHello;}
    public Hello_itf init_forClientHello() {return this;}
    //attributs et méthodes générées pour la configuration avec un composant de protocole
    ...
}
```

Figure A-6. Programmation des composants de "Hello World"

1.1.3 Déploiement de l'application "Hello World"

Dans OpenCCM, le déploiement est fait par des programmes de déploiement Java. Le programme de déploiement pour l'application "Hello World" effectue les opérations suivantes (Figure A-7.). Tout d'abord, il initialise l'environnement d'exécution des composants OpenCCM, et récupère les références de deux serveurs d'exécution, `server1` et `server2`. Il retrouve ensuite les maisons des composants sur ces deux serveurs et y crée les deux instances du serveur et du client : `serveur` sur `server1` et `client` sur `server2`. Il récupère la référence

de l’interface fournie par l’instance `serveur` et connecte les deux instances. Enfin, il termine la configuration et lance l’exécution.

```
public void deployApplication() {
    // Initialiser l'environnement d'exécution openCCM
    org.omg.CORBA.ORB orb = OpenCCM.Components.Runtime.init(...);
    // Récupérer la référence vers le service de nommage.
    CORBA.Object obj = orb.resolve_initial_references("NameService");
    CosNaming.NamingContext nc = CosNaming.NamingContextHelper.narrow(obj);
    // Récupérer les références vers deux serveurs d'exécution openCCM
    OpenCCM.ComponentServer.Server server1 = ...
    OpenCCM.ComponentServer.Server server2 = ...
    // Créer les maisons d'instances pour Server_Comp et Client_Comp
    ServerHome sh = ...
    ClientHome ch = ...
    // Créer les instances de composant
    Server_Comp serveur = sh.create();
    Client_Comp client = ch.create();
    // Connecter le client au serveur
    Hello_itf forClientHello = serveur.provide_forClientHello();
    client.connect_serverHello(forClientHello);
    // Terminer la configuration
    serveur.configuration_complete();
    client.configuration_complete();}
}
```

Figure A-7. Déploiement de l’application “Hello World”

1.2 Construction du protocole de tolérance aux pannes

Pour intégrer un protocole de tolérance aux fautes dans l’application “Hello World”, l’application doit être instrumentée, le protocole doit être implémenté et le processus de déploiement doit augmenter l’architecture de l’application avec des structures de protocole générés de manière spécifique.

1.2.1 Instrumentation de l’application “Hello World”

L’instrumentation de l’application “Hello World” doit permettre l’accès à l’état du composant serveur, le contrôle de l’état stable de ce serveur, ainsi que le contrôle de la cohérence globale de l’application lors d’une opération de reconfiguration du protocole. Les deux derniers points sont assurés par l’implémentation augmentée des composants qui permet l’intégration d’objets d’indirection. Ces derniers jouent le rôle des talons et des squelettes de FAR en interceptant et en contrôlant les appels aux composants. Le point concernant l’accès à l’état du serveur n’a pas été abordé et est discuté dans ce qui suit.

Pour assurer l’accès à l’état du serveur et permettre l’acheminement de son état capturé à distance, l’implémentation du composant doit changer. Elle doit intégrer une définition d’état en utilisant les types valeurs de CORBA, ainsi qu’une définition d’interface d’accès à cet état (cf. Chapitre 7, Section 7.2.1, p. 143)

- *Définition d’état en utilisant les types valeurs de CORBA*. L’état du serveur est une table de hachage `HashTable` qui peut être modélisée comme une séquence de couples de type `HashElement`. La définition de ce type en termes de types valeurs CORBA est

donnée à la Figure A-8.

- *Définition de l'interface d'accès*. Pour définir l'interface d'accès à l'état du serveur, il est possible d'utiliser des attributs pour les composants CCM. En effet, la définition d'un attribut `state` pour le composant `Server_Comp` sera représentée par deux opérations de consultation et de modification de cet attribut.

```
module HelloWorld {
   interface Hello_itf {void print(in string s);};
   abstract valuetype State {            //type valeur générique
      State get_state();
      set_state(in State s);
   }
   valuetype HashElement {
     string key;
     int value;
   }
   valuetype ServerState:State {         //définition du type d'état du serveur
     private <sequence>HashElement elements;
     put(in string);
     int get(in string);
   }
   component Server_Comp {
      attribute ServerState state;       //définition de l'état sous forme d'attribut
      provides Hello_itf forClientHello;}; //interface fournie, porte d'entrée, facette
};
```

Figure A-8. Définition de l'état du serveur en termes de types valeurs CORBA

1.2.2 Programmation du protocole de tolérance aux fautes

La Figure A-9. donne la description IDL des composants de protocole. Elle contient les définitions des cinq types de composants de protocole : `Timer`, `Manager`, `Client`, `Primaire` et `Secondaire`[1].

- *Composant* `Timer`. Ce composant reçoit les événements de type `startCount` et `okCount` qui servent à débuter et à arrêter la mesure d'un délai de temps. Il émet un événement de type `stopCount` quand ce délai est expiré.
- *Composant* `Manager`. Ce composant reçoit l'événement de type `tooLong` qui signale une trop longue attente de réponse pour un `Client`. Il reçoit également l'événement de type `tooManyHallos`, émis par un `Primaire` quand il contient trop de chaînes commençant par "Hello".
- *Composant* `Client`. Ce composant requière une connexion à un service d'interface `Service_itf`. Il émet, avant et après un appel au service, les événements `startCount` et `okCount` et reçoit des événements `stopCount`.
- *Composant* `Primaire`. Ce composant fournit une interface `Service_itf` et émet l'événement `tooManyHallos`. Il a deux portes de sortie pour ces deux connexions avec les deux secondaires avec lesquels il interagit en utilisant une interface `Consistency_itf`. Cette interface définit l'opération `update(State s)` qui sert au

1. Ceci n'est qu'une transcription en CCM IDL du fonctionnement déjà présenté dans le Chapitre 6.

`Primaire` de mettre périodiquement à jour les secondaires avec des captures de son état.

- *Composant* `Secondaire`. Ce composant fournit, en plus de `Service_itf`, l'interface `Consistency_itf` pour la gestion de la cohérence avec les composants de type `Primaire`.

```
module FT_Protocol {
   eventtype tooLong {};      //événement émis par un Client, après attente de résultat trop longue
   eventtype tooManyHallos{}; //emis par un Serveur avec trop d'élements débutant par Hello
   eventtype startCount {...}; //demande de début de mesure de délai d'attente par un Client
   eventtype okCount {...};    //signale une réception de réponse à temps par un Client
   eventtype stopCount {...};  //émis par l'horloge pour signaler une fin de délai d'attente
   interface Consistency_itf { void update(in State s, ...);};
   interface Service_itf { void callService();};

   component Timer {                    //composant service d'horloge
     consumes startCount beforeClient;
     consumes okCount afterClient;
     emits stopCount toClient;};

   component Manager:ManagerComponent { //composant gestionnaire
      consumes tooLong fromClient;
      consumes tooManyHallos fromServer;};

   component Client {                   //composant client
      uses Service_itf serveur;
      emits tooLong toControl;
      emits startCount toBeforeTimer;
      emits okCount toAfterTimer;
      consumes stopCount fromTimer;};

   component Primaire {                 //composant serveur primaire
      provides Service_itf forClients;
      emits tooManyHallos toControl;
      uses Consistency_itf toSecondaire1;
      uses Consistency_itf toSecondaire2;};

   component Secondaire {               //composant serveur secondaire
      provides Service_itf forClients;
      provides Consistency_itf forPrimaire;};
};
```

Figure A-9. Description IDL des composants de protocole

L'implémentation des composants de protocole est faite de manière standard en suivant la même procédure que celle pour les composants de l'application "Hello World". Les seules différences concernent les composants clients et les composants copies qui implémentent une partie de leurs méthodes en les structurant en pré et post traitements. Vu les limitations d'OpenCCM en termes de gestion non fonctionnelle, ces composants sont implémentés en tant que composants simples et peuvent être directement intégrés dans les structures augmentées des composants métier.

1.2.3 Intégration du protocole

Le protocole de tolérance aux fautes est intégré lors du déploiement de l'application "Hello World" et est défini, comme dans FAR, dans la méthode `deployApplication()` du composant gestionnaire de configuration. Cette méthode effectue les traitements donnés dans le programme de déploiement de "Hello World" (Figure A-7.) tout en rajoutant les instructions qui instancient et qui attachent des composants de protocole aux composant métier (Figure A-10.).

```
void deployApplication() {
	//instructions de déploiement de "HelloWorld"
	...
	//instructions d'intégration de protocole
	client.attach("FT_Protocol.Client");
	serveur.attach("FT_Protocol.Primaire");
	//opérations appelées pour établir les connexions entre composants de protocole
	serveur.protocol_configuration_complete();
	client.protocol_configuration_complete();
}
```

Figure A-10. Instruction de déploiement concernant l'intégration du protocole

Comme dans la mise en œuvre de FAR, les opérations `attach` sont responsables du processus de génération de composants de protocole spécifiques aux types de composants métier.

- *Intégration au niveau du client de "Hello World"*. Dans le cas du composant de protocole `Client`, attaché au client de "Hello World", cette génération produit un objet d'indirection qui entoure l'appel à `print(String s)` par des appels à `pre_print()` et à `post_print()`. L'opération `pre_print()` se charge de l'émission de l'événement `startCount` vers le composant `Timer` pour la mesure du délai d'attente, alors que `post_print()` émet l'événement `okCount` pour signaler que la réponse est reçue.
- *Intégration au niveau du serveur de "Hello World"*. Dans le cas du composant de protocole `Primaire`, attaché au serveur de "Hello World", la génération ne crée pas d'objet d'indirection spécifique puisque l'appel à `print` ne nécessite pas de traitements liés à la gestion de la duplication ou de la cohérence. Toutefois, l'intégration des composants `Primaire` et `Secondaire` nécessite une spécialisation de ces derniers au type `Server_Comp` du serveur. En effet, pour pouvoir émettre l'événement `tooManyHallos`, ces composants ont besoin de consulter l'état spécifique de `Server_Comp`. Pour cela, il est nécessaire d'implémenter de manière spécifique l'opération de capture de cet état laissée abstraite dans l'implémentation générique du protocole.

Le déploiement termine par des appels à `protocol_configuration_complete()` qui déclenchent les traitements de déploiement prédéfinis par les composants de protocole. En font partie, dans cet exemple, l'instanciation de deux composants de type `Secondaire` et leur interconnexion au composant de type `Primaire`.

Annexe B

Mise en œuvre dans JOnAS

Dans cet annexe nous détaillons notre mise en œuvre des principes de gestion de la duplication et de la cohérence dans la plate-forme JOnAS. Nous illustrons les différentes étapes de construction et d'intégration des applications métier et des protocoles en utilisant les exemples de l'application métier "Hello World" et du protocole de tolérance aux fautes.

B.1 Construction de l'application "Hello World"

Le processus EJB standard de construction d'une application métier est composé d'une étape de programmation de composants, d'une étape de description pour les besoins de déploiement, d'une étape de génération des structures de conteneur et finalement, d'une étape de déploiement (Figure B-1.).

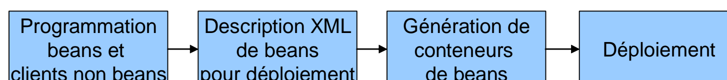


Figure B-1. Construction EJB d'une application métier

Afin d'introduire la gestion de la duplication et de la cohérence dans les applications EJB, nous modifions ce processus en lui rajoutant des étapes supplémentaires (Figure B-2.). Ces étapes sont la description déclarative des dépendances, la génération de squelettes d'implémentation, la modification des structures de conteneur générées par le processus standard, ainsi que la description de l'architecture de l'application à déployer.

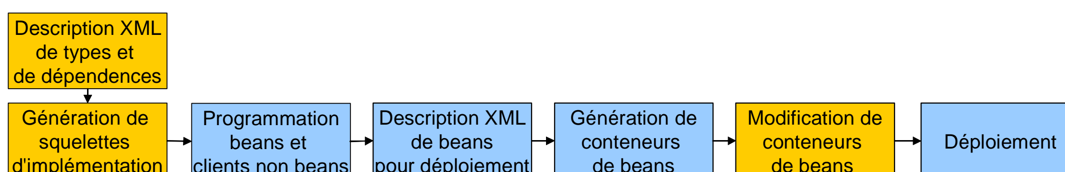


Figure B-2. Construction d'une application métier configurable
en duplication et en cohérence

Après une brève présentation de la modélisation de "Hello World" en termes de beans, nous détaillons chacune des étapes de la construction de cette application.

B.1.1 Modélisation de l'application "Hello World"

L'application "Hello World" permet à des clients de s'adresser à un serveur pour l'affichage de chaînes de caractères. Ce dernier stocke les chaînes affichées, ainsi que le nombre de leurs affichages.

Nous modélisons cette application en termes de trois types d'entités : un client Java, un bean session et un bean entité (Figure B-3.). Le client Java représente le côté client de l'application, alors que le bean session et le bean entité sont utilisés pour structurer le côté serveur de "Hello World". Le client Java permet l'utilisation du service d'affichage et n'est pas programmé en tant que bean. Le bean session est celui qui interagit avec un client et qui traite ses appels de la méthode `print(String s)`. Il est contacté à distance par les clients. Le bean entité est utilisé pour stocker de manière persistante les informations concernant les chaînes affichées. Ses instances sont référencées localement par le bean session.

Étant donné que, pour les besoins de duplication et de cohérence, nous enrichissons les structures des beans, ainsi que des clients non beans, dans la suite nous parlons de *composants* de "Hello World".

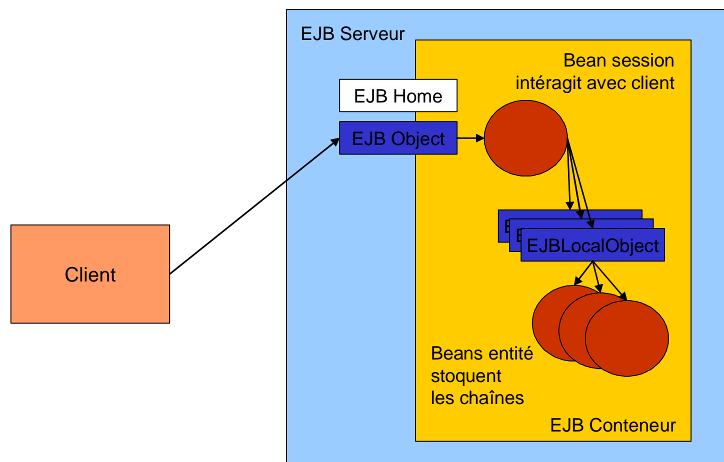


Figure B-3. Modélisation de "Hello World"

B.1.2 Description XML des composants de "Hello World"

Si les types du client, du bean session et du bean entité sont respectivement `Hello_EJBStd_Client`, `Hello_ServerBean` et `Hello_ServerDataBean`, le descripteur de leurs dépendances est donné à la Figure B-4. Il spécifié que le client de type `Hello_EJBStd_Client` a une dépendance d'une instance de type `Hello_itf` qu'il n'initialise qu'une seule fois. Le bean session a une dépendance d'une instance de type `ServerData_itf`[1] dont l'identité est calculée dynamiquement. En effet, le client récupère au lancement la référence de l'instance de bean session avec laquelle il interagit tout au long de son exécution. Une instance de bean session manipule différentes instances entités en fonction de la chaîne qu'il traite pour le client.

1. `ServerData_itf` est l'interface implémentée par `Hello_ServerDataBean`. Elle spécifie une méthode qui permet d'augmenter le nombre d'affichages pour une chaîne.

```
<ejb-client>
     <java_type> Hello_EJBStd_Client </java_type>
     <dep>
        <bean_itf> Hello_itf </bean_itf>
        <name> serverHello </name>
        <number> 1 </number>
        <type> static </type>
     </dep>
</ejb-client>
<ejb-client>
     <bean_type> Hello_ServerBean </bean_type>
     <dep>
        <bean_itf> ServerData_itf </bean_itf>
        <name> currentData </name>
        <number></number>
        <type> dynamic </type>
     </dep>
</ejb-client>
```

Figure B-4. Description XML des dépendances des composants de "Hello World"

B.1.3 Programmation des composants de "Hello World"

La programmation des entités de "Hello World" consiste à augmenter des squelettes d'implémentation générés à partir de la description XML de leurs dépendances. Ces squelettes introduisent les attributs de dépendance, ainsi que les méthodes explicites d'affectation de ces attributs. Ils sont des classes Java héritant d'une classe prédéfinie `Component` qui introduit toutes les structures nécessaires à l'instrumentation d'un composant (bean ou non) en vue d'une configuration de la duplication et de la cohérence. Cette classe `Component` définit, en réalité, une structure de conteneur pour une instance de composant.

Dans le cas de "Hello World" où seulement le client et le bean session ont des dépendances, les squelettes générés sont les suivants (Figure B-5.).

```
public class Hello_EJBStd_Client extends Component {
    //attribut de dépendence
    Hello_itf serverHello;
    //méthode d'affectation pour l'attribut de dépendence
    public void affect_serverHello(Hello_itf sh) {super(...);}
}

public class Hello_ServerBean extends Component {
    //attribut de dépendence
    ServerData_itf currentData;
    //méthode d'affectation pour l'attribut de dépendence
    public void affect_currentData(ServerData_itf cd) {super(...);}
}
```

Figure B-5. Squelettes d'implémentation des composants de "Hello World"

La classe `Hello_EJBStd_Client` définit un attribut `serverHello` qui représente sa dépendance statique vers un bean de type `Hello_itf`. La classe `Hello_ServerBean` définit l'attribut de dépendance dynamique `currentData` vers un bean de type `ServerData_itf`.

Toutes les deux classes définissent les méthodes `affect_<nom attribut>` pour la manipulation explicite des dépendances.

Le développeur reprend ces classes pour définir les traitements applicatifs. Ces traitements ne suivent pas les patrons de programmation utilisés traditionnellement dans les programmes EJB mais utilisent les attributs de dépendance et les méthodes pour leur affectation.

Considérons le cas du client (Figure B-6.). Le code EJB typique crée une nouvelle instance du bean session Hello_ServerBean et affecte sa référence à une variable interne. (Figure B-6.a). Avec un attribut de dépendance, la référence de l'instance est affectée explicitement à l'attribut de dépendance (Figure B-6.b).

```
classe Hello_EJBStd_Client { ...
   //recherche de la maison d'instances auprès du service de nommage
   Hello_ServerHome h = ...lookup("java:env/comp/ejb/Hello_ServerBean");
   Hello_itf b = h.create(...);          //création de l'instance de session, affectation de variable interne
   b.print(...);                         //utilisation de la référence, appel
}                                                                                  (a)
```

```
classe Hello_EJBStd_Client ...{
   Hello_itf serverHello;                //attribut de dépendence
   void affect_serverHello(Hello_itf s) {...}
   ...
   affect_serverHello(h.create(...));    //création instance, affectation de l'attribut
   serverHello.print(...);               //appel
}                                                                                  (b)
```

Figure B-6. Gestion explicite de références dans un client de bean

Dans le cas du bean session, le développeur fournit le code de la méthode `print`, ainsi que l'implémentation d'interfaces obligatoires du modèle EJB dont, par exemple, l'interface `SessionBean`.

B.1.4 Génération des structures de conteneur

Après l'étape de programmation, le développeur de beans doit fournir des descripteurs de déploiement qui spécifient la gestion des transactions et de la persistance. Par exemple, dans le cas du bean session de “Hello World”, le descripteur peut spécifier que les transactions sont gérées par le conteneur et que la méthode `print` doit être exécutée dans un contexte transactionnel. Pour le bean entité de “Hello World”, il peut spécifier que la persistance est également gérée par le conteneur et définir les attributs persistants comme la clé primaire des données représentées par ce bean.

Les descripteurs de déploiement et les classes Java d'implémentation des beans sont utilisés pour générer les classes des objets d'interposition des beans. Ce sont les objets `EJBObject` (cf. Chapitre 8) qui s'occupent de la gestion non fonctionnelle des beans et qui font partie de leur conteneur. Dans JOnAS, par exemple, les classes d'interposition générées pour le bean session `Hello_ServerBean` et pour le bean entité `Hello_ServerDataBean` de “Hello World” sont `JOnASHello_ServerBeanRemote` et `JOnASHello_ServerDataBeanLocal`. La classe `JOnASHello_ServerBeanRemote` assure l'accès à distance par Java RMI. La classe `JOnASHello_ServerDataBeanLocal` assure l'accès par des appels Java normaux[1]

1. Rappelons que les beans de type `Hello_ServerDataBean` ne sont accédés que de manière locale par les beans `Hello_ServerBean`.

Afin de pouvoir intégrer la gestion de la duplication et de la cohérence, nous modifions les objets d'interposition générés. En modifiant l'arbre de héritage des classes d'interposition, nous introduisons des indirections, ainsi que des traitements additionnels pour l'interception et le contrôle des transactions. Par exemple, dans le cas de la classe d'interposition `JOnASHello_ServerBeanRemote` qui hérite de la classe prédéfinie de JOnAS `JSession-Remote`, nous créons une classe `JOnASHello_ServerBeanProtocolRemote` qui hérite de `JSessionRemote` et qui devient la classe mère de `JOnASHello_ServerBeanRemote` (Figure B-7.).

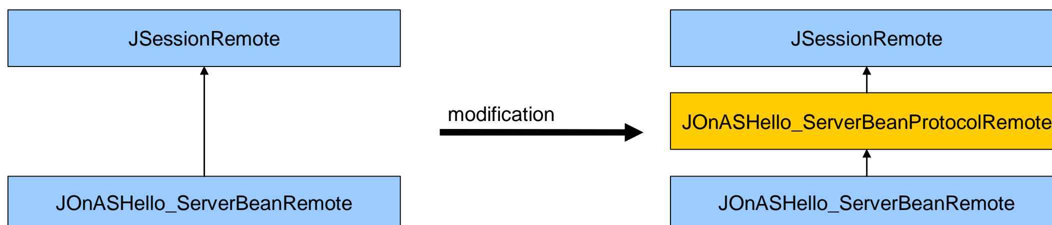


Figure B-7. Modification des classes d'interposition

Lors de la modification des classes d'interposition, nous préservons au maximum les structures standards de JOnAS. Nous ne modifions pas des classes sur lesquelles repose toute l'architecture de la plate-forme et bénéficions des outils de génération de code. La transformation que nous effectuons est limitée au code d'interposition de JOnAS et permet la définition des traitements additionnels de manière groupée dans les classes nouvellement générées. En effet, c'est dans ces classes que nous définissons toutes les indirections vers les traitements de duplication et de cohérence, ainsi que tous les traitements de contrôle des transactions. Dans le code d'interposition de JOnAS nous ne faisons qu'insérer des appels à ces traitements. Pour cela, nous exploitons le fait que tout traitement métier est intercepté et entourée par des appels à deux méthodes, `preInvoke` et `postInvoke`, qui définissent les pré et post traitements de gestion non fonctionnelle.

B.1.5 Déploiement

Nous fournissons l'information sur l'architecture de l'application déployée dans un descripteur XML qui précise quels sont les types de beans qui correspondent aux interfaces des dépendances des beans. Pour "Hello World" ce descripteur est le suivant (Figure B-8.).

```
<application>
     <components>
        <java_client>Hello_EJBStd_Client</java_client>
        <bean>Hello_ServerBean</bean>
        <bean>Hello_ServerDataBean</bean></components>
     <dependency>
        <out>Hello_EJBStd_Client.serverHello</out>
        <in>Hello_ServerBean</in></dependency>
     <dependency>
        <out>Hello_ServerBean.currentData</out>
        <in>Hello_ServerDataBean</in>
     </dependency>
</application>
```

Figure B-8. Description de l'architecture de "Hello World"

Cette description précise que la dépendance du client sera un bean de type Hello_ServerBean. De même, la dépendance des beans de type Hello_ServerBean serons des beans de type Hello_ServerDataBean.

B.2 Construction du protocole de tolérance aux fautes

Après avoir décrit la construction de l'application métier "Hello World", nous présentons dans cette section la construction du protocole de tolérance aux pannes. Nous commençons par discuter de l'instrumentation de l'application "Hello World", parlons brièvement de la structuration du protocole dans le modèle EJB et terminons par son intégration dans "Hello World".

B.2.1 Instrumentation de l'application "Hello World"

L'instrumentation de l'application "Hello World" consiste en l'introduction de primitives d'accès à l'état des beans et en la définition de structures permettant la gestion de la cohérence l'application[1]. Étant donné que les conditions d'état stable sont garanties par la gestion des différents types de composants EJB ou alors font partie de la gestion des transactions (cf. Chapitre 8), la gestion de la cohérence applicative est assurée par les traitements de contrôle des transactions insérés dans les objets d'interposition modifiés. Quant à l'accès à l'état, il est traité comme suit.

Nous donnons accès à l'état des beans qui composent le serveur de "Hello World", `Hello_ServerBean` et `Hello_ServerDataBean`, en utilisant des interfaces d'accès à l'état. Dans le cas de `Hello_ServerDataBean`, l'interface d'accès à l'état est fournie par le conteneur qui est chargé de sa persistance, alors que dans le cas `Hello_ServerBean`, l'interface est fournie par le développeur de bean. En effet, l'état d'une instance de `Hello_ServerDataBean` est constitué de deux attributs persistants, chaîne et nombre d'affichages, et peuvent être manipulés à l'aide de primitives automatiquement générées. L'état d'une instance de `Hello_ServerBean` représente, quant à lui, l'état du serveur d'affichage de chaînes et correspond donc à une collection des états de toutes les instances de `Hello_ServerDataBean`. Par conséquent, nous définissons la capture et la restauration de son état comme une suite de captures et de restaurations des états des instances de `Hello_ServerDataBean`.

B.2.2 Programmation du protocole de tolérance aux fautes

Le protocole de tolérance aux fautes est composé d'un gestionnaire, d'un composant de type `Primaire`, de deux composants de type `Secondaire`, de clients de type `Client` et de horloges de type `Timer` (cf. Chapitre 6). Dans le modèle EJB, le composant `Client` qui représente le côté client dans le protocole est programmé en tant qu'un client Java. Le composant `Timer` qui est utilisé par le client est défini comme un bean session. Finalement, les composants `Gestionnaire`, `Primaire` et `Secondaire` qui doivent avoir des identités persistantes sont modélisés sous forme de beans entités.

La programmation des composants de protocole suit les mêmes règles que celles définies pour les applications métier. Les seules différences concernent les composants clients et les

1. Rappelons que les conditions d'état stable sont garanties par la gestion des différents types de composants EJB ou alors font partie de la gestion des transactions. (cf. Chapitre 8)

composants copies, c'est à dire les composants `Client`, `Primaire` et `Secondaire`. Leurs classes d'implémentation doivent fournir une interface d'accès locale et hériter d'une classe `ProtocolComponent`. L'interface locale est nécessaire pour établir, lors de l'intégration du protocole au sein d'une application, des communications efficaces entre les composants applicatifs et les composants de protocole. Le héritage de `ProtocolComponent` est nécessaire pour permettre à un composant de protocole de référencer le composant métier qu'il contrôle[1].

B.2.3 Intégration du protocole

L'intégration du protocole de tolérance aux fautes dans l'application "Hello World" passe par une étape de description des correspondances entre les entités métier et les entités de protocole. Les correspondances sont les suivantes :

- *Interfaces* `Hello_itf` *et* `Service_itf`. La correspondance entre l'interface `Hello_itf` de "Hello World" et l'interface `Service_itf` du protocole est nécessaire pour la mise en place de l'interception des appels des clients et l'introduction de la mesure du l'attente de résultat fournie par les composants `Timer`.
- *Composants clients*. La correspondance entre le composant `Client` de "Hello World" et le Client du protocole repose sur la correspondance entre interfaces et met effectivement en place l'interception et la mesure de l'attente pour les appels au `print` du serveur d'affichage de chaînes.
- *Composant* `Hello_ServerBean` *et* `Primaire`. La correspondance entre le composant `Hello_ServerBean` de "Hello World" et le composant `Primaire` du protocole introduit les traitements de sauvegarde pour le serveur d'affichage de chaînes de caractères. Nous rattachons les traitements définis dans le `Primaire` uniquement au composant `Hello_ServerBean`, même si le serveur est également composé d'instances de `Hello_ServerDataBean`. Nous supposons que les traitements de capture et de restauration d'état de `Hello_ServerBean` se chargent de recréer la structure composée du serveur. Cette information pourrait être explicitement modélisée avec un modèle *composite* de composants (cf. Chapitre 1), ce qui n'est pas notre cas.

Une possible architecture de l'application "Hello World", après intégration du protocole de tolérance aux fautes, est donnée à la Figure B-9.

1. La classe `ProtocolComponent` permet aux composants de protocole de référencer le composant métier, alors que la classe `Component` permet à un composant métier de référencer les structures de protocole. La structuration en `Component` et en `ProtocolComponent` est la même que dans FAR.

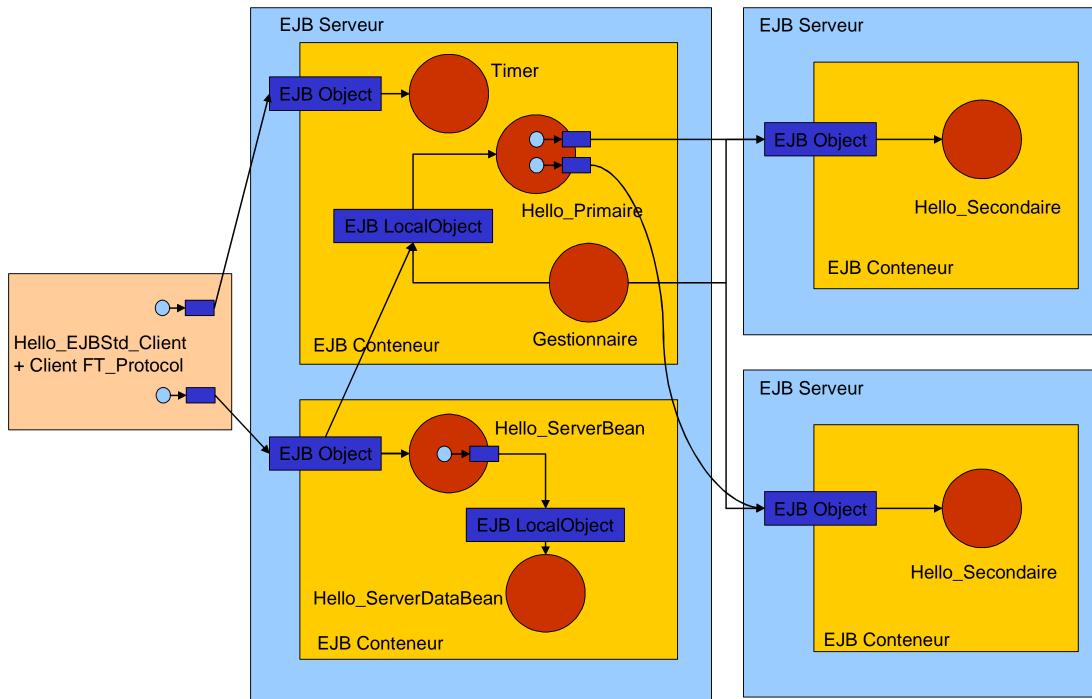


Figure B-9. L'application “Hello World” après intégration du protocole de tolérance aux fautes